

Basic CUDA Optimization

Bin ZHOU @ NVIDIA & USTC
Jan. 2015

Agenda

- ▶ Parallel Reduction
- ▶ Warp Partitioning
- ▶ Memory Coalescing
- ▶ Bank Conflicts
- ▶ Dynamic Partitioning of SM Resources
- ▶ Data Prefetching
- ▶ Instruction Mix
- ▶ Loop Unrolling

Efficient data-
parallel algorithms +

Optimizations based
on GPU Architecture =

Maximum
Performance

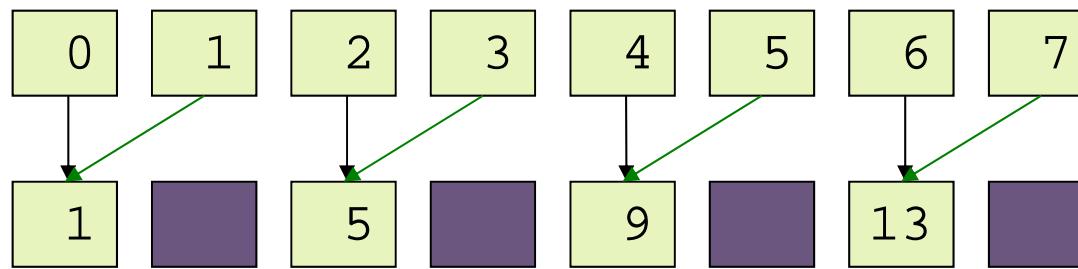
Parallel Reduction

- Recall *Parallel Reduction* (sum)

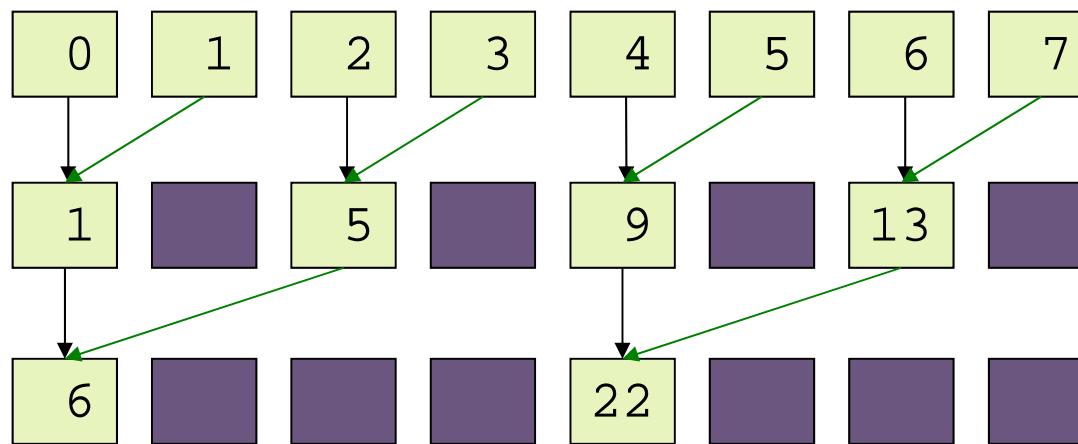


- $S = \sum_{i=0}^N a_i$

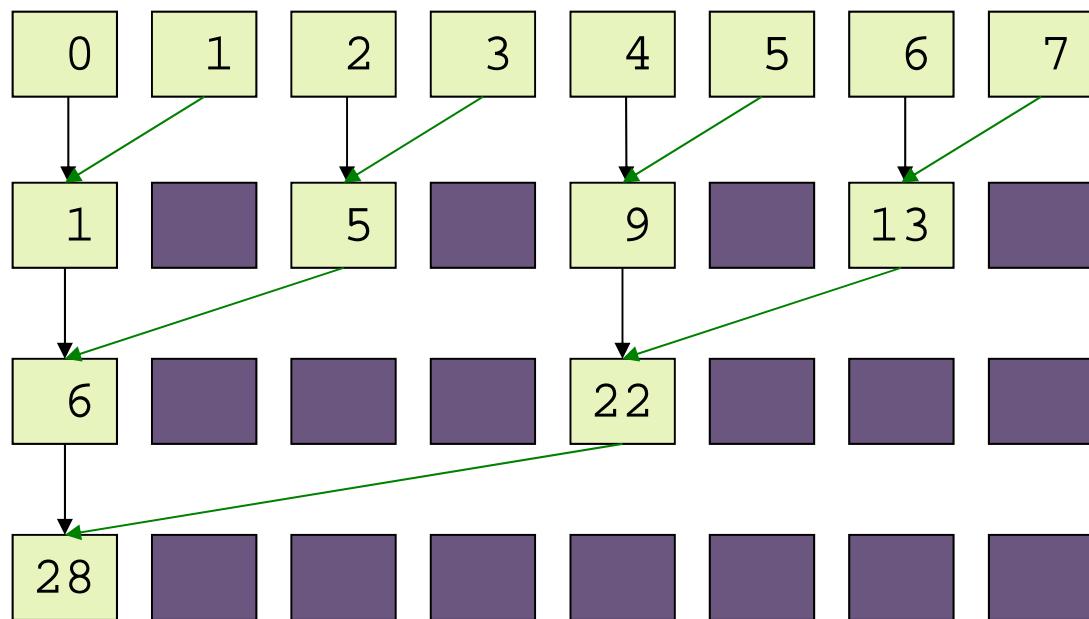
Parallel Reduction



Parallel Reduction



Parallel Reduction

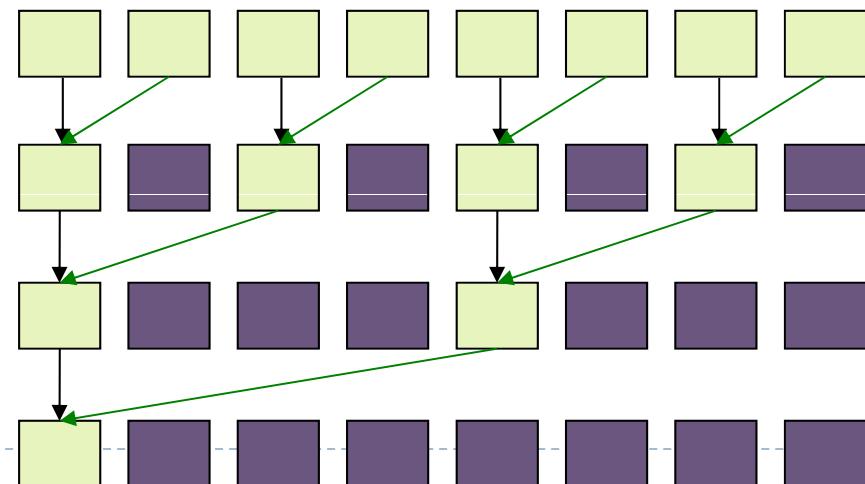


Parallel Reduction

Similar to brackets for a basketball tournament

$\log(n)$ passes for n elements

How would you implement this in CUDA?



```
__shared__ float partialSum[ ];  
// ... load into shared memory  
unsigned int t = threadIdx.x;  
for (unsigned int stride = 1;  
     stride < blockDim.x;  
     stride *= 2)  
{  
    __syncthreads();  
    if (t % (2 * stride) == 0)  
        partialSum[t] +=  
            partialSum[t + stride];  
}
```

```
__shared__ float partialSum[ ];  
// ... load into shared memory
```

```
unsigned int t = threadIdx.x;  
for (unsigned int stride = 1;  
     stride < blockDim.x;  
     stride *= 2)  
{
```

Computing the sum for the
elements in shared memory

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

▶ }
0

```

__shared__ float partialSum[ ] ;

// ... load into shared memory

unsigned int t = threadIdx.x;

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

{

    __syncthreads();

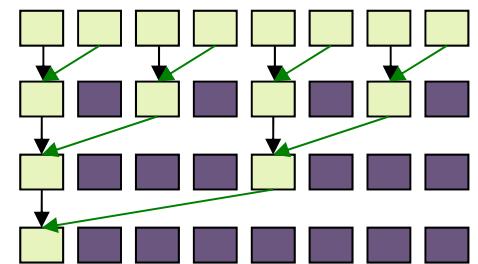
    if (t % (2 * stride) == 0)

        partialSum[t] +=
            partialSum[t + stride];

}

```

Stride:
1, 2, 4, ...

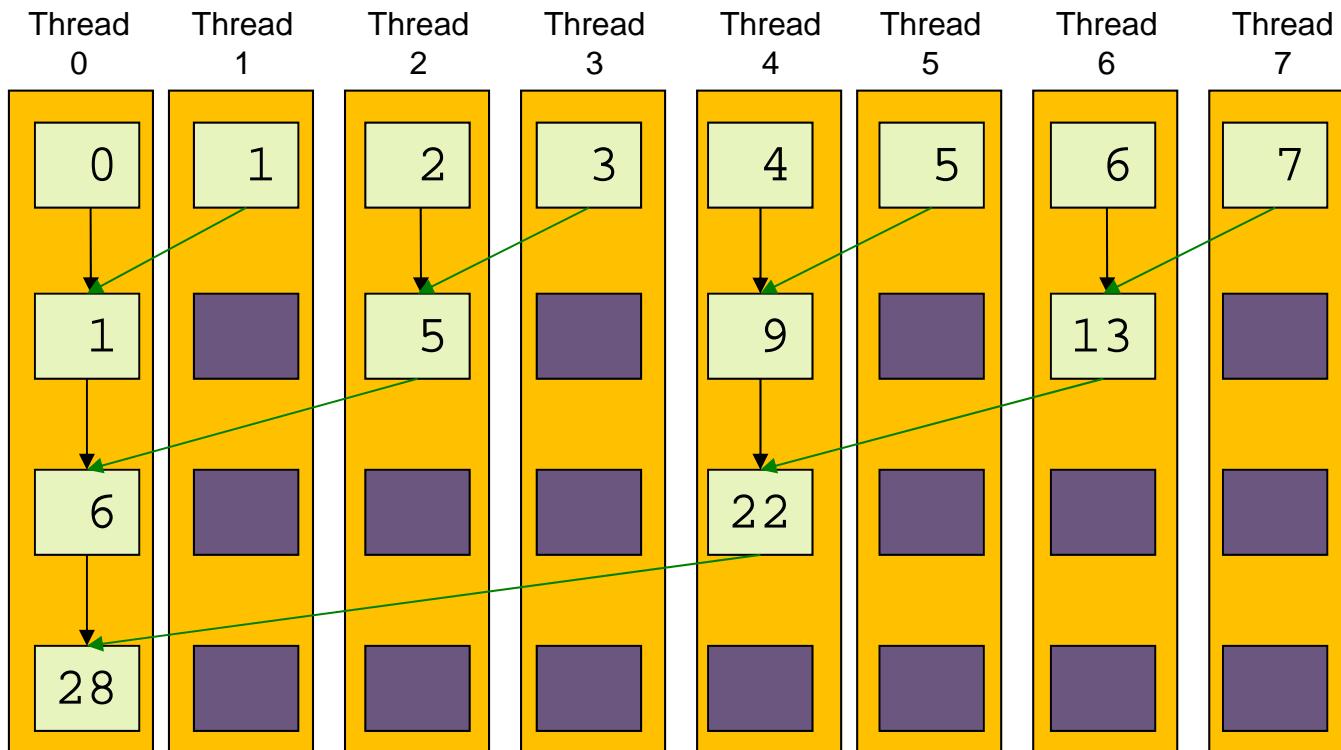


```
__shared__ float partialSum[ ];  
// ... load into shared memory  
unsigned int t = threadIdx.x;  
for (unsigned int stride = 1;  
     stride < blockDim.x;  
     stride *= 2)  
{  
    __syncthreads(); ← Why?  
    if (t % (2 * stride) == 0)  
        partialSum[t] +=  
            partialSum[t + stride];  
}
```

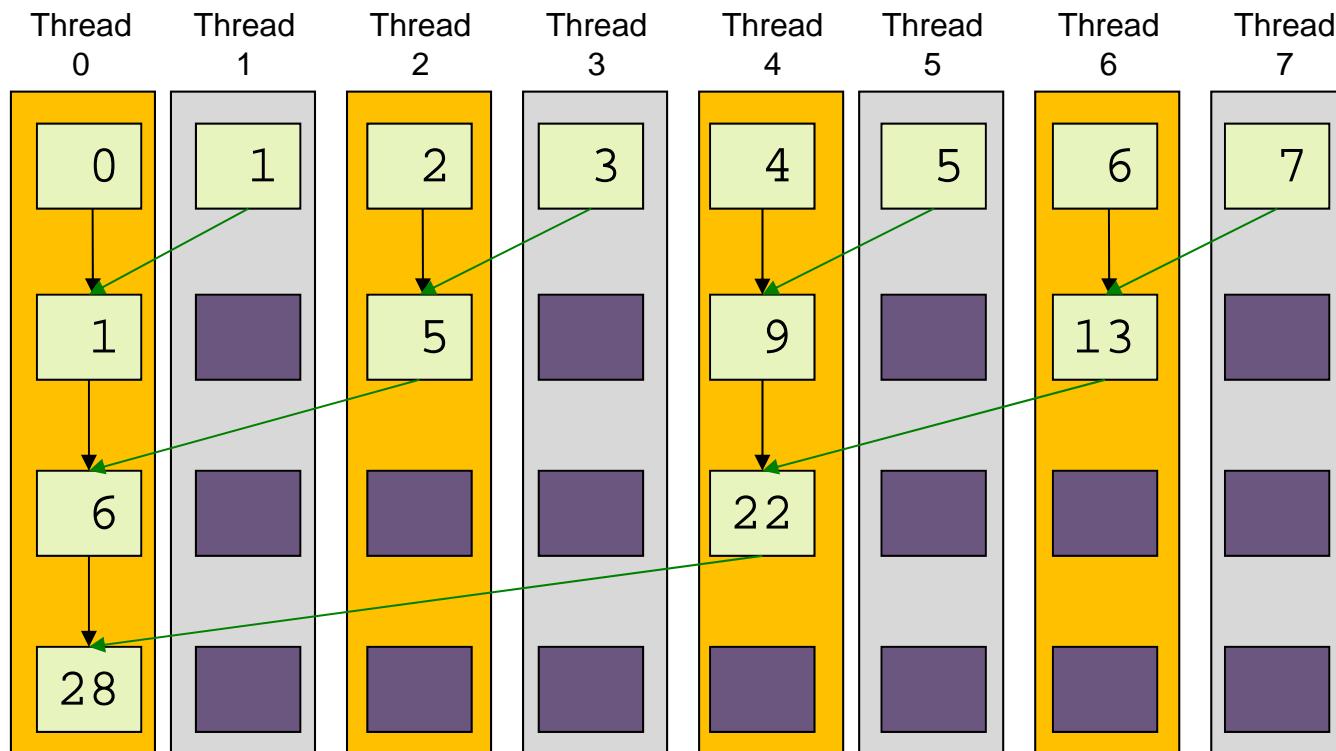
```
__shared__ float partialSum[ ];  
// ... load into shared memory  
unsigned int t = threadIdx.x;  
for (unsigned int stride = 1;  
     stride < blockDim.x;  
     stride *= 2)  
{  
    __syncthreads();  
    if (t % (2 * stride) == 0)  
        partialSum[t] +=  
            partialSum[t + stride];  
}
```

- Compute sum in same shared memory
- As stride increases, what do more threads do?

Parallel Reduction



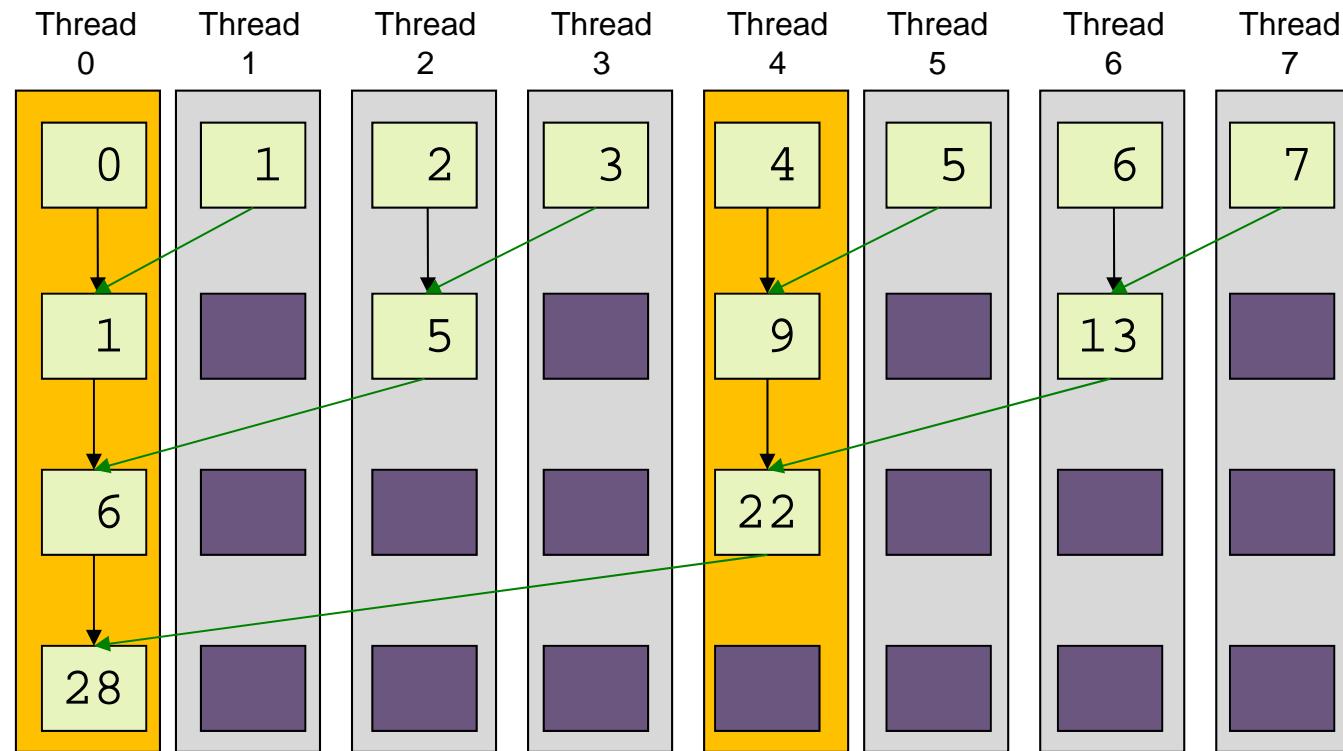
Parallel Reduction



1st pass: threads 1, 3, 5, and 7 don't do anything

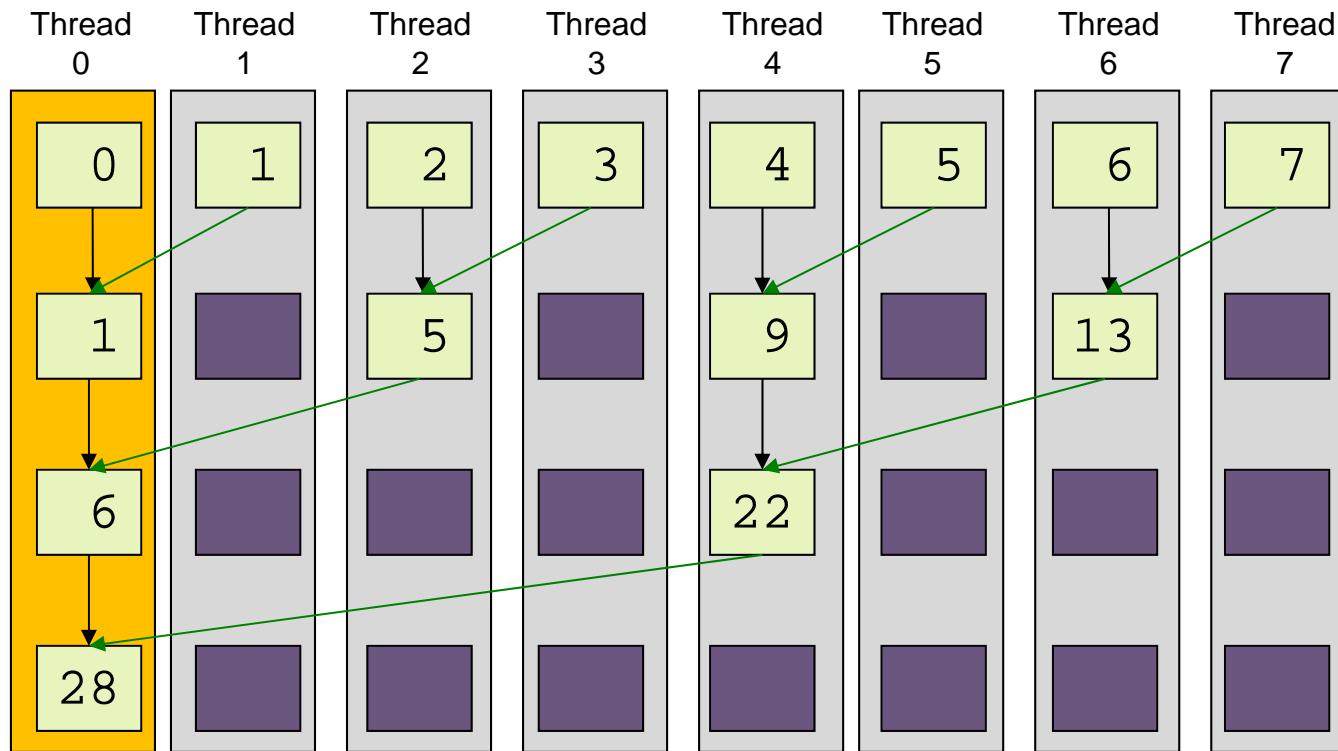
Really only need $n/2$ threads for n elements

Parallel Reduction



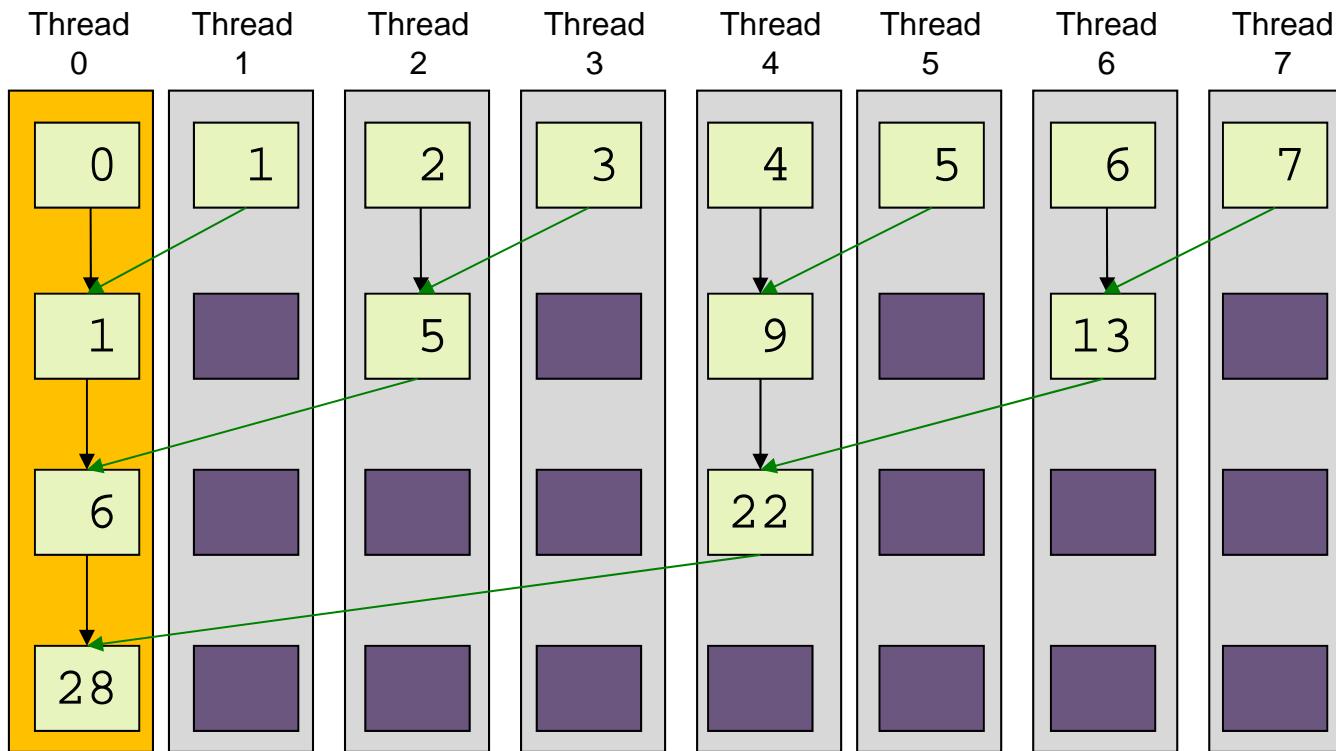
2nd pass: threads 2 and 6 also don't do anything

Parallel Reduction



3rd pass: thread 4 also doesn't do anything

Parallel Reduction



In general, number of required threads cuts in half after each pass

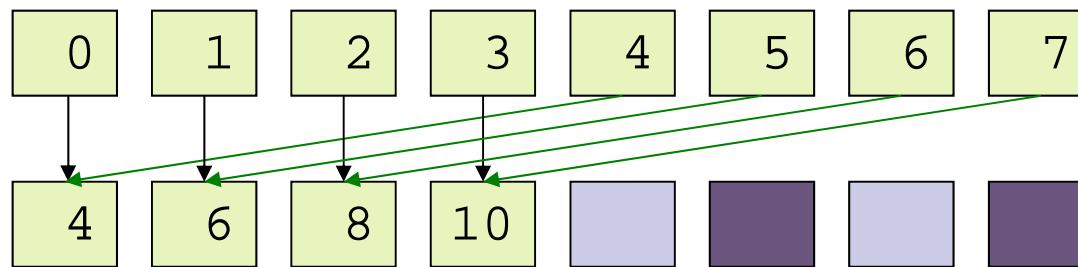
Parallel Reduction

What if we *tweaked* the implementation?

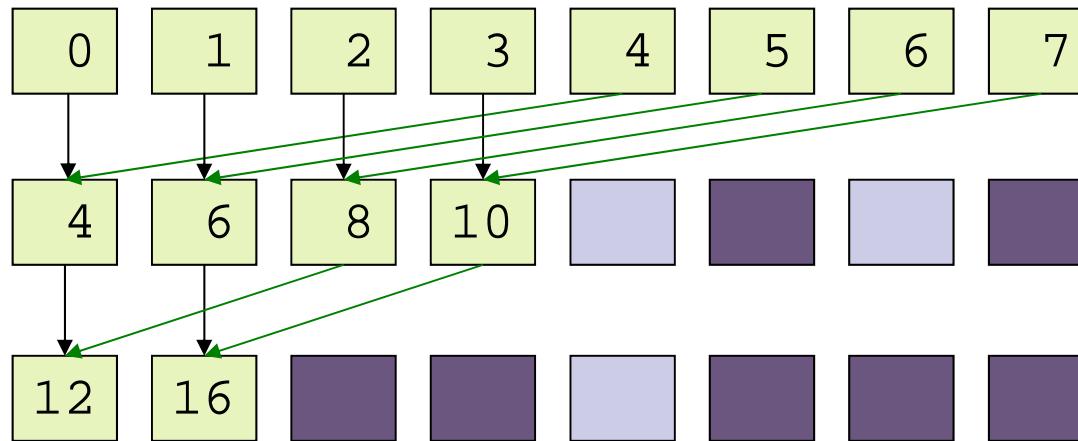
Parallel Reduction



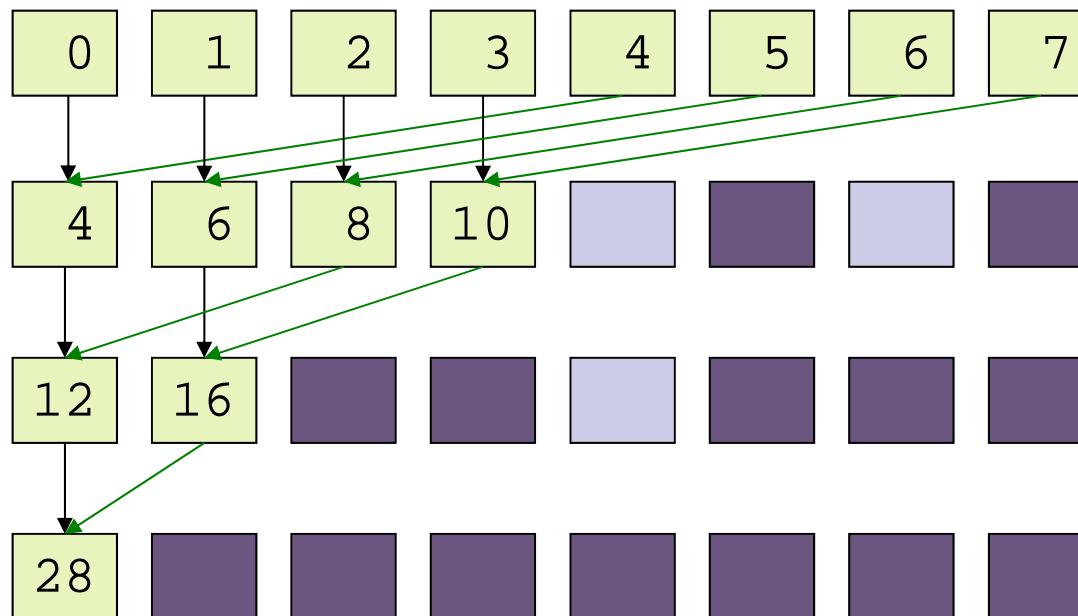
Parallel Reduction



Parallel Reduction



Parallel Reduction

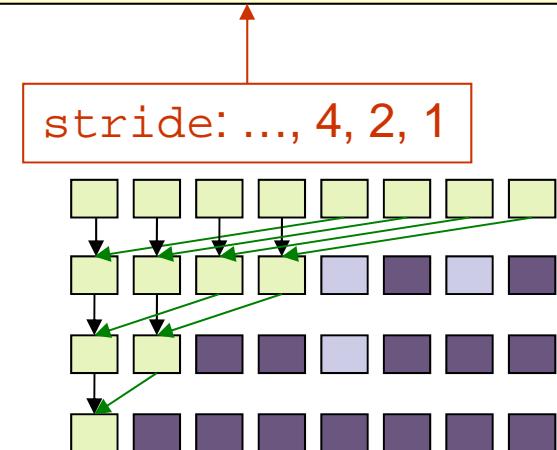


```

__shared__ float partialSum[ ]
// ... load into shared memory
unsigned int t = threadIdx.x;
for(unsigned int stride = blockDim.x / 2;
    stride > 0;
    stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] +=
            partialSum[t + stride];
}

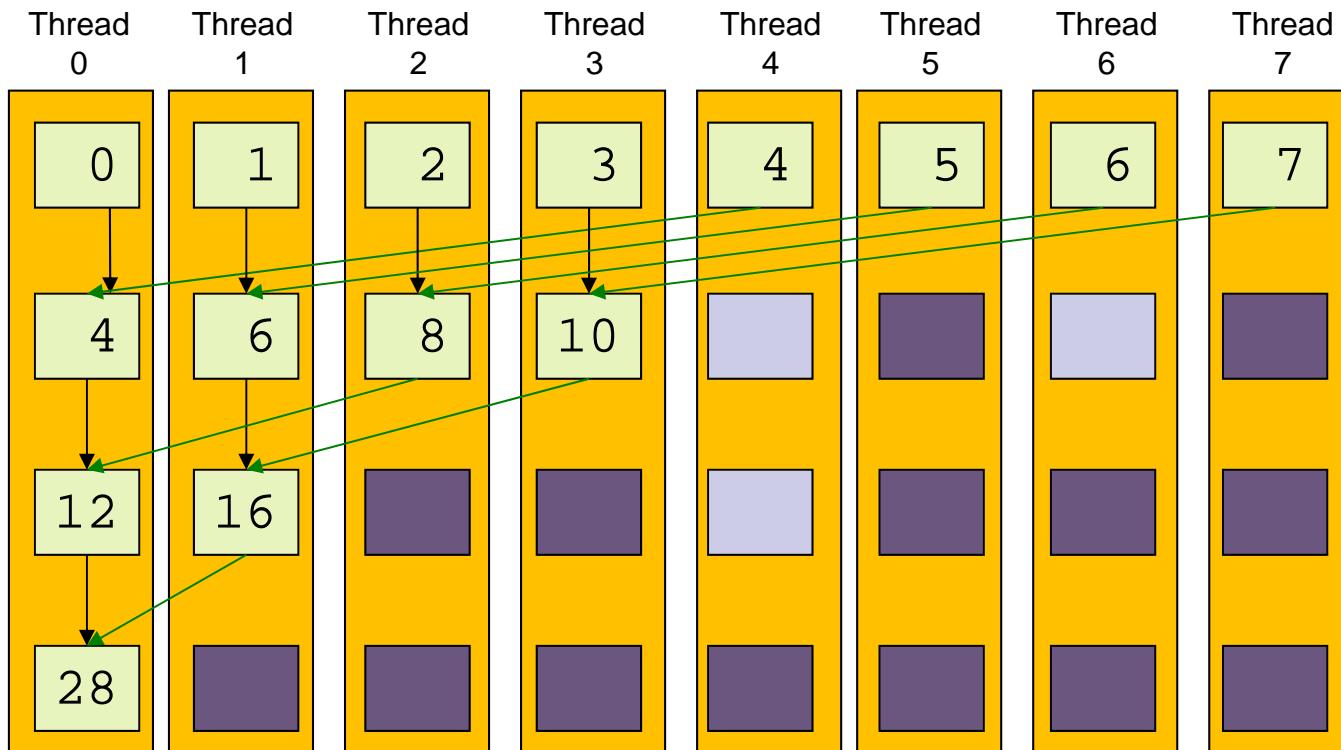
```

▶ 24

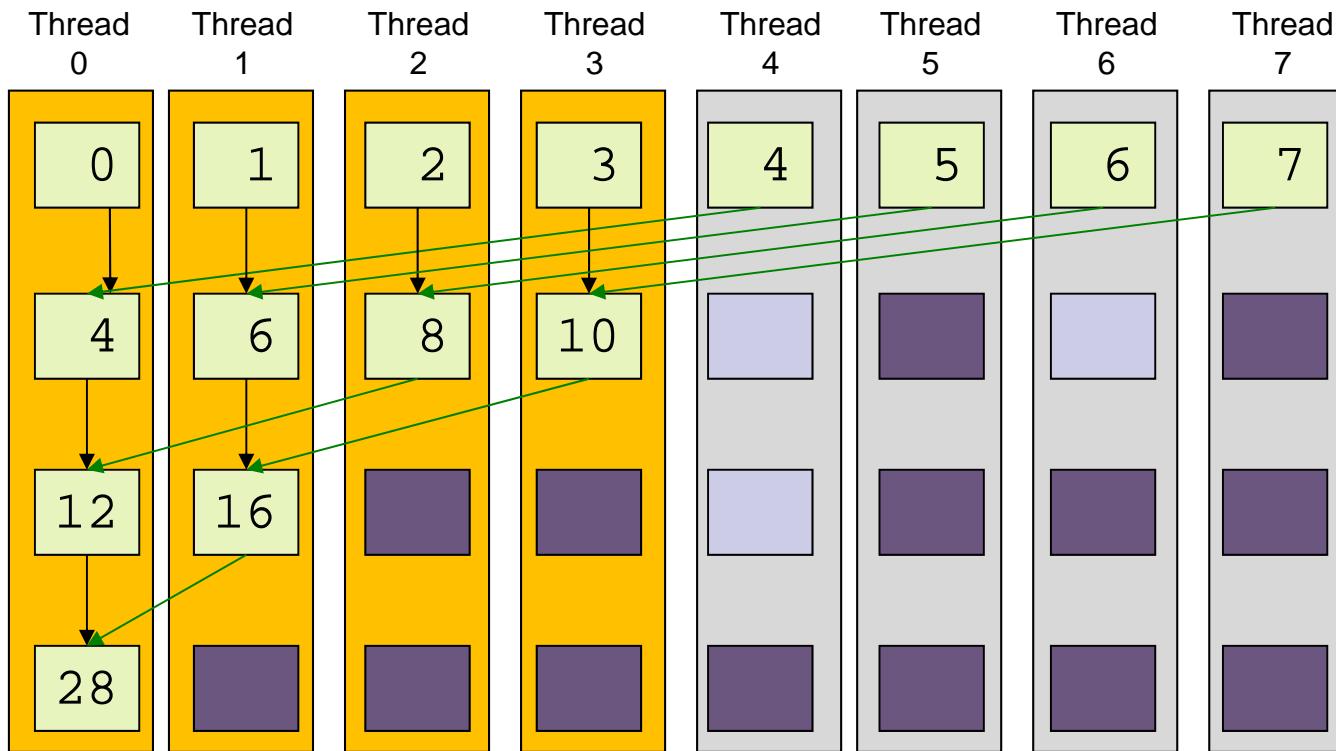


```
__shared__ float partialSum[ ]  
// ... load into shared memory  
unsigned int t = threadIdx.x;  
for(unsigned int stride = blockDim.x / 2;  
    stride > 0;  
    stride /= 2)  
{  
    __syncthreads();  
    if (t < stride)  
        partialSum[t] +=  
            partialSum[t + stride];  
}
```

Parallel Reduction



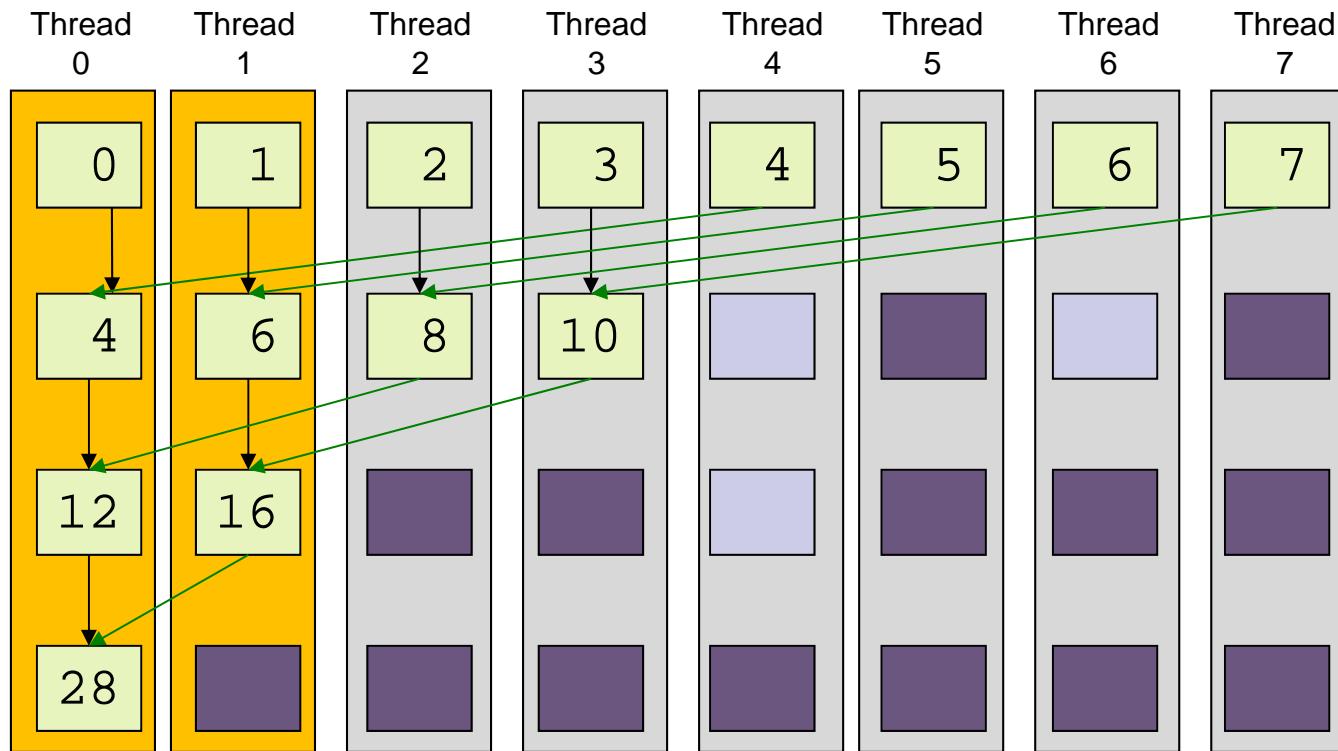
Parallel Reduction



1st pass: threads 4, 5, 6, and 7 don't do anything

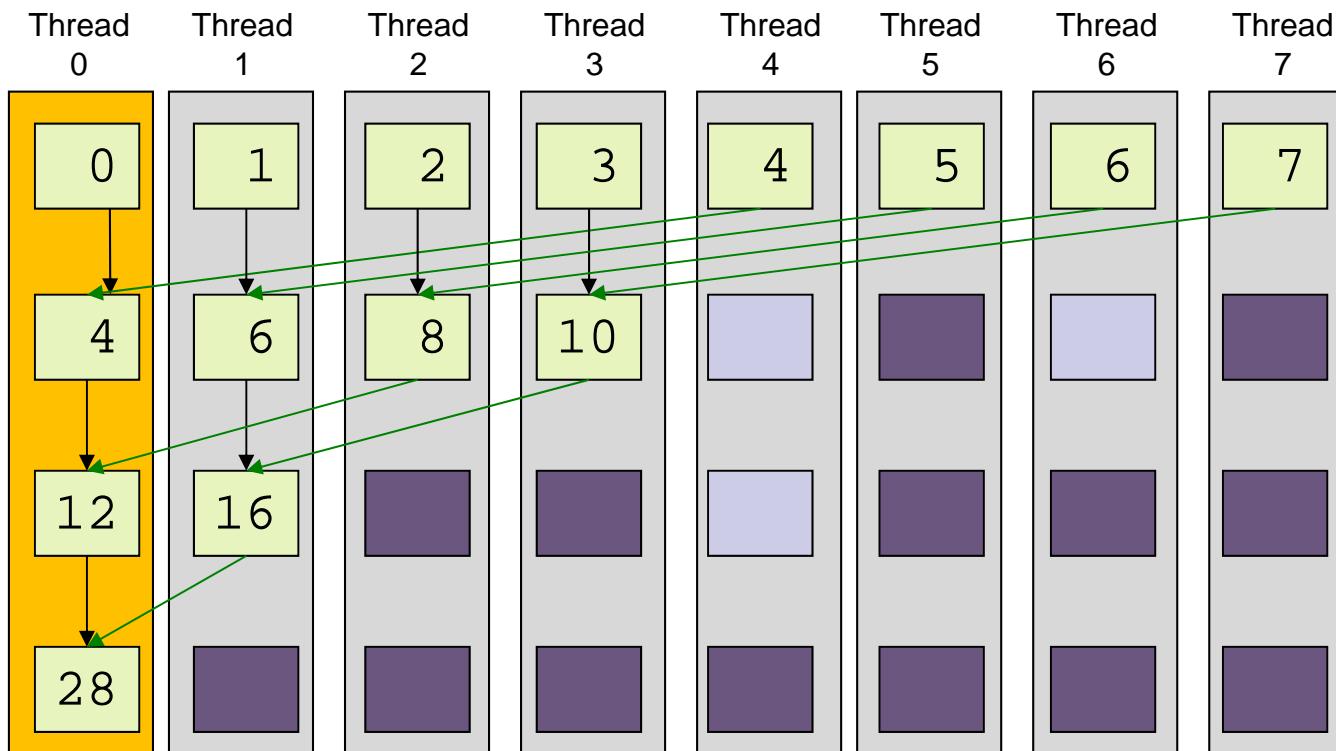
Really only need $n/2$ threads for n elements

Parallel Reduction



2nd pass: threads 2 and 3 also don't do anything

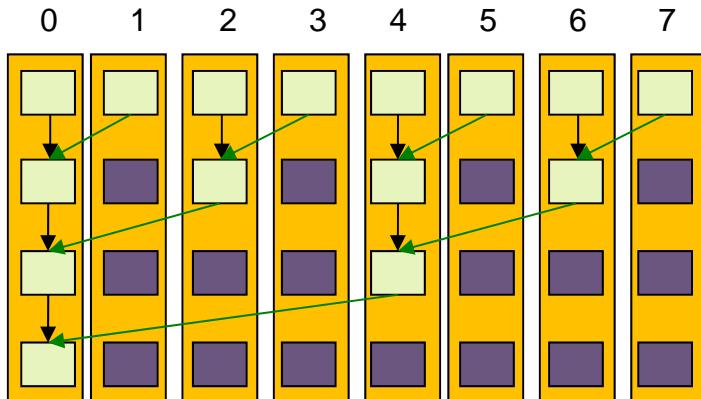
Parallel Reduction



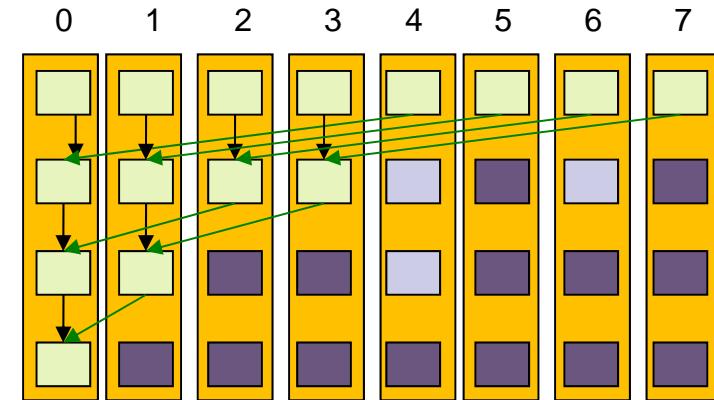
3rd pass: thread 1 also doesn't do anything

Parallel Reduction

What is the difference?



stride = 1, 2, 4, ...



stride = 4, 2, 1, ...

Parallel Reduction

What is the difference?

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

stride = 1, 2, 4, ...

```
if (t < stride)
    partialSum[t] +=
        partialSum[t + stride];
```

stride = 4, 2, 1, ...

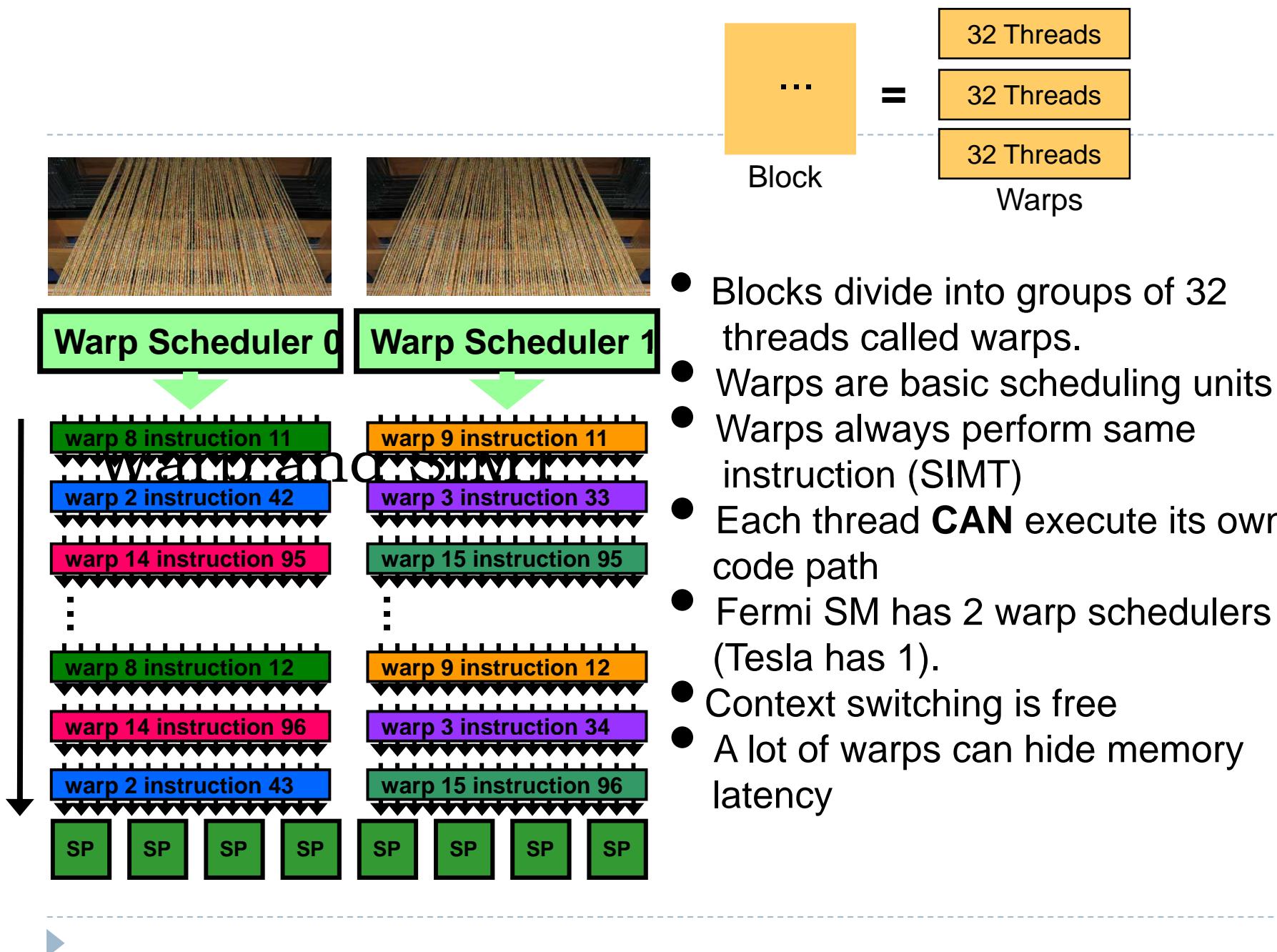


Warp Partitioning

Warp Partitioning: how threads from a block are divided
into warps

Knowledge of warp partitioning can be used to:
Minimize divergent branches
Retire warps early





Warp Partitioning

Partition based on *consecutive increasing* threadIdx



Warp Partitioning

ID Block

`threadIdx.x` between 0 and 512 (G80/GT200)

Warp n

Starts with thread $32n$

Ends with thread $32(n + 1) - 1$

Last warp is padded if block size is not a multiple of 32

35

Warp 0

0...31

Warp 1

32...63

Warp 2

64...95

Warp 3

96...127

...



Warp Partitioning

2D Block

Increasing `threadIdx` means

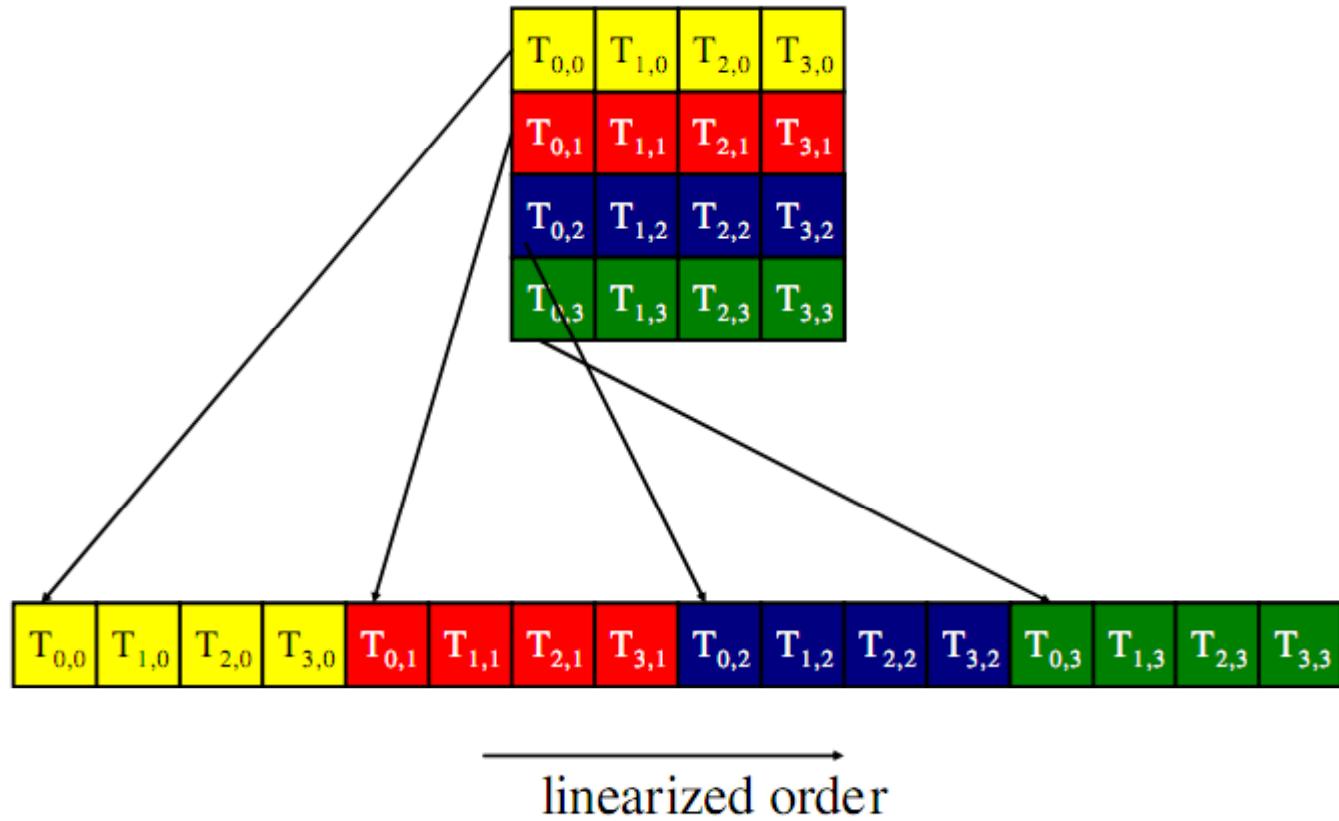
Increasing `threadIdx.x`

Starting with row `threadIdx.y == 0`



Warp Partitioning

2D Block



37

Image from <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf>

Warp Partitioning

3D Block

Start with `threadIdx.z == 0`

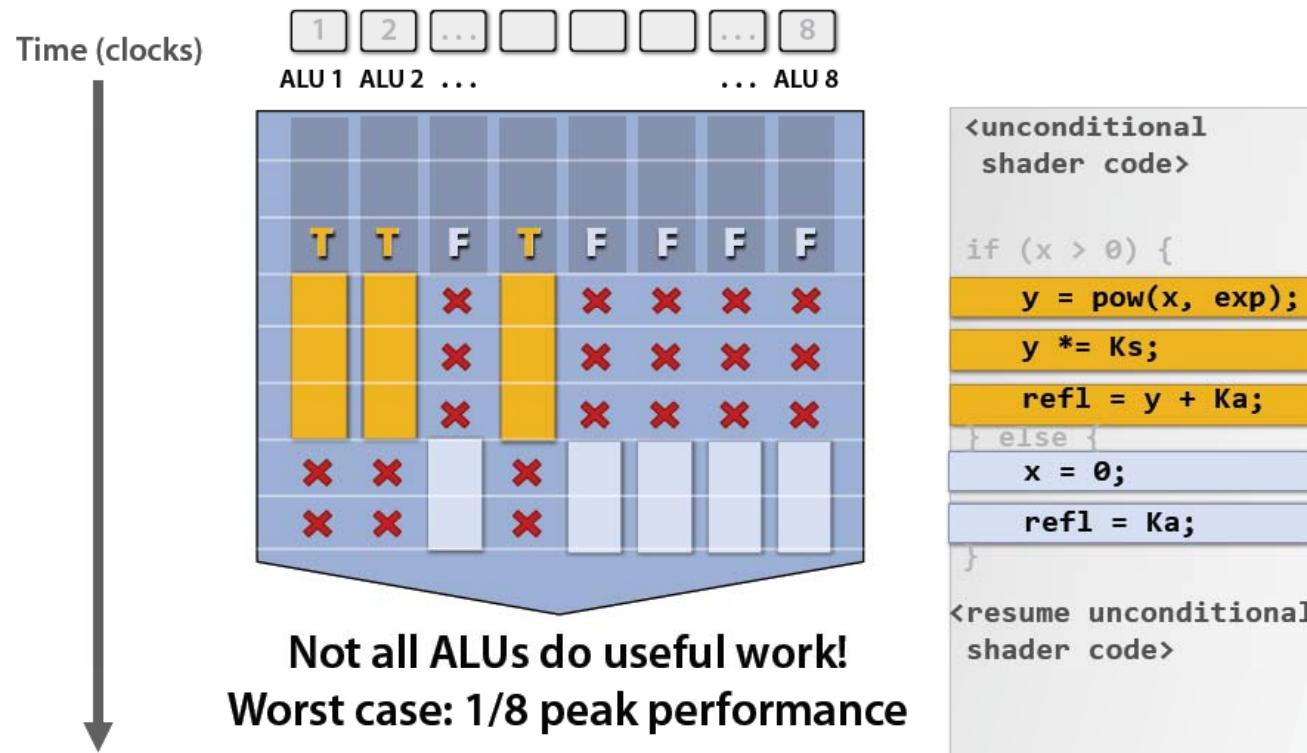
Partition as a 2D block

Increase `threadIdx.z` and repeat



Warp Partitioning

Divergent branches are within a warp!



Warp Partitioning

For `warpSize == 32`, does any warp have a divergent branch with this code:

```
if (threadIdx.x > 15)  
{  
    // ...  
}
```



Warp Partitioning

For any `warpSize > 1`, does any warp have a divergent branch with this code:

```
if (threadIdx.x > warpSize - 1)  
{  
    // ...  
}
```



Warp Partitioning

Given knowledge of warp partitioning, which parallel reduction is better?

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

stride = 1, 2, 4, ...

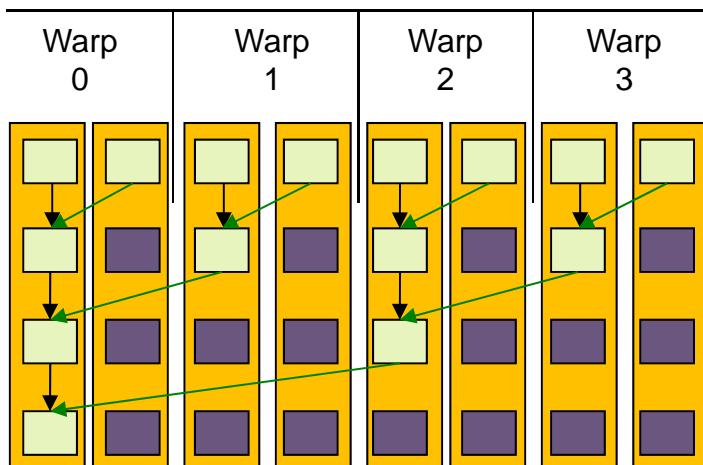
```
if (t < stride)  
    partialSum[t] +=  
    partialSum[t + stride];
```

stride = 4, 2, 1, ...

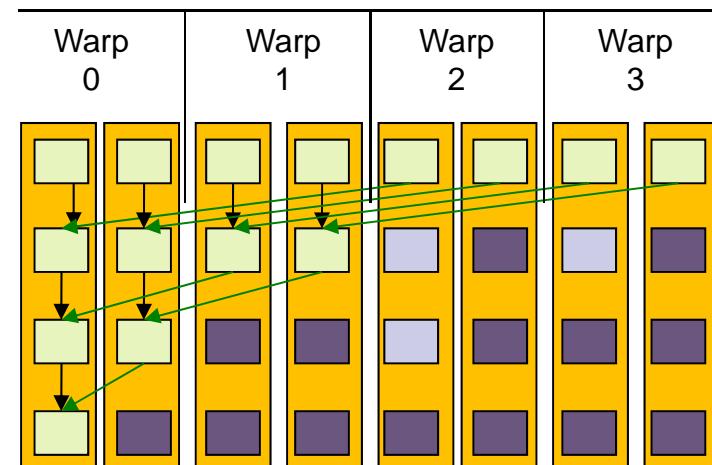


Warp Partitioning

Pretend `warpSize == 2`



`stride = 1, 2, 4, ...`

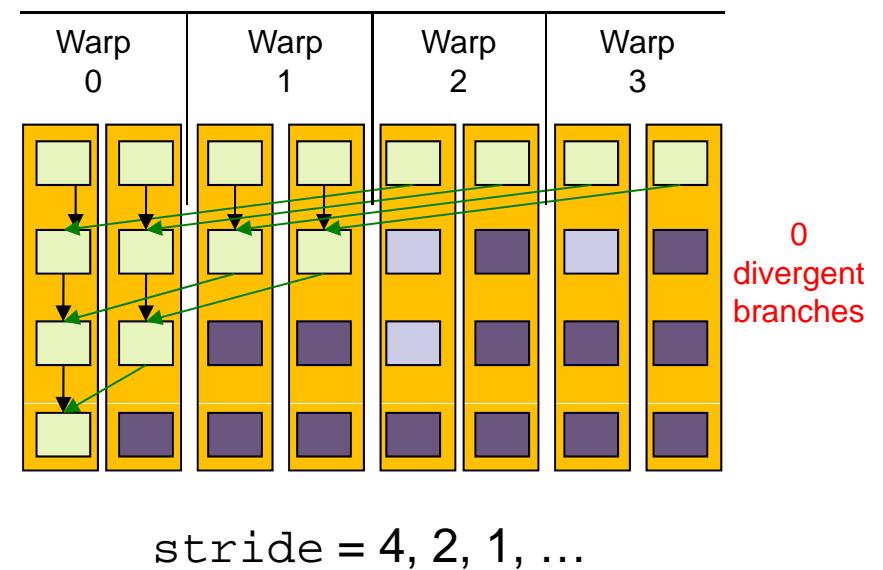
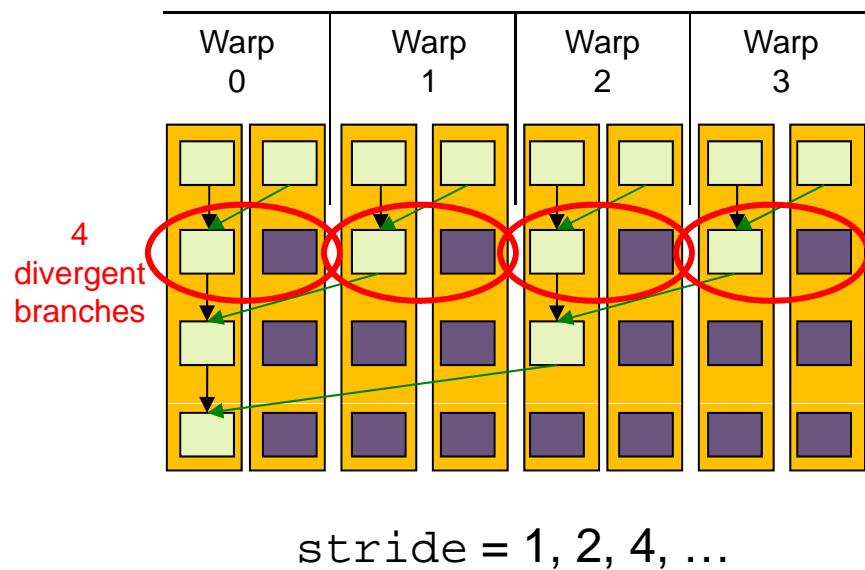


`stride = 4, 2, 1, ...`



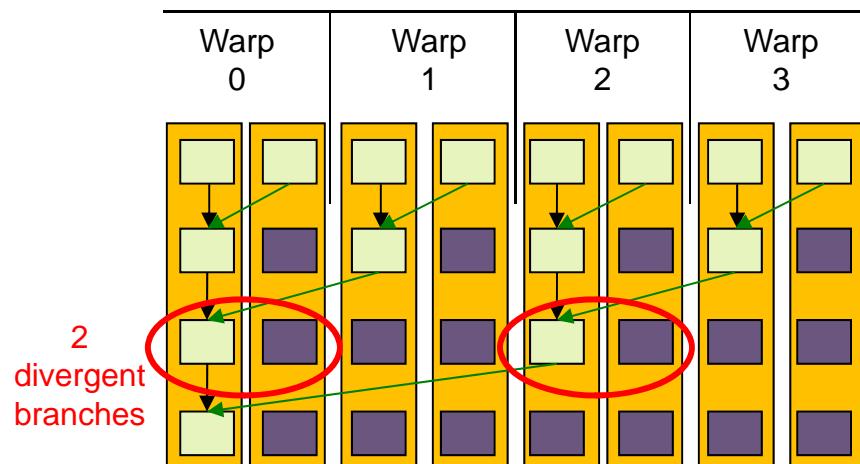
Warp Partitioning

1st Pass

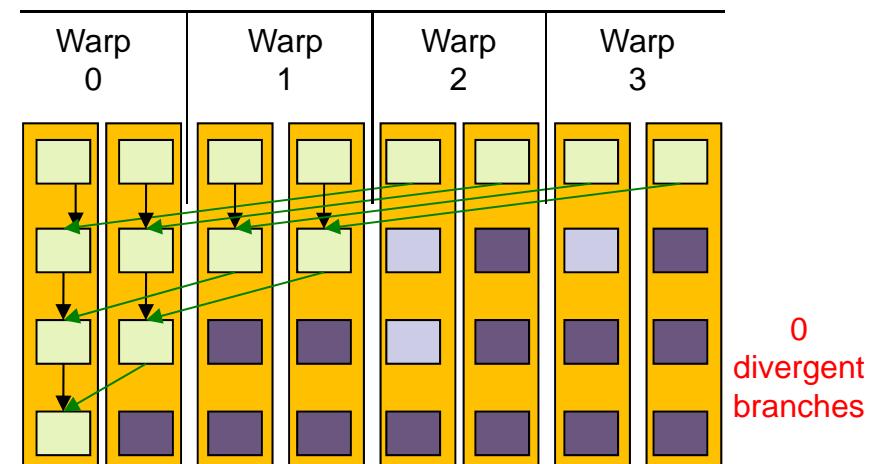


Warp Partitioning

2nd Pass



stride = 1, 2, 4, ...

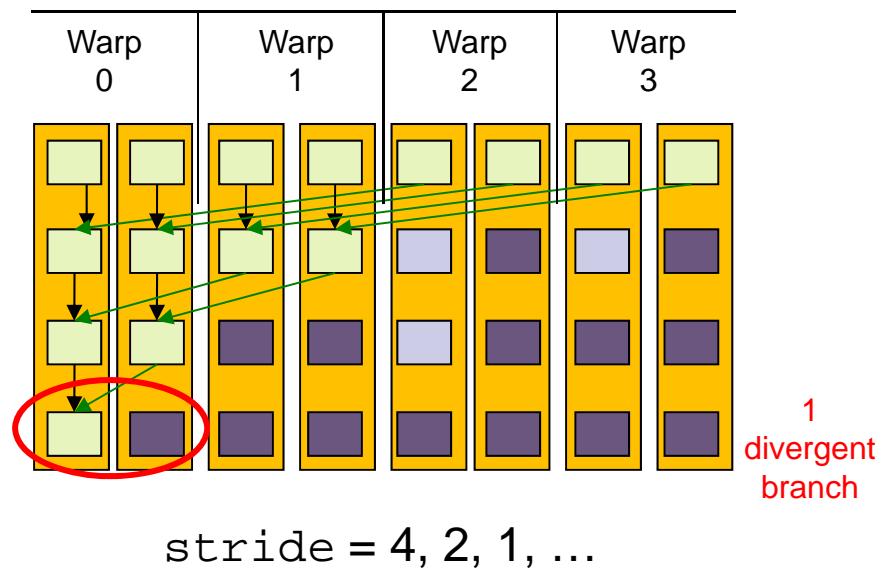
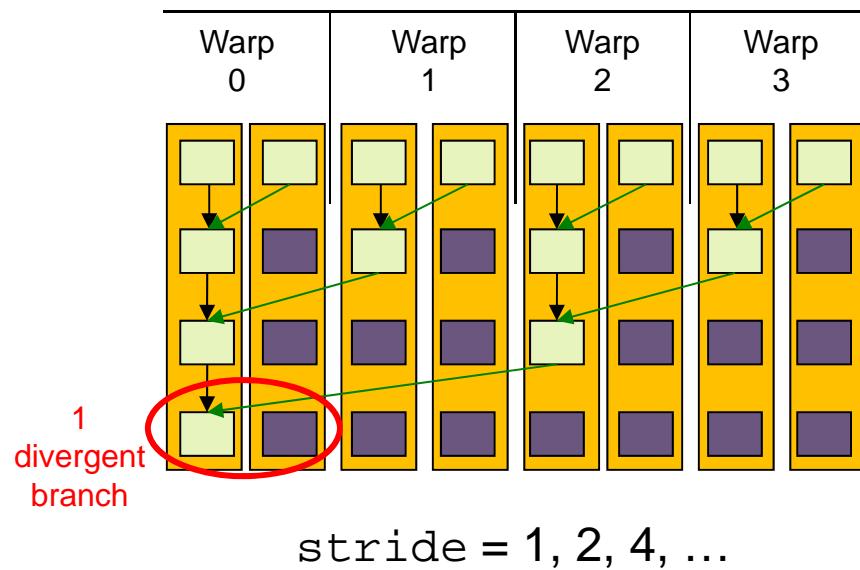


stride = 4, 2, 1, ...



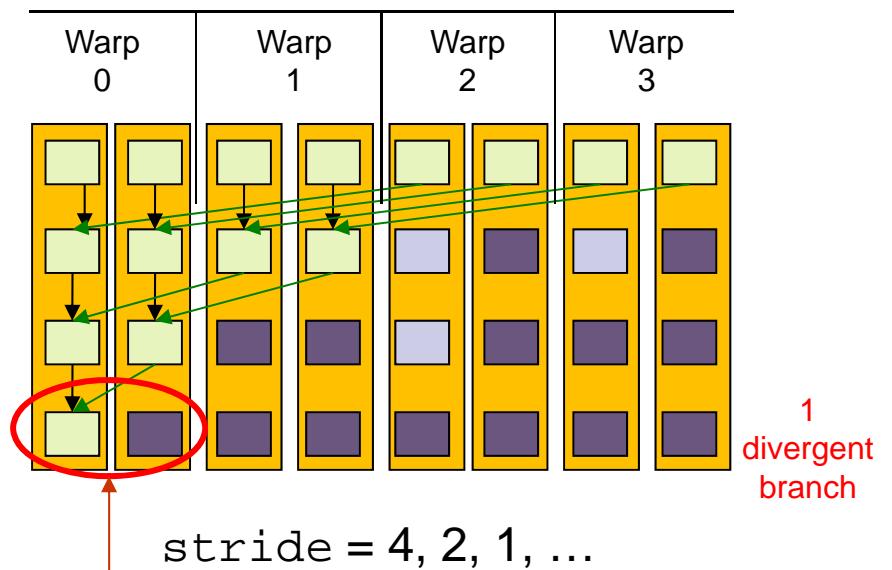
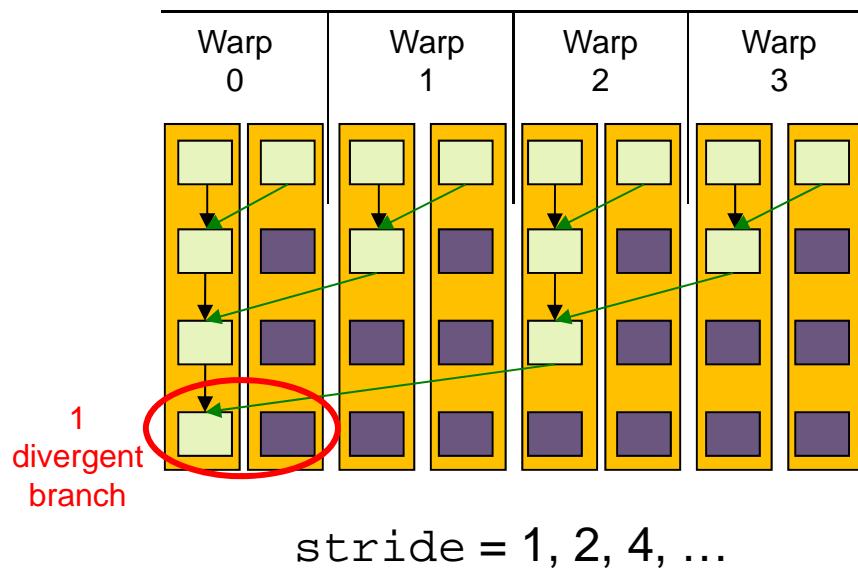
Warp Partitioning

2nd Pass



Warp Partitioning

2nd Pass



Still diverge when number of elements left is $\leq \text{warpSize}$

Warp Partitioning

Good partitioning also allows warps to be retired early.

Better hardware utilization

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

stride = 1, 2, 4, ...

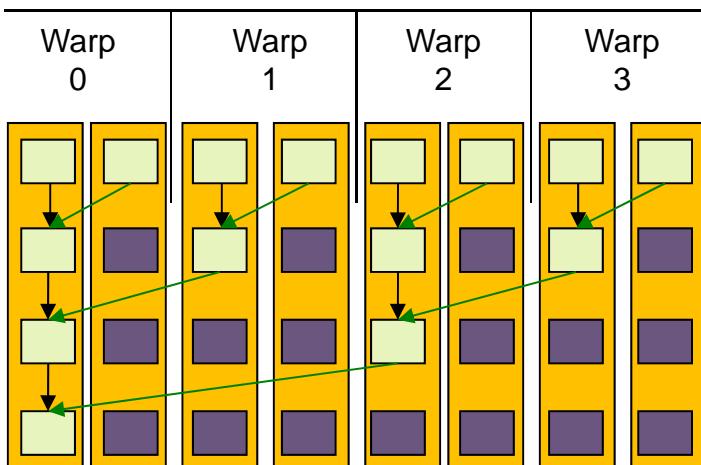
```
if (t < stride)  
    partialSum[t] +=  
        partialSum[t + stride];
```

stride = 4, 2, 1, ...

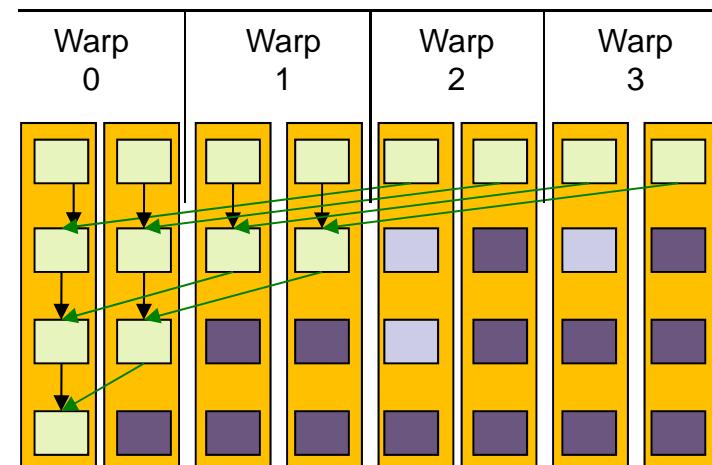


Warp Partitioning

Parallel Reduction



stride = 1, 2, 4, ...

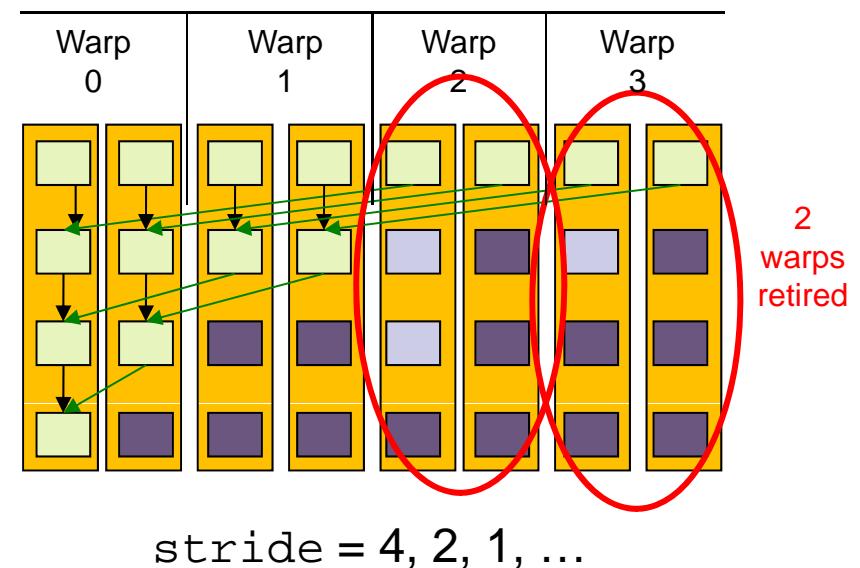
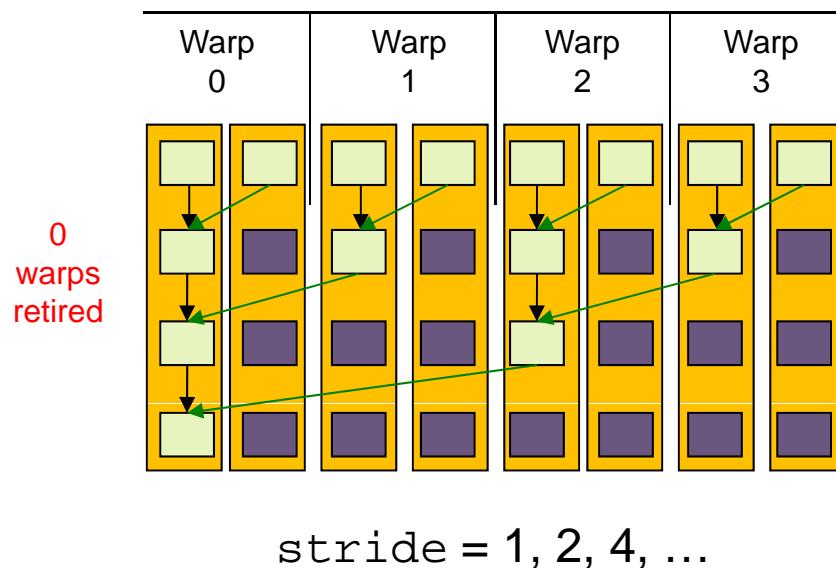


stride = 4, 2, 1, ...



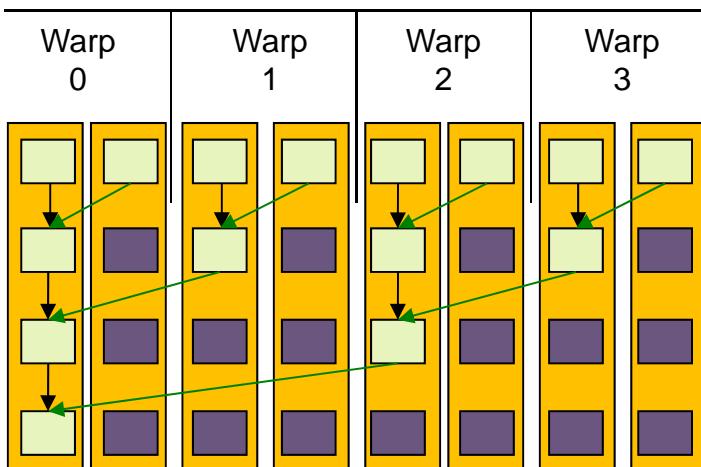
Warp Partitioning

1st Pass

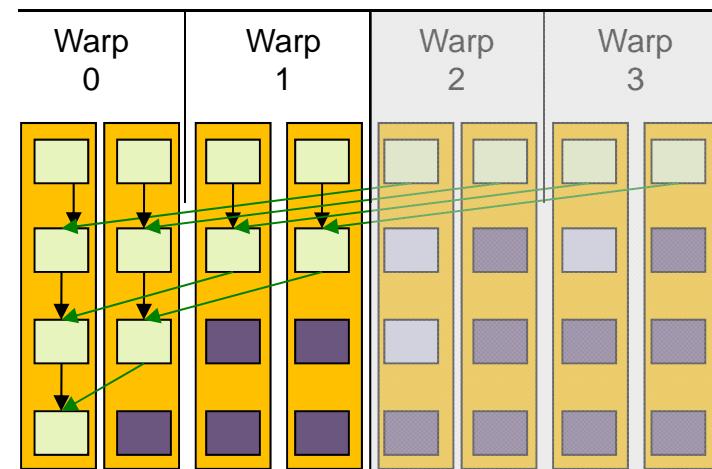


Warp Partitioning

1st Pass



stride = 1, 2, 4, ...

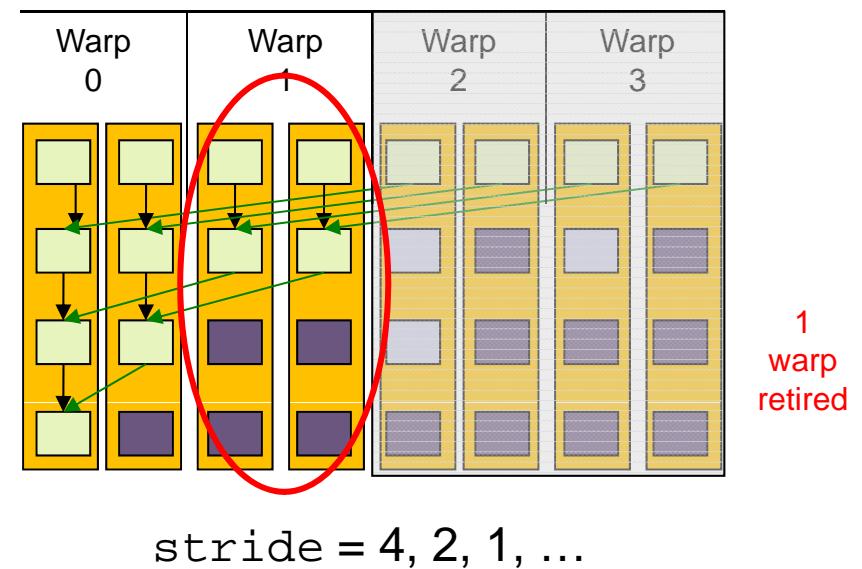
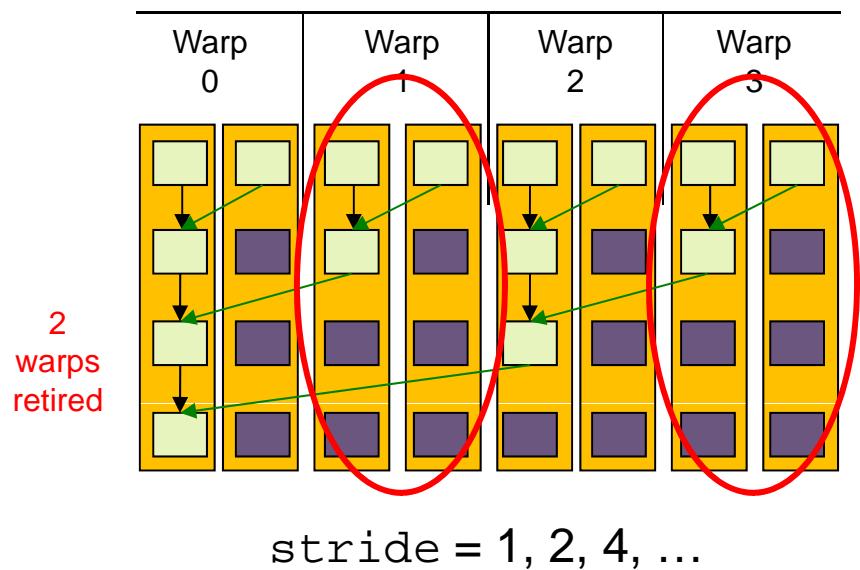


stride = 4, 2, 1, ...



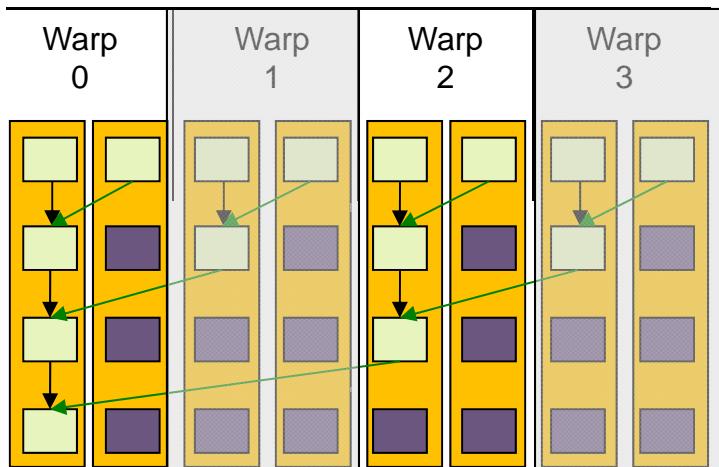
Warp Partitioning

2nd Pass

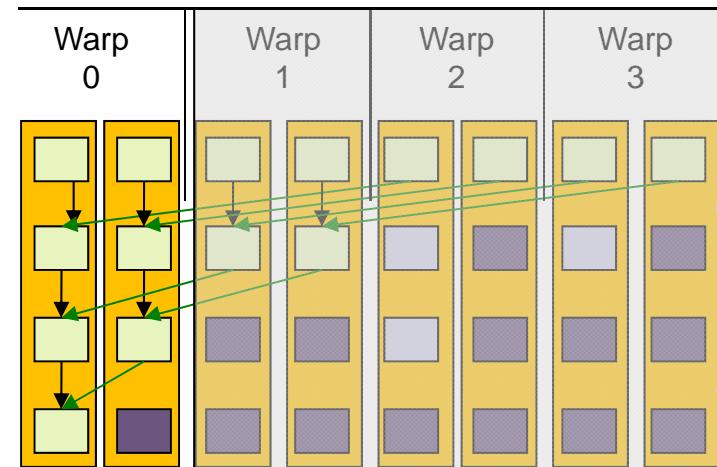


Warp Partitioning

2nd Pass



stride = 1, 2, 4, ...

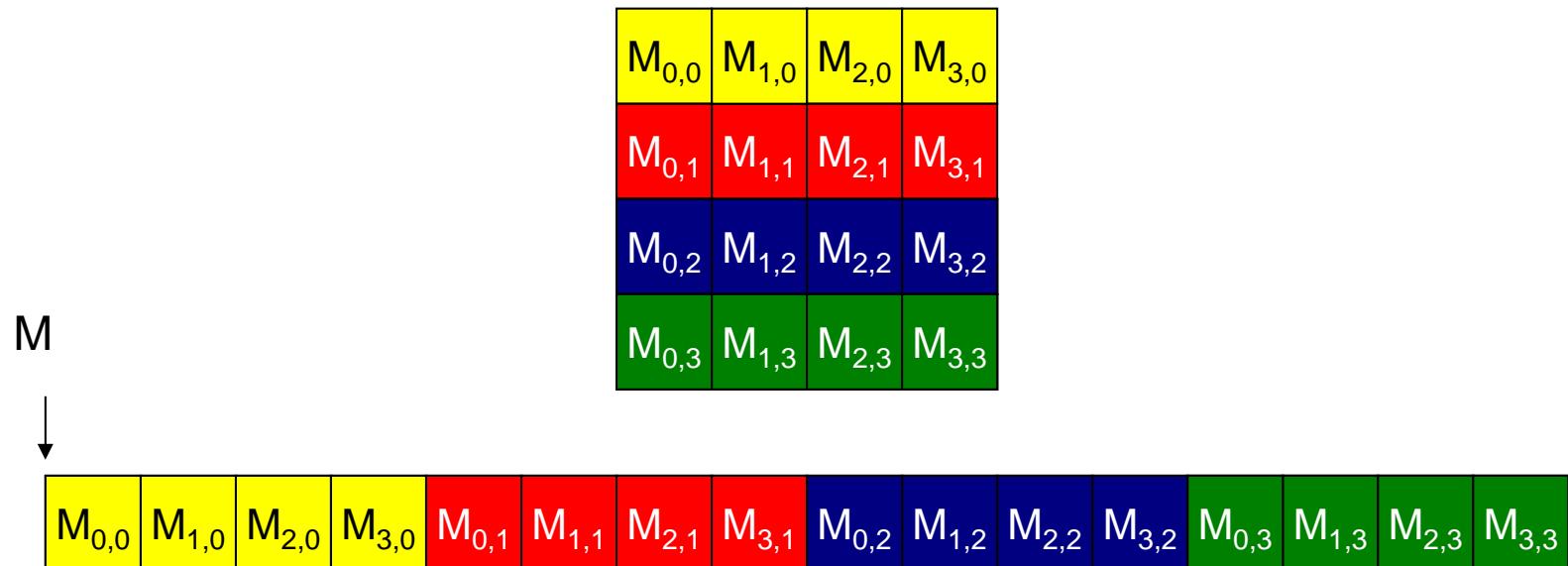


stride = 4, 2, 1, ...



Memory Coalescing

Given a matrix stored *row-major* in *global memory*, what is a *thread'* s desirable access pattern?



Memory Optimization

Minimizing CPU-GPU data transfer

- Host<->device data transfer has much lower bandwidth than global memory access.
 - 8 GB/s (PCIe x16 Gen2) vs 156 GB/s & 515 Ginst/s (C2050)
- Minimize transfer
 - Intermediate data can be allocated, operated, de-allocated directly on GPU
 - Sometimes it's even better to recompute on GPU
 - Move CPU codes to GPU that do not have performance gains if it can reduce data transfer
- Group transfer
 - One large transfer much better than many small ones: 10 microsec latency, 8 GB/s => latency dominated if data size < 80 KB

Overlap memory transfer with computation

Double buffering



Coalescing

- Global memory latency: 400-800 cycles.

The single most important performance consideration!

On Fermi, by default all global memory access are cached in L1.

L1 can be by-passed by passing “-Xptxas –dlcm=cg” to nvcc:
cache only in L2

If cached: requests from a warp falling in a L1 cache line, one transaction

transaction = # L1 line accessed

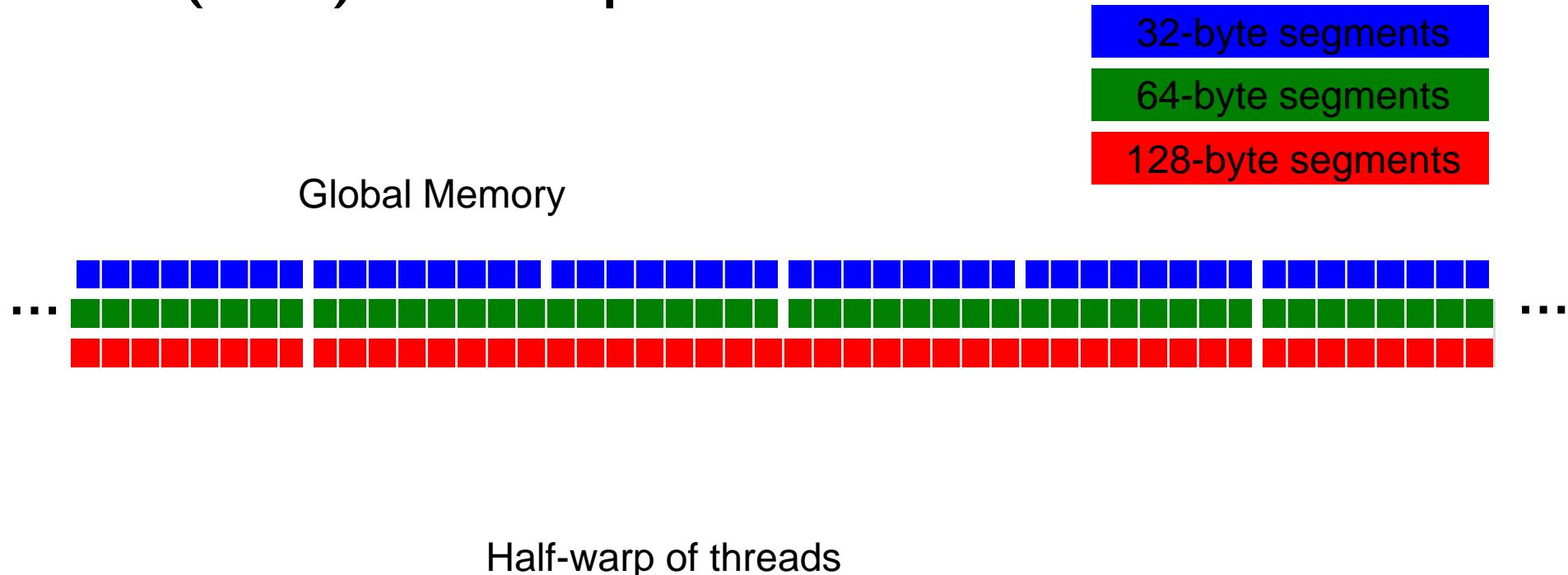
If non-cached: same coalescing criterion

But transaction size can be reduced to 32B segment



Coalescing

- Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met
- Depends on compute capability
 - 1.0 and 1.1 have stricter access requirements
- **Float (32-bit) data example:**

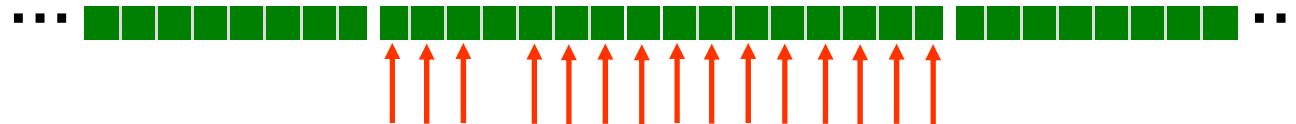


Coalescing

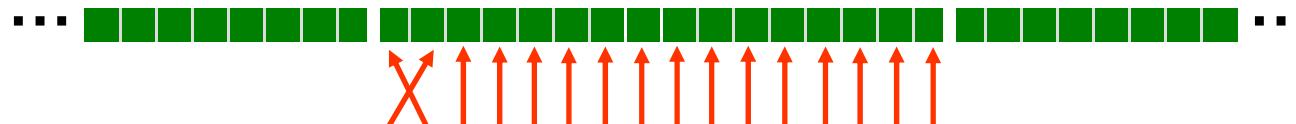
Compute capability 1.0 and 1.1

- K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate

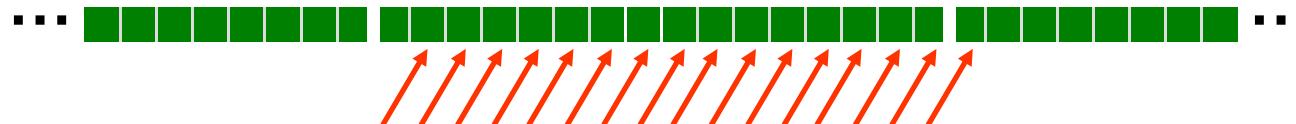
Coalesces – 1 transaction



Out of sequence – 16 transactions



Misaligned – 16 transactions

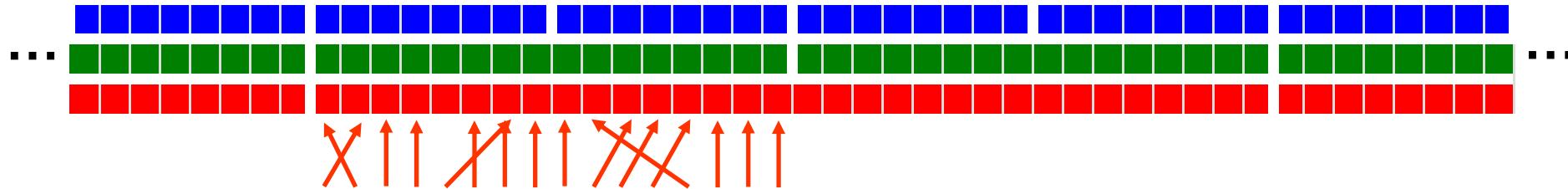


Coalescing

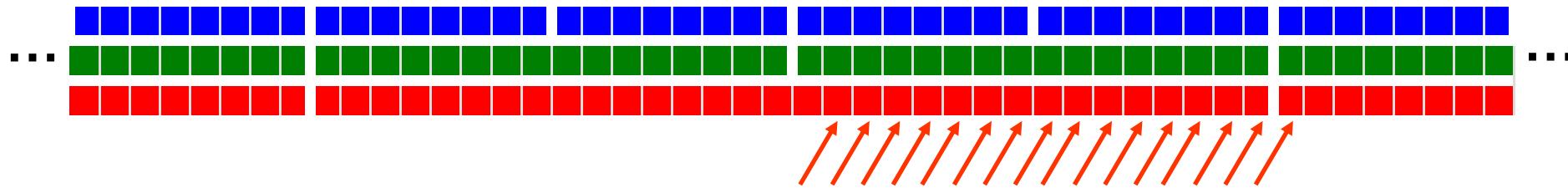
Compute capability 1.2 and higher

- Issues transactions for segments of 32B, 64B, and 128B
- Smaller transactions used to avoid wasted bandwidth

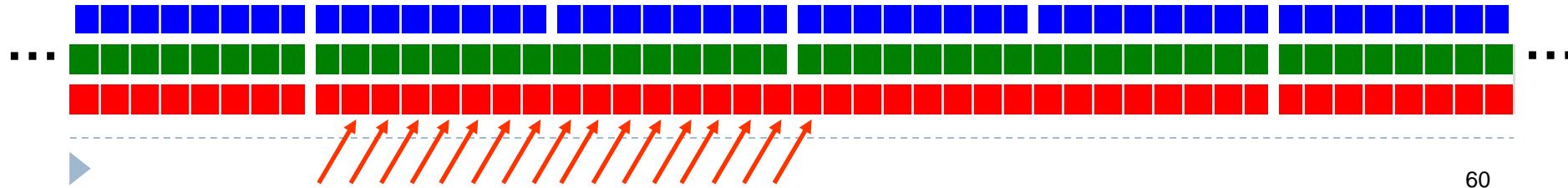
1 transaction - 64B segment



2 transactions - 64B and 32B segments



1 transaction - 128B segment



Coalescing Examples

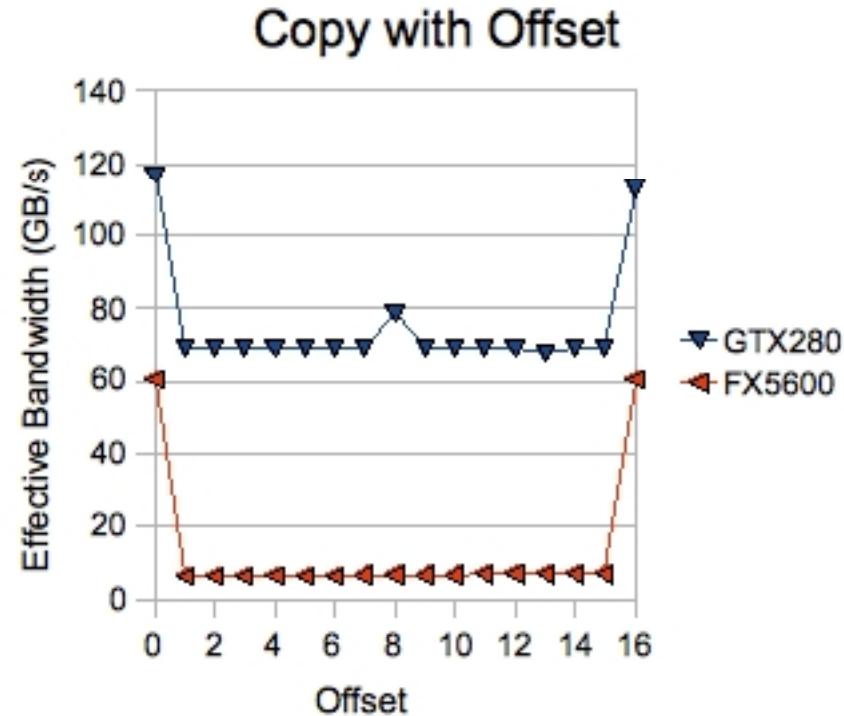
- ➊ Effective bandwidth of small kernels that copy data
 - ➌ Effects of offset and stride on performance

- ➋ Two GPUs
 - ➌ GTX 280
 - ➌ Compute capability 1.3
 - ➌ Peak bandwidth of 141 GB/s
 - ➌ FX 5600
 - ➌ Compute capability 1.0
 - ➌ Peak bandwidth of 77 GB/s



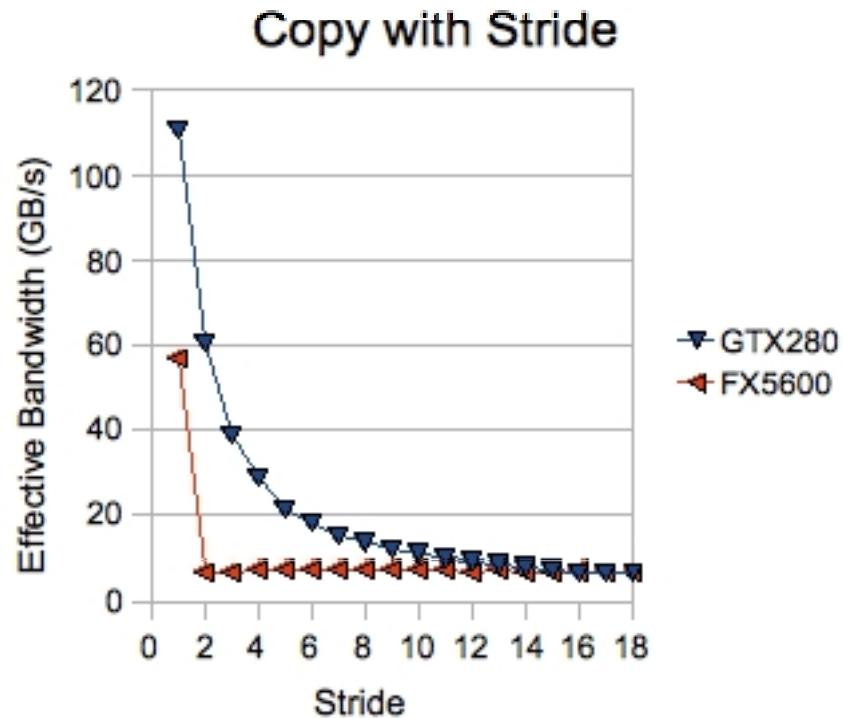
Coalescing Examples

```
__global__ void offsetCopy(float *odata, float *idata,
                           int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```



Coalescing Examples

```
__global__ void strideCopy(float *odata, float *idata,
                           int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

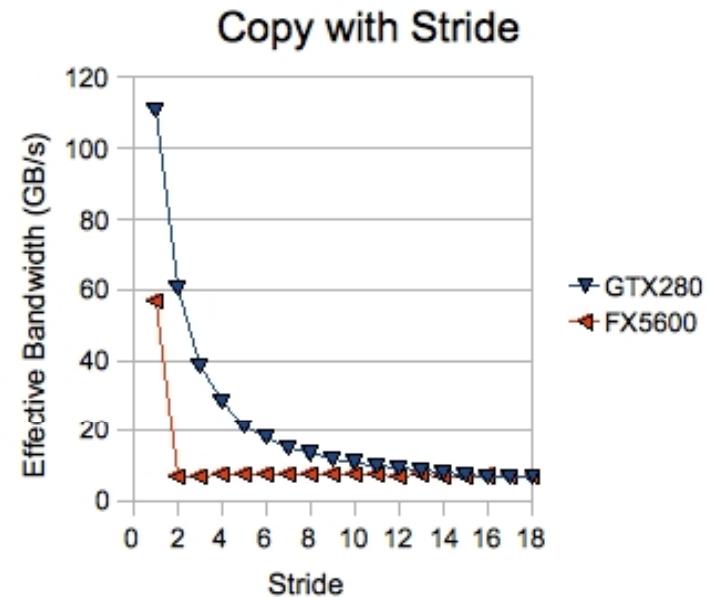


Coalescing Examples

- ➊ Strided memory access is inherent in many multidimensional problems
 - ➋ Stride is generally large (>> 18)

However ...

- ➊ Strided access to *global memory* can be avoided using *shared memory*



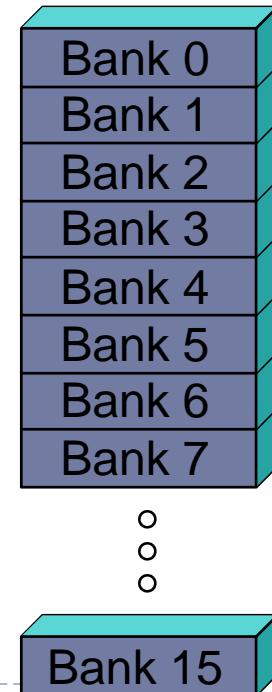
Shared Memory

- ➊ ~Hundred times faster than global memory
- ➋ Cache data to reduce global memory accesses
- ➌ Threads can cooperate via shared memory
- ➍ Use it to avoid non-coalesced access
 - ➎ Stage loads and stores in shared memory to re-order non-coalesceable addressing

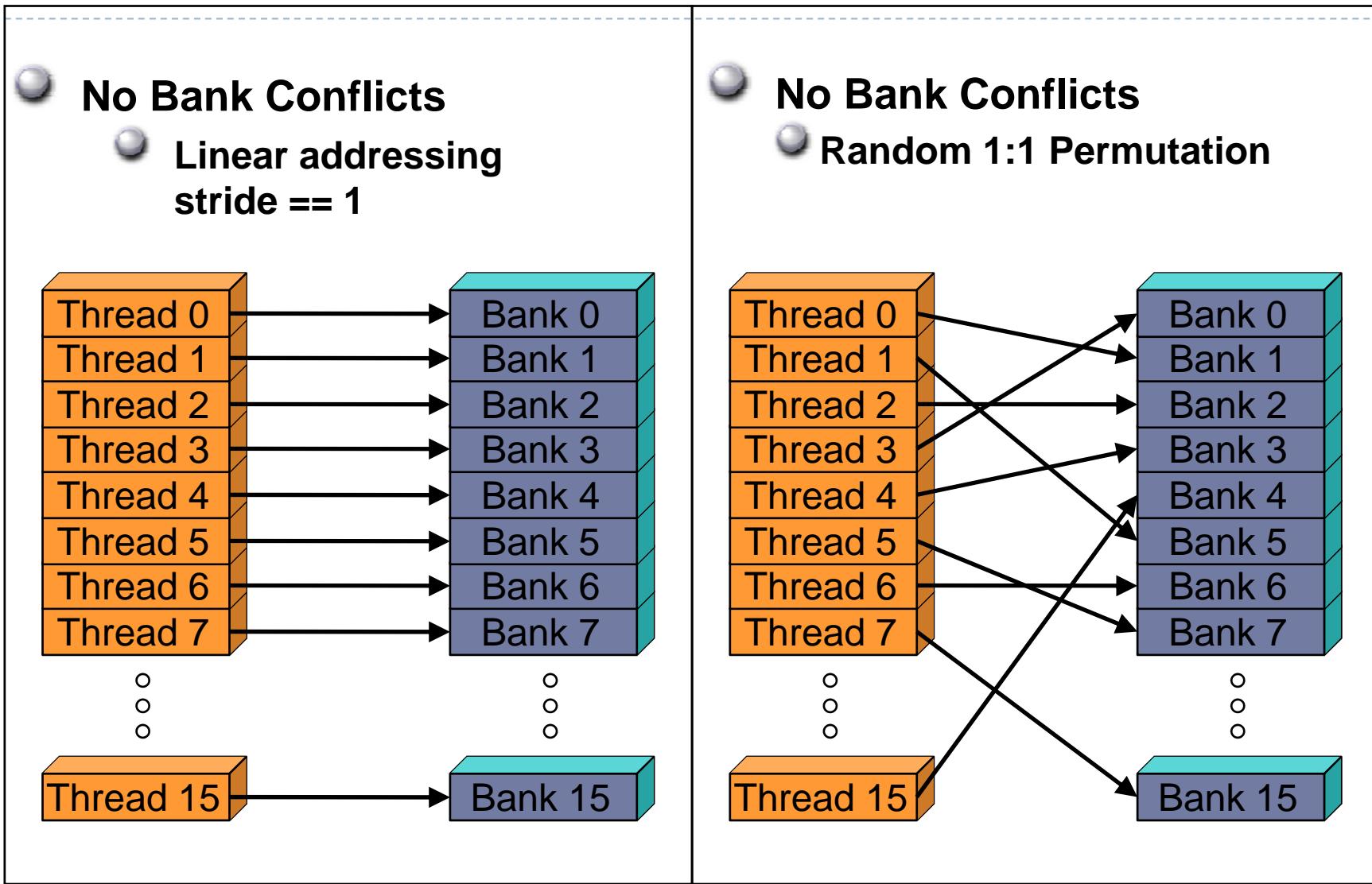


Shared Memory Architecture

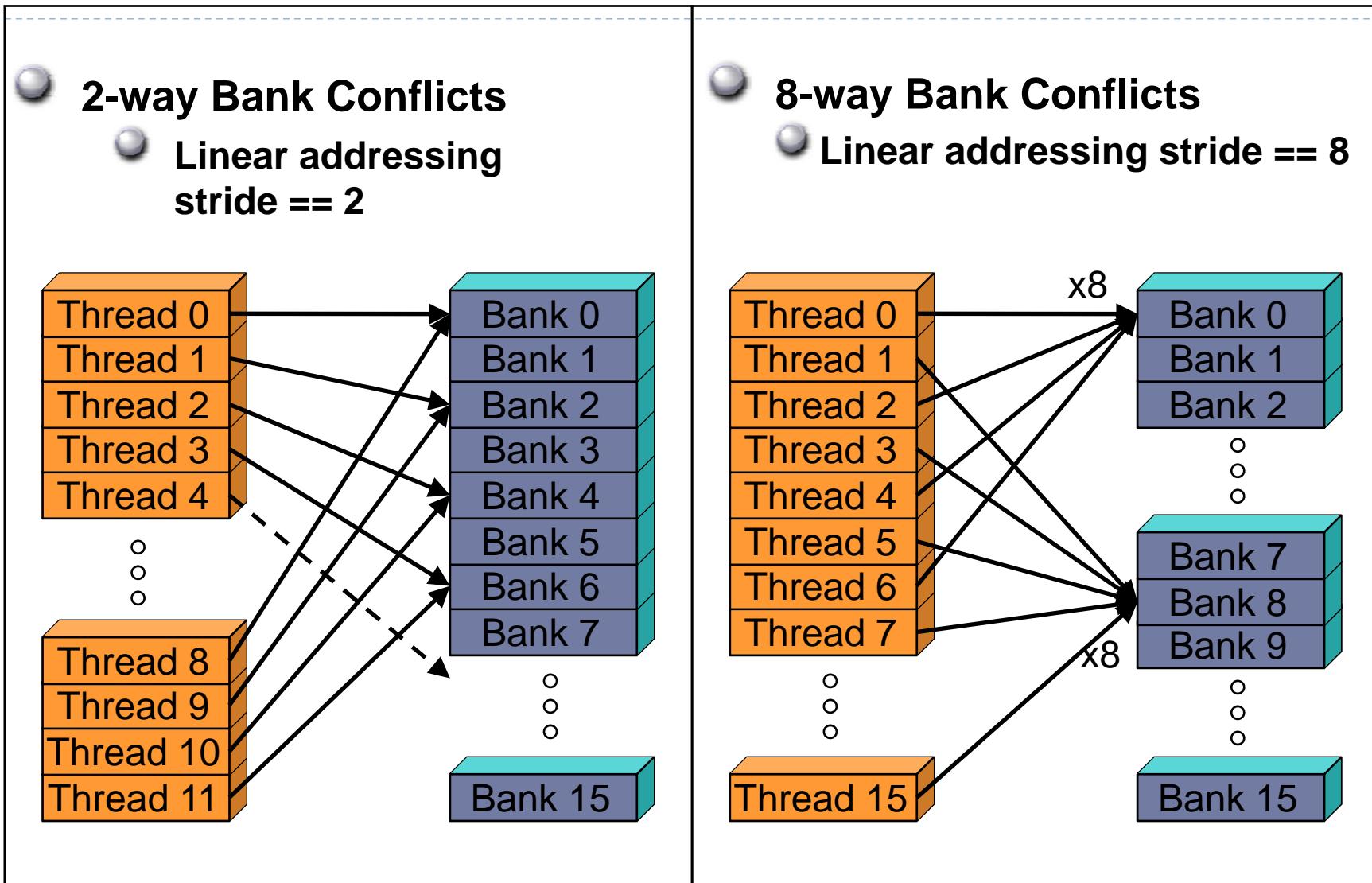
- Many threads accessing memory
 - Therefore, memory is divided into banks
 - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
 - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
 - Conflicting accesses are serialized



Bank Addressing Examples



Bank Addressing Examples



Shared memory bank conflicts

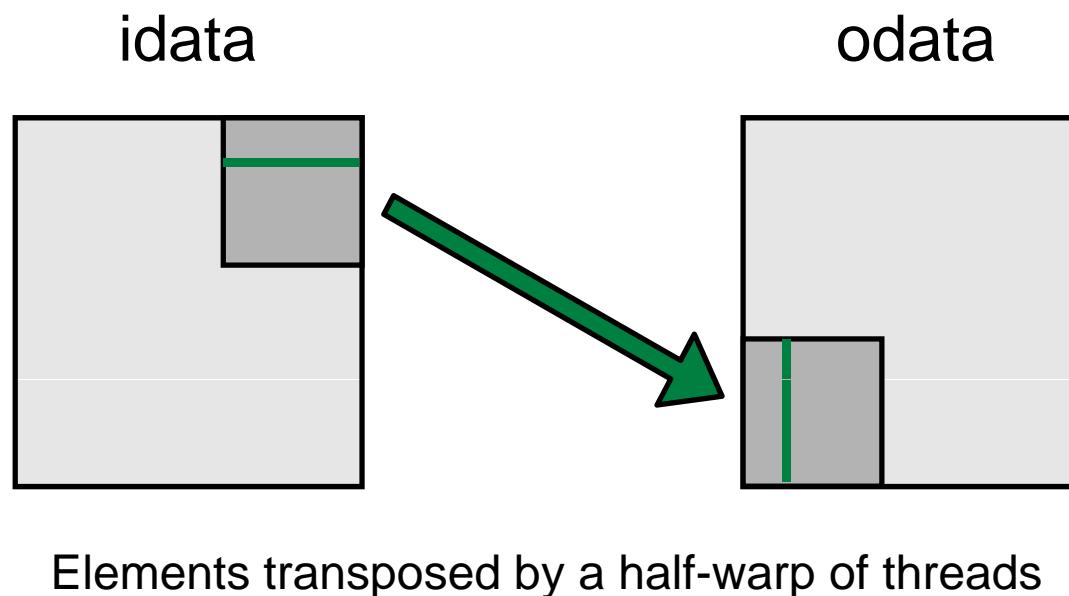
- ➊ Shared memory is ~ as fast as registers if there are no bank conflicts
- ➋ `warp_serialize` profiler signal reflects conflicts
- ➌ The fast case:
 - ➊ If all threads of a half-warp access different banks, there is no bank conflict
 - ➋ If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- ➍ The slow case:
 - ➊ Bank Conflict: multiple threads in the same half-warp access the same bank
 - ➋ Must serialize the accesses
 - ➌ Cost = max # of simultaneous accesses to a single bank



Shared Memory Example:

Transpose

- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory



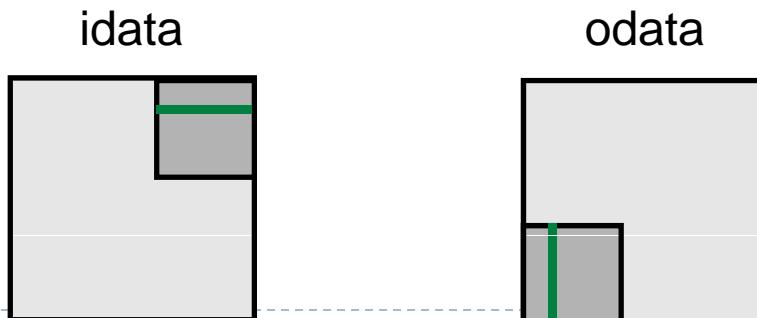
Naïve Transpose

- Loads are coalesced, stores are not (strided by height)

```
__global__ void transposeNaive(float *odata, float *idata,
                                int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

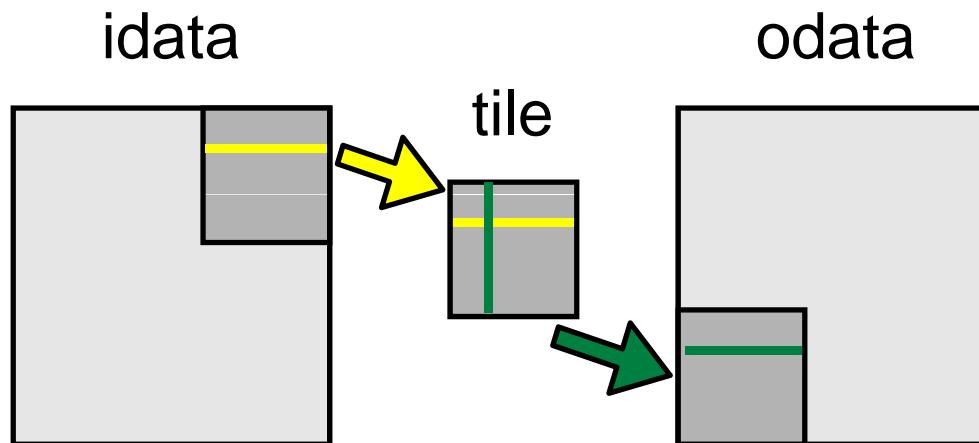
    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;

    odata[index_out] = idata[index_in];
}
```



Coalescing through shared memory

- **Access columns of a tile in shared memory to write contiguous data to global memory**
- **Requires `__syncthreads()` since threads access data in shared memory stored by other threads**



Elements transposed by a half-warp of threads



Coalescing through shared memory

```
__global__ void transposeCoalesced(float *odata, float *idata,
                                    int width, int height)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

    tile[threadIdx.y][threadIdx.x] = idata[index_in];

    __syncthreads();

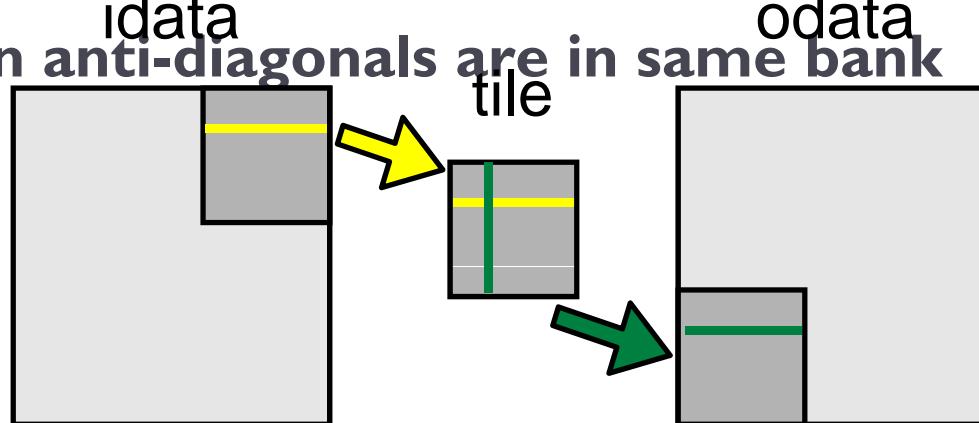
    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```



Bank Conflicts in Transpose

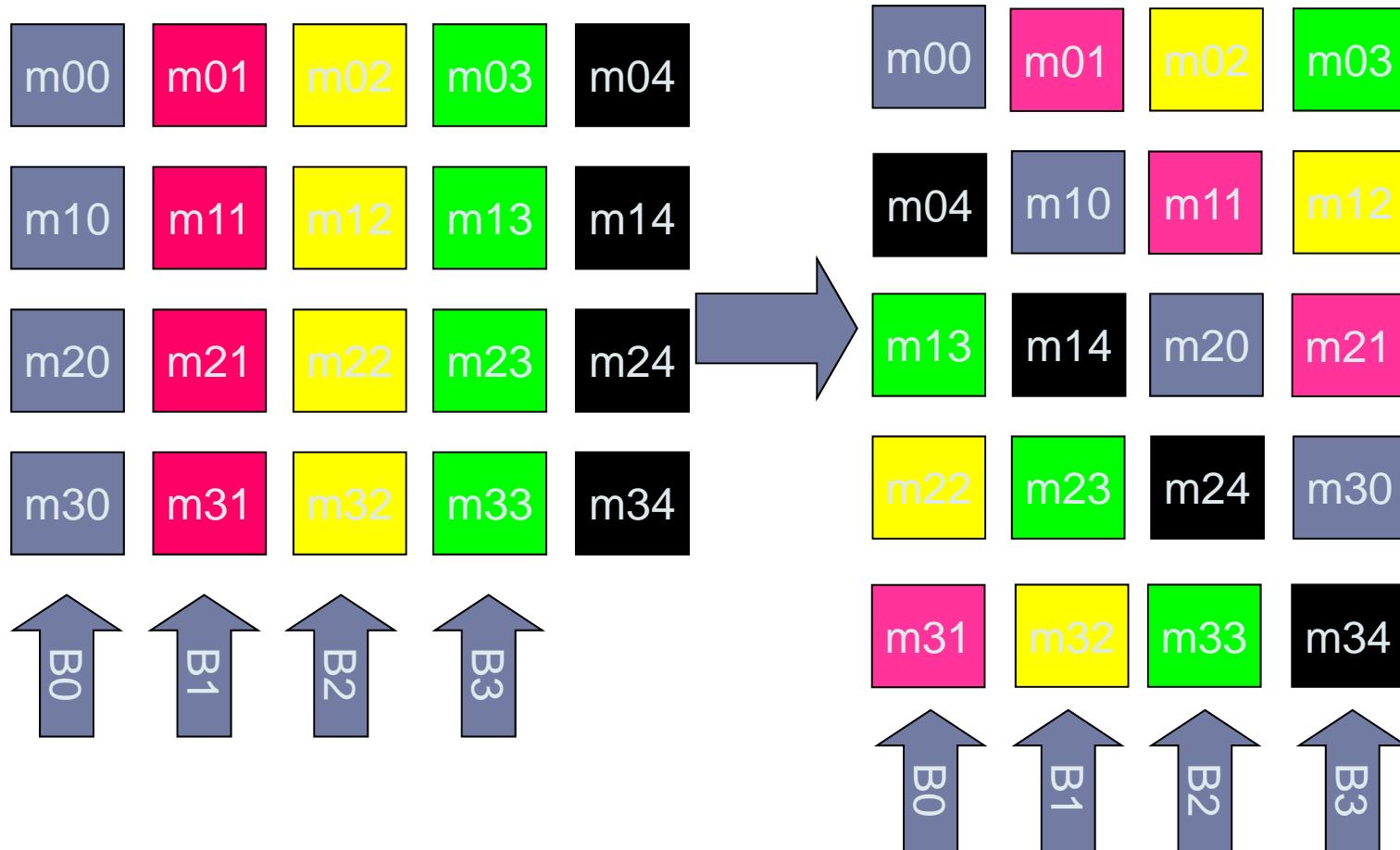
- **16x16 shared memory tile of floats**
 - Data in columns are in the same bank
 - 16-way bank conflict reading columns in tile
- **Solution - pad shared memory array**

- ```
__shared__ float
tile[TILE_DIM][TILE_DIM+1];
```
- Data in anti-diagonals are in same bank



Elements transposed by a half-warp of threads

# Padding Shared Memory



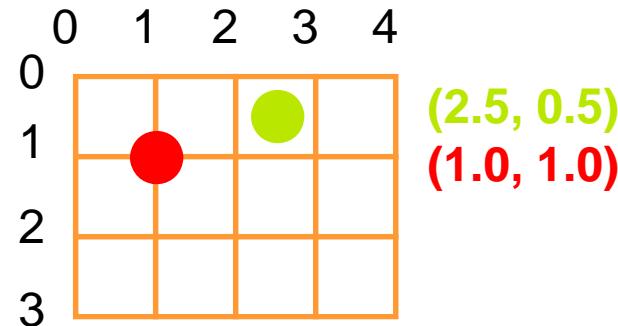
# Textures in CUDA

---

- **Texture is an object for *reading* data**
- **Benefits:**
  - Data is cached
  - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear interpolation
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable in 1D, 2D, or 3D
    - Using integer or normalized coordinates

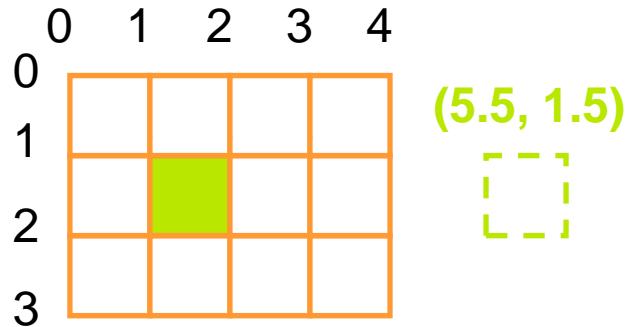


# Texture Addressing

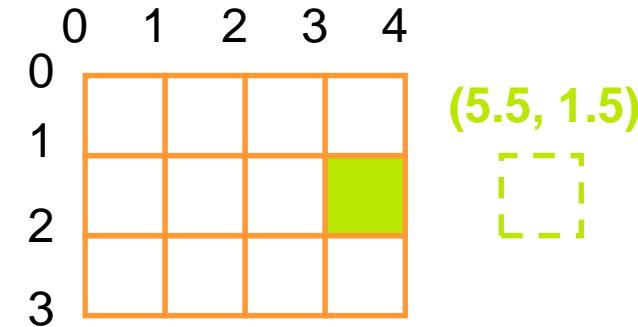


## Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)
- Out-of-bounds coordinate is replaced with the closest boundary



## Clamp



# CUDA Texture Types

---

- **Bound to linear memory**
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes
- **Bound to CUDA arrays**
  - Block linear CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- **Bound to pitch linear (CUDA 2.2)**
  - Global memory address is bound to a texture
  - 2D
  - Float/integer addressing, filtering, and clamp/repeat addressing modes similar to CUDA arrays



# CUDA Texturing Steps

- **Host (CPU) code:**
  - Allocate/obtain memory (**global linear/pitch linear, or CUDA array**)
  - Create a texture reference object
    - Currently must be at file-scope
  - Bind the texture reference to memory/array
  - When done:
    - Unbind the texture reference, free resources
- **Device (kernel) code:**
  - Fetch using texture reference
  - Linear memory textures: **tex1Dfetch()**
  - Array textures: **tex1D() or tex2D() or tex3D()**
  - Pitch linear textures: **tex2D()**

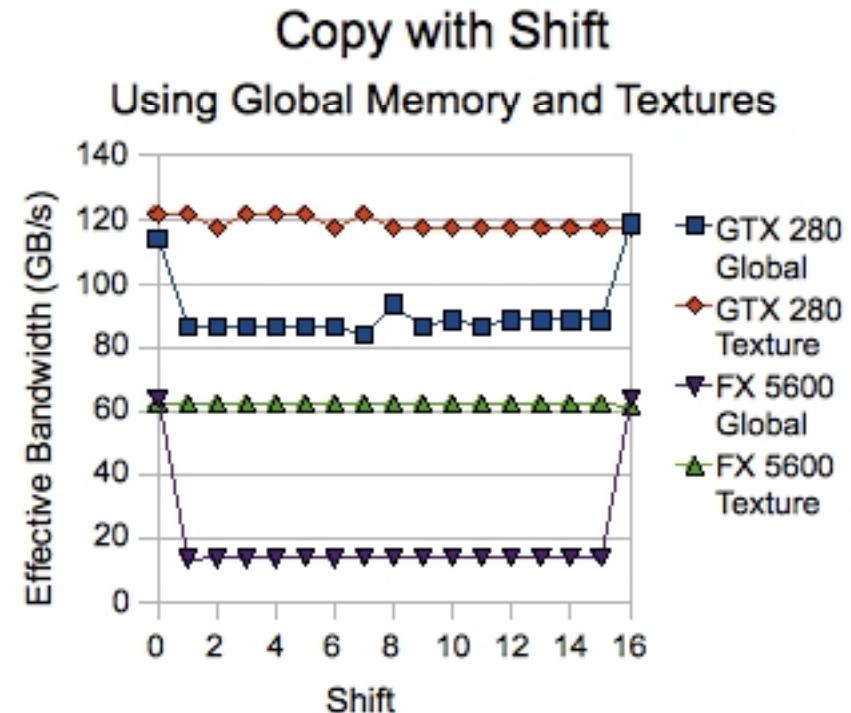


# Texture Example

```
__global__ void
shiftCopy(float *odata,
 float *idata,
 int shift)
{
 int xid = blockIdx.x * blockDim.x
 + threadIdx.x;
 odata[xid] = idata[xid+shift];
}

texture <float> texRef;

__global__ void
textureShiftCopy(float *odata,
 float *idata,
 int shift)
{
 int xid = blockIdx.x * blockDim.x
 + threadIdx.x;
 odata[xid] = tex1Dfetch(texRef, xid+shift);
}
```



# Summary

---

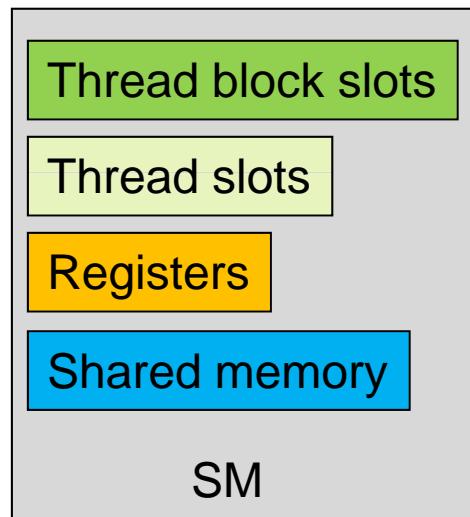
- GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
  - Use parallelism efficiently
  - Coalesce memory accesses if possible
  - Take advantage of shared memory
  - Explore other memory spaces
    - Texture
    - Constant
  - Reduce bank conflicts



# SM Resource Partitioning

---

Recall a SM dynamically partitions resources:



# SM Resource Partitioning

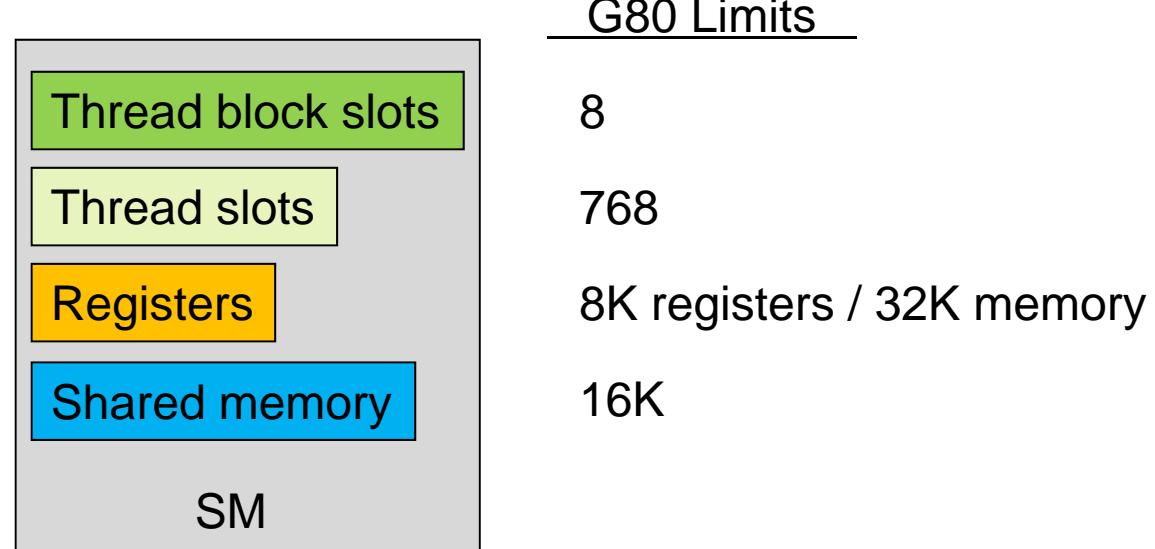
---

Recall a SM dynamically partitions resources:

| G80 Limits         |                           |
|--------------------|---------------------------|
| Thread block slots | 8                         |
| Thread slots       | 768                       |
| Registers          | 8K registers / 32K memory |
| Shared memory      | 16K                       |
| SM                 |                           |

# SM Resource Partitioning

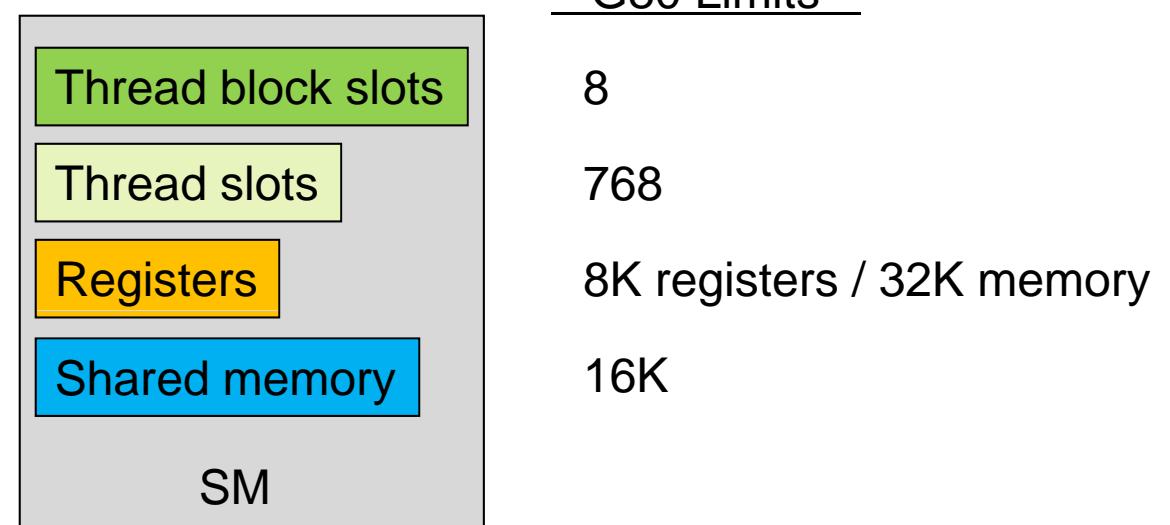
We can have  
8 blocks of 96 threads  
4 blocks of 192 threads  
But not 8 blocks of 192 threads



# SM Resource Partitioning

We can have (assuming 256 thread blocks)

768 threads (3 blocks) using 10 registers each  
512 threads (2 blocks) using 11 registers each

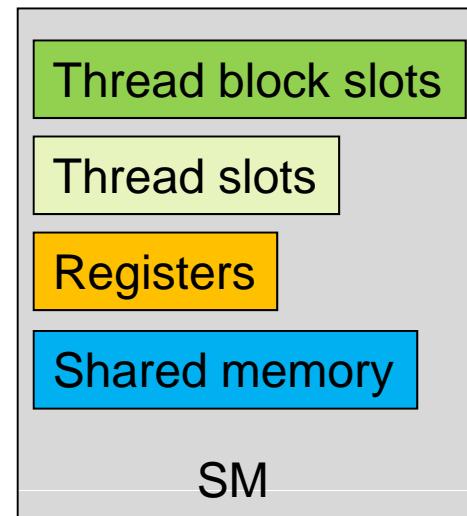


# SM Resource Partitioning

We can have (assuming 256 thread blocks)

768 threads (3 blocks) using 10 registers each  
512 threads (2 blocks) using 11 registers each

- More registers decreases thread-level parallelism
  - Can it ever increase performance?



## G80 Limits

|                           |
|---------------------------|
| 8                         |
| 768                       |
| 8K registers / 32K memory |
| 16K                       |



# SM Resource Partitioning

---

*Performance Cliff*: Increasing resource usage leads to a dramatic reduction in parallelism

For example, increasing the number of registers, unless doing so hides latency of global memory access

# SM Resource Partitioning

---

## CUDA Occupancy Calculator

[http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)

# Kernel Launch Configuration

# Grid Size Heuristics

---

- **# of blocks > # of SM**
  - Each SM has at least one work-group to execute
- **# of blocks / # of SM > 2**
  - Multi blocks can run concurrently on a SM
  - Work on another block if one block is waiting on barrier
- **# of blocks / # of SM > 100 to scale well to future device**



# Block Size Heuristics

---

- Block size should be a multiple of 32 (warp size)
- Want as many warps running as possible to hide latencies
- Minimum: 64. I generally use 128 or 256. But use whatever is best for your app.
- Depends on the problem, do experiments!



# Latency Hiding

---

Key to understanding:

Instructions are issued in order

A thread blocks when one of the operands isn't ready:

Latency is hidden by switching threads

Conclusion:

Need enough threads to hide latency



# Occupancy

---

- Occupancy: ratio of active warps per SM to the maximum number of allowed warps

Maximum number: 32 in Tesla, 48 in Fermi



# Dynamical Partitioning of SM Resources

---

Shared memory is partitioned among blocks

Registers are partitioned among threads:  $\leq 63$

Thread block slots:  $\leq 8$

Thread slots:  $\leq 1536$

Any of those can be the limiting factor on how many threads can be launched at the same time on a SM



# Latency Hiding Occupancy Calculation

---

- Assume global memory takes 400 cycles, we need  $400/2 = 200$  arithmetic instructions to hide the latency.
- For example, assume the code has 8 independent arithmetic instructions for every one global memory access. Thus  $200/8 \sim 26$  warps would be enough (54% occupancy).
- Note beyond 54%, in this example higher occupancy won't lead to performance increase.



# Register Dependency Latency Hiding

---

- If an instruction uses a result stored in a register written by an instruction before it, this is  $\sim 24$  cycles latency
- So, we need  $24/2=13$  warps to hide register dependency latency. This corresponds to 27% occupancy



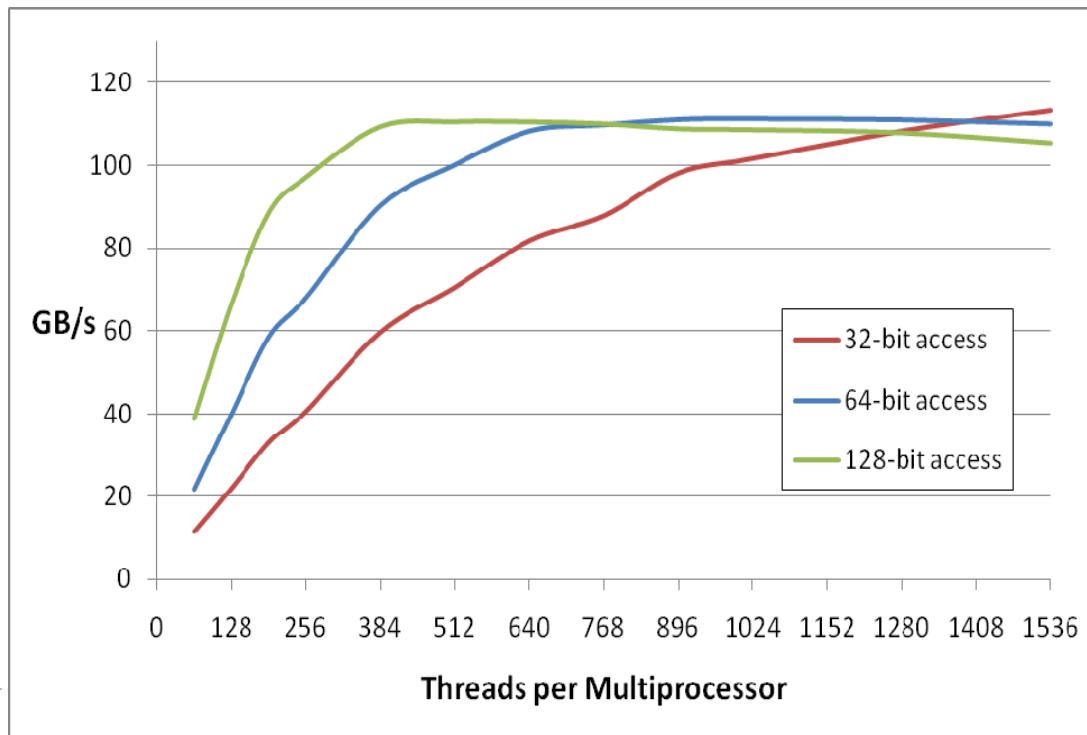
# Concurrent Accesses and Performance

Increment a 64M element array

Two accesses per thread (load then store, but they are dependent)

Thus, each warp (32 threads) has one outstanding transaction at a time

Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s



*Several independent smaller accesses have the same effect as one larger one.*

For example:

Four 32-bit ~ = one 128-bit

# Occupancy Optimizations

---

- Increase occupancy to achieve latency hiding

If adding a single instruction leads to significant perf drop, occupancy is the primary suspect

--ptxas-options=-v: output resource usage info

Compiler option --maxrregcount=n: per file

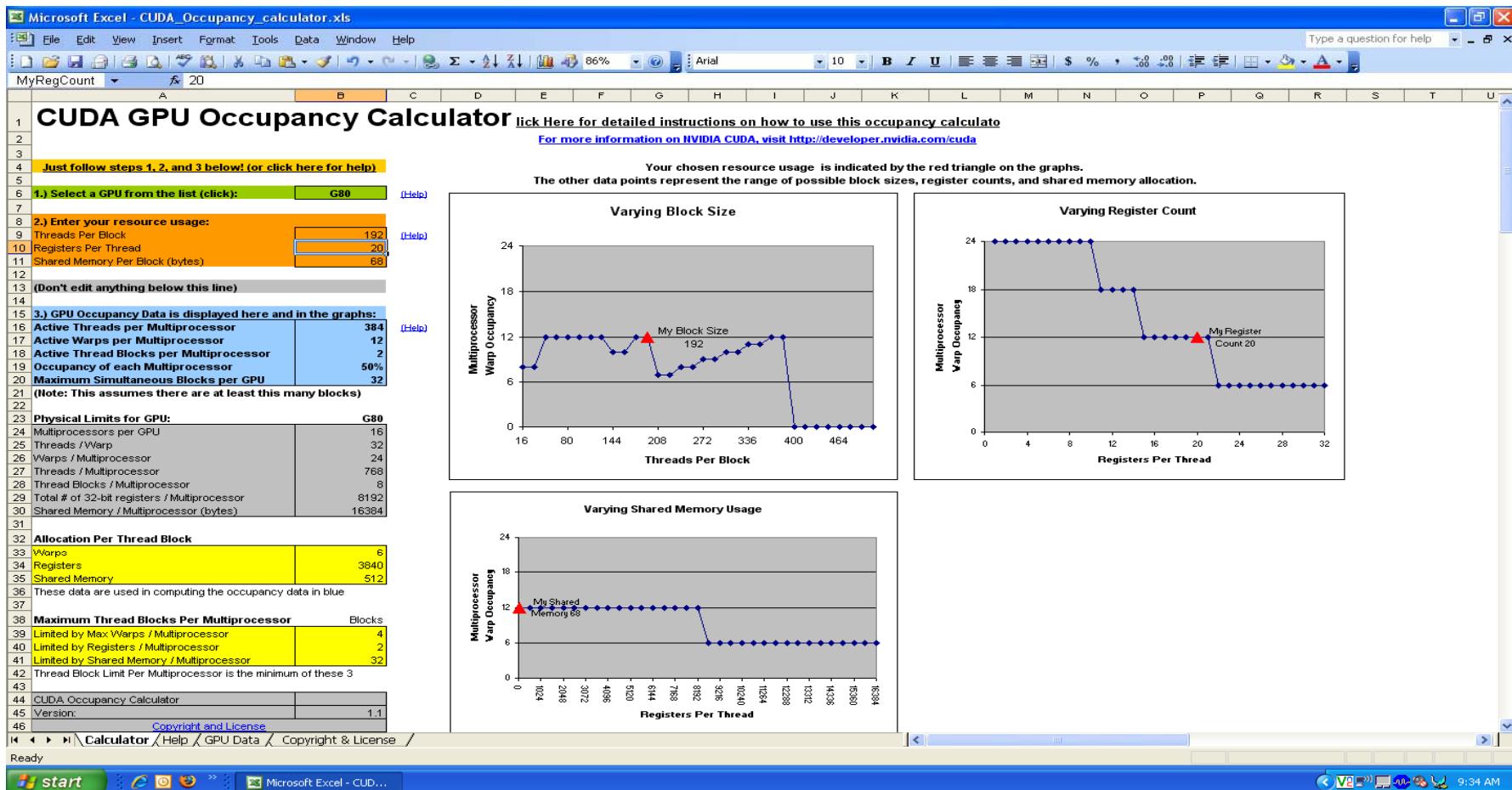
    --launch\_bounds : per kernel

Use of template to reduce register usage

Dynamically allocating shared memory

After some point (generally 50%), further increase in occupancy won't lead to performance increase: got enough warps for latency hiding





# Data Prefetching

---

Independent instructions between a global memory read  
and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

# Data Prefetching

---

Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Read global memory

# Data Prefetching

Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f;
```

Execute instructions  
that are not dependent  
on memory read

# Data Prefetching

---

Independent instructions between a global memory read and its use can hide memory latency

```
float m = Md[i];
float f = a * b + c * d;
float f2 = m * f; // Use global memory after
// the above line from
// enough warps hide the
// memory latency
```

# Data Prefetching

---

**Prefetching** data from global memory can effectively increase the number of independent instructions between global memory read and use

# Data Prefetching

---

Recall tiled matrix multiply:

```
for (/* ... */)
{
 // Load current tile into shared memory
 __syncthreads();
 // Accumulate dot product
 __syncthreads();
}
```

# Data Prefetching

---

Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
 // Deposit registers into shared memory
 __syncthreads();
 // Load next tile into registers
 // Accumulate dot product
 __syncthreads();
}
```

# Data Prefetching

---

Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
 // Deposit registers into shared memory
 __syncthreads();
 // Load next tile into registers
 // Accumulate dot product
 __syncthreads();
}
```

# Data Prefetching

---

Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
 // Deposit registers into shared memory
 __syncthreads();
 // Load next tile into registers
 // Accumulate dot product
 __syncthreads();
}

▶ 108
```

Prefetch for next iteration of the loop

# Data Prefetching

---

Tiled matrix multiply with prefetch:

```
// Load first tile into registers

for (/* ... */)
{
 // Deposit registers into shared memory
 __syncthreads();
 // Load next tile into registers
 // Accumulate dot product
 __syncthreads();
}
```

These instructions executed by enough threads will hide the memory latency of the prefetch

# Instruction Throughput Optimizations

# Instruction Optimization

---

If you find out the code is instruction bound

Compute-intensive algorithm can easily become memory-bound if not careful enough

Typically, worry about instruction optimization after memory and execution configuration optimizations



# Fermi Arithmetic Instruction Throughputs

---

Int & fp32: 2 cycles

fp64: 2 cycles

Fp32 transcendental: 8 cycles

Int divide and modulo are expensive

Divide by  $2^n$ , use “`>> n`”

Modulo  $2^n$ , use “`& (2^n - 1)`”

Avoid automatic conversion of double to float

Adding “`f`” to floating literals (e.g. `1.0f`) because the default is double

Fermi default: `-ftz=false`, `-prec-div=true`, `-prec-sqrt=true` for IEEE compliance



# Runtime Math Library and Intrinsics

---

Two types of runtime math library functions

`func()`:

Slower but higher accuracy (5 ulp or less)

Examples: `sin(x)`, `exp(x)`, `pow(x, y)`

`__func()`:

Fast but lower accuracy (see prog. guide for full details)

Examples: `__sin(x)`, `__exp(x)`, `__pow(x, y)`

A number of additional intrinsics:

`__sincos()`, `__rcp()`, ...

Full list in Appendix C.2 of the CUDA Programming Guide

`-use-fast-math`: forces every `func()` to `__func ()`



# Control Flow

Instructions are issued per 32 threads (warp)

Divergent branches:

Threads within a single warp take different paths

if-else, ...

Different execution paths within a warp are serialized

Different warps can execute different code with no impact on performance

Avoid diverging within a warp

Example with divergence:

```
if (threadIdx.x > 2) { ... } else { ... }
```

Branch granularity < warp size

Example without divergence:

```
if (threadIdx.x / WARP_SIZE > 2) { ... } else { ... }
```

Branch granularity is a whole multiple of warp size



# Profiler and Instruction Throughput

Visual Profiler derives:

Instruction throughput

Fraction of SP arithmetic instructions that could have been issued in the same amount of time

- So, not a good metric for code with DP arithmetic or transcendentals

Extrapolated from one multiprocessor to GPU

Change the conditional statement and see how that affect the instruction throughput

| Method                | GPU usec    | %GPU time | glob mem read throughput (GB/s) | glob mem write throughput (GB/s) | glob mem overall throughput (GB/s) | instruction throughput |
|-----------------------|-------------|-----------|---------------------------------|----------------------------------|------------------------------------|------------------------|
| 1 fwd_3D_16x16_order8 | 3.09382e+06 | 82.15     | 46.9465                         | 11.6771                          | 58.6236                            | 0.763973               |
| 2 memcpyHtoD          | 503094      | 13.35     |                                 |                                  |                                    |                        |
| 3 memcpyDtoH          | 168906      | 4.48      |                                 |                                  |                                    |                        |



Optimizing CPU/GPU interaction

# Pinned (non-pageable) memory

---

Pinned memory enables:

- faster PCIe copies (~2x throughput on FSB systems)

- memcopies asynchronous with CPU

- memcopies asynchronous with GPU

Usage

`cudaHostAlloc / cudaFreeHost`

**instead of malloc / free**

Additional flags if pinned region is to be shared between lightweight CPU threads

Implication:

- pinned memory is essentially removed from virtual memory

- `cudaHostAlloc` is typically very expensive



# Streams and Async API

---

## Default API:

Kernel launches are asynchronous with CPU

Memcopies (D2H, H2D) block CPU thread

CUDA calls are serialized by the driver

## Streams and async functions provide:

Memcopies (D2H, H2D) asynchronous with CPU

Ability to concurrently execute a kernel and a memcpy

Concurrent kernel in Fermi

Stream = sequence of operations that execute in issue-order on GPU

Operations from different streams can be interleaved

A kernel and memcpy from different streams can be overlapped



# Overlap kernel and memory copy

---

Requirements:

- D2H or H2D memcpy from pinned memory
- Device with compute capability  $\geq 1.1$  (G84 and later)
- Kernel and memcpy in different, non-0 streams

Code:

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<<grid, block, 0, stream2>>>(...);
```

} **potentially  
overlapped**



# Stream Examples

Kernel      Memcpy

---

K1,M1,K2,M2



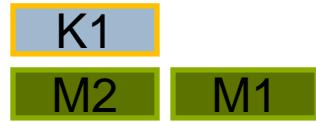
K1,K2,M1,M2



K1,M1,M2,



K1,M2,M1,



K1,M2,M2,



# Summary

---

Optimization needs an understanding of GPU architecture

Memory optimization: coalescing, shared memory

Execution configuration: latency hiding

Instruction throughput: use high throughput inst

**Do measurements!**

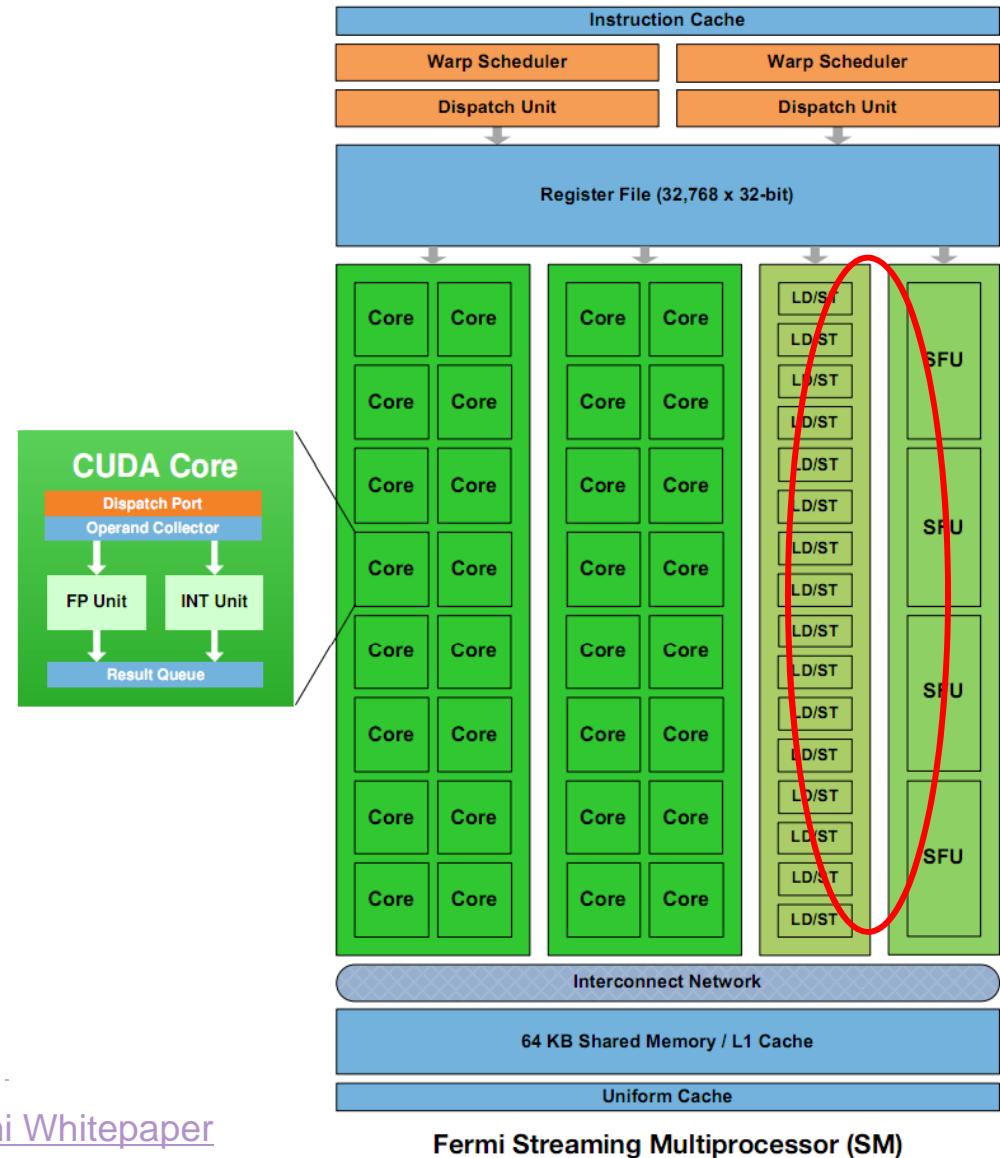
Use the Profiler, simple code modifications

Compare to theoretical peaks



# Instruction Mix

- Special Function Units (SFUs)
  - Use to compute `__sinf()`, `__expf()`
  - Only 4, each can execute 1 instruction per clock



# Loop Unrolling

---

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
 Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Instructions per iteration

One floating-point multiply

One floating-point add

What else?

# Loop Unrolling

---

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
 Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Other instructions per iteration

Update loop counter

# Loop Unrolling

---

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
 Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Other instructions per iteration

    Update loop counter

    Branch

# Loop Unrolling

---

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
 Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

Other instructions per iteration

Update loop counter

Branch

Address arithmetic

# Loop Unrolling

---

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
 Pvalue += Ms[ty][k] * Ns[k][tx];
}
```

## Instruction Mix

2 floating-point arithmetic instructions

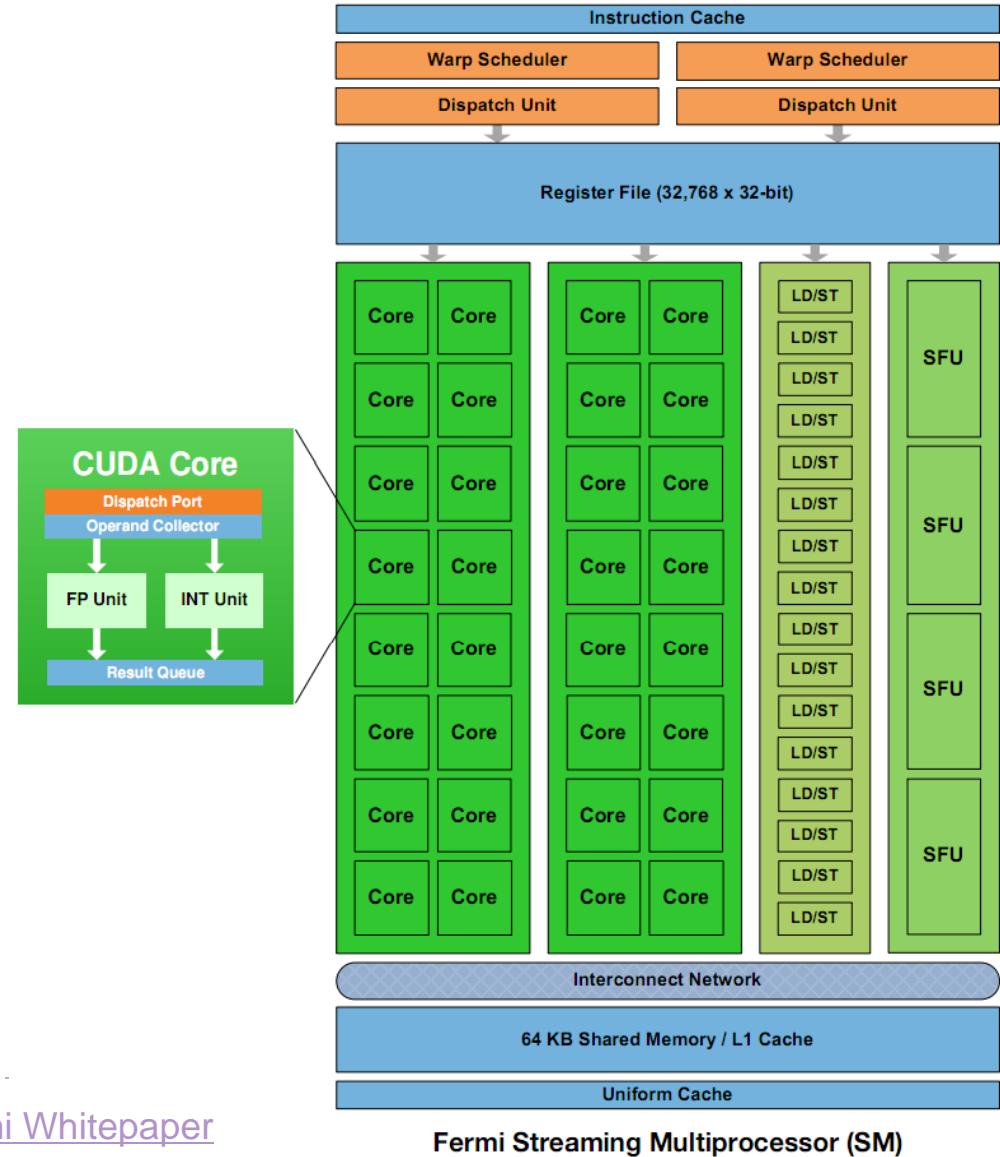
1 loop branch instruction

2 address arithmetic instructions

1 loop counter increment instruction

# Loop Unrolling

- Only 1/3 are floating-point calculations
  - But I want my full theoretical 1 TFLOP (Fermi)
  - Consider *loop unrolling*



# Loop Unrolling

---

```
Pvalue +=
 Ms[ty][0] * Ns[0][tx] +
 Ms[ty][1] * Ns[1][tx] +
 ...
 Ms[ty][15] * Ns[15][tx]; // BLOCK_SIZE = 16
```

- No more loop
  - No loop count update
  - No branch
  - Constant indices – no address arithmetic instructions



# Loop Unrolling

---

Automatically:

```
#pragma unroll BLOCK_SIZE
for (int k = 0; k < BLOCK_SIZE; ++k)
 {
 Pvalue += Ms[ty][k] * Ns[k][tx];
 }
```

Disadvantages to unrolling?

# CNN & cuDNN

Bin ZHOU

USTC Jan. 2015

## Acknowledgement

---

- ▶ Reference:
- ▶ 1) Introducing NVIDIA® cuDNN, Sharan Chetlur,  
Software Engineer,
- ▶ CUDA Libraries and Algorithms Group
  
- ▶ 深度卷积神经网络CNNs的多GPU并行框架 及其在图像识别的应用 --  
<http://data.qq.com/article?id=1516>



# CNN

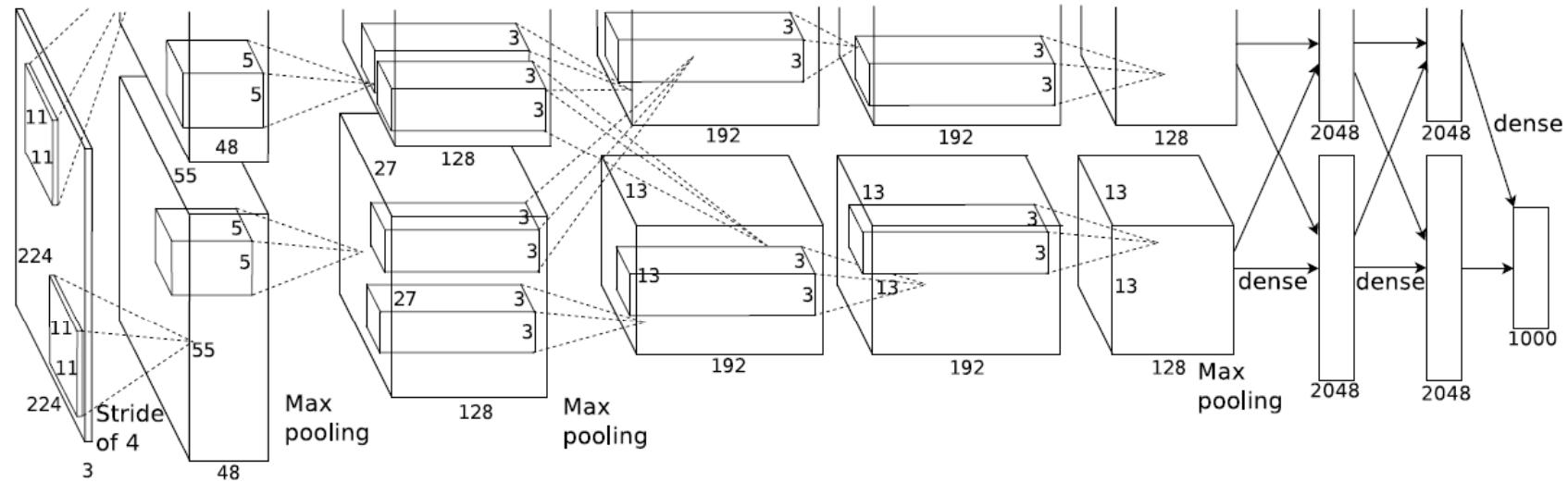
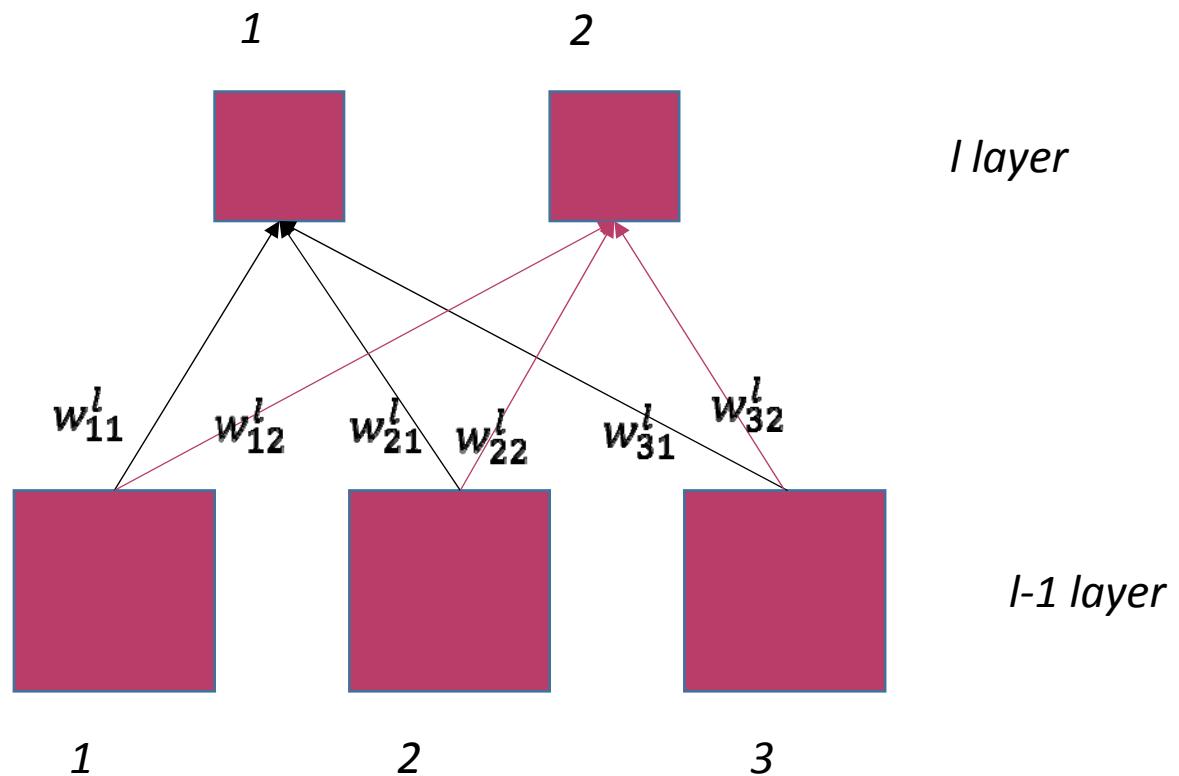


Figure1. ImageNet CNN Model



# Recall BP Network



# BP Brief Review

---

- Cost (Loss) Function To Evaluate the output of the network
- Common Cost Function
  - MSE (Mean Squared Error)
  - Cross Function



## 2D Convolution

$$\begin{pmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 3 & 1 \\ 0 & 5 & 0 \\ 2 & 1 & 2 \end{pmatrix}$$

full      same      valid

|    |     |     |     |     |     |    |
|----|-----|-----|-----|-----|-----|----|
| 17 | 75  | 90  | 35  | 40  | 53  | 15 |
| 23 | 159 | 165 | 45  | 105 | 137 | 16 |
| 38 | 198 | 120 | 165 | 205 | 197 | 52 |
| 56 | 95  | 160 | 200 | 245 | 184 | 35 |
| 19 | 117 | 190 | 255 | 235 | 106 | 53 |
| 20 | 89  | 160 | 210 | 75  | 90  | 6  |
| 22 | 47  | 90  | 65  | 70  | 13  | 18 |

# CNN Brief

---

- Interpret AI task as the evaluation of complex function
  - Facial Recognition: Map a bunch of pixels to a name
  - Handwriting Recognition: Image to a character
- Neural Network: Network of interconnected simple “neurons”
- Neuron typically made up of 2 stages:
  - Linear Transformation of data
  - Point-wise application of non-linear function
- In a CNN, Linear Transformation is a convolution



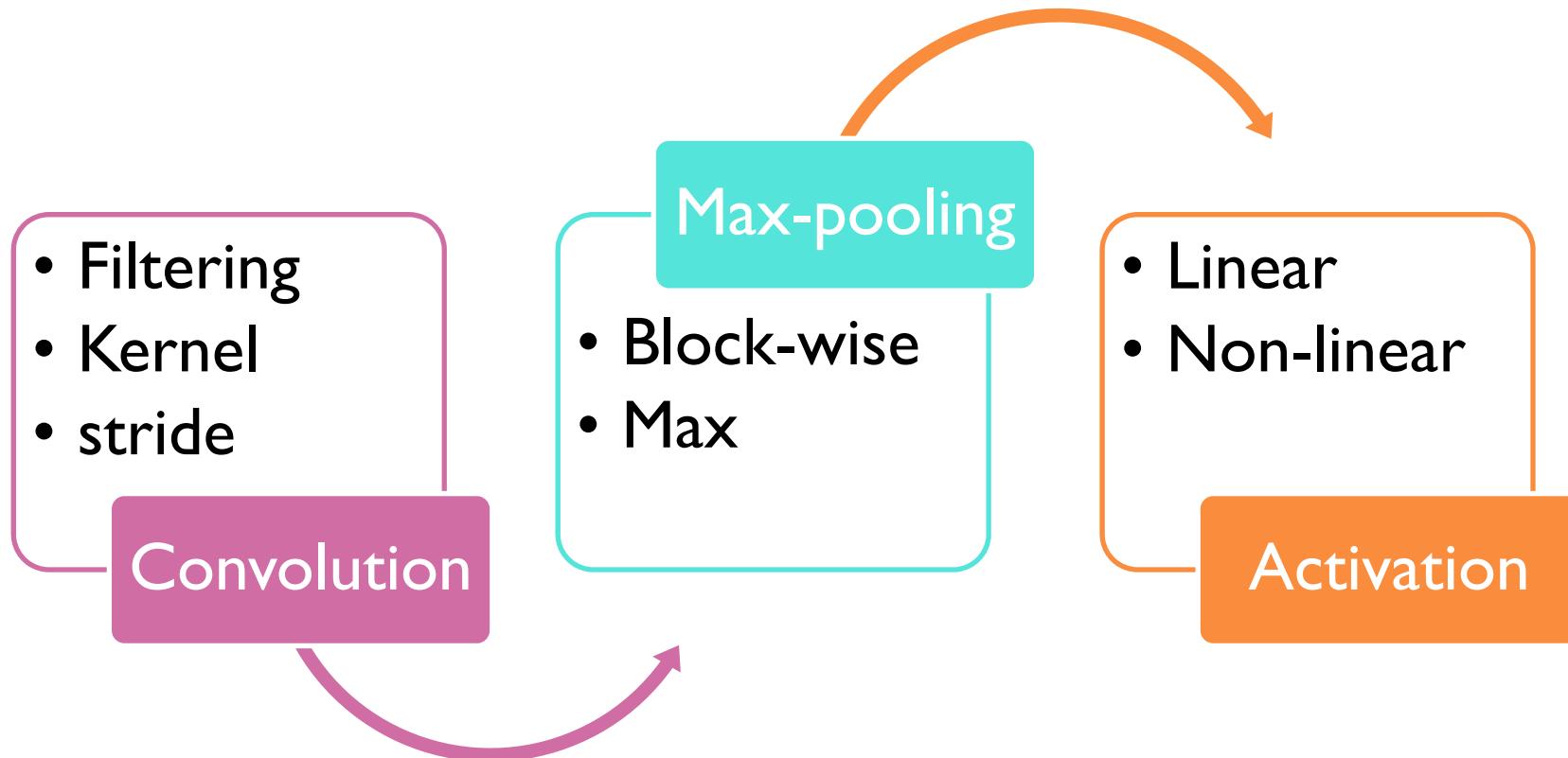
## cuDNN

---

- ▶ implementations of routines
  - Convolution
  - Pooling
  - softmax
  - neuron activations, including:
    - Sigmoid
    - Rectified linear (ReLU)
    - Hyperbolic tangent (TANH)



# CNNs: Stacked Repeating Triplets



# Applications ?

---

- ▶ Anyone Enlighten me?
- ▶ You can bring more brilliant applications



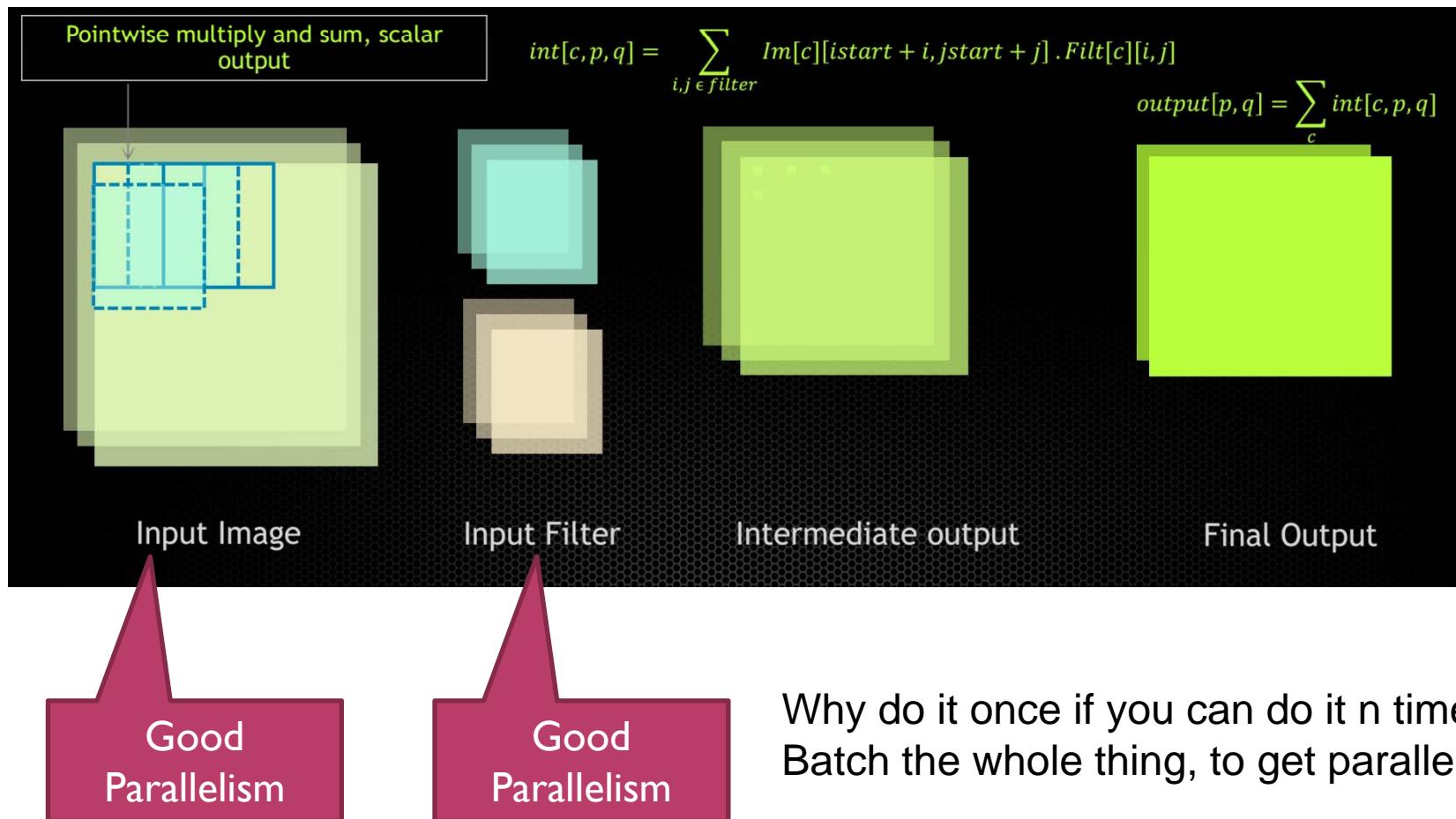
## Multi-convolve overview

---

- Linear Transformation part of the CNN neuron
  - Main computational workload
  - 80–90% of execution time
- Generalization of the 2D convolution (a 4D tensor convolution)
- Very compute intensive, therefore good for GPUs
- However, not easy to implement efficiently



# Multi-convolve, pictorially



## cuDNN—GPU accelerated CNN lib

---

- ▶ Low-level Library of GPU-accelerated routines;  
similar  
in intent to BLAS
- ▶ Out-of-the-box speedup of Neural Networks
- ▶ Developed and maintained by NVIDIA
- ▶ Optimized for current and future NVIDIA GPU  
generations
- ▶ First release focused on Convolutional Neural  
Networks



## cuDNN Features

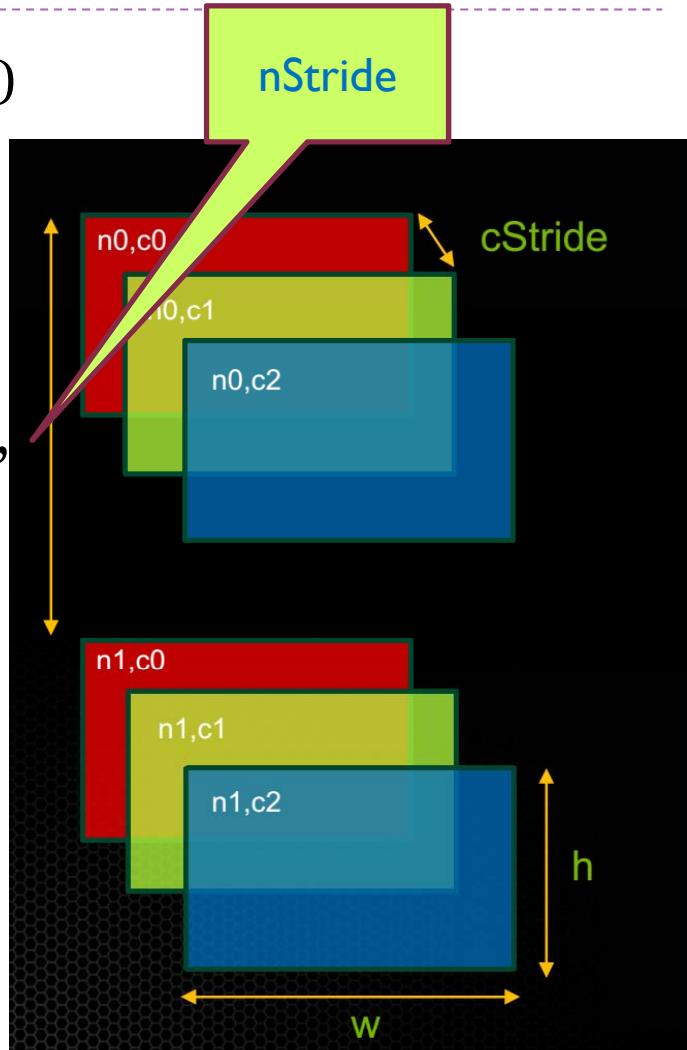
---

- ▶ Flexible API : arbitrary dimension ordering, striding, and sub-regions for 4d tensors
- ▶ Less memory, more performance : Efficient forward and backward convolution routines with zero memory overhead
- ▶ Easy Integration : black box implementation of convolution and other routines - ReLu, Sigmoid, Tanh, Pooling, Softmax



# Tensor-4d: Important

- ▶ Image Batches described as 4D Tensor  
[n, c, h, w] with stride support  
[nStride, cStride, hStride, wStride]
- ▶ Allows flexible data layout
- ▶ Easy access to subsets of features (Caffe's “groups” )
- ▶ Implicit cropping of sub-images
- ▶ Plan to handle negative



# Example - OverFeat Layer 1

```
/* Allocate memory for Filter and ImageBatch, fill with data */
cudaMalloc(&ImageInBatch , ...);
cudaMalloc(&Filter , ...);
...

/* Set descriptors */
cudnnSetTensor4dDescriptor(InputDesc, CUDNN_TENSOR_NCHW, 128, 96, 221, 221);
cudnnSetFilterDescriptor(FilterDesc, 256, 96, 7, 7);
cudnnSetConvolutionDescriptor(convDesc, InputDesc, FilterDesc,
 pad_x, pad_y, 2, 2, 1, 1, CUDNN_CONVOLUTION);

/* query output layout */
cudnnGetOutputTensor4dDim(convDesc, CUDNN_CONVOLUTION_FWD, &n_out, &c_out, &h_out, &w_out);

/* Set and allocate output tensor descriptor */
cudnnSetTensor4dDescriptor(&OutputDesc, CUDNN_TENSOR_NCHW, n_out, c_out, h_out, w_out);
cudaMalloc(&ImageBatchOut, n_out * c_out * h_out * w_out * sizeof(float));

/* launch convolution on GPU */
cudnnConvolutionForward(handle, InputDesc, ImageInBatch, FilterDesc, Filter, convDesc,
 OutputDesc, ImageBatchOut, CUDNN_RESULT_NO_ACCUMULATE);
```



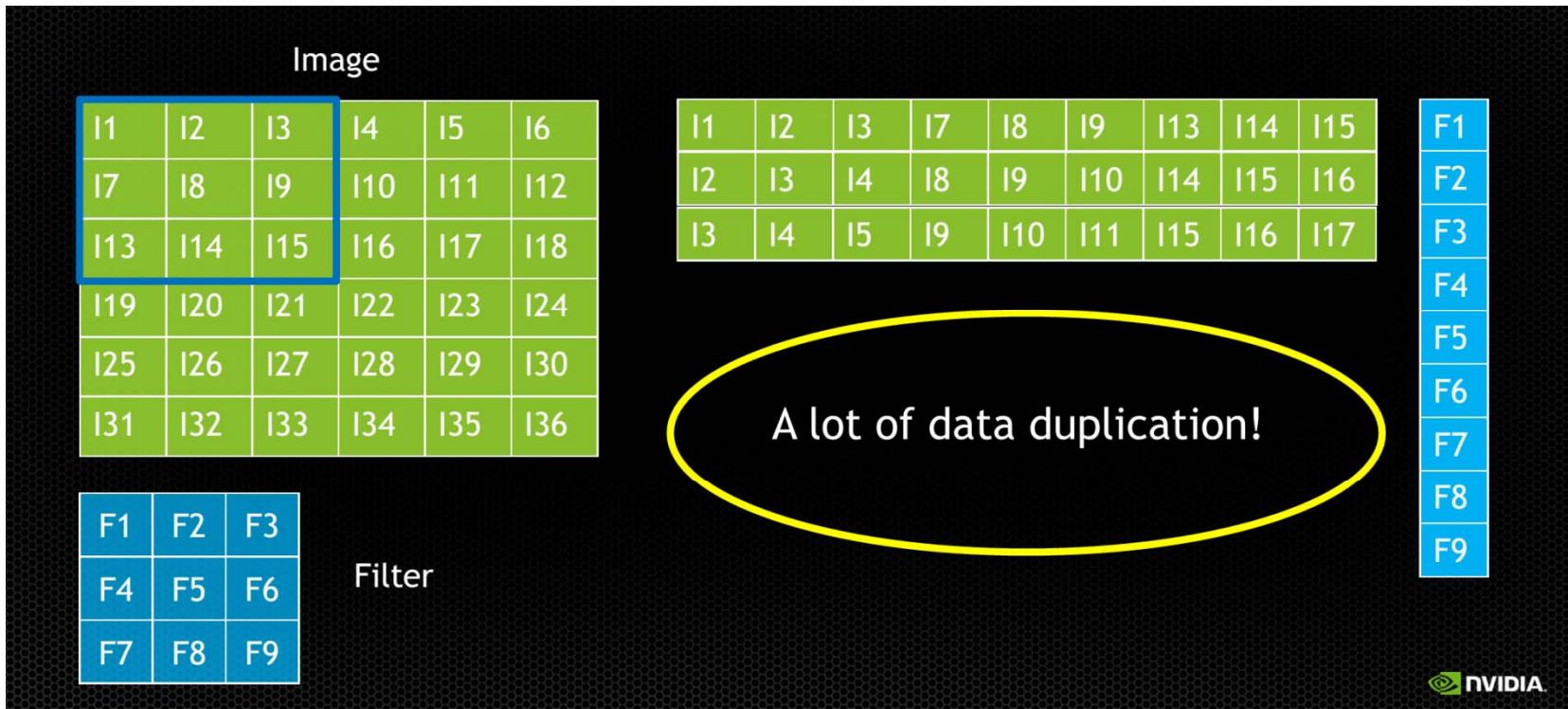
# Real Code that runs

---

- ▶ Under Linux
- ▶ Demostration



# Implementation 1: 2D conv as a GEMV



## Multi-convolve

---

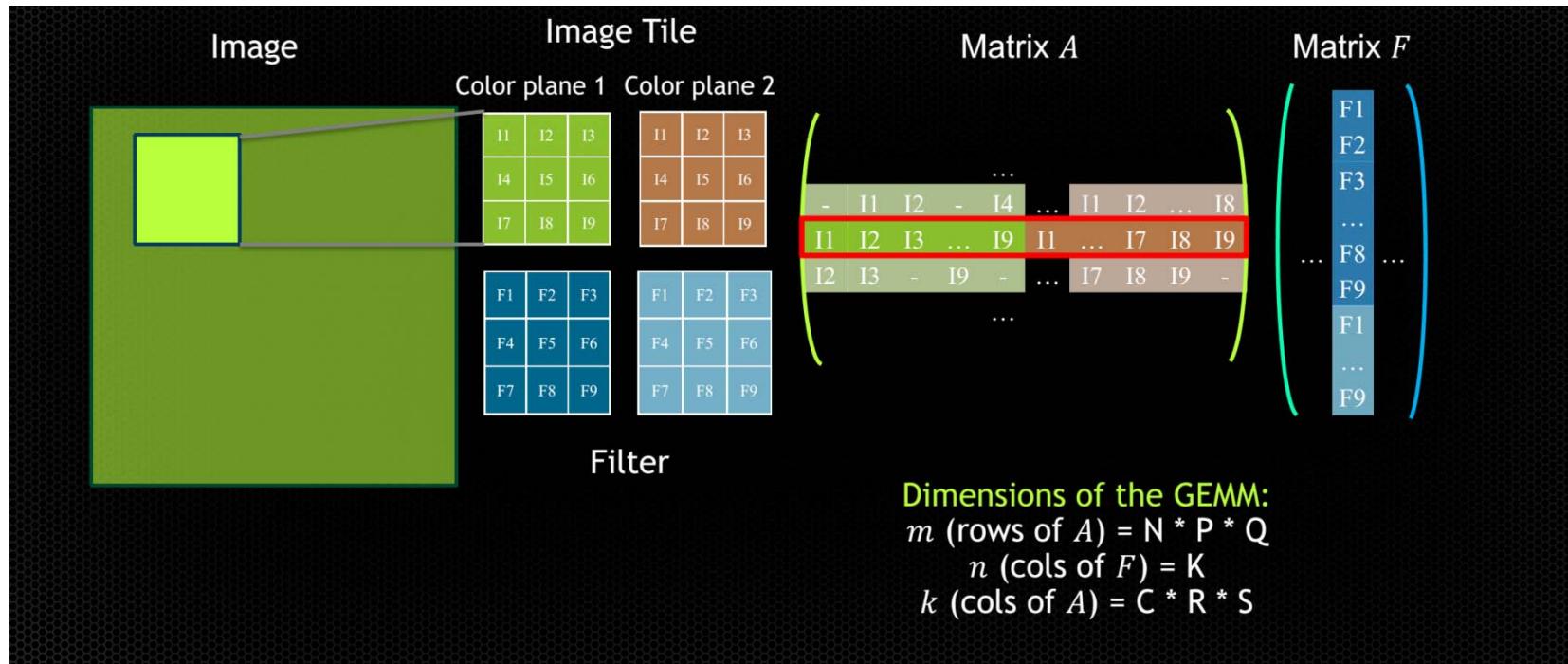
- ▶ More of the same, just a little different
  - Longer dot products
  - More filter kernels
  - Batch of images, not just one
  - Mathematically:

$$out[k, p, q] = \sum_{c \in \text{input color planes}} \left( \sum_{i, j \in \text{filter}} Im[c][istart + i, jstart + j] \cdot Filt[k][c][i, j] \right)$$

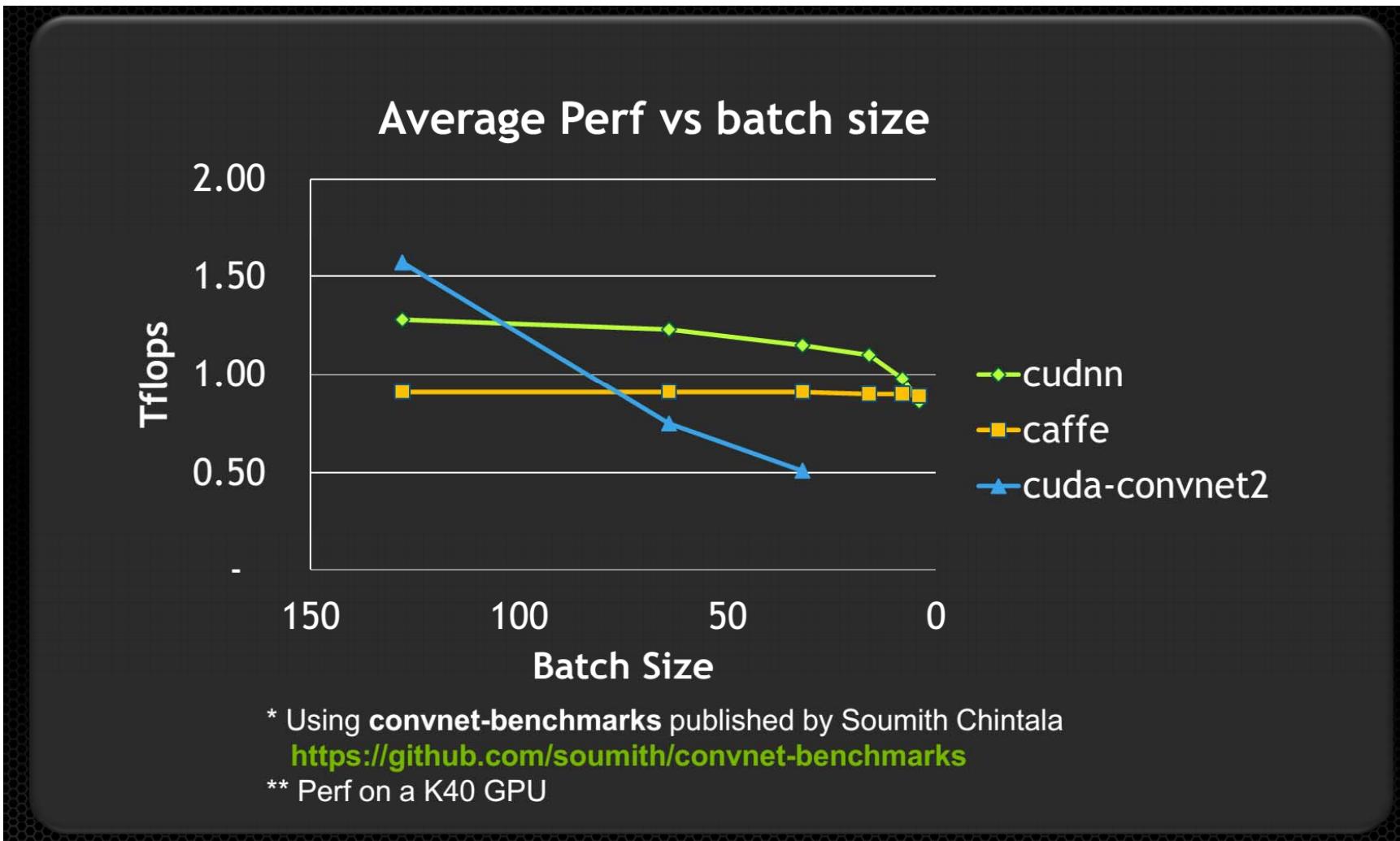
$\forall k \in \text{output color planes}, (p, q) \in \text{output image}$



# Implementation 2: Multi-convolve as GEMM



# Performance



# cuDNN Integration

---

- ▶ cuDNN is already integrated in major open-source frameworks
  - ▶ Caffe
  - ▶ Torch



# Using Caffe with cuDNN

- ▶ Accelerate Caffe layer types by 1.2 - 3x
- ▶ On average, 36% faster overall for training on Alexnet
- ▶ Integrated into Caffe dev branch today! (official release with Caffe 1.0)
- ▶ Seamless integration with a global switch



\*CPU is 24 core E5-2697v2 @ 2.4GHz Intel MKL 11.1.3

# Caffe with cuDNN: No Programming Required

---

- ▶ layers {  
    name: "MyData"  
    type: DATA  
    top: "data"  
    top: "label"  
}
  
- ▶ layers {  
    name: "Conv1"  
    type: CONVOLUTION  
    bottom: "MyData"  
    top: "Conv1"  
    convolution\_param {  
        num\_output: 96  
        kernel\_size: 11  
        stride: 4  
    }

---



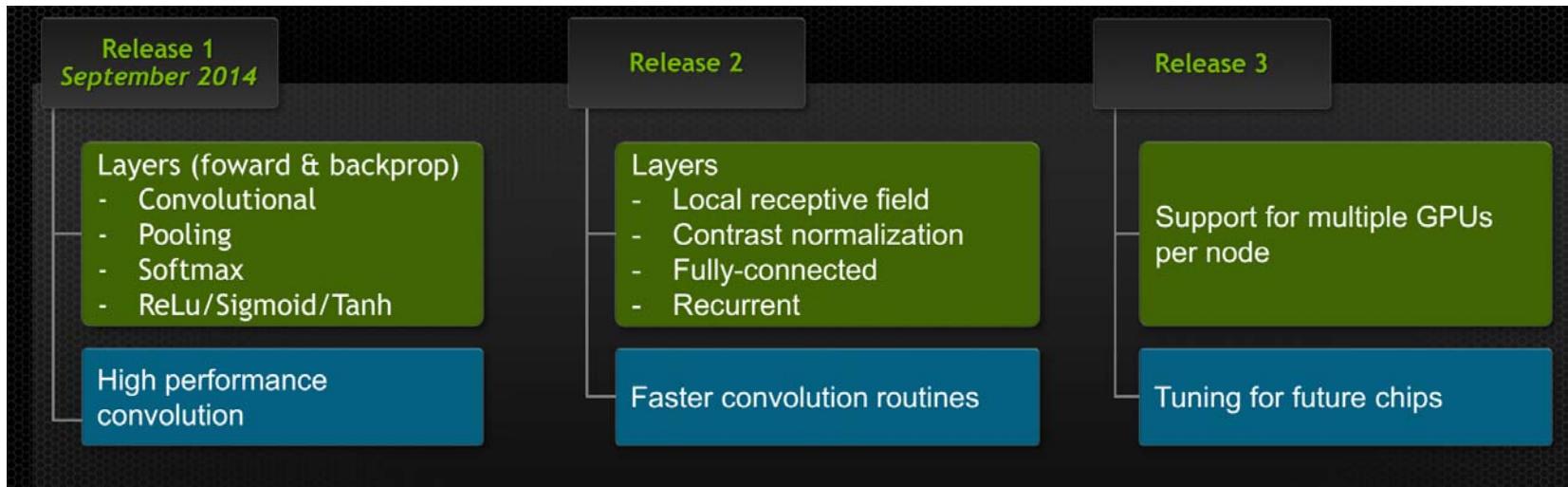
## Caffe with cuDNN : Life is easy

---

- ▶ install cuDNN
- ▶ uncomment the USE\_CUDNN := 1 flag in Makefile.config when installing Caffe.
- ▶ Acceleration is automatic



# NVIDIA® cuDNN Roadmap



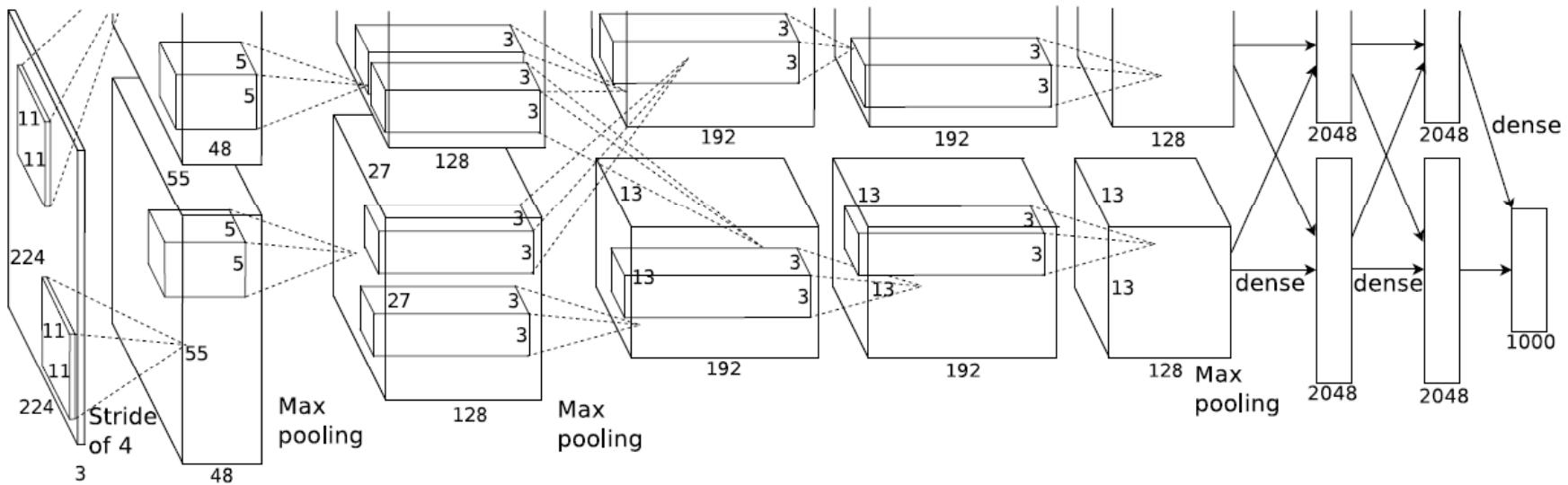
# cuDNN availability

---

- ▶ Free for registered developers!
- ▶ Release 1 / Release 2 - RC
  - ▶ available on Linux/Windows 64bit
  - ▶ GPU support for Kepler and newer
- ▶ Already Done:
  - Tegra K1 (Jetson board)
  - Mac OSX support



# Multi-GPU with CNN



Problem:

- 1) Single GPU has limited memory, which limits the size of the network
- 2) Single GPU is still too slow for some very large scale network



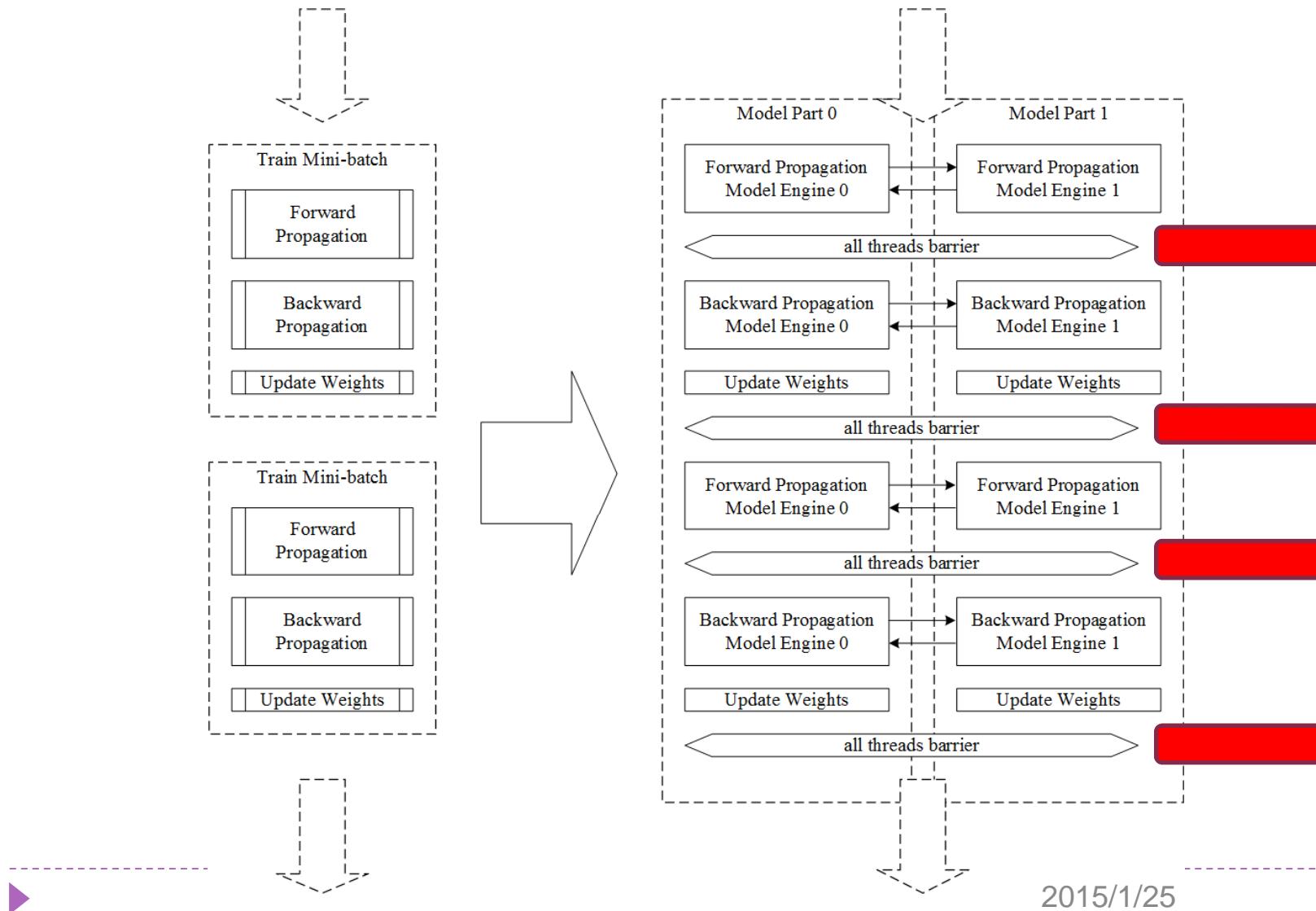
# Multi-GPU Challenge

---

- ▶ First, How to parallelize the whole process, to avoid or reduce data dependency between different nodes
- ▶ Data I/O and distribution to different Nodes
  - ▶ Pipeline and I/O/Execution overlap to hide latency
- ▶ Synchronize between all the nodes??

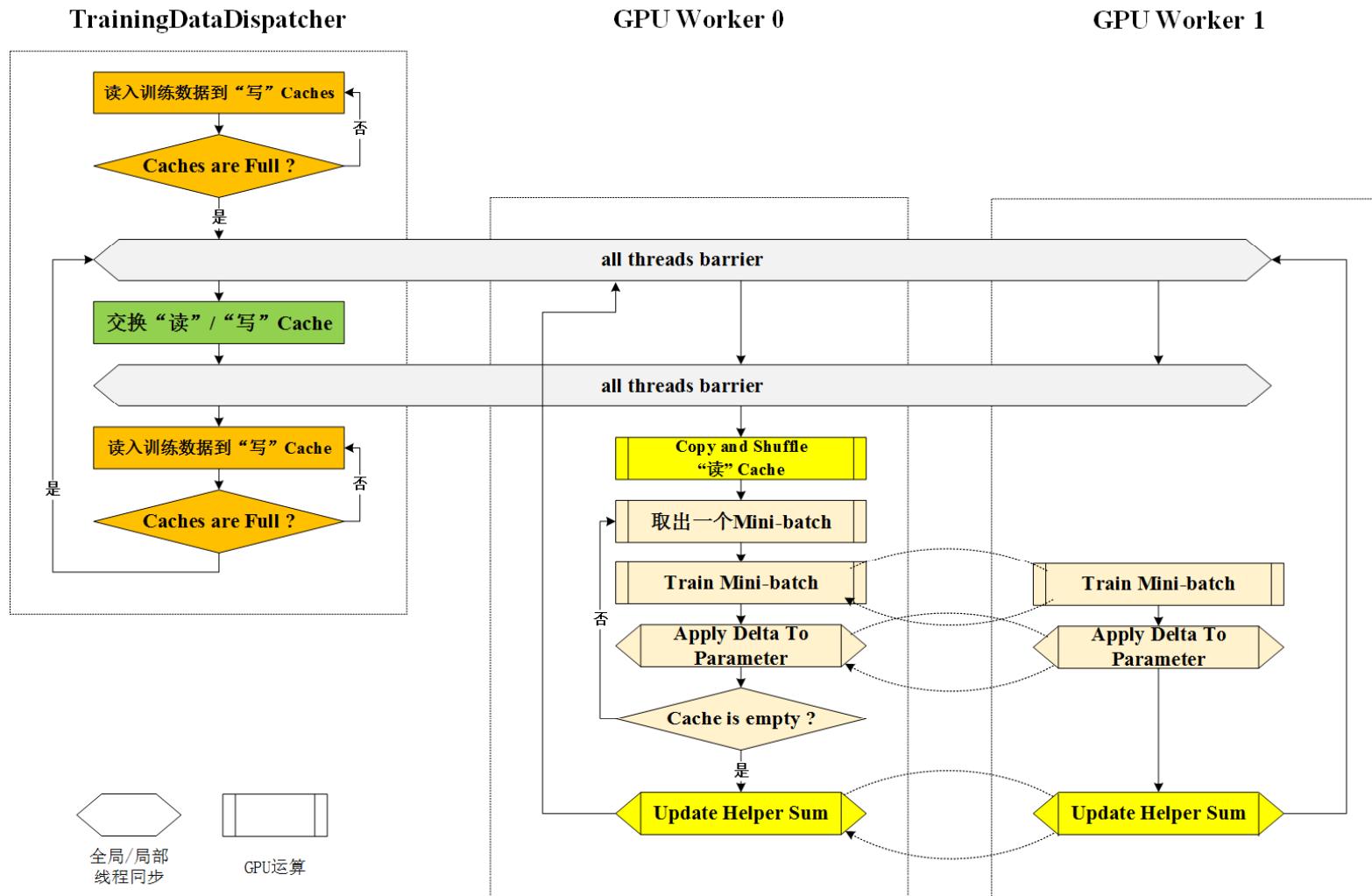


# Multi-GPU Strategy



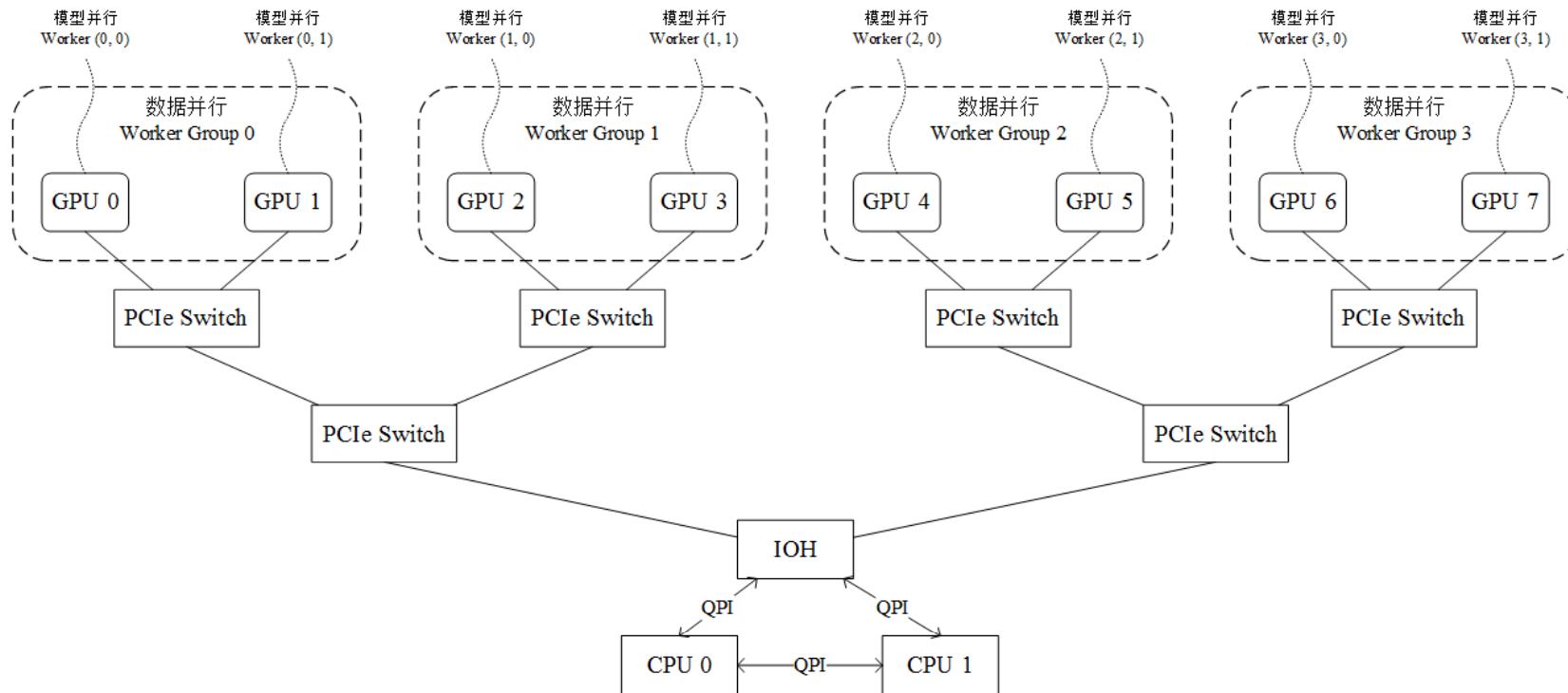
2015/1/25

# Data distribution I/O/Exe Overlap

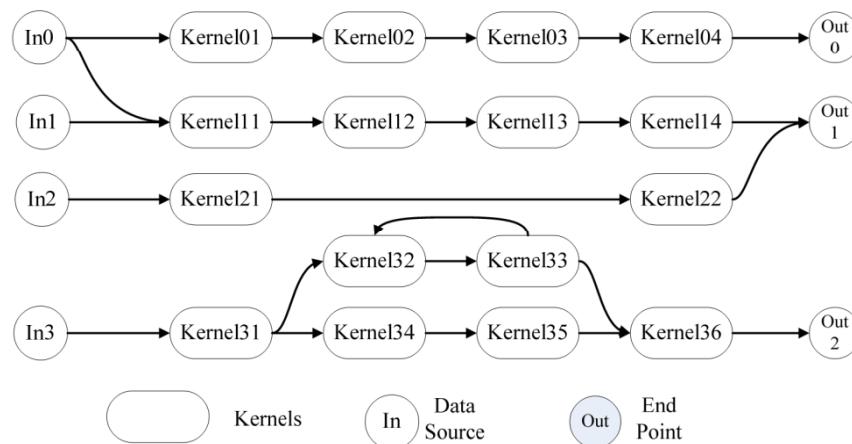
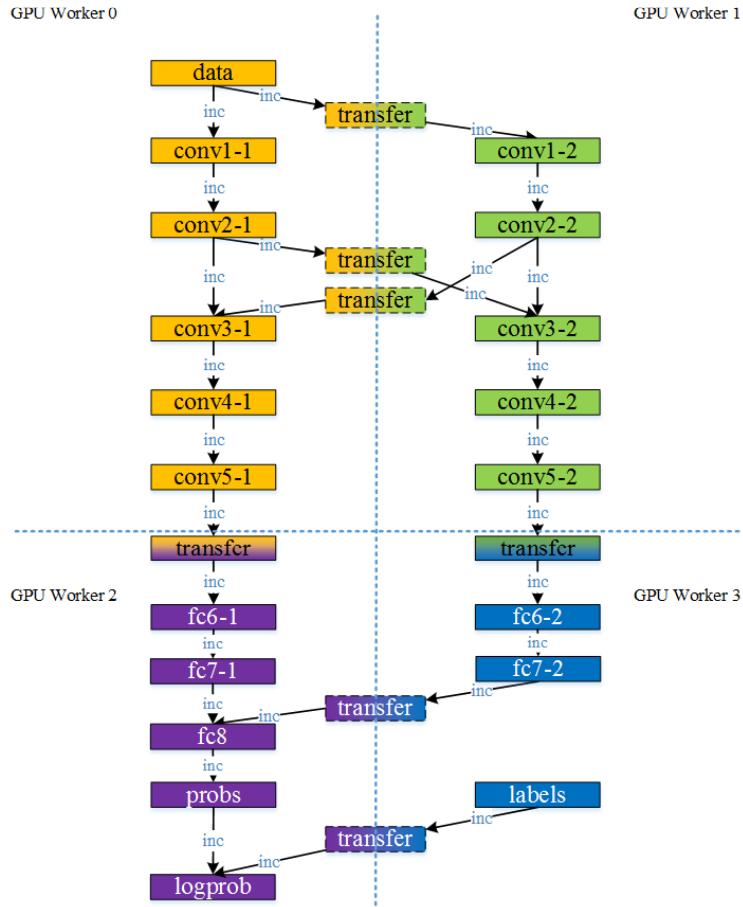


2015/1/25

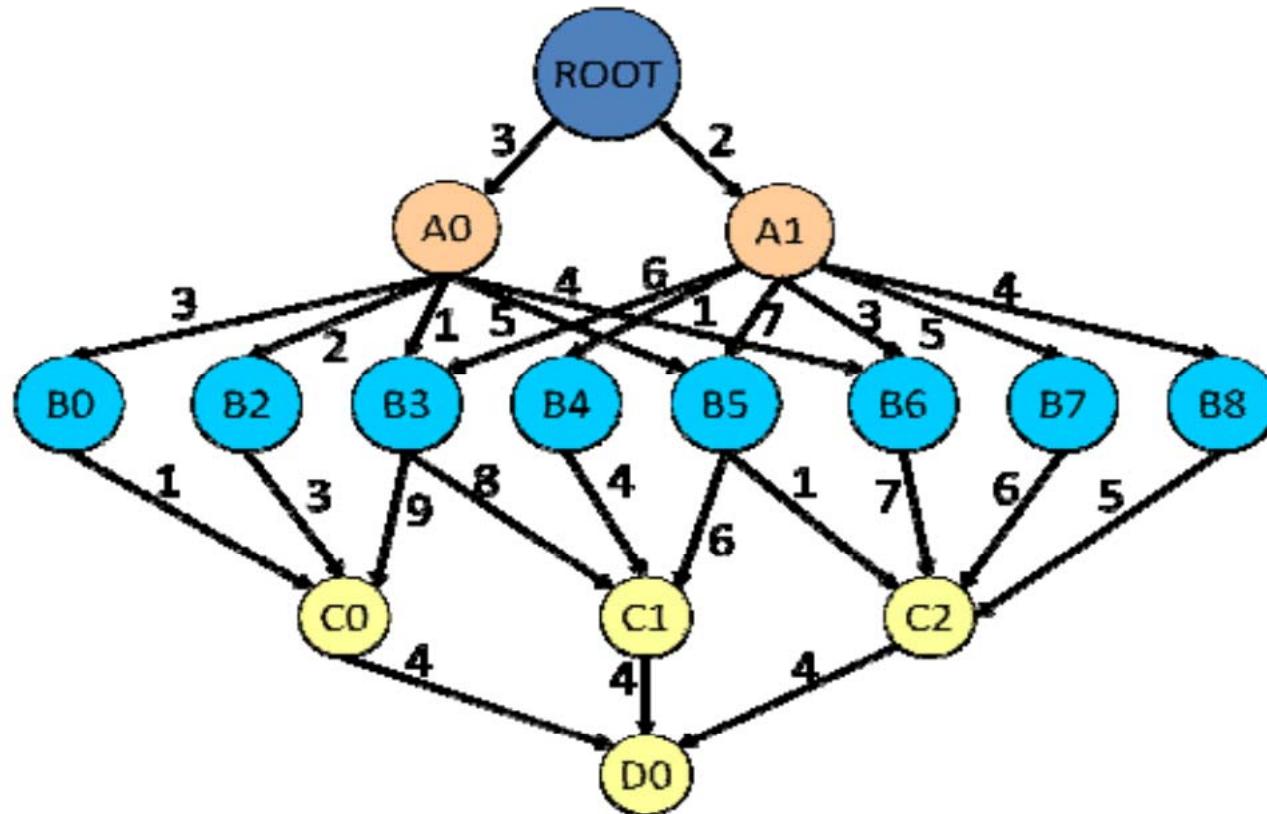
# 8-GPU server



# Pipeline and Stream processing in CNN



Familiar?? It's a DAG!



# My Algorithm for DAG auto-Parallelization

---

**WHILE** *Task\_List* !=  $\emptyset$  **DO**

**GET** first task  $n_i$ ;

**FOR** every  $p_j$  **DO**

$$EST(n_i, p_j) = \max\{available(p_j), \max_{n_m \in pred(n_i)}(AFT(n_m) + c_{m,i})\};$$

$$f(n_i, p_j) = EST(n_i, p_j) + \sum_{n_m \in pred(n_i)} c_{m,i} / speed_j$$

**ENDFOR**

**SORT**( $f(n_i, p_j)$ );

**SCHEDULE**( $n_i, p_j$ ) with smallest  $f(n_i, p_j)$ ;

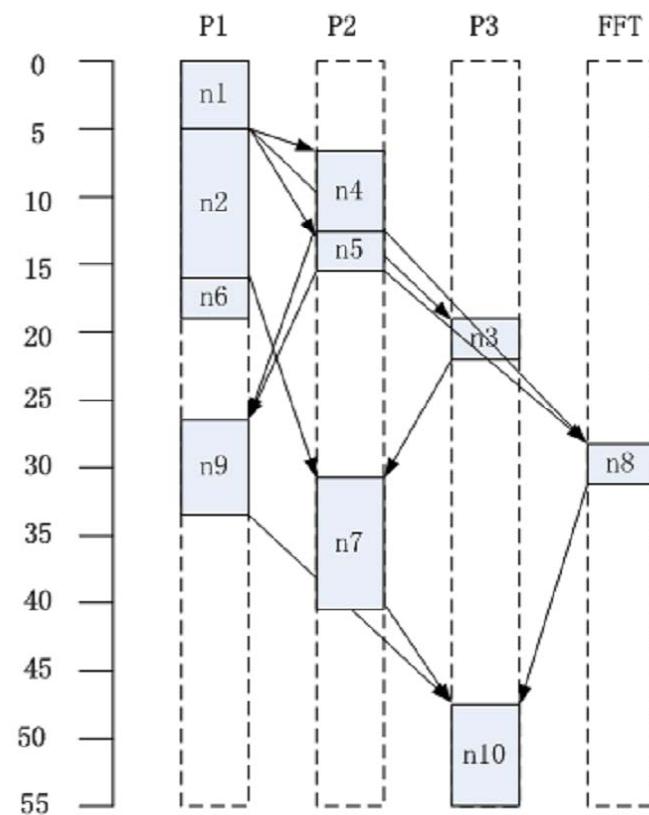
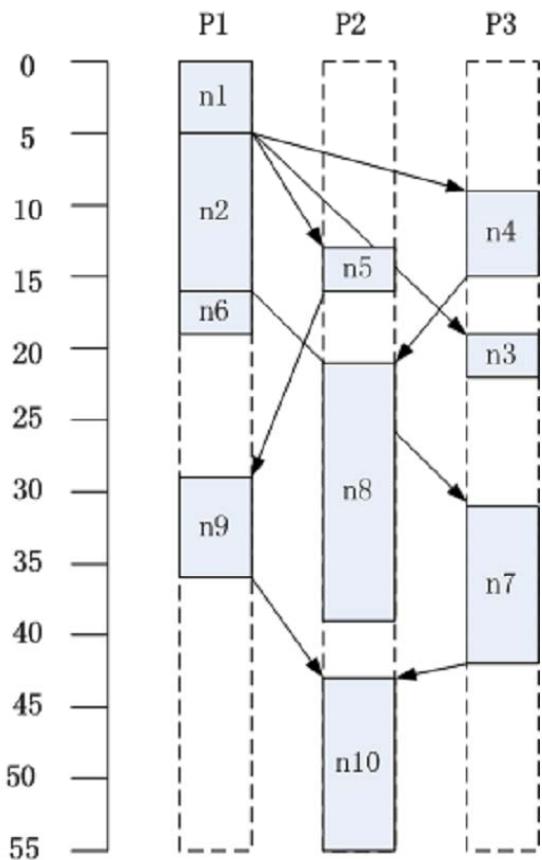
$$AST(n_i) = EST(n_i, p_j); AFT(n_i) = AST(n_i) + w'_{i,j};$$

$$available(p_j) = EFT(n_i, p_j);$$

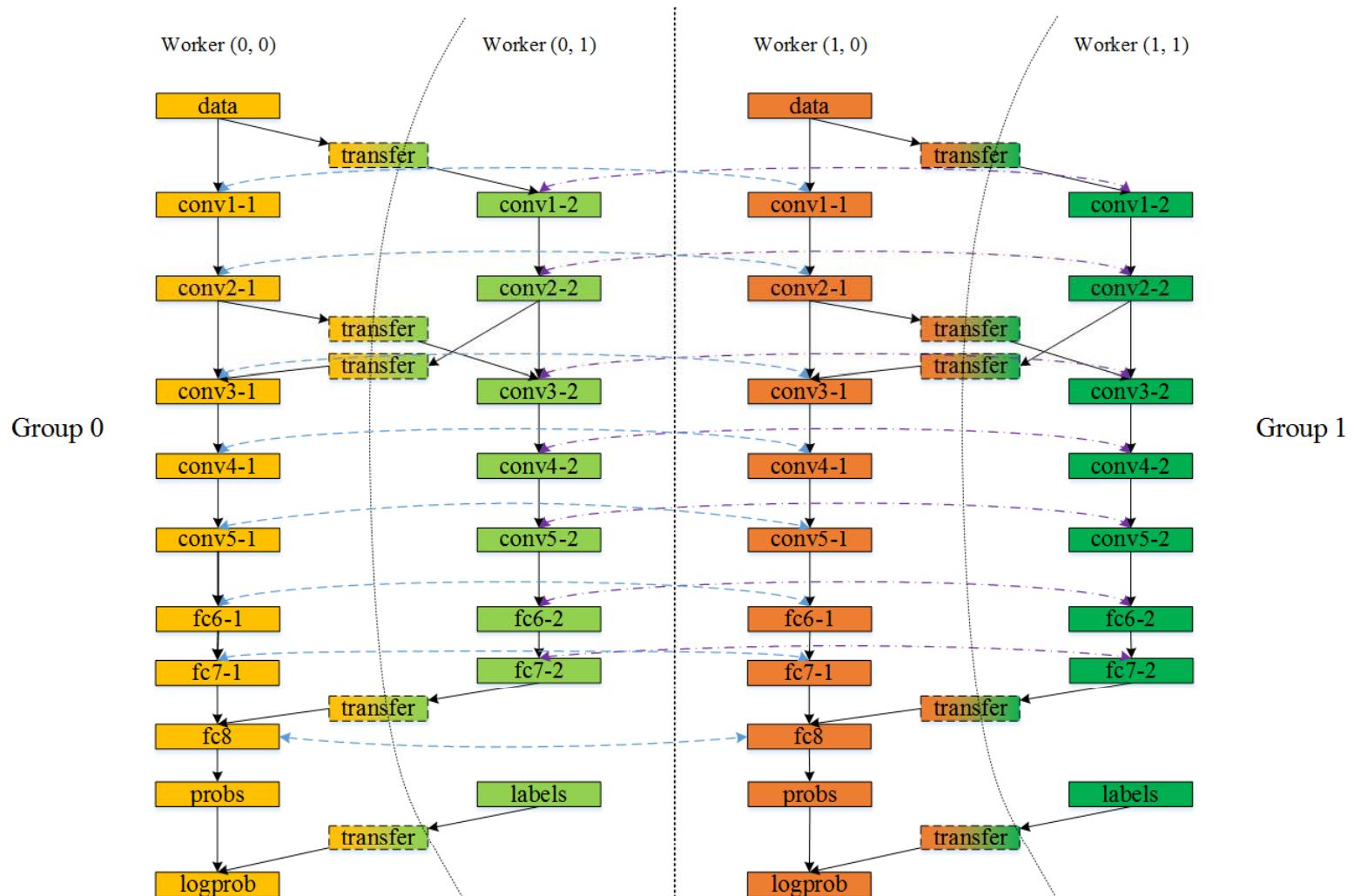
**ENDWHILE**



# Test Case



# A More Complex Case



# Speed with Multi-GPUs

---

| Configuration                | Speedup vs. 1 GPU |
|------------------------------|-------------------|
| 2 GPUs Model P.              | 1.71              |
| 2 GPUs Data P                | 1.85              |
| 4 GPUs Data P. +<br>Model P. | 2.52              |
| 4 GPUs Data P.               | 2.67              |



# Conclusion

---

- ▶ GPU is very well suitable for CNN
- ▶ cuDNN is easy to use and good performance
- ▶ Multi-GPU is improving more.
- ▶ Carefully Designed parallel design on multi-GPU could get adequate scalability





# CPU Architecture Overview

Bin ZHOU  
USTC. Spring 2015



# CPU Pictures





# Contents

- Modern CPU Architecture and Performance Consideration
  - Pipelining
  - Branch Prediction
  - Superscalar
  - Out-of-Order (OoO) Execution
  - Memory Hierarchy
  - Vector Operations
  - SMT
  - Multicore
- Quick/Simple Question
  - What is a CPU????



# What is a CPU anyways?

- Execute instructions
  - ALU, Arithmetic and Logic Unit
- Now so much more
  - Interface to main memory (DRAM)
  - I/O functionality
  - Ports
- Composed of transistors
  - Millions and Billions of them



# Instructions

- Examples: arithmetic, memory, control flow

**add r3, r4 -> r4**

**load [r4] -> r7**

**jz end**

- Given a compiled program, minimize

$$\frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

– CPI (cycles per instruction) & clock period

– Reducing one term may increase the other

- 增长到天空才是尽头？彼此不独立。
- Paradox or dilemma??

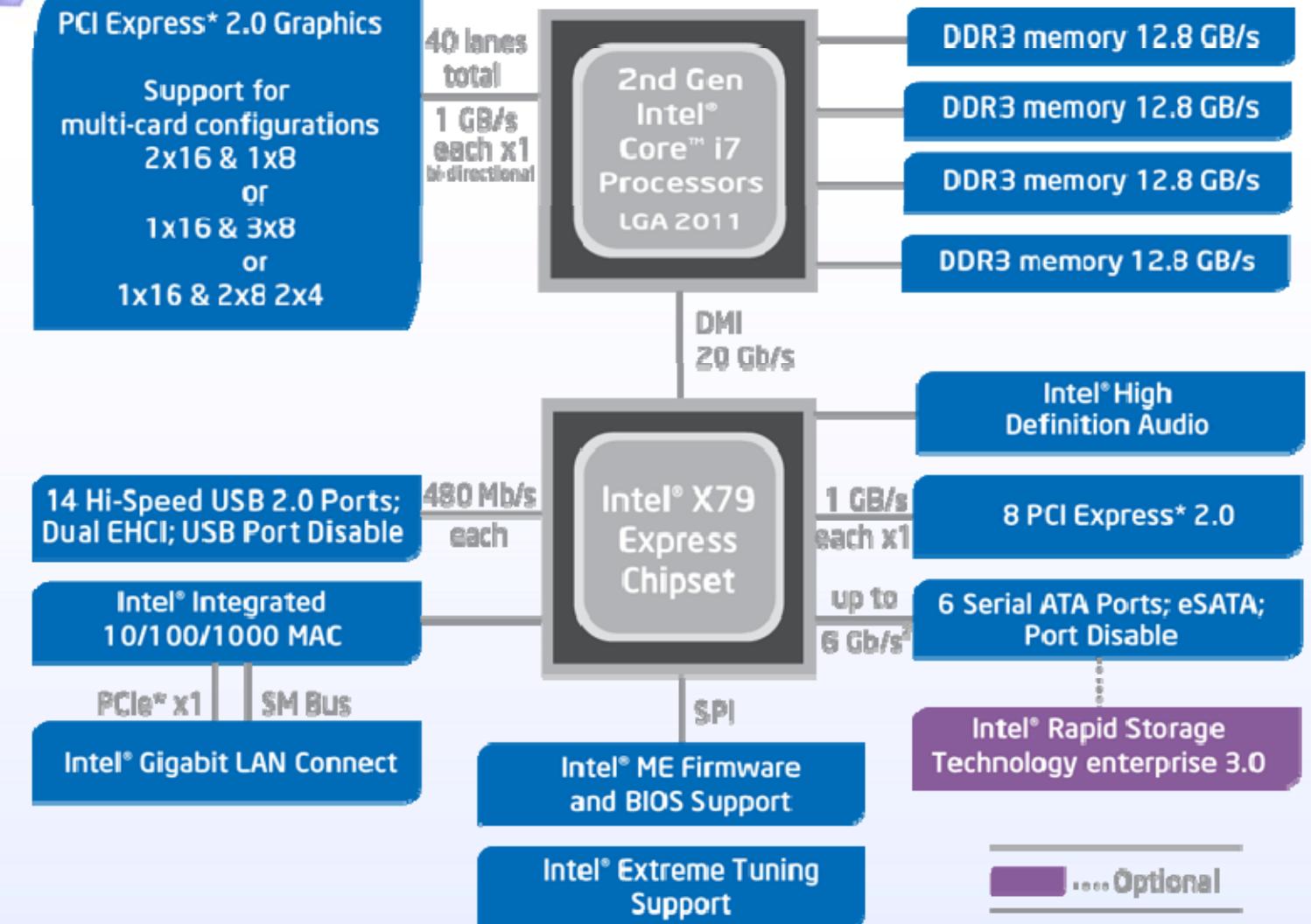


# Desktop Programs

- Lightly threaded
- Lots of branches
- Lots of memory accesses
- Only a few Vectors and Arithmetic operations

|                      | vim   | ls    |
|----------------------|-------|-------|
| Conditional branches | 13.6% | 12.5% |
| Memory accesses      | 45.7% | 45.7% |
| Vector instructions  | 1.1%  | 0.2%  |

on Linux



<sup>1</sup>Theoretical maximum bandwidth

<sup>2</sup>All SATA ports capable of 3 Gb/s, 2 ports capable of 6 Gb/s.

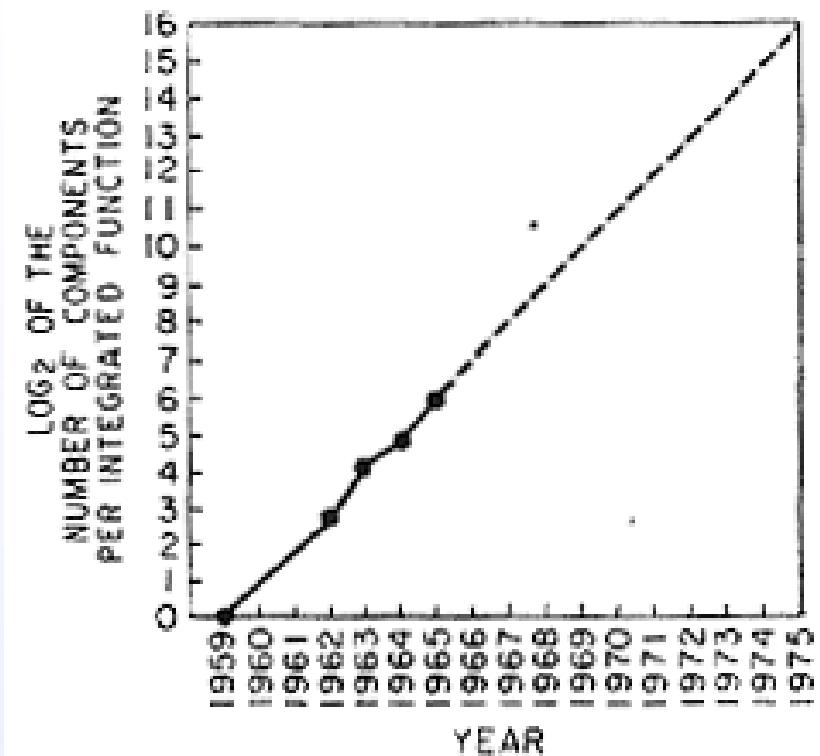
## Intel® X79 Express Chipset Block Diagram

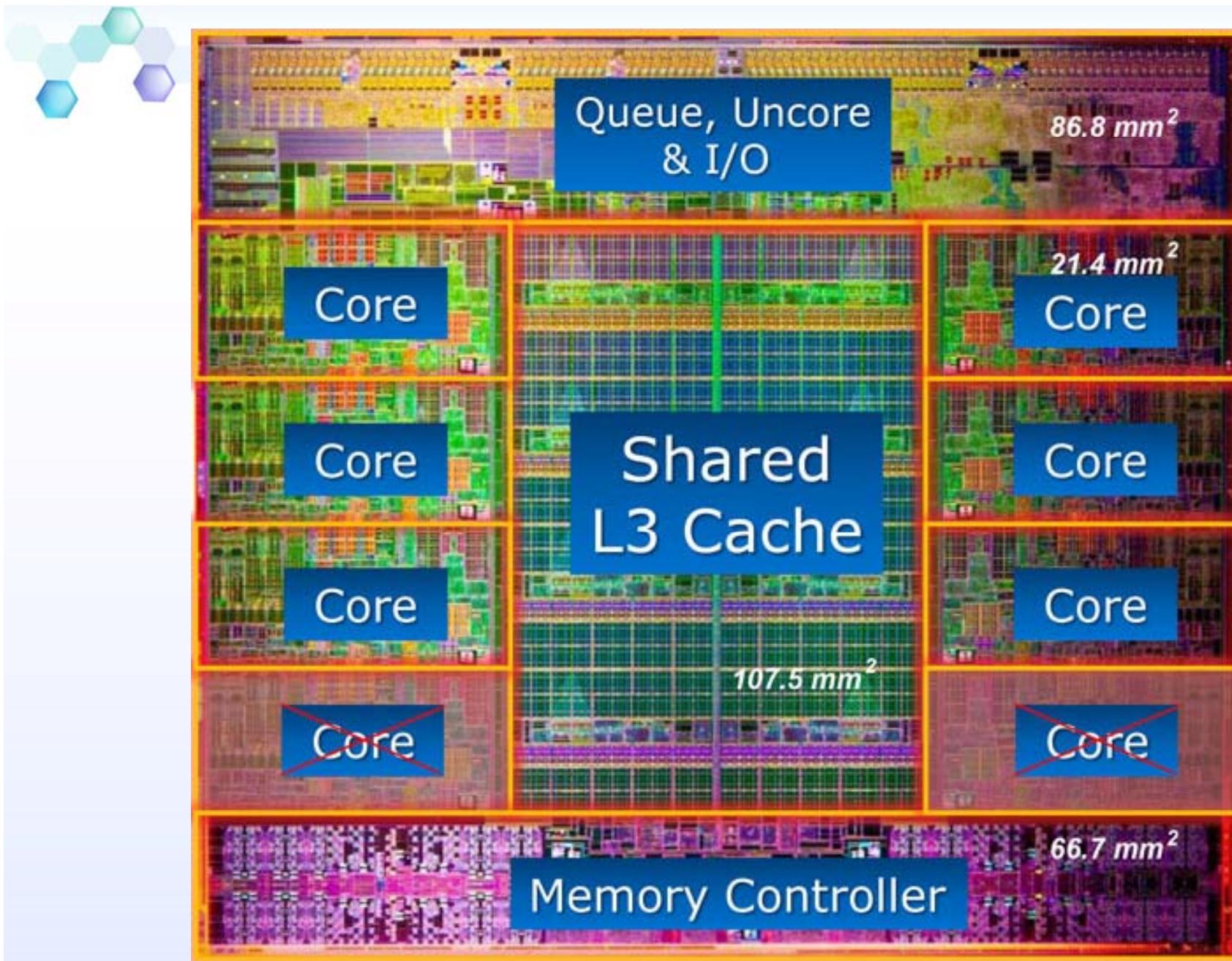
Source: [intel.com](http://intel.com)



# Moore's Law

- “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year”
- What do we do with our transistor budget?

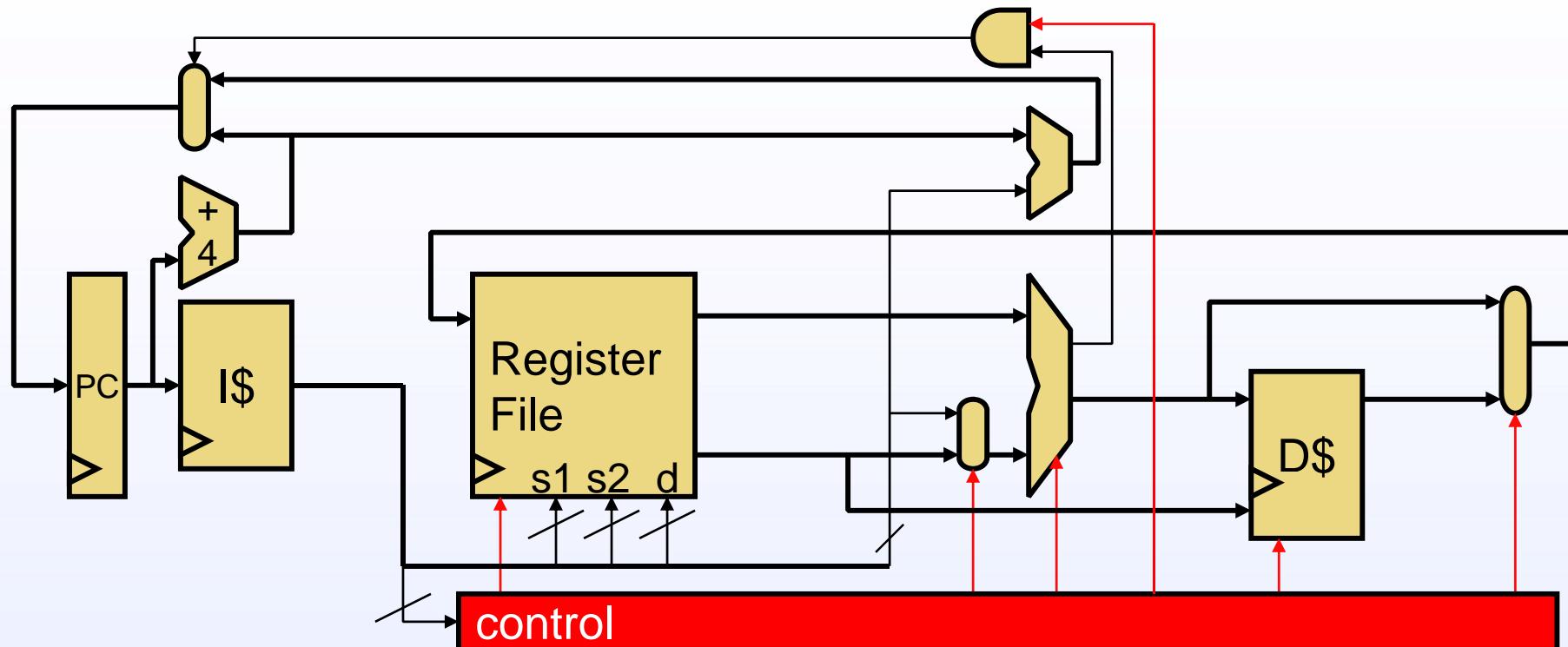




Intel Core i7 3960X (Codename Sandy Bridge-E) – 2.27B transistors, Total Size  
435mm<sup>2</sup>

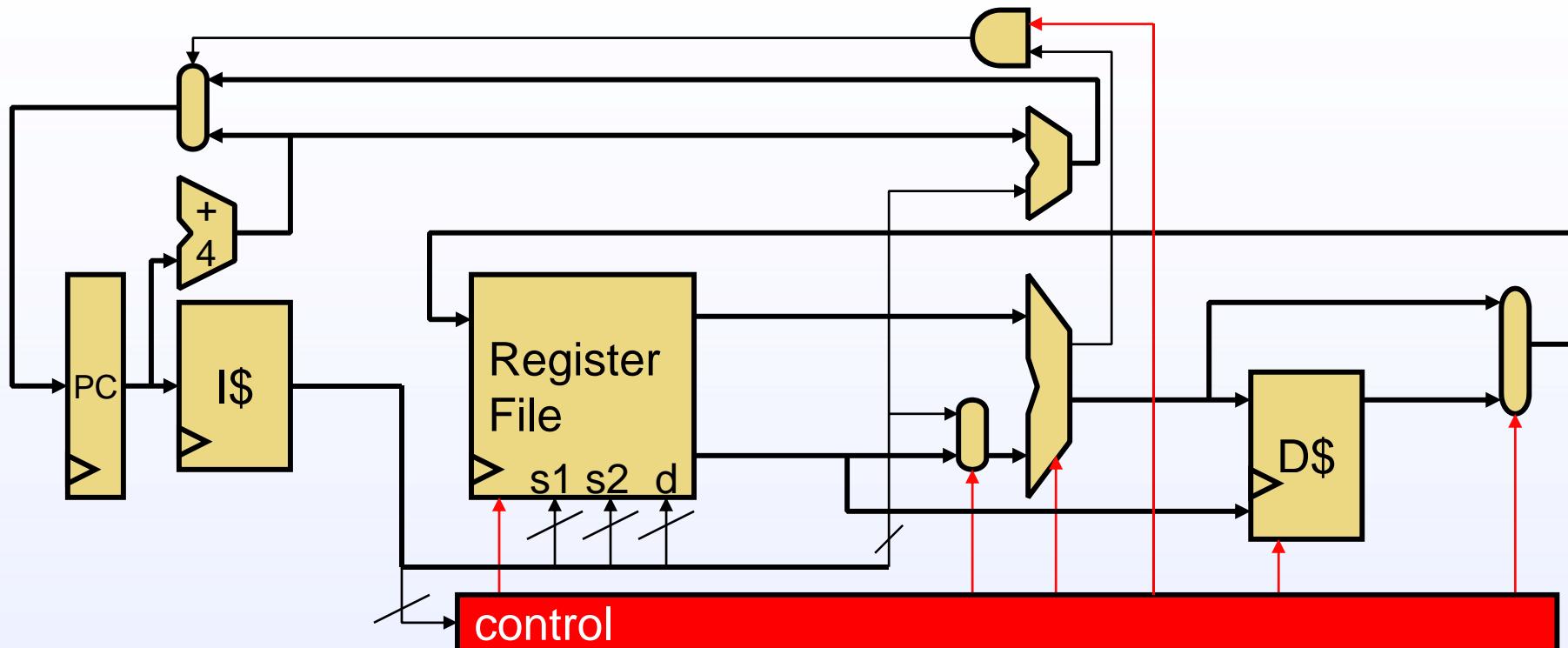


# A Simple CPU Core





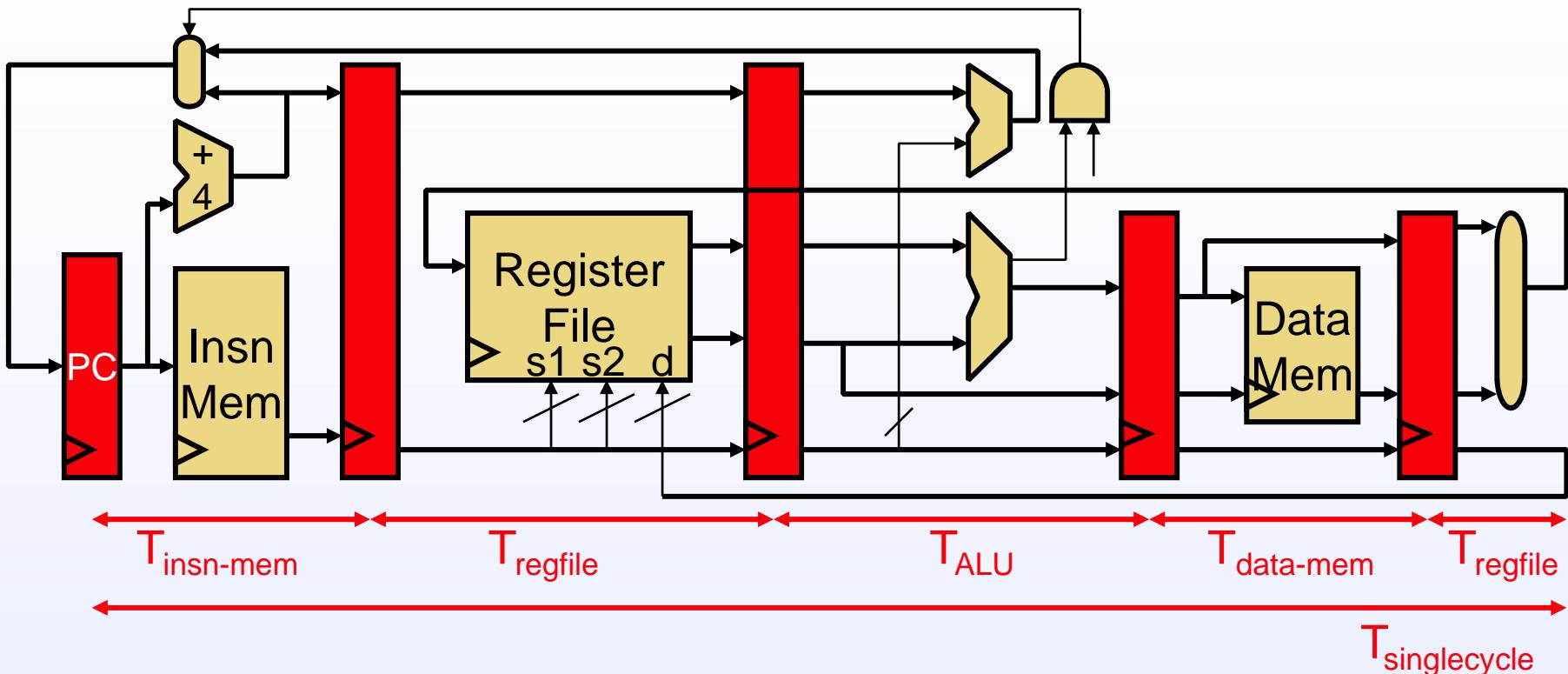
# A Simple CPU Core



Fetch → Decode → Execute → Memory → Writeback



# Pipelining



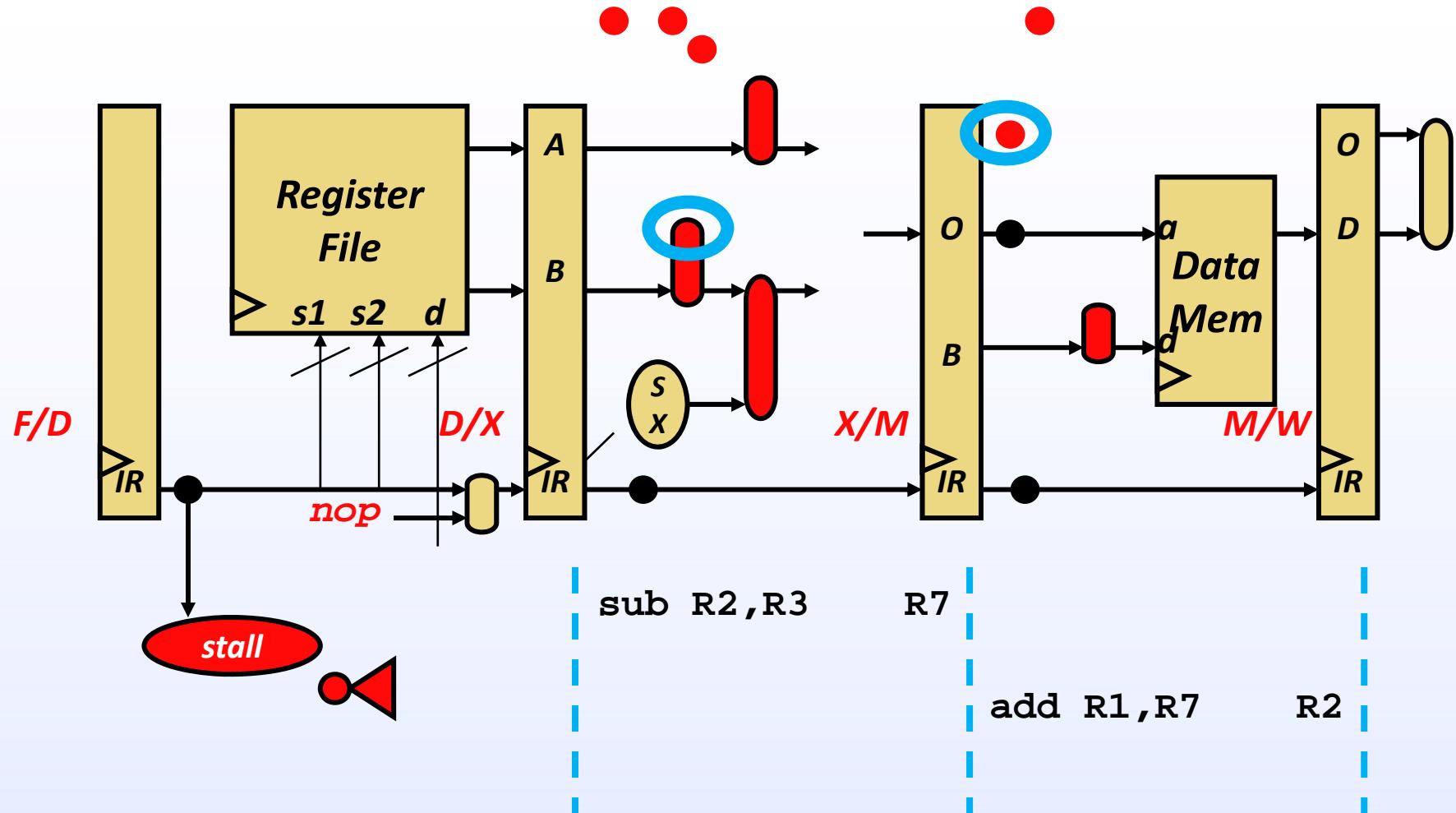


# Pipelining

- Capitalize on instruction-level parallelism (ILP)
  - + Significantly reduced clock period
  - Slight latency & area increase (pipeline latches)
  - ? Dependent instructions
  - ? Branches
- Alleged Pipeline Lengths:
  - Core 2: 14 stages
  - Pentium 4 (Prescott): > 20 stages
  - Sandy Bridge: in between



# Bypassing



# Stalls

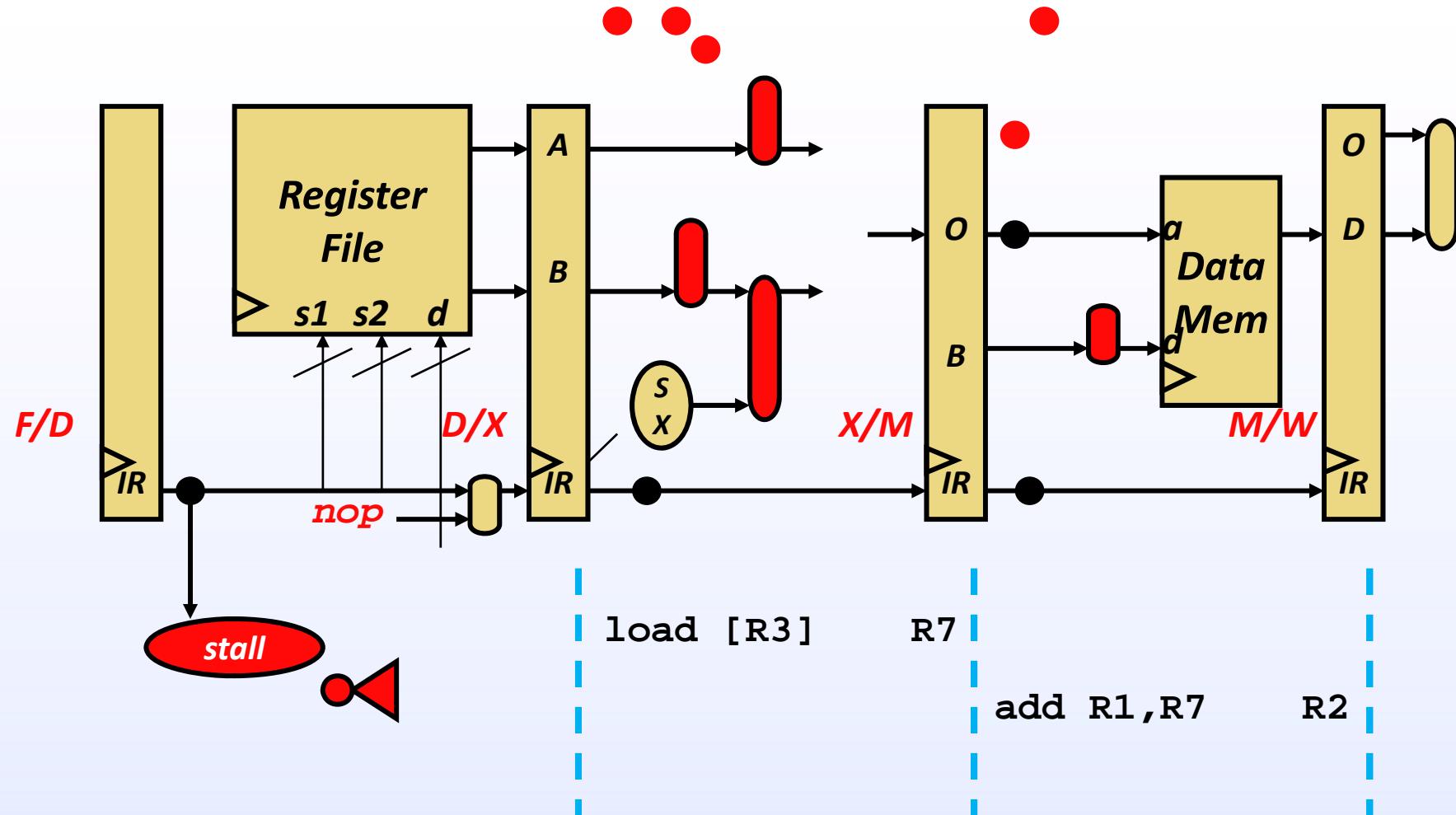
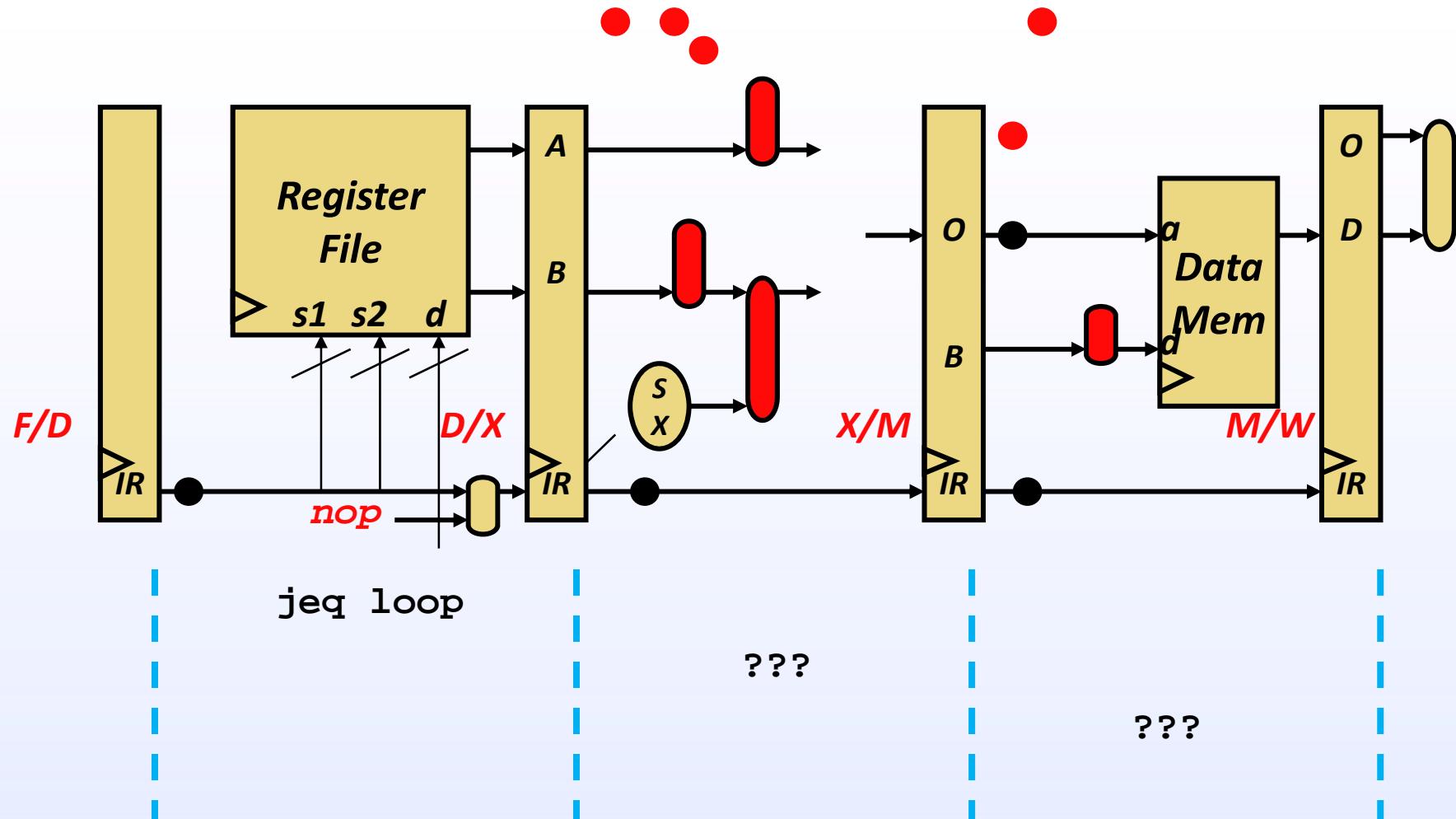


Image: [Penn CIS501](#)



# Branches





# Branch Prediction

- Guess what instruction comes next
- Based off branch history
- Example: two-level predictor with global history
  - Maintain history table of all outcomes for  $M$  successive branches
  - Compare with past  $N$  results (history register)
  - Sandy Bridge employs 32-bit history register
- Pretty Hard Work!





# Branch Prediction

- + Modern predictors > 90% accuracy
  - o Raise performance *and* energy efficiency (why?)
- Area increase
- Potential fetch stage latency increase



## Another option: Predication

- Replace branches with conditional instructions

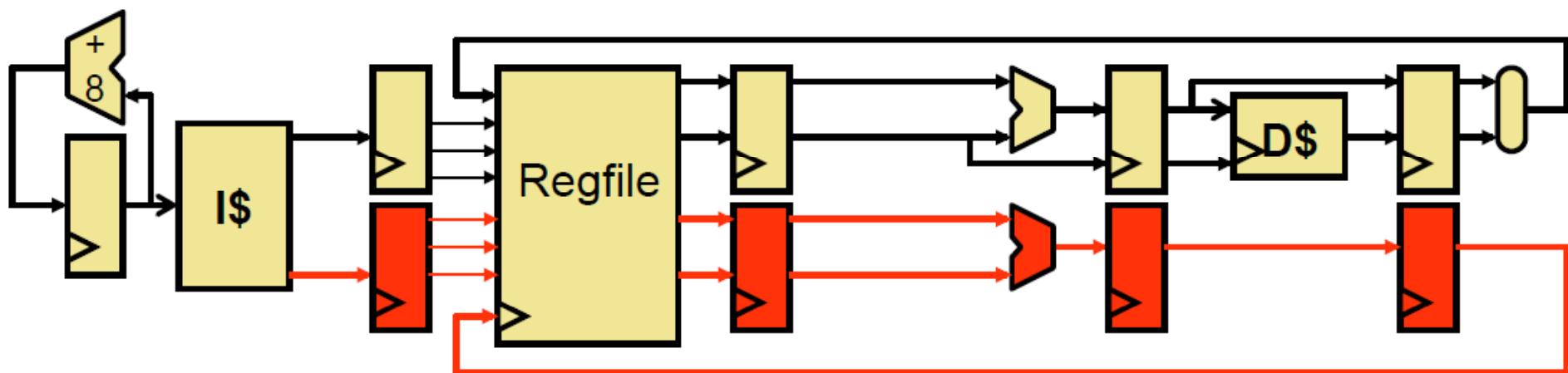
```
; if (r1==0) r3=r2
cmoveq r1, r2 -> r3
```

- + Avoids branch predictor
  - Avoids area penalty, misprediction penalty
- Avoids branch predictor
  - Introduces unnecessary nop if predictable branch
- GPUs also use predication



# Improving IPC

- IPC (instructions/cycle) bottlenecked at 1 instruction / clock
- Superscalar – increase pipeline width



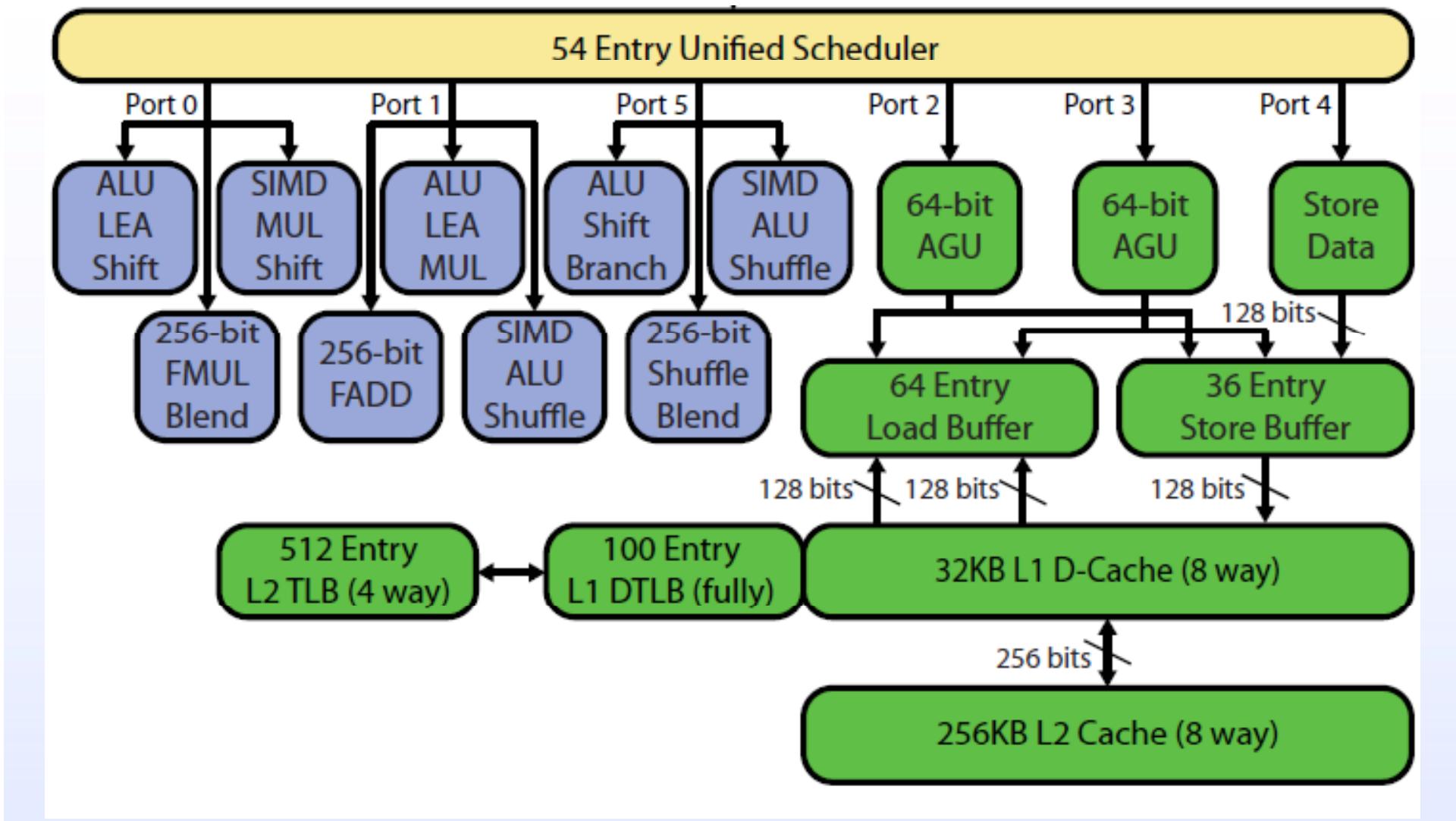


# Superscalar

- + Peak IPC now at  $N$  (for  $N$ -way superscalar)
  - o Branching and scheduling impede this
  - o Need some more tricks to get closer to peak (next)
- Area increase
  - o Doubling execution resources
  - o Bypass network grows at  $N^2$
  - o Need more register & memory bandwidth



# Superscalar in Sandy Bridge





# Scheduling

- Consider instructions:

**xor** r1,r2 → r3

**add** r3,r4 → r4

**sub** r5,r2 → r3

**addi** r3,1 → r1

- xor and add are dependent (Read-After-Write, RAW)
- sub and addi are dependent (RAW)
- xor and sub are *not* (Write-After-Write, WAW)



# Register Renaming

- How about this instead:

```
xor p1,p2 -> p6
```

```
add p6,p4 -> p7
```

```
sub p5,p2 -> p8
```

```
addi p8,1 -> p9
```

- xor and sub can now execute in parallel

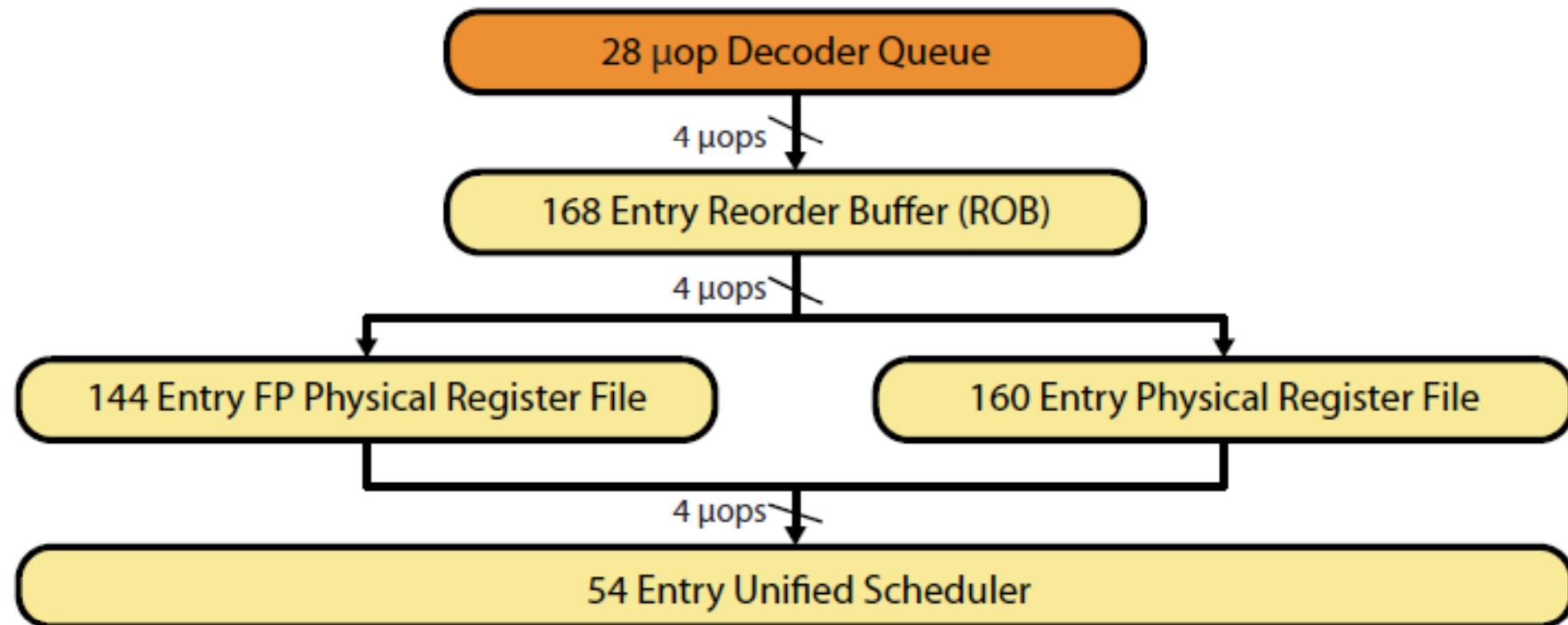


# Out-of-Order Execution

- Reordering instructions to maximize throughput
- Fetch → Decode → Rename → Dispatch → Issue → Register-Read → Execute → Memory → Writeback → Commit
- Reorder Buffer (ROB)
  - Keeps track of status for in-flight instructions
- Physical Register File (PRF)
- Issue Queue/Scheduler
  - Chooses next instruction(s) to execute



# OoO in Sandy Bridge





# Out-of-Order Execution

- + Brings IPC much closer to ideal
  - Area increase
  - Energy increase
- Modern Desktop/Mobile In-order CPUs
  - Intel Atom
  - ARM Cortex-A8 (Apple A4, TI OMAP 3)
  - Qualcomm Scorpion
- Modern Desktop/Mobile OoO CPUs
  - Intel Pentium Pro and onwards
  - ARM Cortex-A9 (Apple A5, NV Tegra 2/3, TI OMAP 4)
  - Qualcomm Krait



# Memory Hierarchy

- Memory: the larger it gets, the slower it gets
- Rough numbers:

|                   | Latency | Bandwidth | Size       |
|-------------------|---------|-----------|------------|
| SRAM (L1, L2, L3) | 1-2ns   | 200GBps   | 1-20MB     |
| DRAM (memory)     | 70ns    | 20GBps    | 1-20GB     |
| Flash (disk)      | 70-90μs | 200MBps   | 100-1000GB |
| HDD (disk)        | 10ms    | 1-150MBps | 500-3000GB |

SRAM & DRAM latency, and DRAM bandwidth for Sandy Bridge from [Lostcircuits](#)

Flash and HDD latencies from [AnandTech](#)

Flash and HDD bandwidth from AnandTech [Bench](#)

SRAM bandwidth guesstimated.



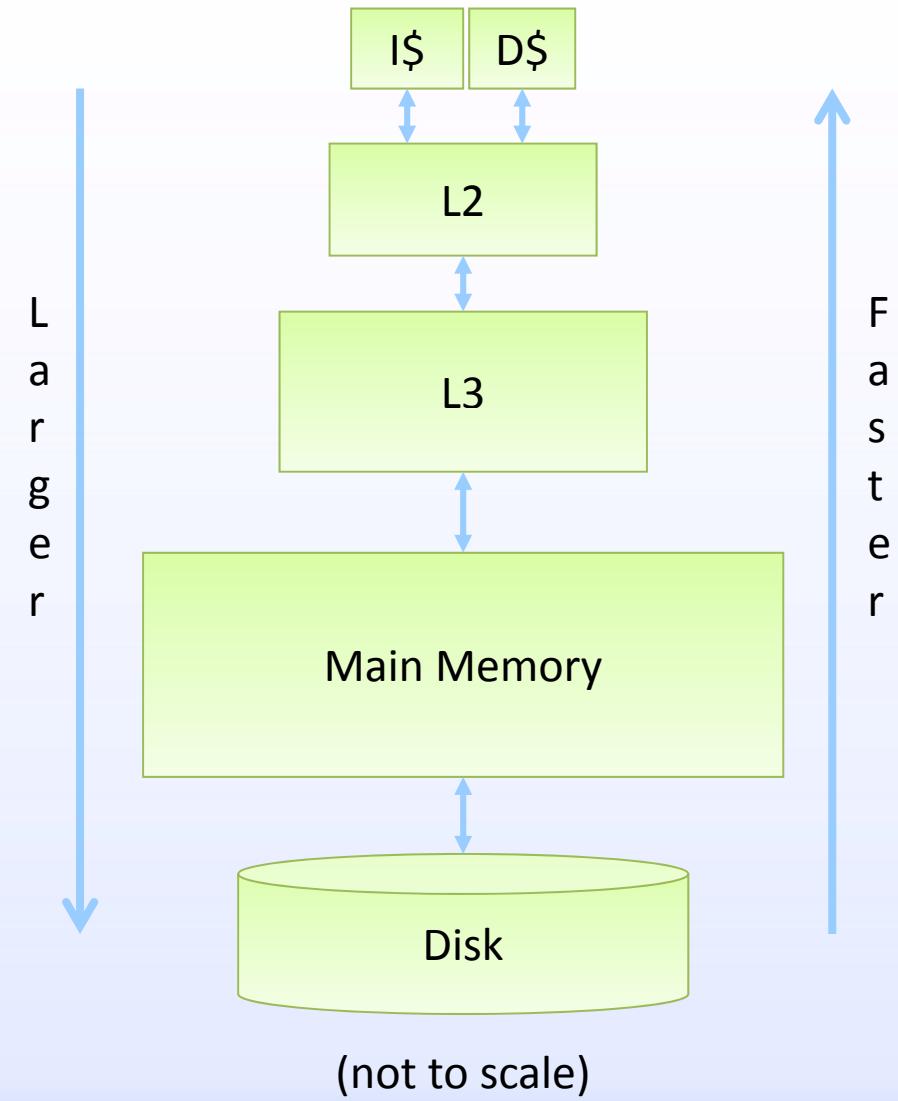
# Caching

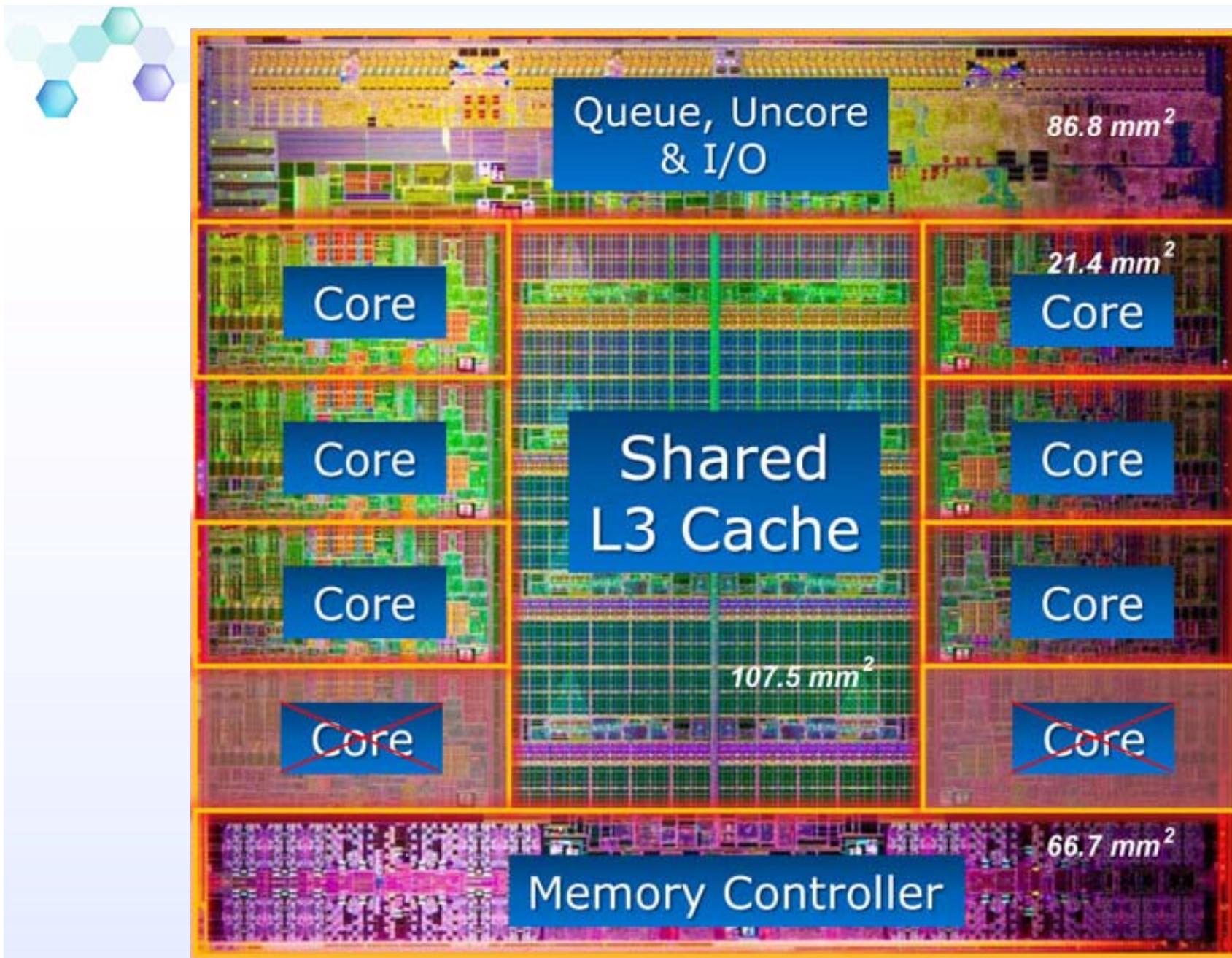
- Keep data you need close
- Exploit:
  - Temporal locality
    - Chunk just used likely to be used again soon
  - Spatial locality
    - Next chunk to use is likely close to previous



# Cache Hierarchy

- Hardware-managed
  - L1 Instruction/Data caches
  - L2 unified cache
  - L3 unified cache
- Software-managed
  - Main memory
  - Disk





Intel Core i7 3960X – 15MB L3 (25% of die). 4-channel Memory Controller, 51.2GB/s total



# Some Memory Hierarchy Design Choices

- Banking
  - Avoid multi-porting
- Coherency
- Memory Controller
  - Multiple channels for bandwidth



# Parallelism in the CPU

- Covered Instruction-Level (ILP) extraction
  - Superscalar
  - Out-of-order
- Data-Level Parallelism (DLP)
  - Vectors
- Thread-Level Parallelism (TLP)
  - Simultaneous Multithreading (SMT)
  - Multicore



# Vectors Motivation

```
for (int i = 0; i < N; i++)
 A[i] = B[i] + C[i];
```



# CPU Data-level Parallelism

- Single Instruction Multiple Data (SIMD)

- Let's make the execution unit (ALU) really wide
- Let's make the registers really wide too

```
for (int i = 0; i < N; i+= 4) {
 // in parallel
 A[i] = B[i] + C[i];
 A[i+1] = B[i+1] + C[i+1];
 A[i+2] = B[i+2] + C[i+2];
 A[i+3] = B[i+3] + C[i+3];
}
```



# Vector Operations in x86

- SSE2
  - 4-wide packed float and packed integer instructions
  - Intel Pentium 4 onwards
  - AMD Athlon 64 onwards
- AVX
  - 8-wide packed float and packed integer instructions
  - Intel Sandy Bridge
  - AMD Bulldozer



# Thread-Level Parallelism

- Thread Composition
  - Instruction streams
  - Private PC, registers, stack
  - Shared globals, heap
- Created and destroyed by programmer
- Scheduled by programmer or by OS



# Simultaneous Multithreading

- Instructions can be issued from multiple threads
- Requires partitioning of ROB, other buffers
  - + Minimal hardware duplication
  - + More scheduling freedom for OoO
  - Cache and execution resource contention can reduce single-threaded performance



## Multicore

- Replicate full pipeline
- Sandy Bridge-E: 6 cores
  - + Full cores, no resource sharing other than last-level cache
  - + Easier way to take advantage of Moore's Law
  - Utilization

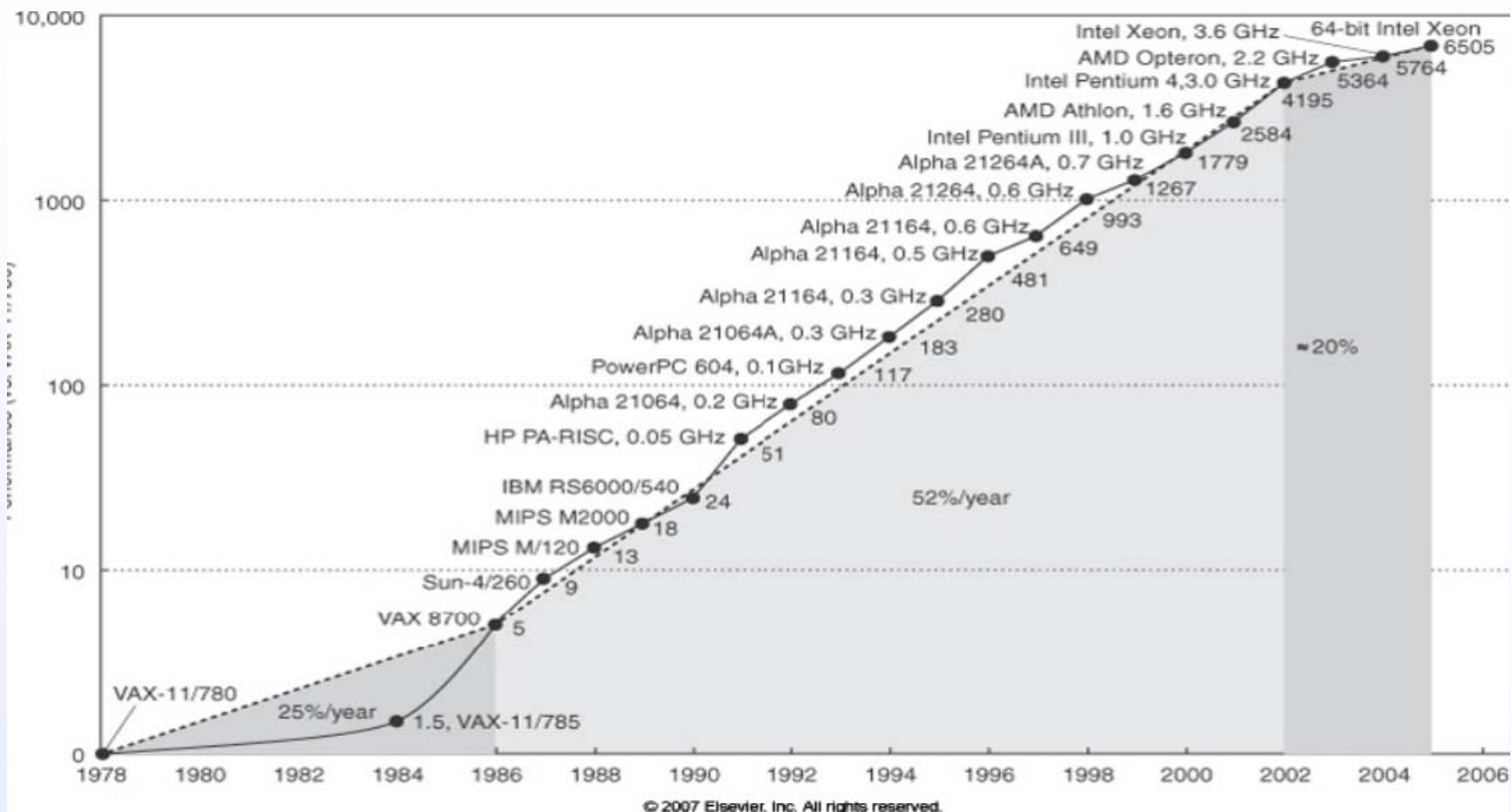


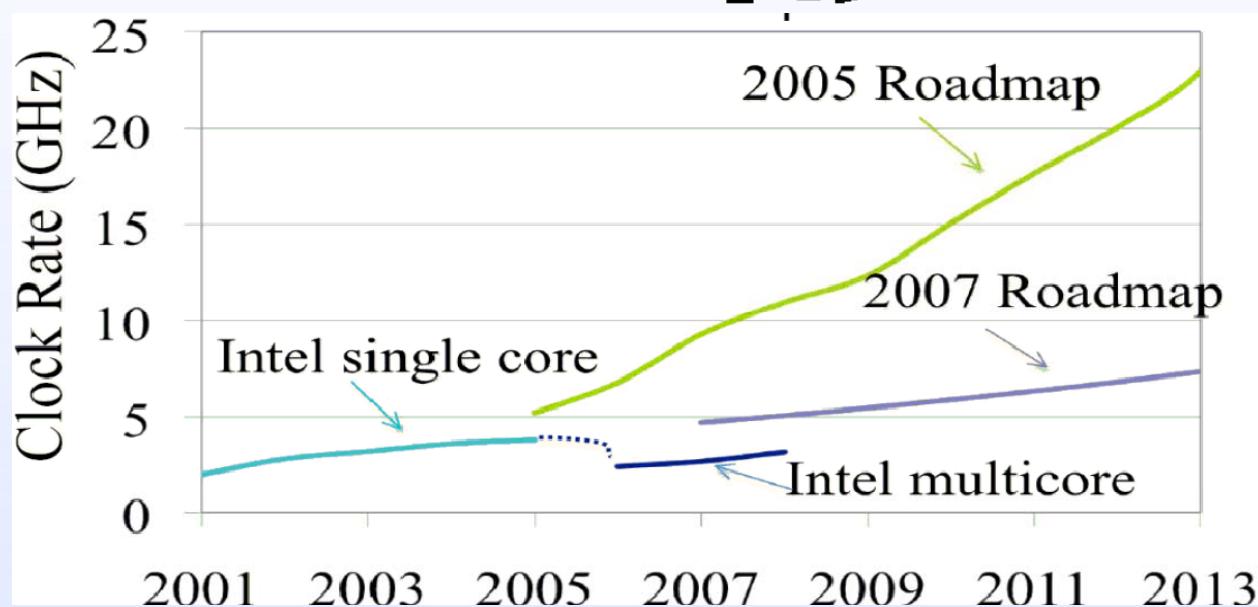
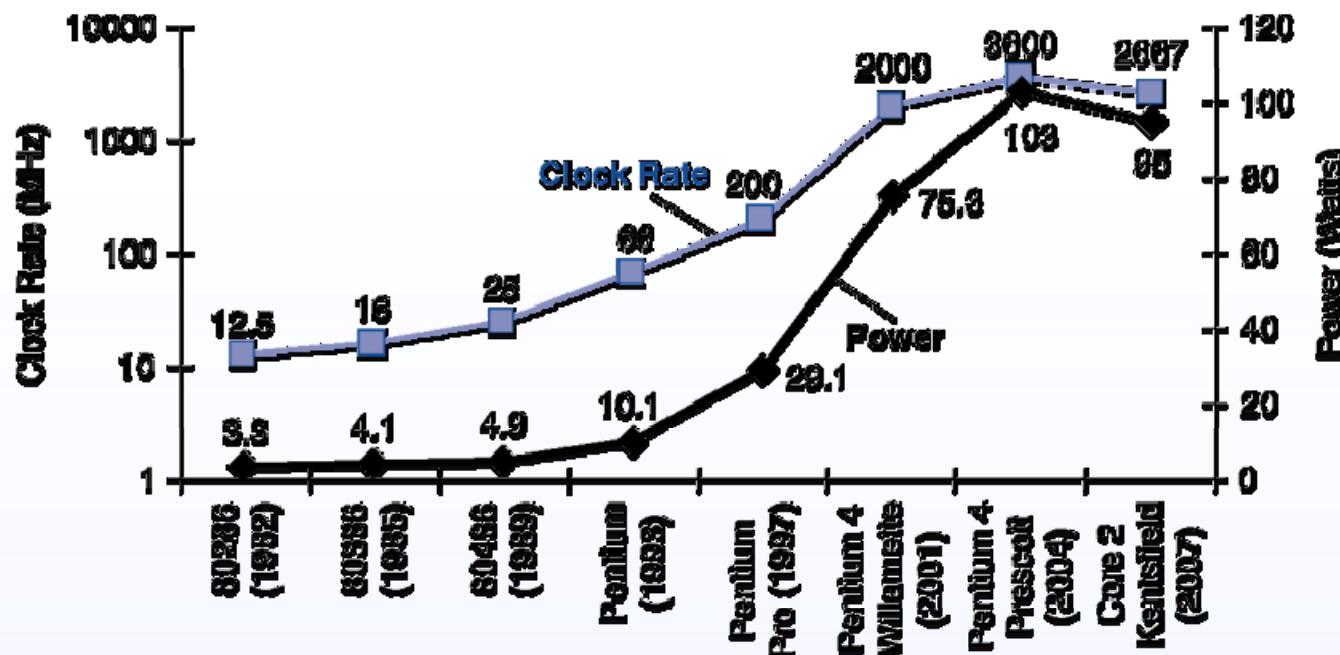
# Locks, Coherence, and Consistency

- **Problem:** multiple threads reading/writing to same data
- A solution: Locks
  - Implement with test-and-set, load-link/store-conditional instructions
- **Problem:** Who has the correct data?
- A solution: cache coherency protocol
- **Problem:** What is the correct data?
- A solution: memory consistency model



# Difficulty : Power Wall

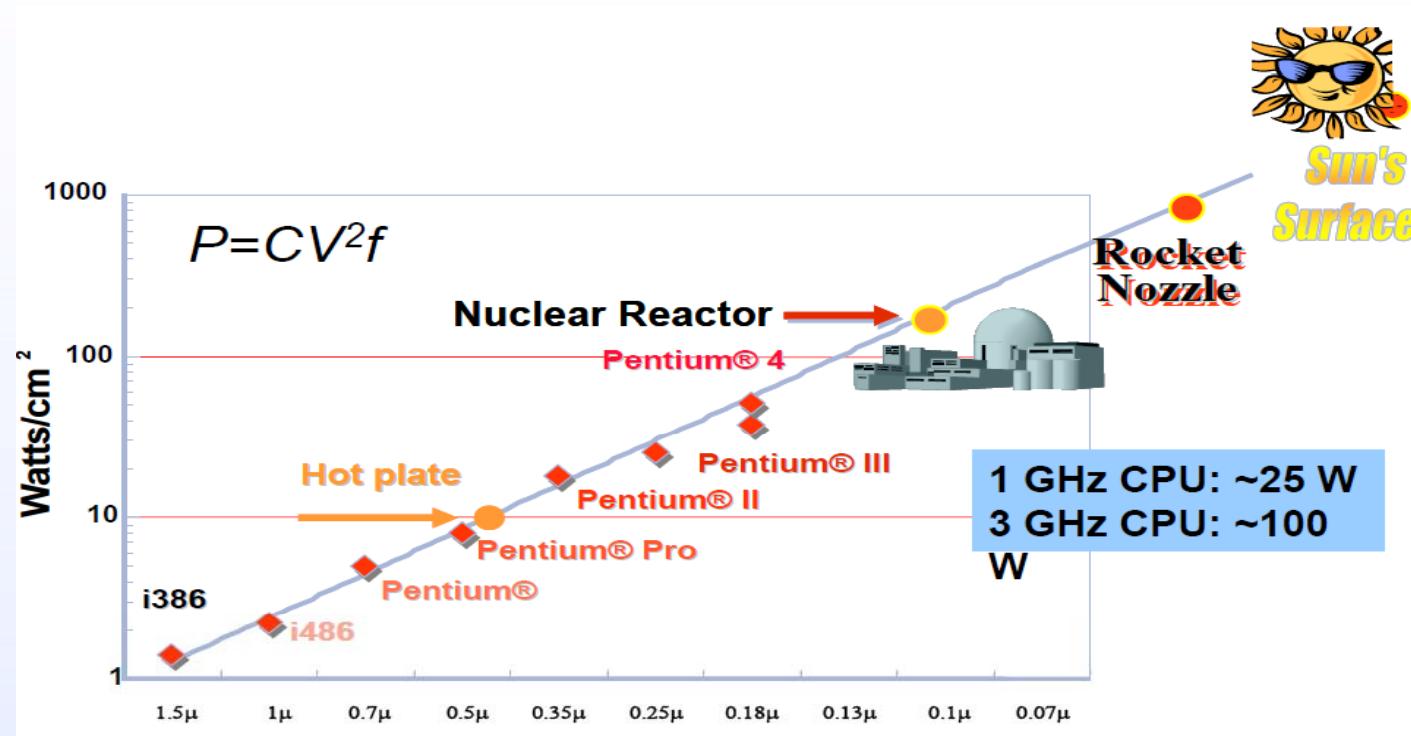






# New Era of Computing: Parallel Computing

- General purpose single processor hits the power wall
  - Clock frequency (perf/clock) no longer increases linearly



The free lunch is over.



## New Moore's Law

- CPUs becomes fatter and fatter , with more cores,
- Rather than faster...

| CPU Type            | CORES/Freq. | CPU Type           | CORES/Freq. |
|---------------------|-------------|--------------------|-------------|
| Intel i7 3960X      | 6/3.3GHz    | AMD FX-8150        | 8/3.6GHz    |
| Intel Xeon E5-2687W | 8/3.1GHz    | AMD Opteron 6282SE | 16/2.6GHz   |

- Hit Another Wall: Memory Wall
- growing disparity of speed between CPU and memory outside the CPU chip.





# Conclusions

- CPU optimized for sequential programming
  - Pipelines, branch prediction, superscalar, OoO
  - Reduce execution time with high clock speeds and high utilization
- Slow memory is a constant problem
- Parallelism
  - Sandy Bridge-E great for 6-12 active threads
  - How about 12,000?



## Additional Slides

# CUDA/GPU Programming (1)

Bin ZHOU  
Jan. 2015

# Acknowledgement

---

- ▶ Some Slides are from David Kirk Wen-mei Hwu's UIUC GPU Course
- ▶ Some Slides are from Patrick Cozzi University of Pennsylvania CIS 565



# GPU Architecture Review

---

- ▶ GPUs are specialized for
  - ▶ Compute-intensive, highly parallel computation
  - ▶ Graphics!
- ▶ Transistors are devoted to:
  - ▶ Processing
  - ▶ Not:
    - ▶ Data caching
    - ▶ Flow control



# GPU Architecture Review

---

## Transistor Usage



Figure 1-2. The GPU Devotes More Transistors to Data Processing



Image from: [http://developer.download.nvidia.com/compute/cuda/3\\_2\\_prod/toolkit/docs/CUDA\\_C\\_Programming\\_Guide.pdf](http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf)

---

Let's CUDA  
Now



# GPU Computing History

---

- ▶ 2001/2002 - researchers see GPU as data-parallel coprocessor
  - ▶ The *GPGPU* field is born
- ▶ 2007 - NVIDIA releases CUDA
  - ▶ *CUDA* - Compute Uniform Device Architecture
  - ▶ GPGPU shifts to *GPU Computing*
- ▶ 2008 - Khronos releases *OpenCL* specification
- ▶ 2013 - Khronos releases *OpenGL* compute shaders



# CUDA Abstractions

---

- ▶ A hierarchy of thread groups
- ▶ Shared memories
- ▶ Barrier synchronization



# CUDA Terminology

---

- ▶ *Host* - typically the CPU
  - ▶ Code written in ANSI C
- ▶ *Device* - typically the GPU (data-parallel)
  - ▶ Code written in *extended* ANSI C
- ▶ Host and device have separate memories
- ▶ CUDA Program
  - ▶ Contains both host and device code



# CUDA Terminology

---

- ▶ *Kernel* - data-parallel function
  - ▶ Invoking a kernel creates lightweight threads on the device
  - ▶ Threads are generated and scheduled with hardware
- Similar to a *shader* in OpenGL?



# CUDA Kernels

- ▶ Executed  $N$  times in parallel by  $N$  different *CUDA threads*

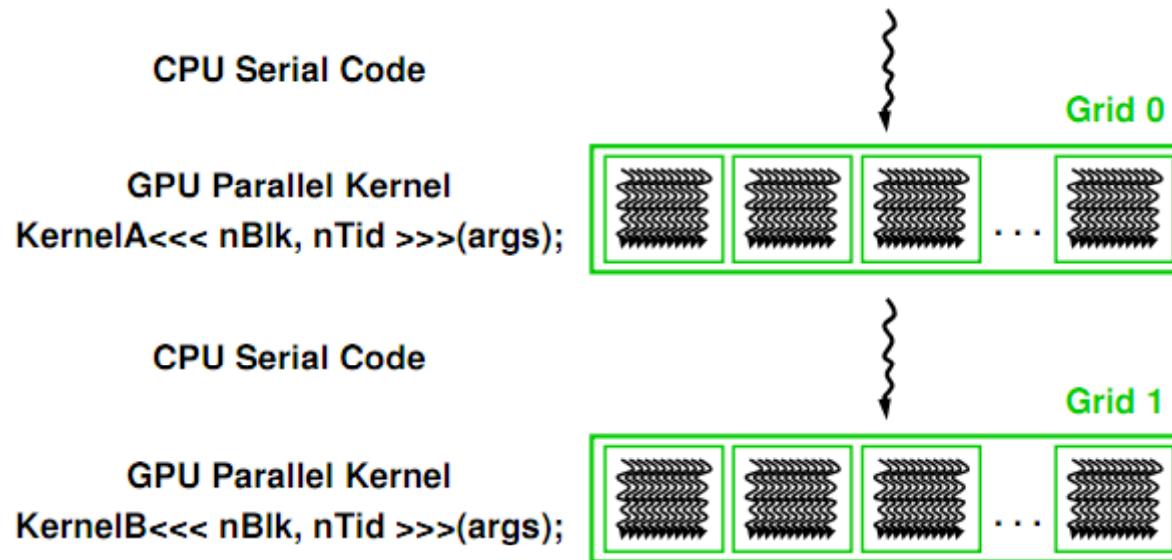
The diagram illustrates the components of a CUDA kernel. A red box labeled "Declaration Specifier" points to the `__global__` keyword in the kernel definition. Another red box labeled "Thread ID" points to the `threadIdx.x` variable used to index array `C`. A third red box labeled "Execution Configuration" points to the `<<<1, N>>>` parameters in the kernel invocation.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
 int i = threadIdx.x;
 C[i] = A[i] + B[i];
}

int main()
{
 ...
 // Kernel invocation with N threads
 VecAdd<<<1, N>>>(A, B, C);
}
```



# CUDA Program Execution



# Thread Hierarchies

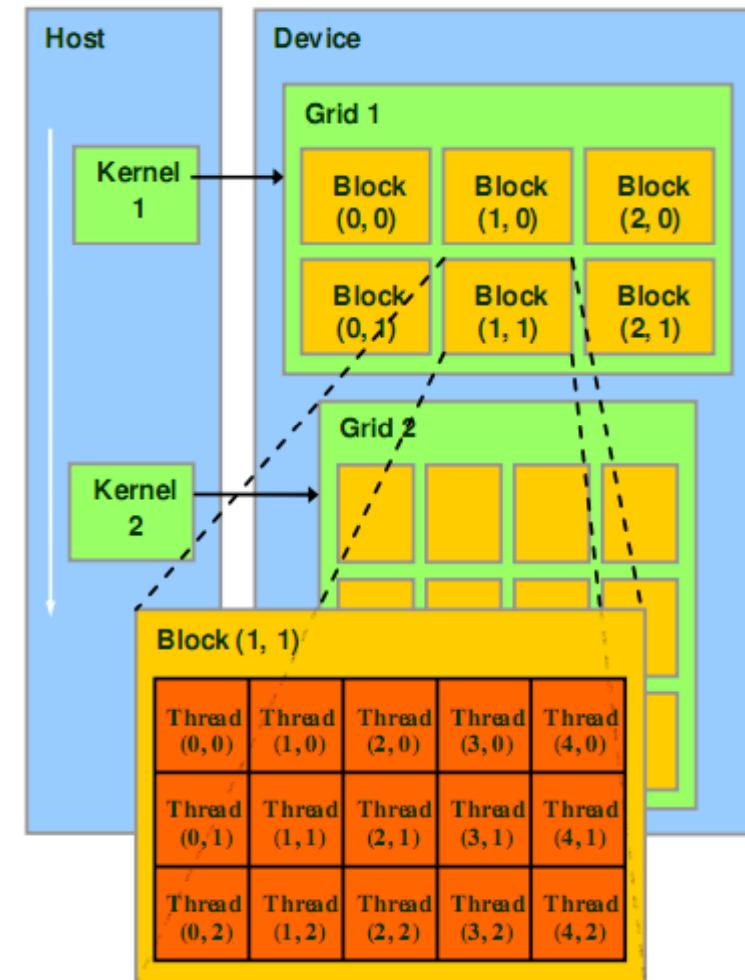
---

- ▶ *Grid* - one or more thread blocks
  - ▶ 1D or 2D
- ▶ *Block* - array of threads
  - ▶ 1D, 2D, or 3D
  - ▶ Each block in a grid has the same number of threads
  - ▶ Each thread in a block can
    - ▶ Synchronize
    - ▶ Access shared memory



# Thread Hierarchies Recall

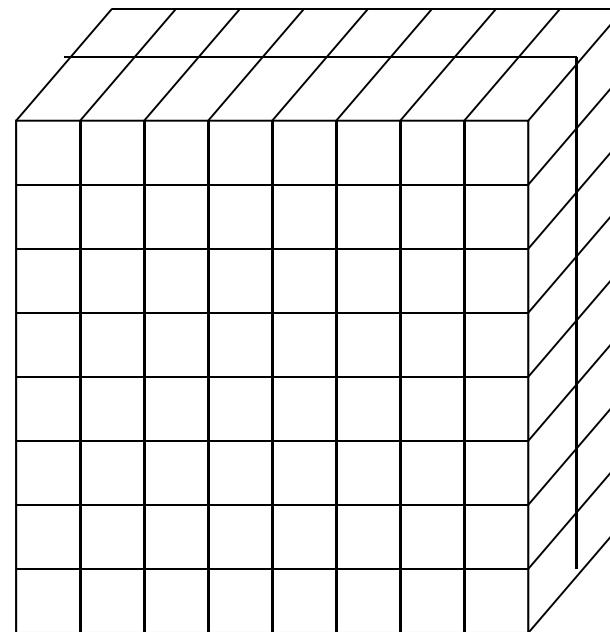
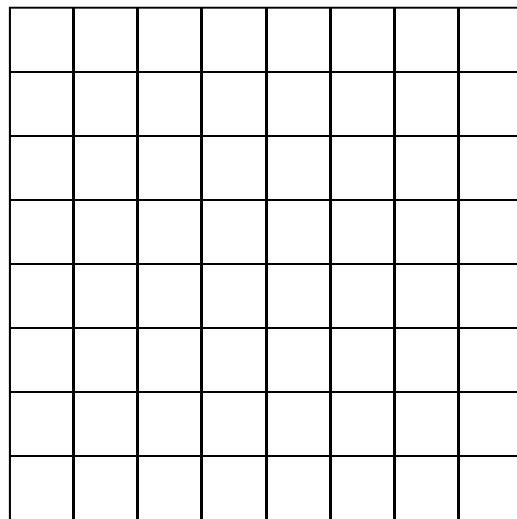
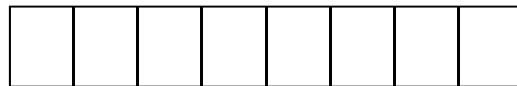
- A **thread block** is a batch of threads that can **cooperate** with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



# Thread Hierarchies

---

- ▶ *Block* - 1D, 2D, or 3D
  - ▶ Example: Index into vector, matrix, volume



# Thread Hierarchies

---

- ▶ *Thread ID*: Scalar thread identifier
- ▶ Thread Index: `threadIdx`
  
- ▶ 1D: Thread ID == Thread Index
- ▶ 2D with size  $(D_x, D_y)$ 
  - ▶ Thread ID of index  $(x, y) == x + y D_x$
- ▶ 3D with size  $(D_x, D_y, D_z)$ 
  - ▶ Thread ID of index  $(x, y, z) == x + y D_x + z D_x D_y$



# Thread Hierarchies

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
 float C[N][N])
{
 int i = threadIdx.x;
 int j = threadIdx.y;
 C[i][j] = A[i][j] + B[i][j];
}

int main()
{
 ...
 // Kernel invocation with one block of N * N * 1 threads
 int numBlocks = 1;
 dim3 threadsPerBlock(N, N);
 MatAdd<<numBlocks, threadsPerBlock>>(A, B, C);
}
```

1 Thread Block

2D Index

2D Block



# Thread Hierarchies

---

- ▶ Thread Block
  - ▶ Group of threads
    - ▶ G80 and GT200: Up to 512 threads
    - ▶ Fermi and Kepler: Up to 1024 threads
    - ▶ Maxwell: Up to 1024 threads
  - ▶ Reside on same processor core
  - ▶ Share memory of that core( SM, SMX, SMM...)



# Thread Hierarchies

- ▶ Thread Block
  - ▶ Group of threads
    - ▶ G80 and GT200: Up to 512 threads
    - ▶ Fermi and Kepler: Up to 1024 threads
  - ▶ Reside on same processor core
    - ▶ Share memory of that core

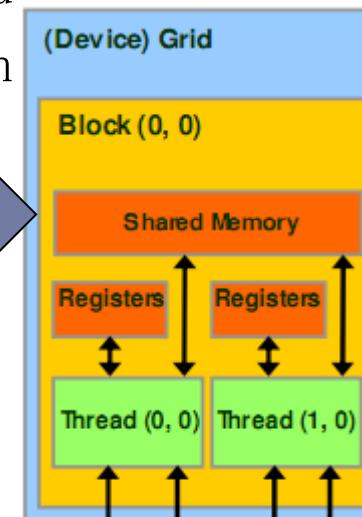


Image from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

# Thread Hierarchies

---

- ▶ Block Index: `blockIdx`
- ▶ Dimension: `blockDim`
  - ▶ 1D or 2D



# Thread Hierarchies

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
 float C[N][N])
{
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 int j = blockIdx.y * blockDim.y + threadIdx.y;
 if (i < N && j < N)
 C[i][j] = A[i][j] + B[i][j];
}

int main()
{
 ...
 // Kernel invocation
 dim3 threadsPerBlock(16, 16);
 dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
 MatAdd<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

16x16  
Threads per block

2D Thread Block



# Thread Hierarchies

---

- ▶ Example:  $N = 32$ 
  - ▶ 16x16 threads per block (independent of  $N$ )
    - ▶ `threadIdx` ( $[0, 15], [0, 15]$ )
  - ▶ 2x2 thread blocks in grid
    - ▶ `blockIdx` ( $[0, 1], [0, 1]$ )
    - ▶ `blockDim` = 16

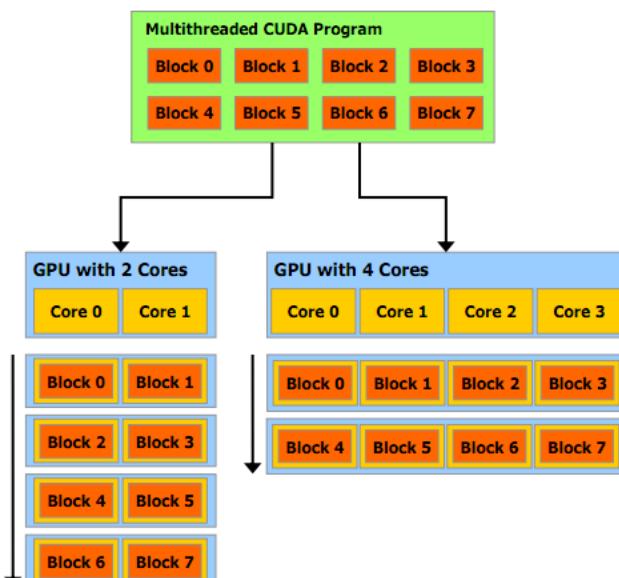
```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

- $i = [0, 1] * 16 + [0, 15]$



# Thread Hierarchies

- ▶ Blocks execute **independently**
  - ▶ In any order: parallel or series
  - ▶ Scheduled in any order by any number of cores
    - ▶ Allows code to scale with core count

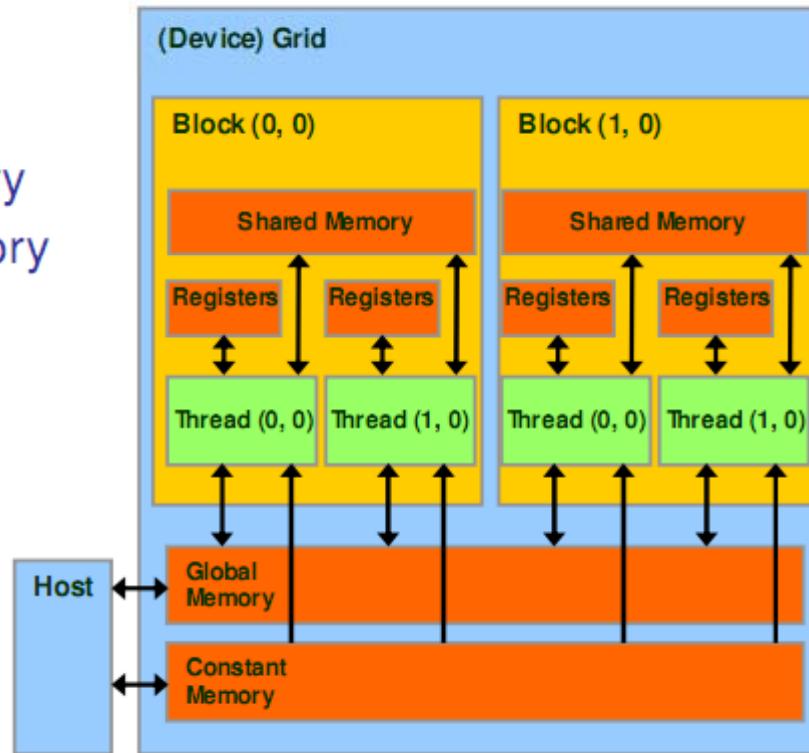


A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4. Automatic Scalability

# CUDA Memory Transfers

- Device code 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
- Host code can
  - R/W per grid global and constant memories



# CUDA Memory Transfers

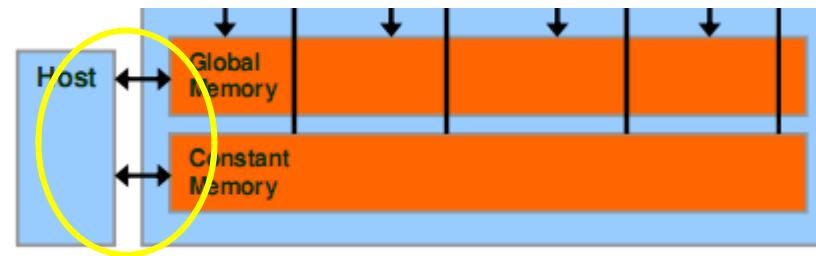
---

- ▶ Host can transfer to/from device

- ▶ *Global* memory
- ▶ *Constant* memory

- ▶ Note:

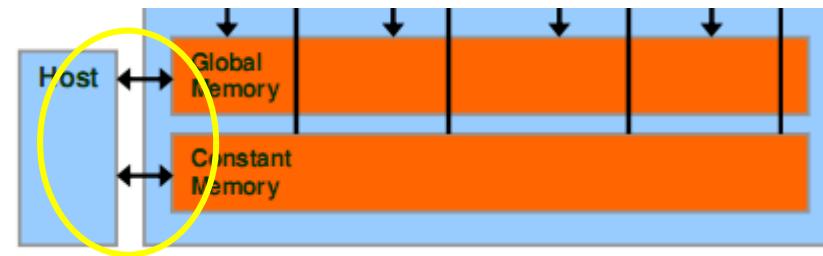
Constant memory is  
constant for GPU code, not CPU host;



# CUDA Memory Transfers

---

- ▶ **cudaMalloc( )**
  - ▶ Allocate global memory on device
- ▶ **cudaFree( )**
  - ▶ Frees memory



- ▶ Many other memory APIs:
- ▶ `cudaMemcpyToSymbol()`
- ▶ `cudaMemcpyFromSymbol()`
- ▶ `cudaMemset()`
- ▶ `cudaMemcpyPeer()`
- ▶ ...



Image from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

# CUDA Memory Transfers

---

```
float *Md
int size = Width * Width * sizeof(float);

cudaMalloc((void**) &Md, size);
...
cudaFree (Md) ;
```



Code from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

# CUDA Memory Transfers

```
float *Md
int size = Width * Width * sizeof(float);

cudaMalloc((void**)&Md, size);
...
cudaFree (Md);
```

Diagram annotations:

- A yellow box highlights the expression `(void**)&Md`.
- A yellow box highlights the variable `size`.
- A red box labeled "Size in bytes" has an arrow pointing to the `size` variable.
- A red box labeled "Pointer to device memory" has an arrow pointing to the `(void**)&Md` expression.

# CUDA Memory Transfers

---

- ▶ **cudaMemcpy( )**
  - ▶ Memory transfer
    - ▶ Host to host
    - ▶ Host to device
    - ▶ Device to host
    - ▶ Device to device



# CUDA Memory Transfers

---

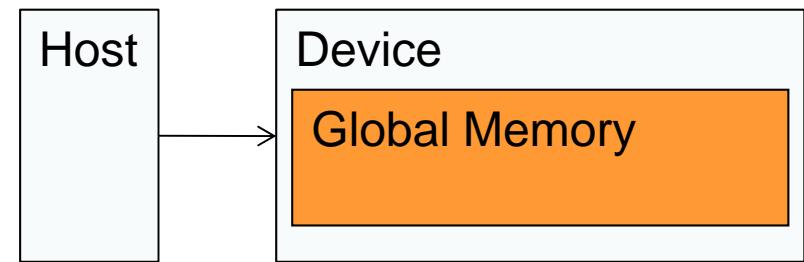
- ▶ **cudaMemcpy( )**
- ▶ Memory transfer
  - ▶ Host to host
  - ▶ Host to device
  - ▶ Device to host
  - ▶ Device to device



# CUDA Memory Transfers

---

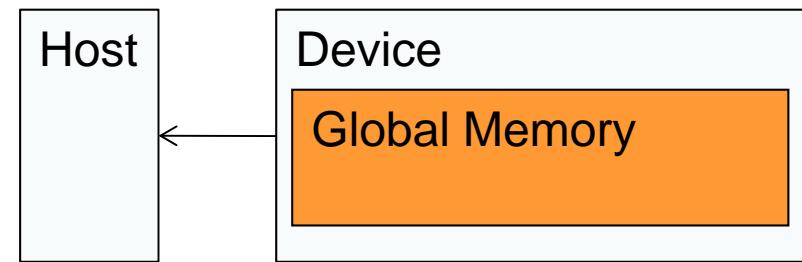
- ▶ **cudaMemcpy( )**
  - ▶ Memory transfer
    - ▶ Host to host
    - ▶ Host to device
    - ▶ Device to host
    - ▶ Device to device



# CUDA Memory Transfers

---

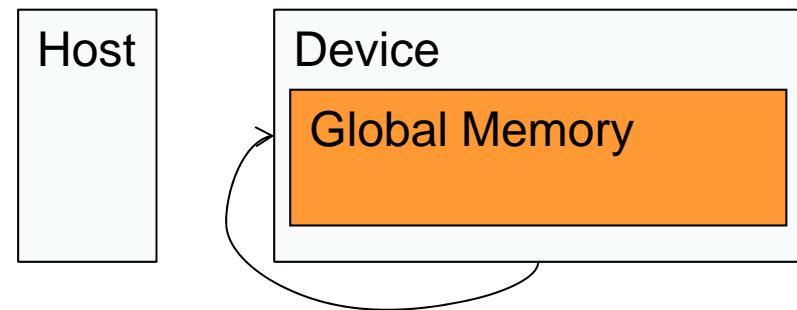
- ▶ **cudaMemcpy( )**
  - ▶ Memory transfer
    - ▶ Host to host
    - ▶ Host to device
    - ▶ Device to host
    - ▶ Device to device



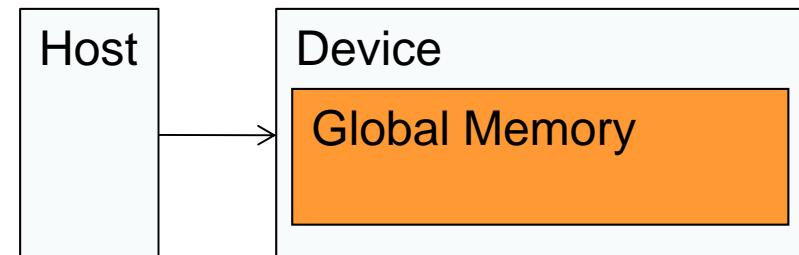
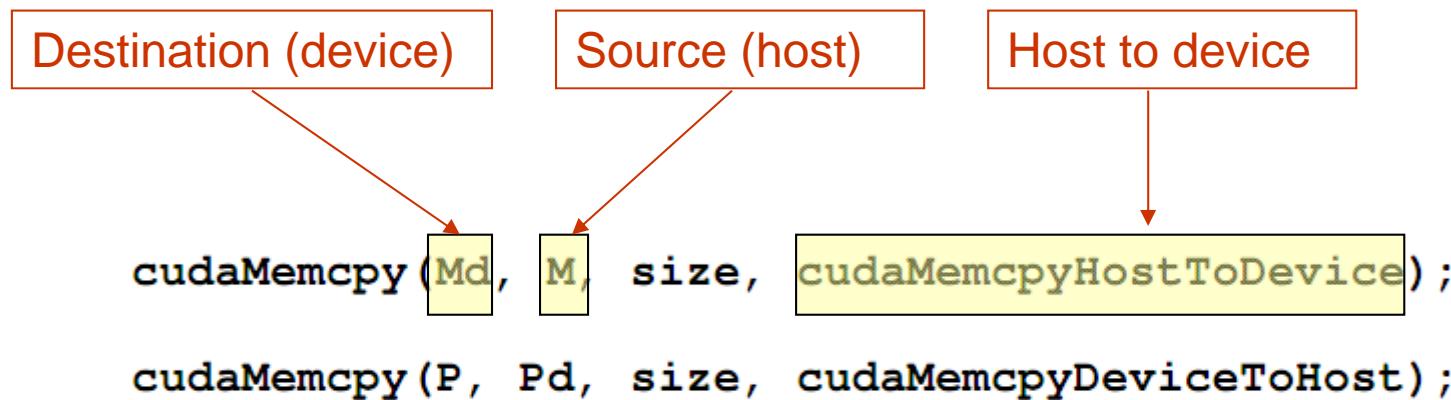
# CUDA Memory Transfers

---

- ▶ **cudaMemcpy( )**
  - ▶ Memory transfer
    - ▶ Host to host
    - ▶ Host to device
    - ▶ Device to host
    - ▶ Device to device



# CUDA Memory Transfers



# CUDA Memory Transfers

---

```
cudaMemcpy (Md, M, size, cudaMemcpyHostToDevice) ;
cudaMemcpy (P, Pd, size, cudaMemcpyDeviceToHost) ;
```



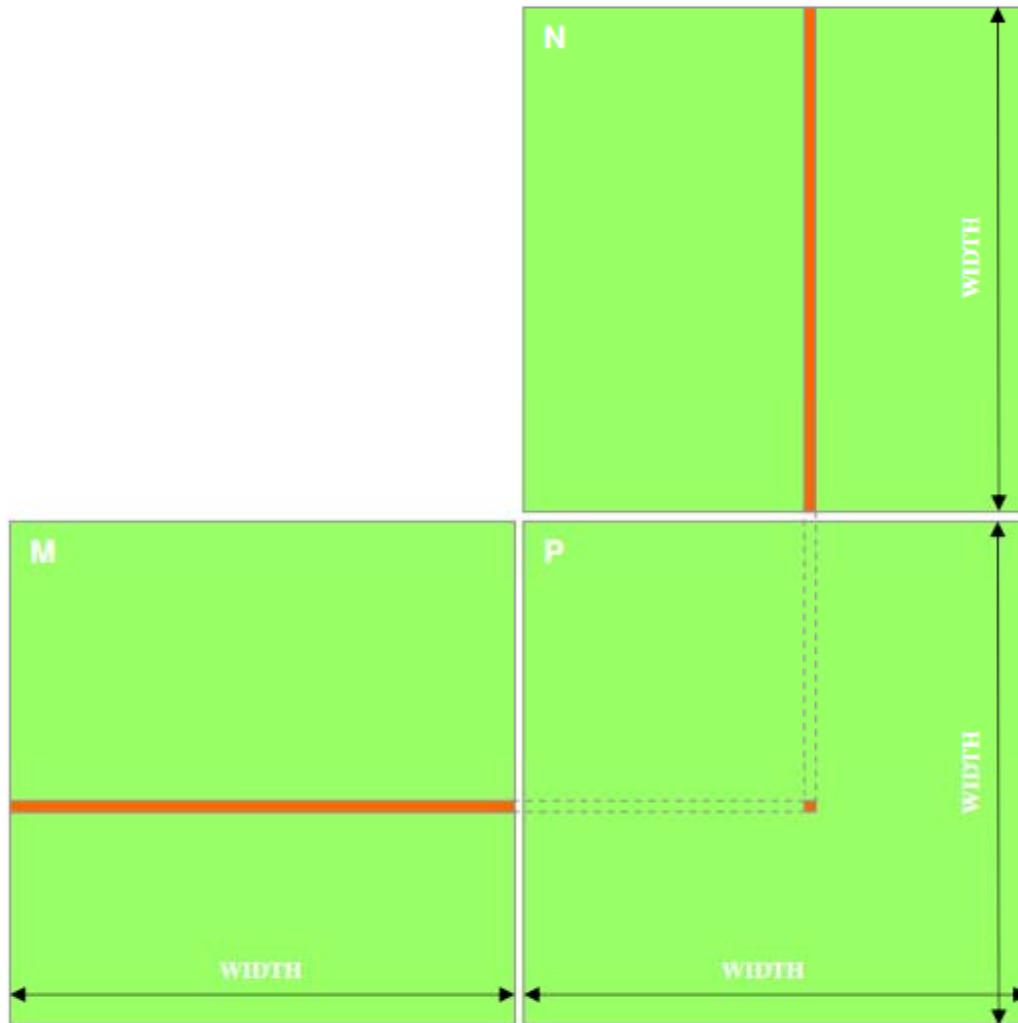
# Matrix Multiply Reminder

---

- ▶ Vectors
- ▶ Dot products
- ▶ Row major or column major?
- ▶ Dot product per output element



# Matrix Multiply



- $P = M * N$
- Assume M and N are square for simplicity

# Matrix Multiply

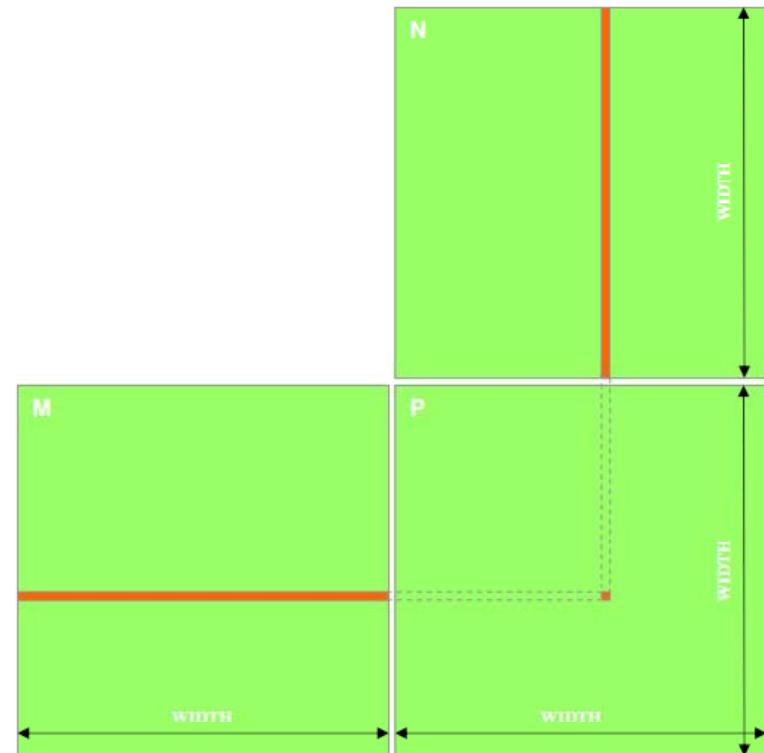
---

- $1,000 \times 1,000$  matrix
  - 1,000,000 dot products
    - Each 1,000 multiples and 1,000 adds



# Matrix Multiply: CPU Implementation

```
void MatrixMulOnHost(float* M, float* N, float* P, int width)
{
 for (int i = 0; i < width; ++i)
 for (int j = 0; j < width; ++j)
 {
 float sum = 0;
 for (int k = 0; k < width; ++k)
 {
 float a = M[i * width + k];
 float b = N[k * width + j];
 sum += a * b;
 }
 P[i * width + j] = sum;
 }
}
```



# Matrix Multiply: CUDA Skeleton

---

```
int main(void) {
 1. // Allocate and initialize the matrices M, N, P
 // I/O to read the input matrices M and N

 2. // M * N on the device
 MatrixMulOnDevice(M, N, P, width);
 ...
 3. // I/O to write the output matrix P
 // Free matrices M, N, P
 ...
 return 0;
}
```



# Matrix Multiply: CUDA Skeleton

---

```
int main(void) {
 1. // Allocate and initialize the matrices M, N, P
 // I/O to read the input matrices M and N

 2. // M * N on the device
 MatrixMulOnDevice(M, N, P, width);
 ...
 3. // I/O to write the output matrix P
 // Free matrices M, N, P
 ...
 return 0;
}
```



# Matrix Multiply: CUDA Skeleton

---

```
int main(void) {
 1. // Allocate and initialize the matrices M, N, P
 // I/O to read the input matrices M and N

 2. // M * N on the device
 MatrixMulOnDevice(M, N, P, width);
 ...
 3. // I/O to write the output matrix P
 // Free matrices M, N, P
 ...
 return 0;
}
```



# Matrix Multiply

---

- ▶ Step 1
  - ▶ Add *CUDA memory transfers* to the skeleton



# Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
 int size = Width * Width * sizeof(float);

 1. // Load M and N to device memory
 cudaMalloc(Md, size);
 cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
 cudaMalloc(Nd, size);
 cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

 // Allocate P on the device
 cudaMalloc(Pd, size);

 2. // Kernel invocation code – to be shown later
 ...
 3. // Read P from the device
 cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
 // Free device matrices
 cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

Allocate input



# Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
 int size = Width * Width * sizeof(float);

 1. // Load M and N to device memory
 cudaMalloc(Md, size);
 cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
 cudaMalloc(Nd, size);
 cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

 // Allocate P on the device ← Allocate output
 cudaMalloc(Pd, size);

 2. // Kernel invocation code – to be shown later
 ...
 3. // Read P from the device
 cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
 // Free device matrices
 cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```



# Matrix Multiply: Data Transfer

---

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
 int size = Width * Width * sizeof(float);

 1. // Load M and N to device memory
 cudaMalloc(Md, size);
 cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
 cudaMalloc(Nd, size);
 cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

 // Allocate P on the device
 cudaMalloc(Pd, size);

 2. // Kernel invocation code – to be shown later
 ...
 3. // Read P from the device
 cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
 // Free device matrices
 cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```



# Matrix Multiply: Data Transfer

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
 int size = Width * Width * sizeof(float);

 1. // Load M and N to device memory
 cudaMalloc(Md, size);
 cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
 cudaMalloc(Nd, size);
 cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

 // Allocate P on the device
 cudaMalloc(Pd, size);

 2. // Kernel invocation code – to be shown later

 ...
 3. // Read P from the device
 cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
 // Free device matrices
 cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

Read back  
from device

# Matrix Multiply

---

- ▶ Step 2
  - ▶ Implement the *kernel* in CUDA C



# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y; ← Accessing a matrix, so using a 2D block

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0;

 for (int k = 0; k < Width; ++k)
 {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0; // Each kernel computes one output
 for (int k = 0; k < Width; ++k)
 {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

Code from: <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0;

 for (int k = 0; k < Width; ++k) {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

Where did the two outer for loops  
in the CPU implementation go?



# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0;

 for (int k = 0; k < Width; ++k)
 {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

No locks or synchronization, why?

# Matrix Multiply

---

- ▶ Step 3
  - ▶ Invoke the *kernel* in CUDA C



# Matrix Multiply: Invoke Kernel

---

```
// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);
```

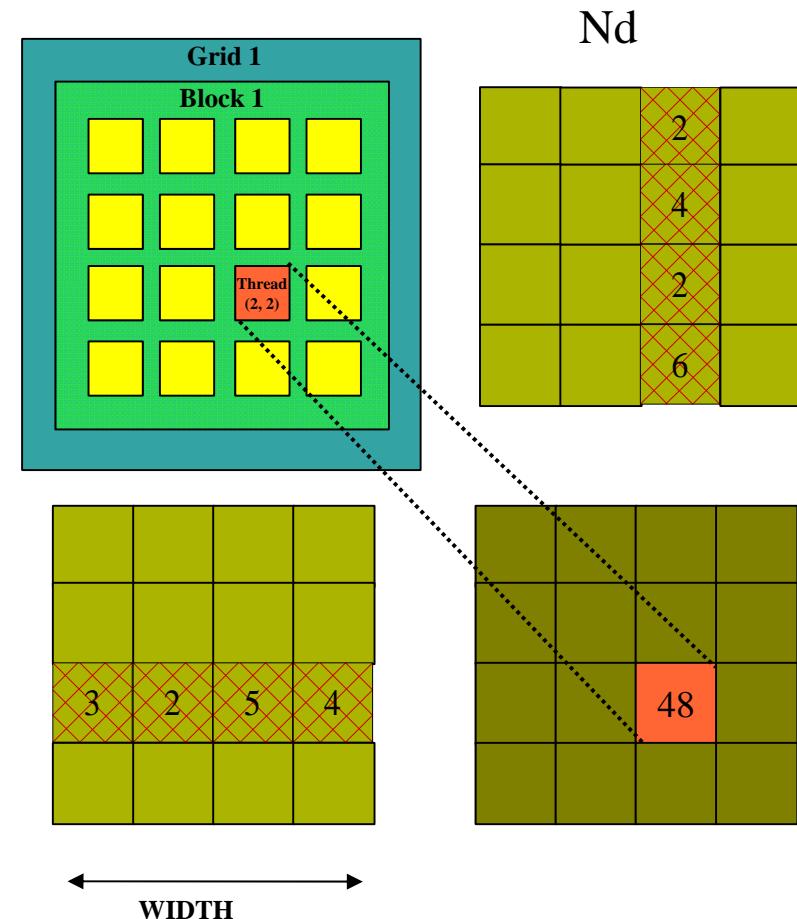
One block with width  
by width threads

```
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);
```



# Matrix Multiply

- One Block of threads compute matrix  $P_d$ 
  - Each thread computes one element of  $P_d$
- Each thread
  - Loads a row of matrix  $M_d$
  - Loads a column of matrix  $N_d$
  - Perform one multiply and addition for each pair of  $M_d$  and  $N_d$  elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



# Matrix Multiply

---

- ▶ What is the major performance problem with our implementation?
- ▶ What is the major limitation?



# CUDA/GPU Programming (2)

Bin ZHOU  
Jan. 2015

# Acknowledgement

---

- ▶ Some Slides are from David Kirk Wen-mei Hwu's UIUC GPU Course
- ▶ Some Slides are from Patrick Cozzi University of Pennsylvania CIS 565



# Contents

---

- ▶ Built-ins and functions
- ▶ Memory model Recall
- ▶ Synchronizing threads
- ▶ Matrix multiply revisited

# Functional Declarations

|                                            | <b>Executed on the:</b> | <b>Only callable from the:</b> |
|--------------------------------------------|-------------------------|--------------------------------|
| <code>__global__ void KernelFunc()</code>  | device                  | host                           |
| <code>__device__ float DeviceFunc()</code> | device                  | device                         |
| <code>__host__ float HostFunc()</code>     | host                    | host                           |

# Functional Declarations

---

- **\_global\_**

- Must return **void**

- **\_device\_**

- Used to be Inlined by default
  - Think as it is.

# Functional Declarations

---

## ■ What do these do?

- `__global__ __host__ void func()`
- `__device__ __host__ void func()`

# Functional Declarations

---

## ■ What do these do?

- **`__global__ __host__ void func()`**
- **`__device__ __host__ void func()`**

```
__host__ __device__ func()
{
 #if __CUDA_ARCH__ == 100
 // Device code path for compute capability 1.0
 #elif __CUDA_ARCH__ == 200
 // Device code path for compute capability 2.0
 #elif !defined(__CUDA_ARCH__)
 // Host code path
 #endif
}
```

# Functional Declarations

---

- Global and device functions
    - Less recursion (ok with Fermi, kepler, Maxwell)
    - OK with malloc() but be cautious
    - Careful with function calls through pointers
  - Consider thousands of threads following the same
- 
- ▶<sup>8</sup> ■ Instruction stream

# Vector Types

---

- ▶ `char[1-4], uchar[1-4]`
- ▶ `short[1-4], ushort[1-4]`
- ▶ `int[1-4], uint[1-4]`
- ▶ `long[1-4], ulong[1-4]`
- ▶ `longlong[1-4], ulonglong[1-4]`
- ▶ `float[1-4]`
- ▶ `double1, double2`

## Vector Types

---

- ▶ Available in host and device code
- ▶ Construct with `make_<type name>`

```
int2 i2 = make_int2(1, 2);
float4 f4 = make_float4(
 1.0f, 2.0f, 3.0f, 4.0f);
```

## Vector Types

---

- ▶ Access with `.x`, `.y`, `.z`, and `.w`

```
int2 i2 = make_int2(1, 2);
int x = i2.x;
int y = i2.y;
```

- **No `.r`, `.g`, `.b`, `.a`, etc. like GLSL**

# Math Functions

---

- ▶ **double** and **float** overloads
  - ▶ No vector overloads
- ▶ On the host, functions use the C runtime implementation if available

# Math Functions

---

- ▶ Partial list:
  - ▶ `sqrt`, `rsqrt`
  - ▶ `exp`, `log`
  - ▶ `sin`, `cos`, `tan`, `sincos`
  - ▶ `asin`, `acos`, `atan2`
  - ▶ `trunc`, `ceil`, `floor`

# Math Functions

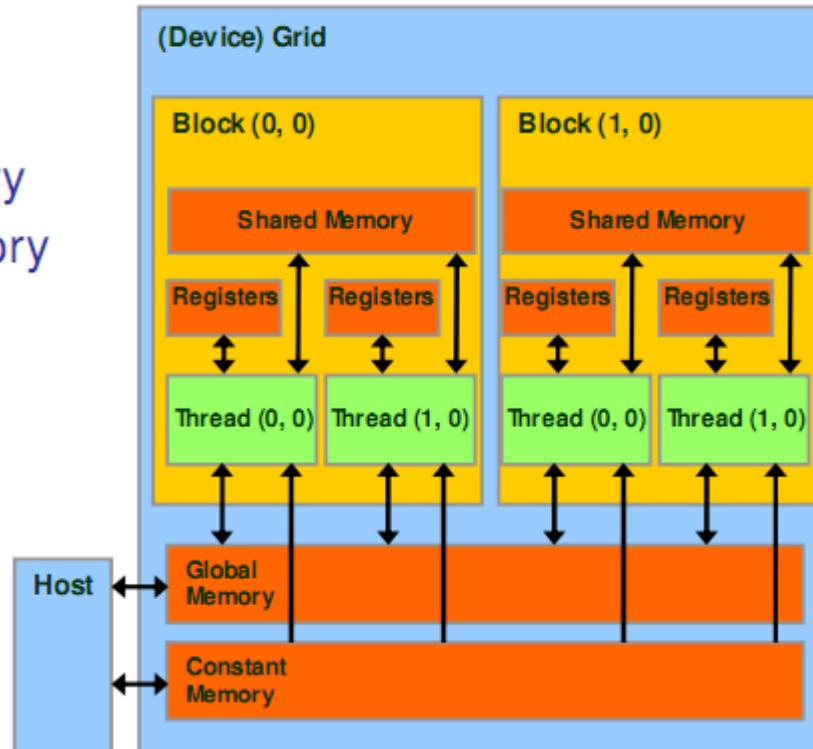
---

- ▶ *Intrinsic* function
  - ▶ Device only
  - ▶ Faster, but less accurate
  - ▶ Prefixed with `_`
  - ▶ `_exp`, `_log` , `_sin` , `_pow` , ...

# Memory Model Recall

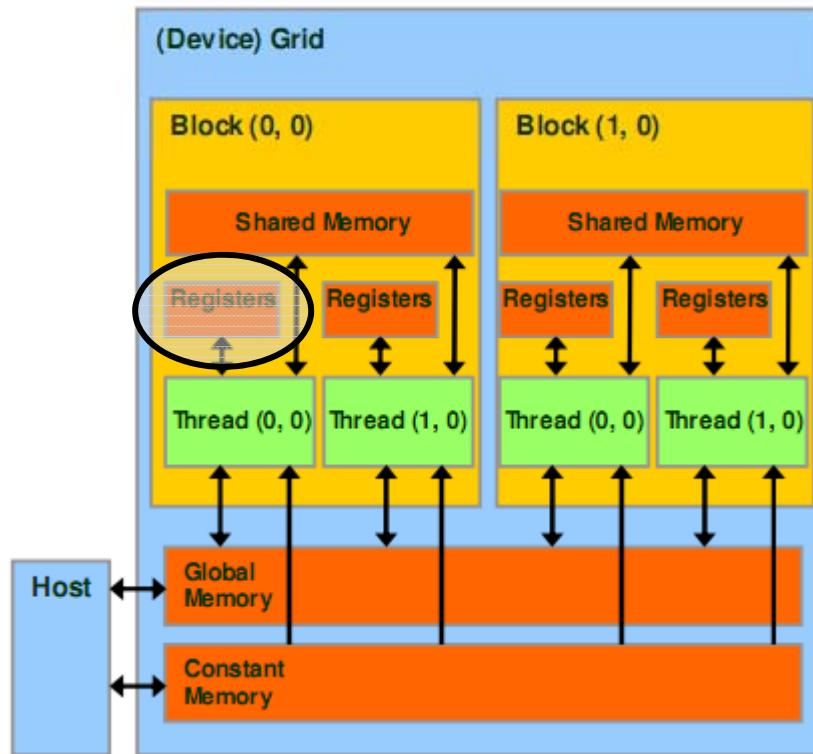
Recall:

- Device code 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
- Host code can
  - R/W per grid global and constant memories



# Memory Model

- ▶ Registers
  - ▶ Per thread
  - ▶ Fast, on-chip, read/write access
  - ▶ Increasing the number of registers used by a kernel has what affect?



# Memory Model

---

- ▶ Registers – G80
  - ▶ Per SM
    - ▶ Up to 768 threads
    - ▶ 8K registers
  - ▶ How many registers per thread?
- ▶ Registers – Kepler – Per SM
  - ▶ Up to 2048 threads
  - ▶ 64K Registers
- ▶ Registers – Maxwell– Per SM
  - ▶ Up to 2048 threads
  - ▶ 64K Registers

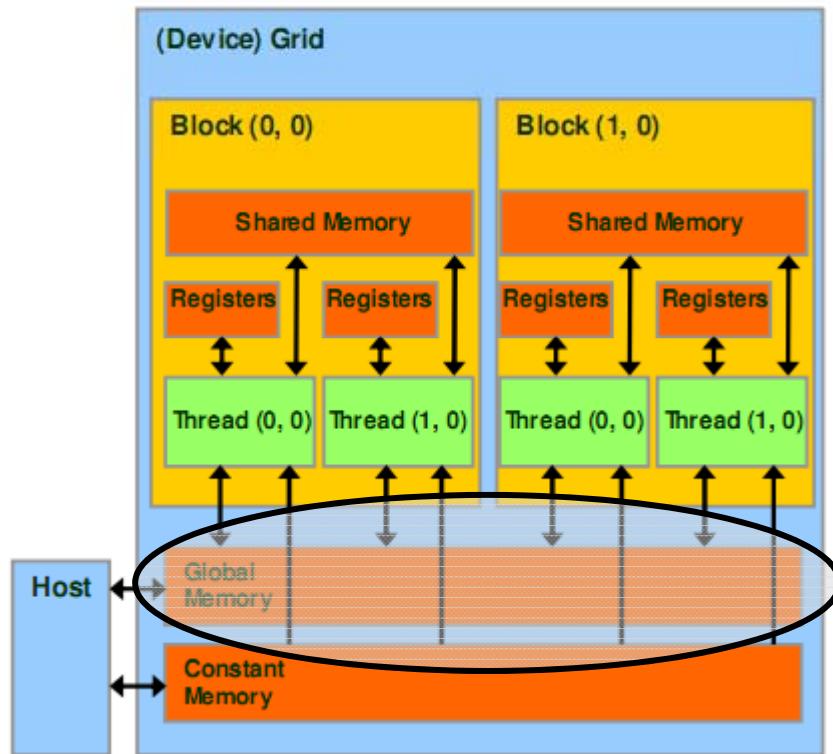
# Memory Model

---

- ▶ Registers – G80
  - ▶  $8K / 768 = 10$  registers per thread
  - ▶ Exceeding limit reduces threads by the block
  - ▶ Example: Each thread uses 11 registers, and each block has 256 threads
    - ▶ How many threads can a SM host?
    - ▶ How many warps can a SM host?
    - ▶ What does having less warps mean?

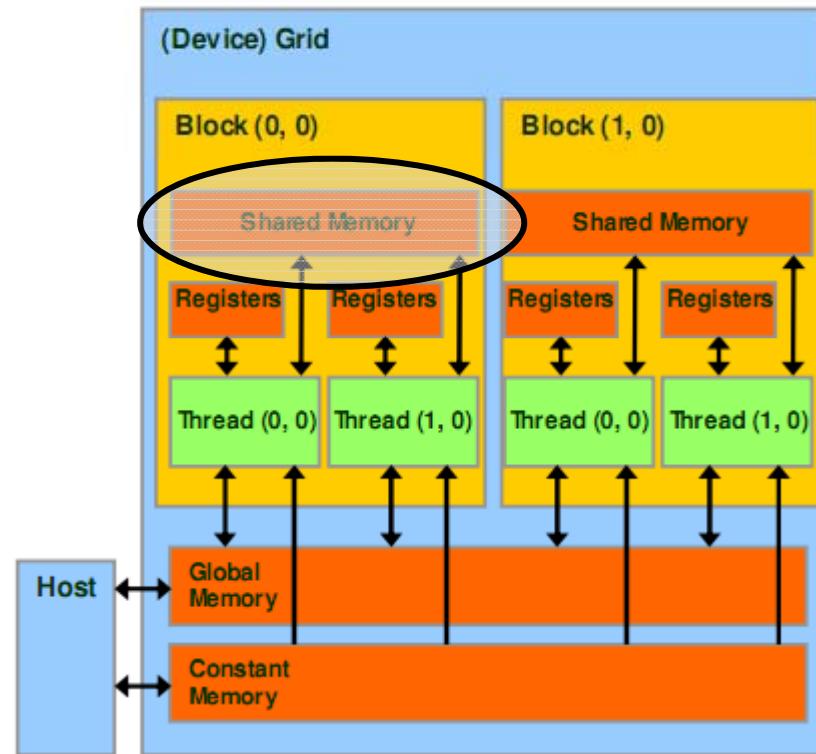
# Memory Model

- ▶ Local Memory
  - ▶ Stored in global memory
  - ▶ Copy per thread
- ▶ Used for automatic arrays
  - ▶ Unless all accessed with constant indices



# Memory Model

- ▶ Shared Memory
  - ▶ Per block
  - ▶ Fast, on-chip, read/write access
  - ▶ Full speed random access



# Memory Model

---

- ▶ Shared Memory - G80
  - ▶ Per SM
    - ▶ Up to 8 blocks
    - ▶ 16 KB
  - ▶ How many KB per block
- ▶ Shared Memory - Maxwell
  - ▶ Per SM
    - ▶ Up to 32 blocks
    - ▶ 96 KB
  - ▶ How many KB per block

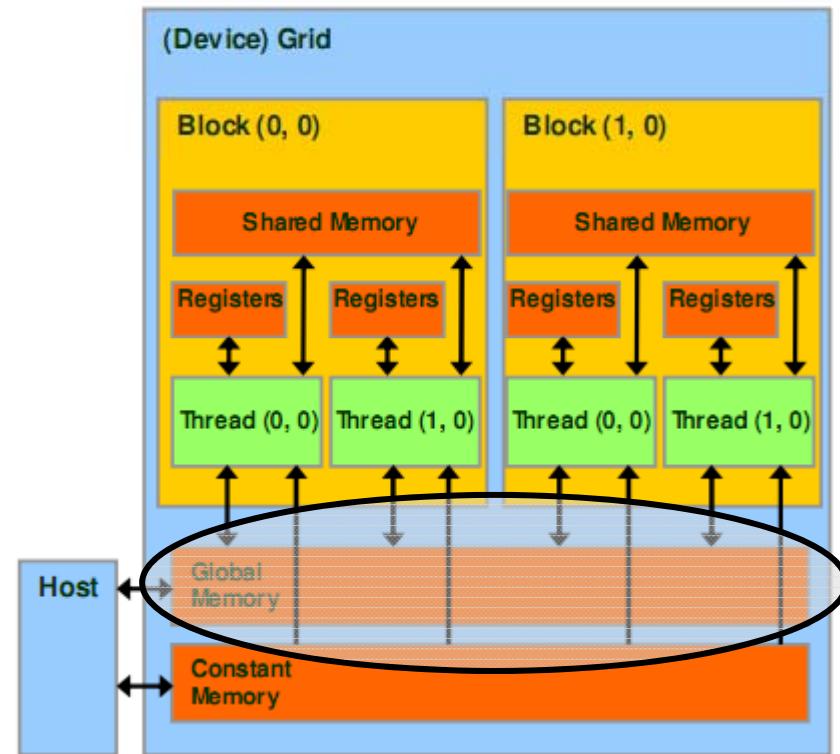
# Memory Model

---

- ▶ Shared Memory - G80
  - ▶  $16\text{ KB} / 8 = 2\text{ KB}$  per block
  - ▶ Example
    - ▶ If each block uses 5 KB, how many blocks can a SM host?

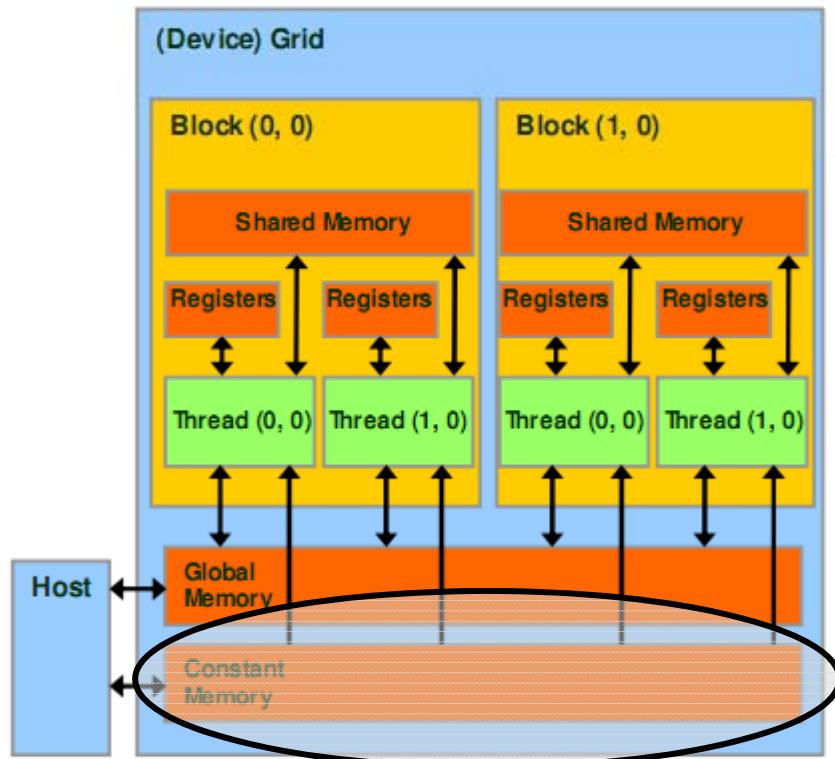
# Memory Model

- ▶ Global Memory
  - ▶ Long latency (100s cycles)
  - ▶ Off-chip, read/write access
  - ▶ Random access causes performance hit
  - ▶ Host can read/write
  - ▶ GT200
    - ▶ 150 GB/s
    - ▶ Up to 4 GB
  - ▶ 23
  - ▶ G80 - 86.4 GB/s



# Memory Model

- ▶ Constant Memory
  - ▶ Short latency, high bandwidth, read only access when all threads access the same location
  - ▶ Stored in global memory but cached
  - ▶ Host can read/write
  - ▶ Up to 64 KB



# Memory Model

---

| Variable Declaration                       | Memory   | Scope  | Lifetime    |
|--------------------------------------------|----------|--------|-------------|
| Automatic variables other than arrays      | register | thread | kernel      |
| Automatic array variables                  | local    | thread | kernel      |
| <code>__shared__ int sharedVar;</code>     | shared   | block  | kernel      |
| <code>__device__ int globalVar;</code>     | global   | grid   | application |
| <code>__constant__ int constantVar;</code> | constant | grid   | application |

# Memory Model

---

- ▶ Global and constant variables
  - ▶ Host can access with
    - ▶ `cudaGetSymbolAddress()`
    - ▶ `cudaGetSymbolSize()`
    - ▶ `cudaMemcpyToSymbol()`
    - ▶ `cudaMemcpyFromSymbol()`
  - ▶ Constants must be declared outside of a function body

## Review: Thread Hierarchies

---

```
int threadID = blockIdx.x *
 blockDim.x + threadIdx.x;

float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

## Review: Thread Hierarchies

```
int threadID = blockIdx.x *
 blockDim.x + threadIdx.x;
```

```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Use grid and block position to  
compute a thread id

## Review: Thread Hierarchies

---

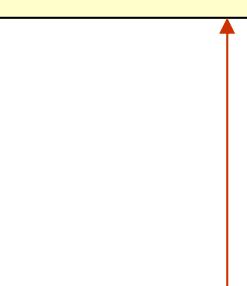
```
int threadID = blockIdx.x *
 blockDim.x + threadIdx.x;
```

```
float x = input[threadID];
```

```
float y = func(x);
```

```
output[threadID] = y;
```

Use thread id to read from input



## Review: Thread Hierarchies

---

```
int threadID = blockIdx.x *
 blockDim.x + threadIdx.x;

float x = input[threadID];

float y = func(x);

output[threadID] = y;
```

Run function on input: data-parallel!

## Review: Thread Hierarchies

---

```
int threadID = blockIdx.x *
 blockDim.x + threadIdx.x;

float x = input[threadID];
float y = func(x);

output[threadID] = y;
```

Use thread id to output result

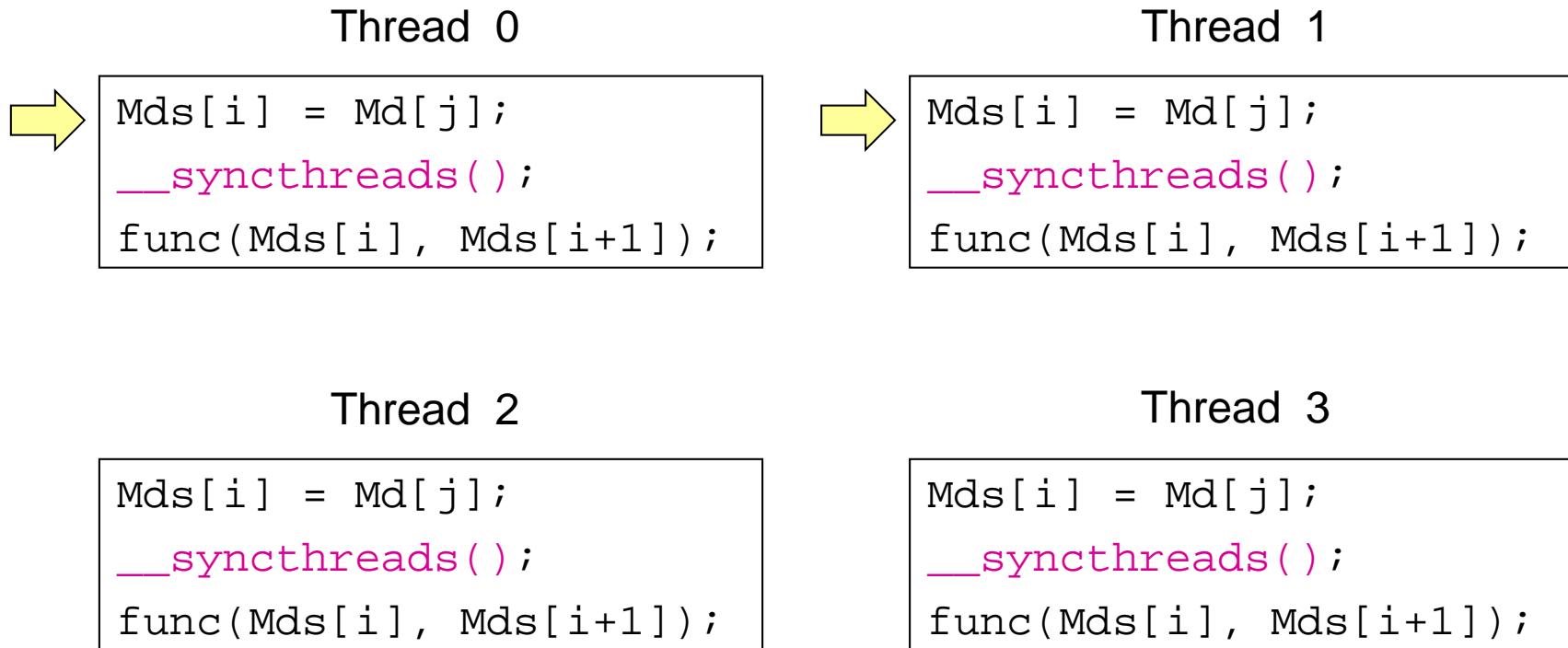
# Thread Synchronization

---

- ▶ Threads in a block can synchronize
  - ▶ call `__syncthreads` to create a barrier
  - ▶ A thread waits at this call until all threads in the block reach it, then all threads continue

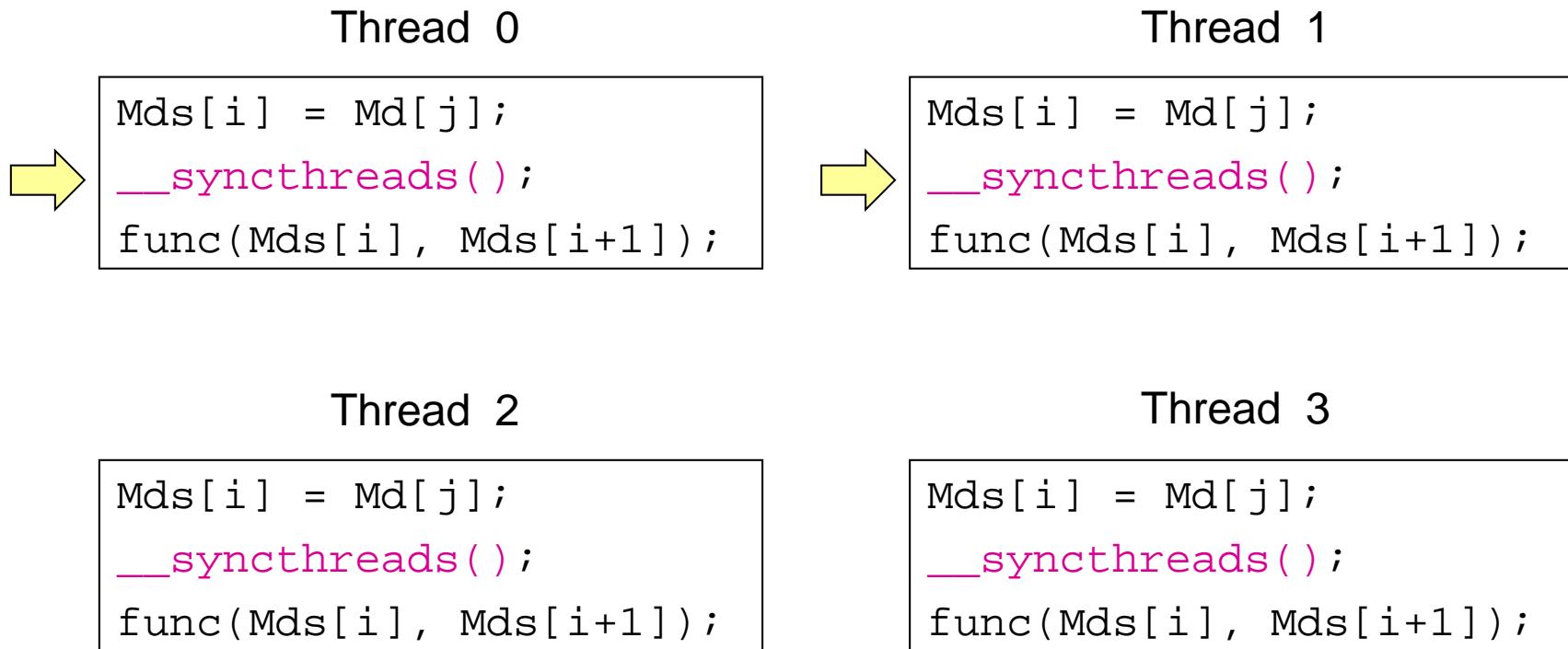
```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i + 1]);
```

# Thread Synchronization

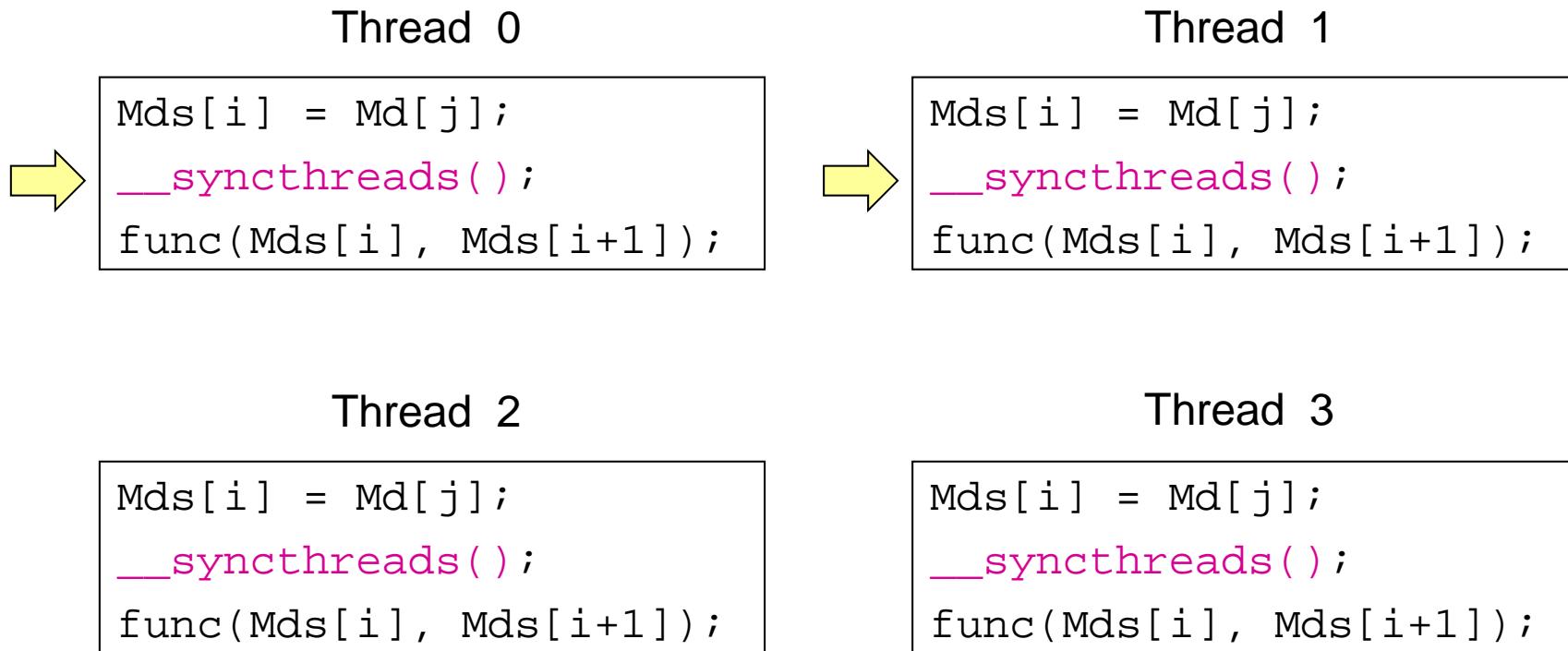


Time: 0

# Thread Synchronization



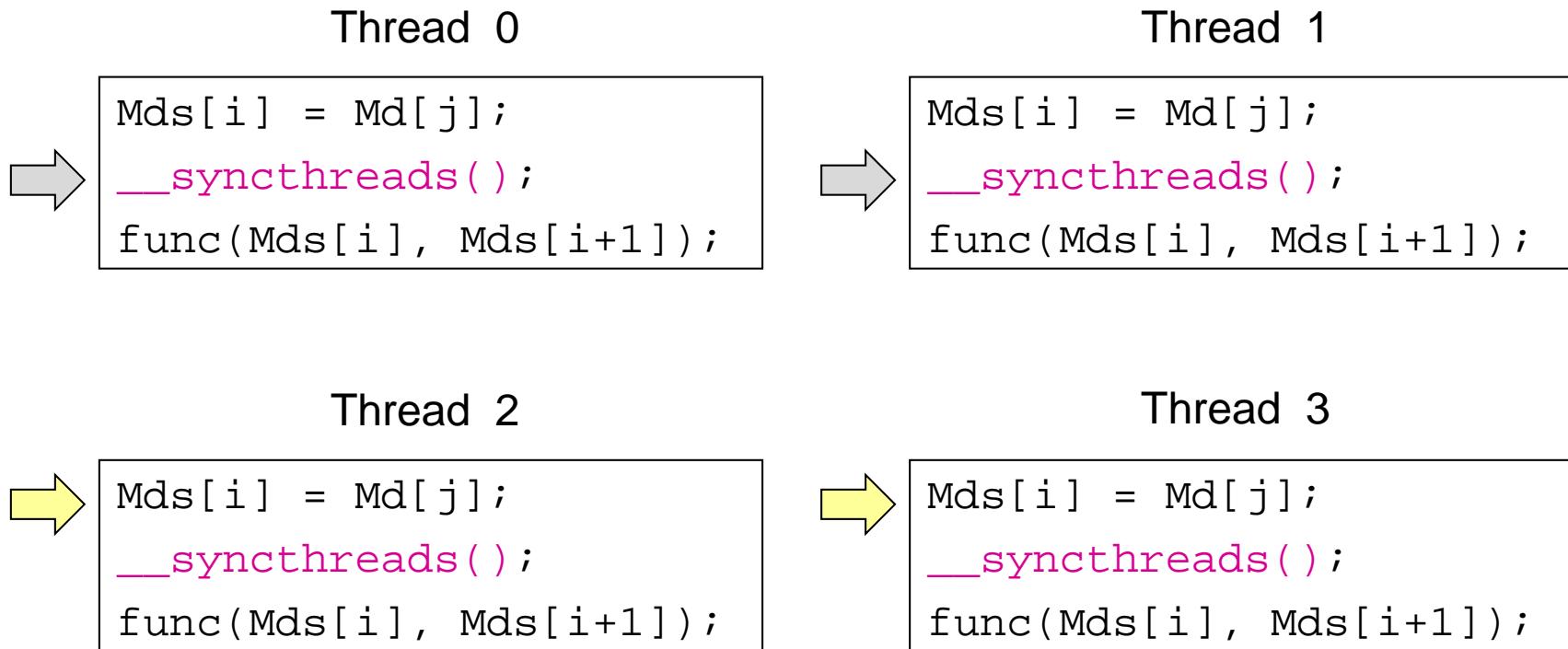
# Thread Synchronization



Threads 0 and 1 are blocked at barrier

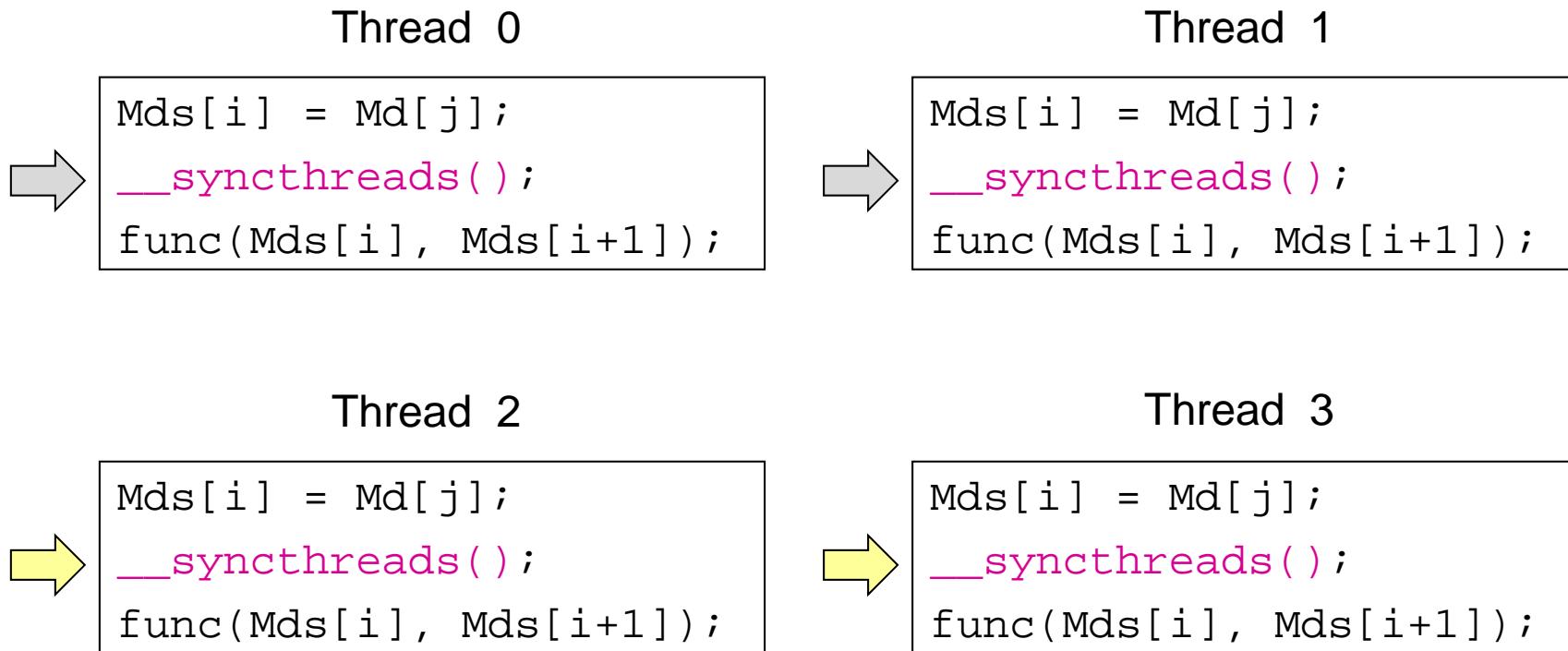
Time: 1

# Thread Synchronization



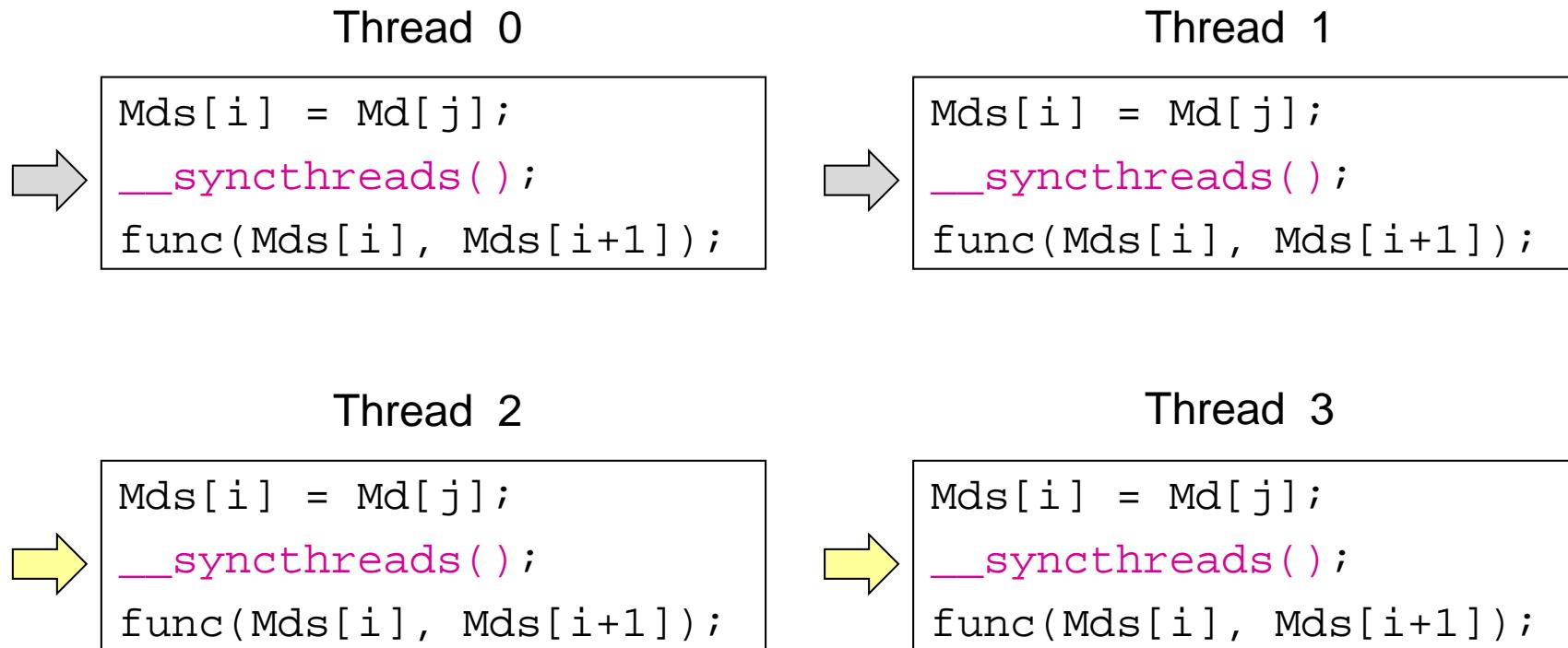
Time: 2

# Thread Synchronization



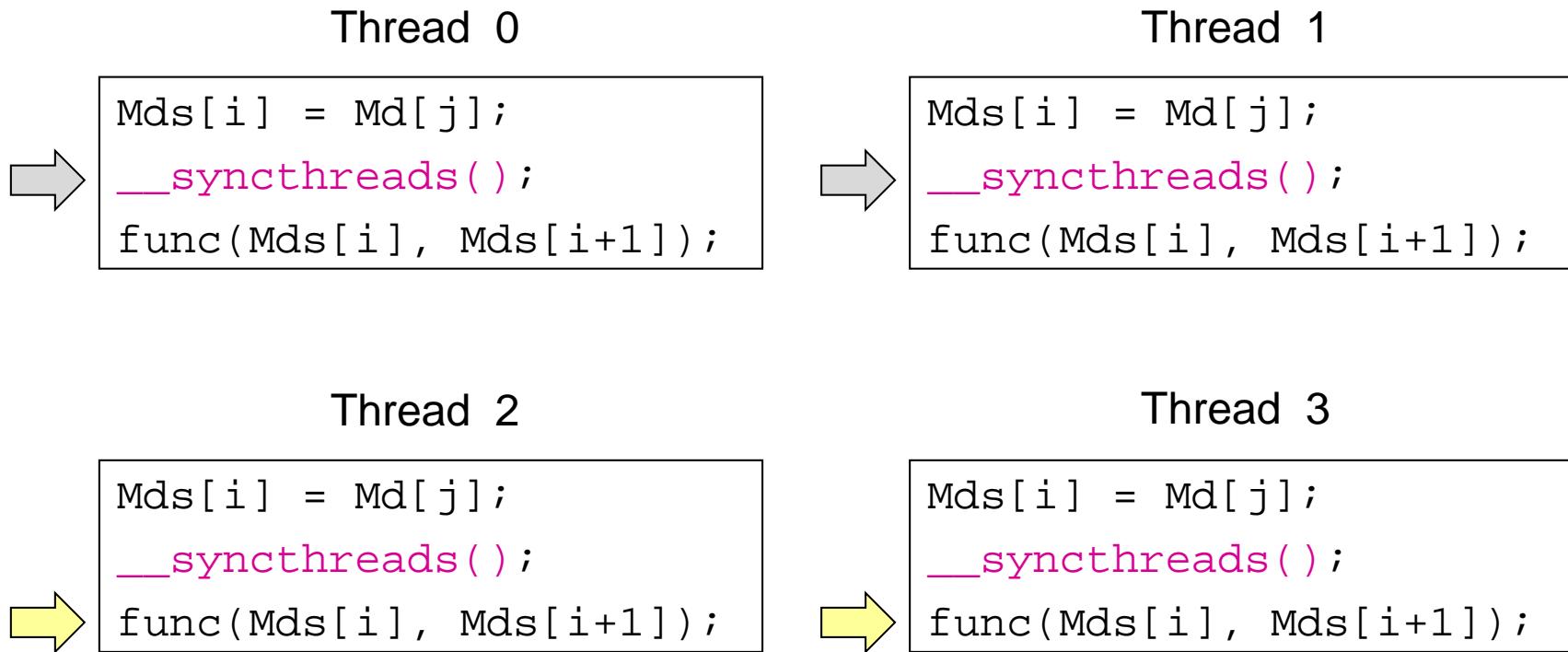
Time: 3

# Thread Synchronization



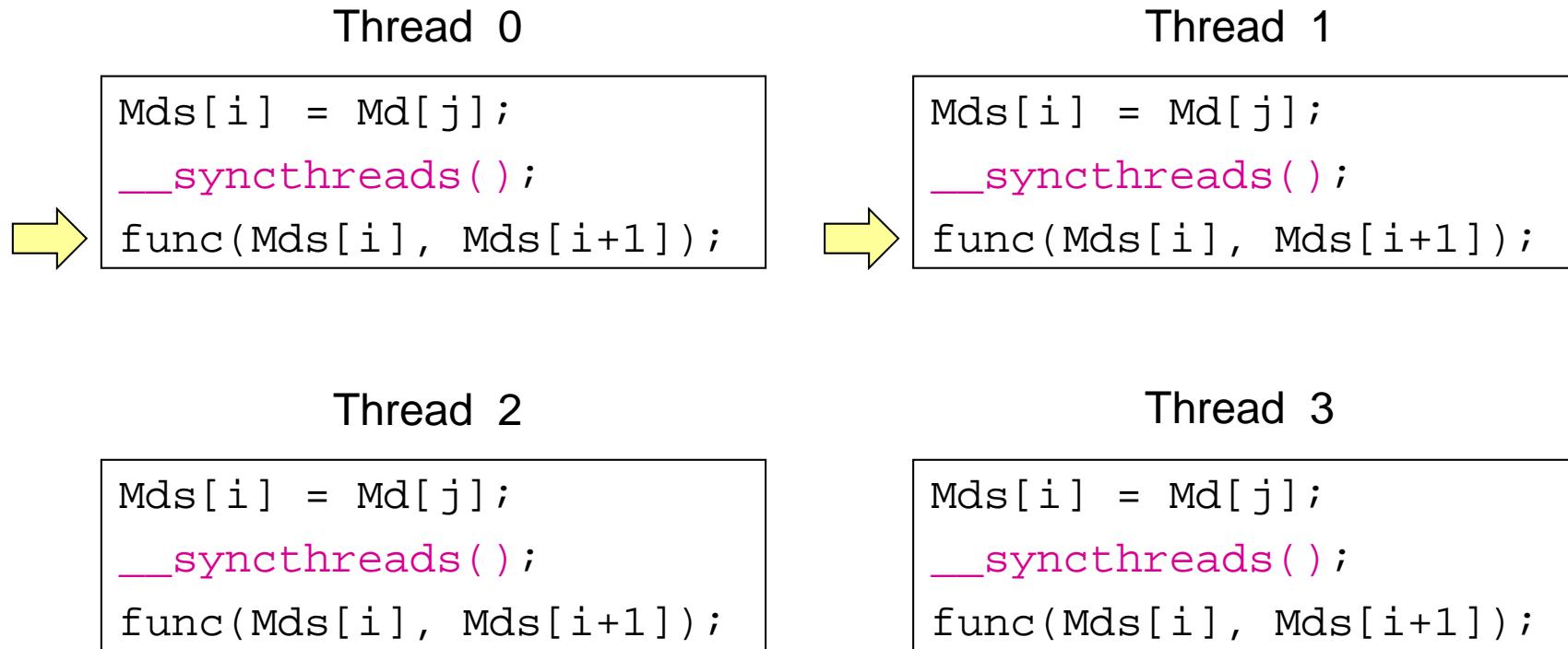
All threads in block have reached barrier, any thread can continue

# Thread Synchronization



Time: 4

# Thread Synchronization



# Thread Synchronization

---

- ▶ Why is it important that execution time be similar among threads?
- ▶ Why does it only synchronize within a block?

# Thread Synchronization

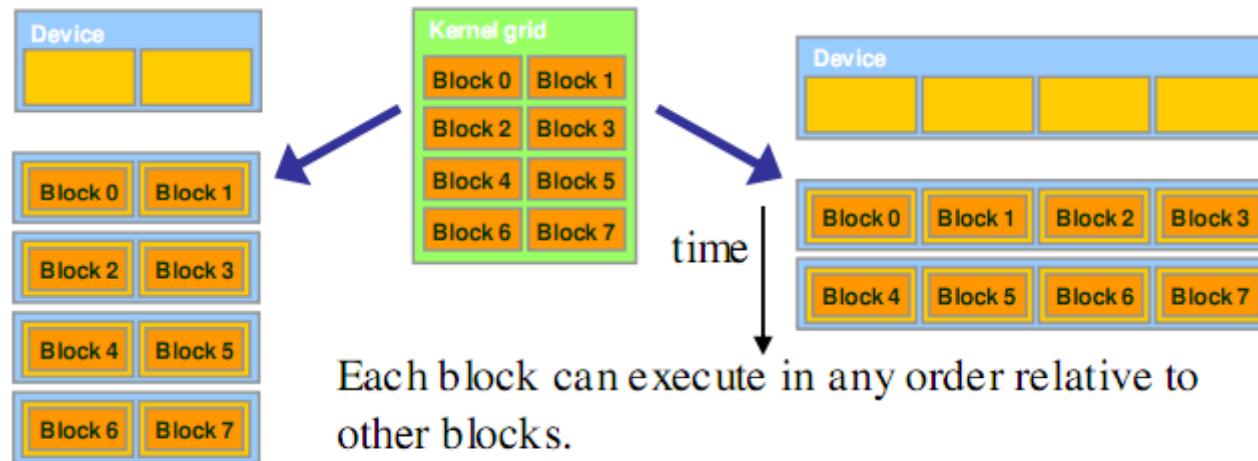


Figure 3.5 Lack of synchronization across blocks enables transparent scalability of CUDA programs

# Thread Synchronization

---

- ▶ Can `__syncthreads()` cause a thread to hang?

# Thread Synchronization

---

```
if (someFunc())
{
 __syncthreads() ;
}
// . . .
```

# Thread Synchronization

---

```
if (someFunc())
{
 __syncthreads() ;
}
else
{
 __syncthreads() ;
}
```

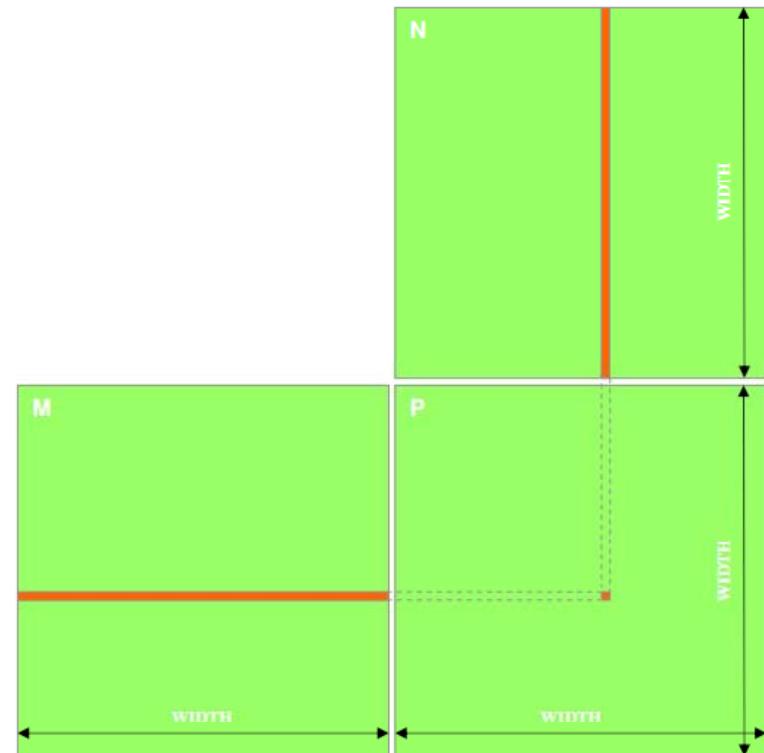
---

Let's  
revisit  
matrix  
multiple

---

# Matrix Multiply: CPU Implementation

```
void MatrixMulOnHost(float* M, float* N, float* P, int width)
{
 for (int i = 0; i < width; ++i)
 for (int j = 0; j < width; ++j)
 {
 float sum = 0;
 for (int k = 0; k < width; ++k)
 {
 float a = M[i * width + k];
 float b = N[k * width + j];
 sum += a * b;
 }
 P[i * width + j] = sum;
 }
}
```



# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y; ← Accessing a matrix, so using a 2D block

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0;

 for (int k = 0; k < Width; ++k)
 {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0; // Each kernel computes one output
 for (int k = 0; k < Width; ++k)
 {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

# Matrix Multiply: CUDA Kernel

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0;

 for (int k = 0; k < Width; ++k) {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

Where did the two outer for loops  
in the CPU implementation go?

# Matrix Multiply: CUDA Kernel

---

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
 // 2D Thread ID
 int tx = threadIdx.x;
 int ty = threadIdx.y;

 // Pvalue stores the Pd element that is computed by the thread
 float Pvalue = 0;

 for (int k = 0; k < Width; ++k)
 {
 float Mdelement = Md[ty * Md.width + k];
 float Ndelement = Nd[k * Nd.width + tx];
 Pvalue += Mdelement * Ndelement;
 }

 // Write the matrix to device memory each thread writes one element
 Pd[ty * Width + tx] = Pvalue;
}
```

No locks or synchronization, why?

# Matrix Multiply

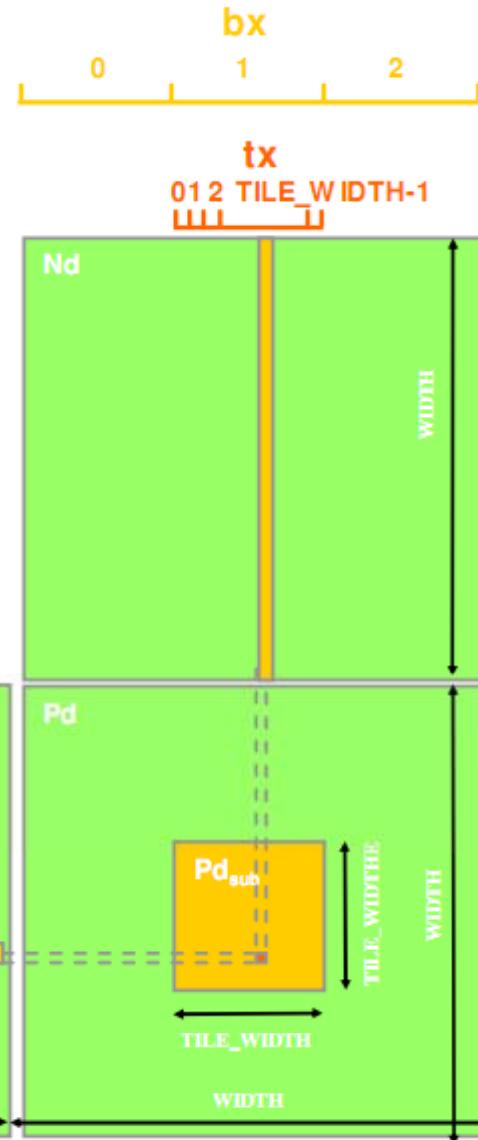
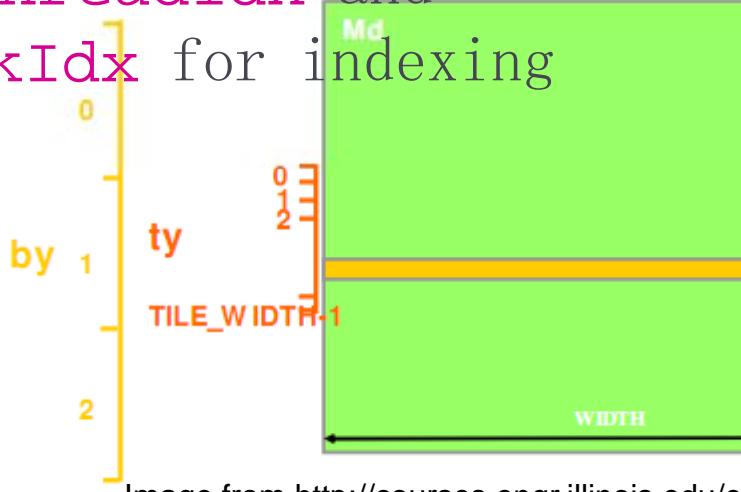
---

- ▶ Problems

- ▶ Limited matrix size
  - ▶ Only uses one block
  - ▶ G80 and GT200 - up to 512 threads per block
  - ▶ Fermi, Kepler, Maxwell - up to 1024 threads per block
- ▶ Lots of global memory access

# Matrix Multiply

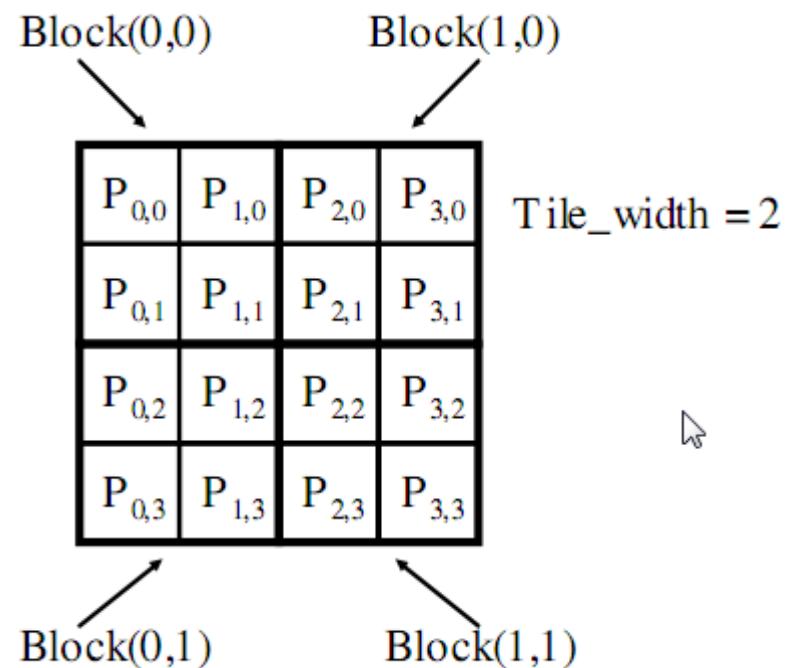
- ▶ Remove size limitation
  - ▶ Break  $Pd$  matrix into tiles
  - ▶ Assign each tile to a block
  - ▶ Use `threadIdx` and `blockIdx` for indexing



# Matrix Multiply

## ▶ Example

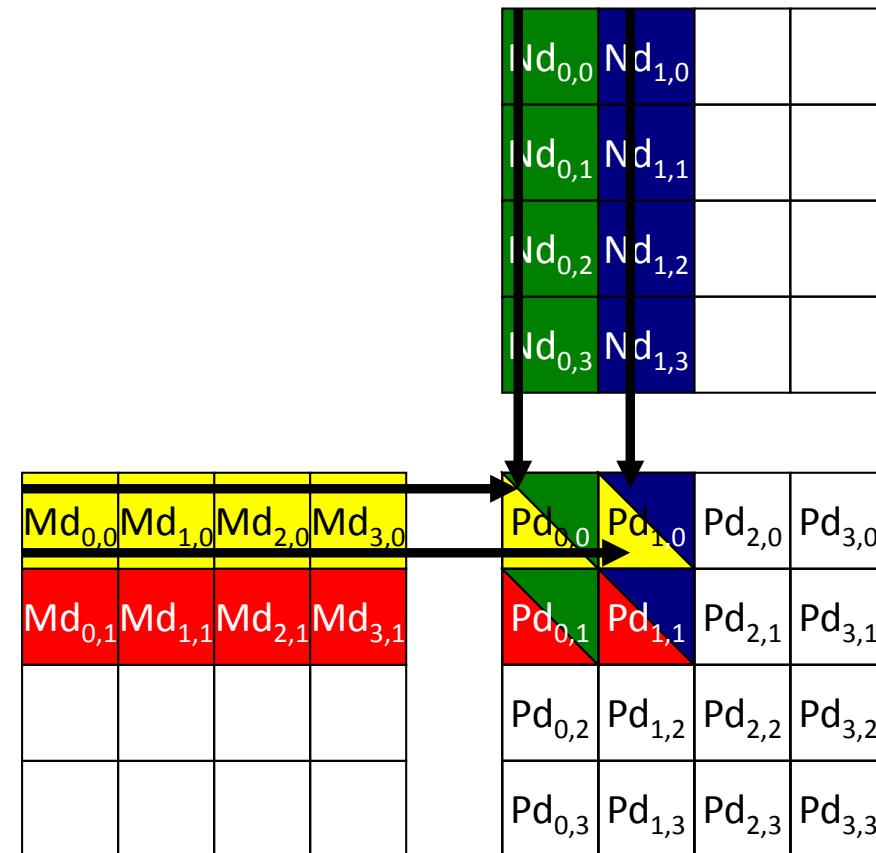
- ▶ Matrix: 4x4
- ▶ TILE\_WIDTH = 2
- ▶ Block size: 2x2



# Matrix Multiply

## ▶ Example

- ▶ Matrix: 4x4
- ▶ TILE\_WIDTH = 2
- ▶ Block size: 2x2



# Matrix Multiply

---

```
__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 float Pvalue = 0;
 for (int k = 0; k < Width; ++k)
 Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

 Pd[Row * Width + Col] = Pvalue;
}
```

# Matrix Multiply

Calculate the row index of the Pd element and M

```
__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 float Pvalue = 0;
 for (int k = 0; k < Width; ++k)
 Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

 Pd[Row * Width + Col] = Pvalue;
}
```

# Matrix Multiply

Calculate the column index of Pd and N

```
__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 float Pvalue = 0;
 for (int k = 0; k < Width; ++k)
 Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

 Pd[Row * Width + Col] = Pvalue;
}
```

# Matrix Multiply

```
__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 float Pvalue = 0;
 for (int k = 0; k < Width; ++k)
 Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

 Pd[Row * Width + Col] = Pvalue;
}
```

Each thread computes one element  
of the block sub-matrix

# Matrix Multiply

---

- ▶ Invoke kernel:

```
dim3 dimGrid(Width / TILE_WIDTH, Height / TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

MatrixMulKernel<<<dimGrid, dimBlock>>>(
 Md, Nd, Pd, TILE_WIDTH);
```

---

What about  
global memory  
access?

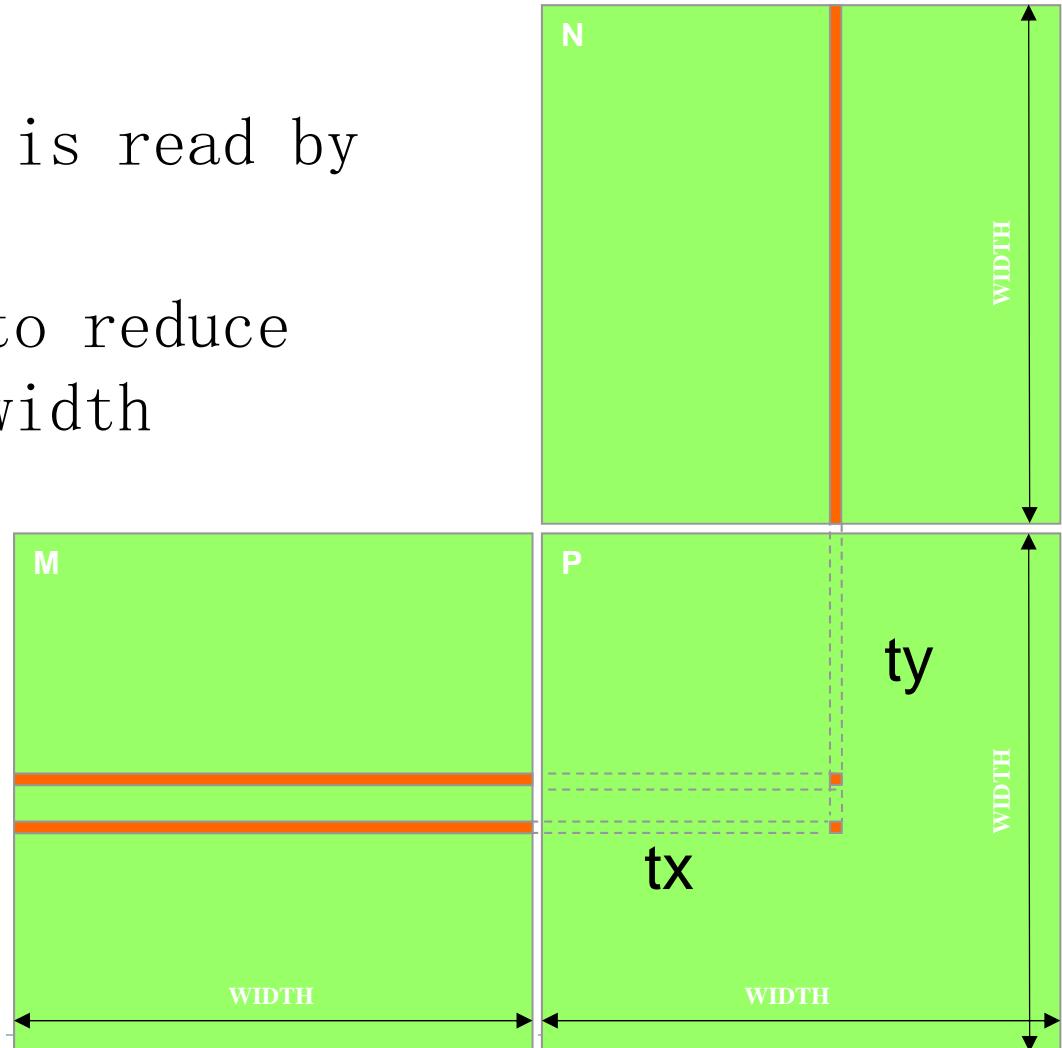
# Matrix Multiply

---

- ▶ Limited by global memory bandwidth
  - ▶ Tesla K40 peak GFLOPS( SP) : 5000
  - ▶ Require 20000 GB/s to achieve this
  - ▶ G80 memory bandwidth: 288 GB/s
    - ▶ Limits code to 72 GFLOPS
    - ▶ In practice, code runs at 60 GFLOPS
- ▶ Must drastically reduce global memory access

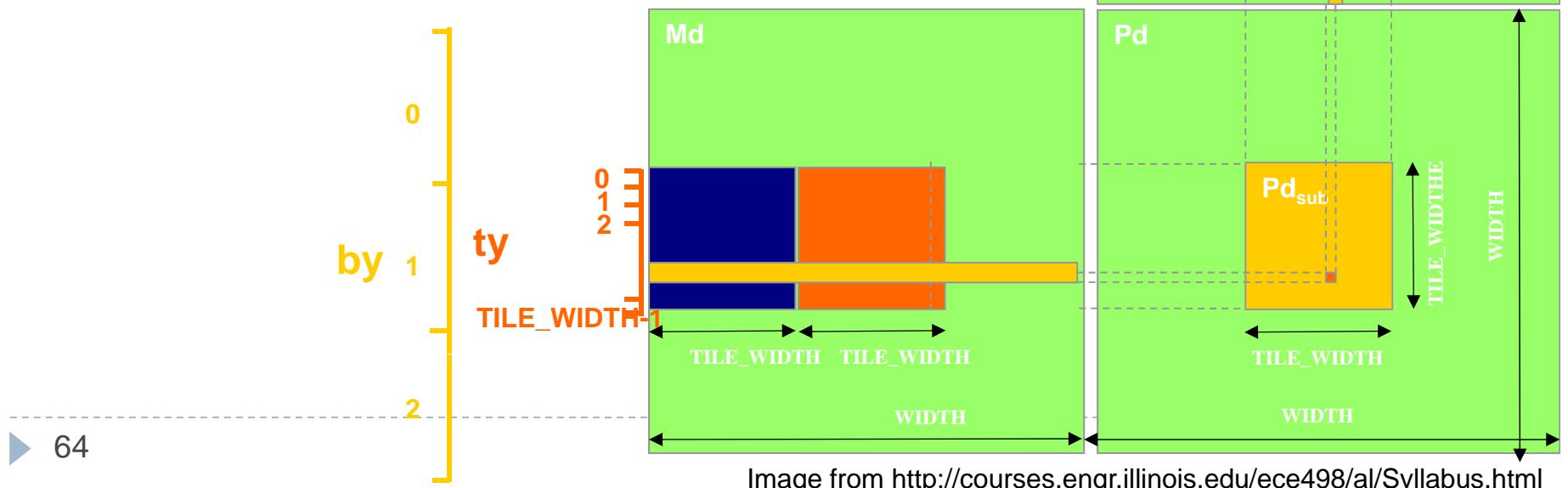
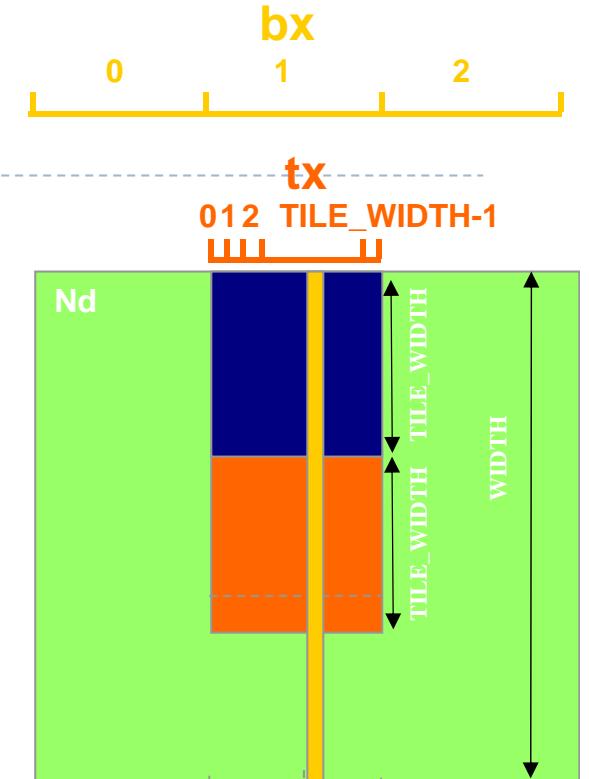
# Matrix Multiply

- ▶ Each input element is read by **width** threads
- ▶ Use shared memory to reduce global memory bandwidth



# Matrix Multiply

- ▶ Break kernel into phases
  - ▶ Each phase accumulates  $P_d$  using a subset of  $M_d$  and  $N_d$
  - ▶ Each phase has good data locality

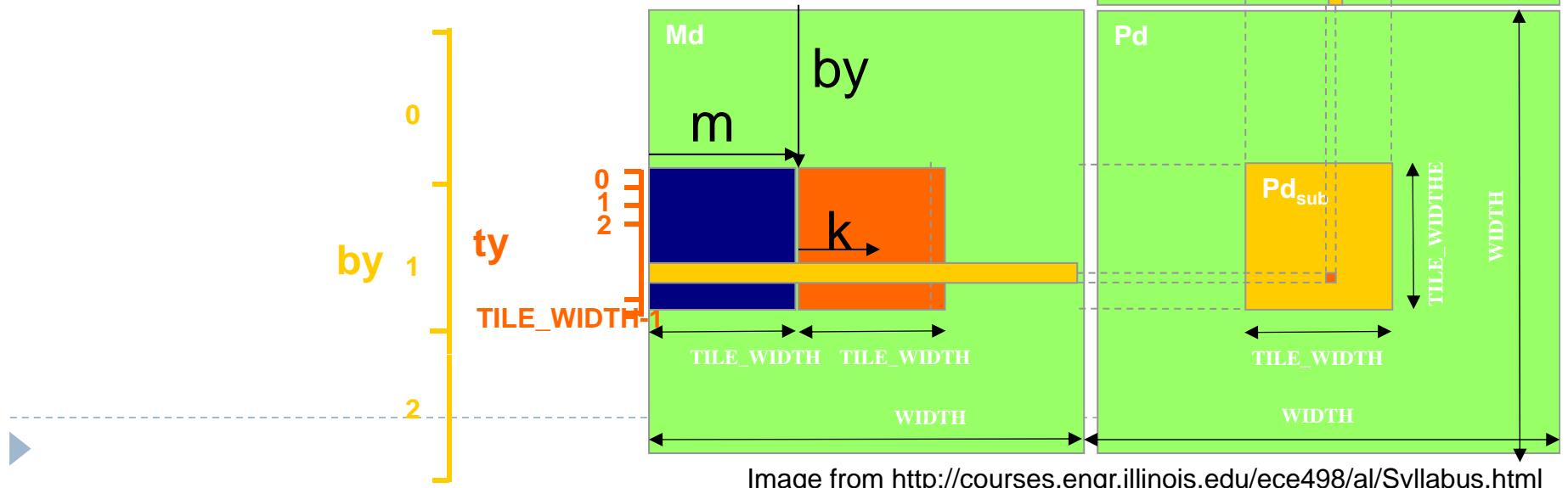


▶ 64

Image from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

# Matrix Multiply

- ▶ Each thread
  - ▶ loads one element of  $M_d$  and  $N_d$  in the tile into shared memory



```
__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*width];
 __syncthreads();

 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
 }
 Pd[Row*Width+Col] = Pvalue;
}
```

```

__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*width];
 __syncthreads();

 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
 }
 Pd[Row*Width+Col] = Pvalue;
}

```

Shared memory for a subset of Md and Nd

```

__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*width];
 __syncthreads();

 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
 }
 Pd[Row*Width+Col] = Pvalue;
}

```

Width/TILE\_WIDTH  
 • Number of phases  
 $m$   
 • Index for current phase

```

__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
 }
 __syncthreads();

 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
}
Pd[Row*Width+Col] = Pvalue;

```

Bring one element  
each from `Md` and `Nd`  
into shared memory

```

__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*width];
 __syncthreads();
 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
 }
 Pd[Row*Width+Col] = Pvalue;
}

```

Wait for every thread  
in the block, i.e., wait  
for the tile to be in  
shared memory

```

__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*width];
 __syncthreads();

 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
 }
 Pd[Row*Width+Col] = Pvalue;
}

```

Accumulate subset of dot product

```

__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*width];
 __syncthreads();

 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
 }
 Pd[Row*Width+Col] = Pvalue;
}

```

```

__global__ void MatrixMulKernel(
 float* Md, float* Nd, float* Pd, int Width)
{
 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 int bx = blockIdx.x; int by = blockIdx.y;
 int tx = threadIdx.x; int ty = threadIdx.y;

 int Row = by * TILE_WIDTH + ty;
 int Col = bx * TILE_WIDTH + tx;

 float Pvalue = 0;
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
 Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*width];
 __syncthreads();

 for (int k = 0; k < TILE_WIDTH; ++k)
 Pvalue += Mds[ty][k] * Nds[k][tx];
 __syncthreads();
 }

 Pd[Row*Width+Col] = Pvalue;
}

```

Write final  
answer to global  
memory

Code from <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>

# Matrix Multiply

---

- ▶ How do you pick `TILE_WIDTH`?
  - ▶ How can it be too large?

# Matrix Multiply

---

- ▶ How do you pick `TILE_WIDTH`?
  - ▶ How can it be too large?
    - ▶ By exceeding the maximum number of threads/block
      - G80 and GT200 - 512
      - Fermi - 1024

# Matrix Multiply

---

- ▶ How do you pick `TILE_WIDTH`?
  - ▶ How can it be too large?
    - ▶ By exceeding the maximum number of threads/block
      - G80 and GT200 - 512
      - Fermi - 1024
    - ▶ By exceeding the shared memory limitations
      - G80: 16KB per SM and up to 8 blocks per SM
        - 2 KB per block
        - 1 KB for `Nds` and 1 KB for `Mds` ( $16 * 16 * 4$ )
        - `TILE_WIDTH = 16`
        - A larger `TILE_WIDTH` will result in less blocks

# Matrix Multiply

---

- ▶ Shared memory tiling benefits
  - ▶ Reduces global memory access by a factor of TILE\_WIDTH
    - ▶ 16x16 tiles reduces by a factor of 16
  - ▶ G80
    - ▶ Now global memory supports 345.6 GFLOPS
    - ▶ Close to maximum of 346.5 GFLOPS

# First-order Size Considerations in G80

---

- Each **thread block** should have many threads
  - TILE\_WIDTH of 16 gives  $16 \times 16 = 256$  threads
- There should be many thread blocks
  - A 1024\*1024 Pd gives  $64 \times 64 = 4K$  Thread Blocks
- Each thread block perform  $2 \times 256 = 512$  float loads from global memory for  $256 \times (2 \times 16) = 8K$  mul/add operations.
  - Memory bandwidth no longer a limiting factor

# CUDA/GPU Programming Model

Bin ZHOU @ NVIDIA & USTC  
Jan. 2015

# Contents

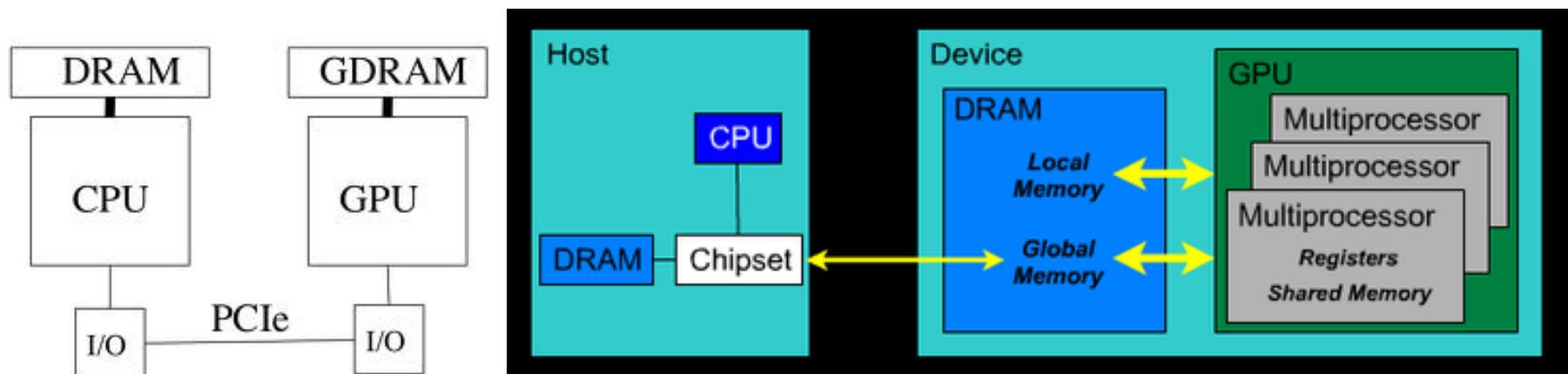
---

- ▶ CPU&GPU Interaction
- ▶ GPU Thread Organization (important)
- ▶ GPU Memory Hierarchy
- ▶ Some Basic Programming



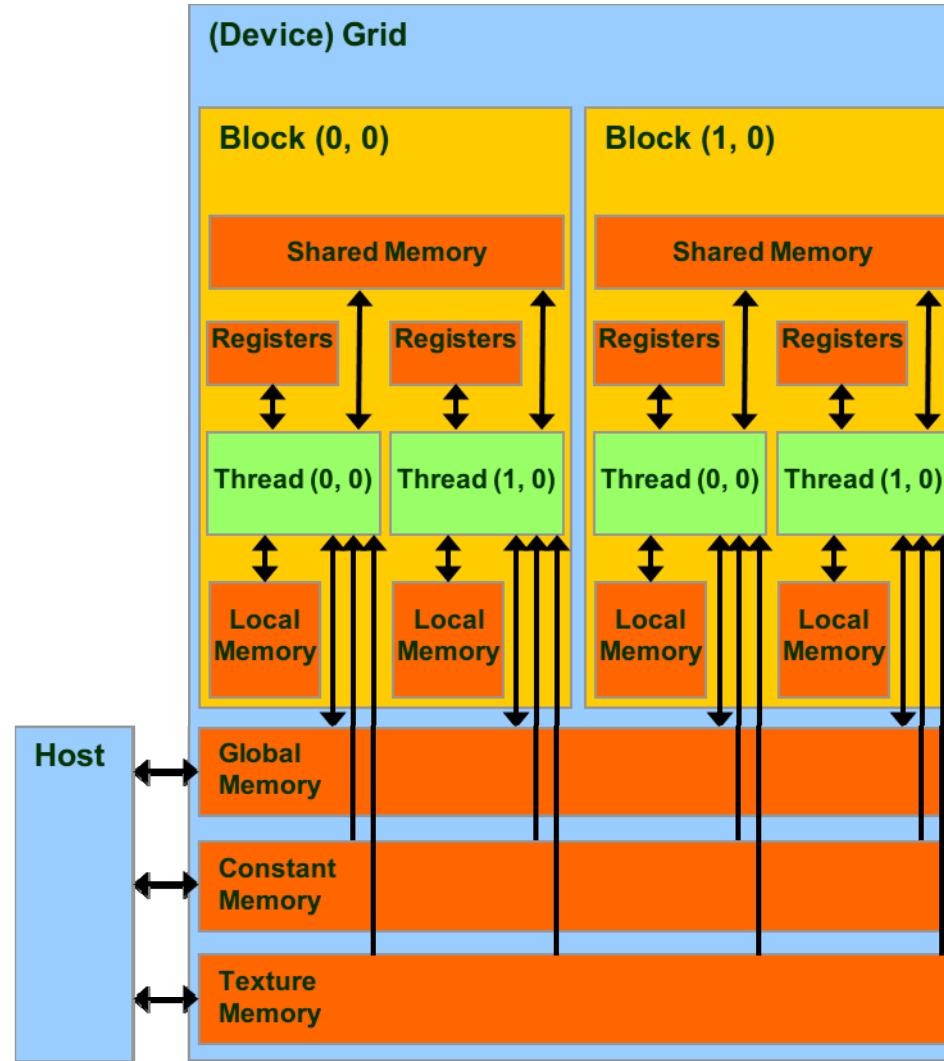
# CPU-GPU Interaction

- ▶ Separate Physical Memory Space
- ▶ Via PCIE Bus (8GB/s~16GB/s)
- ▶ Communication Overhead

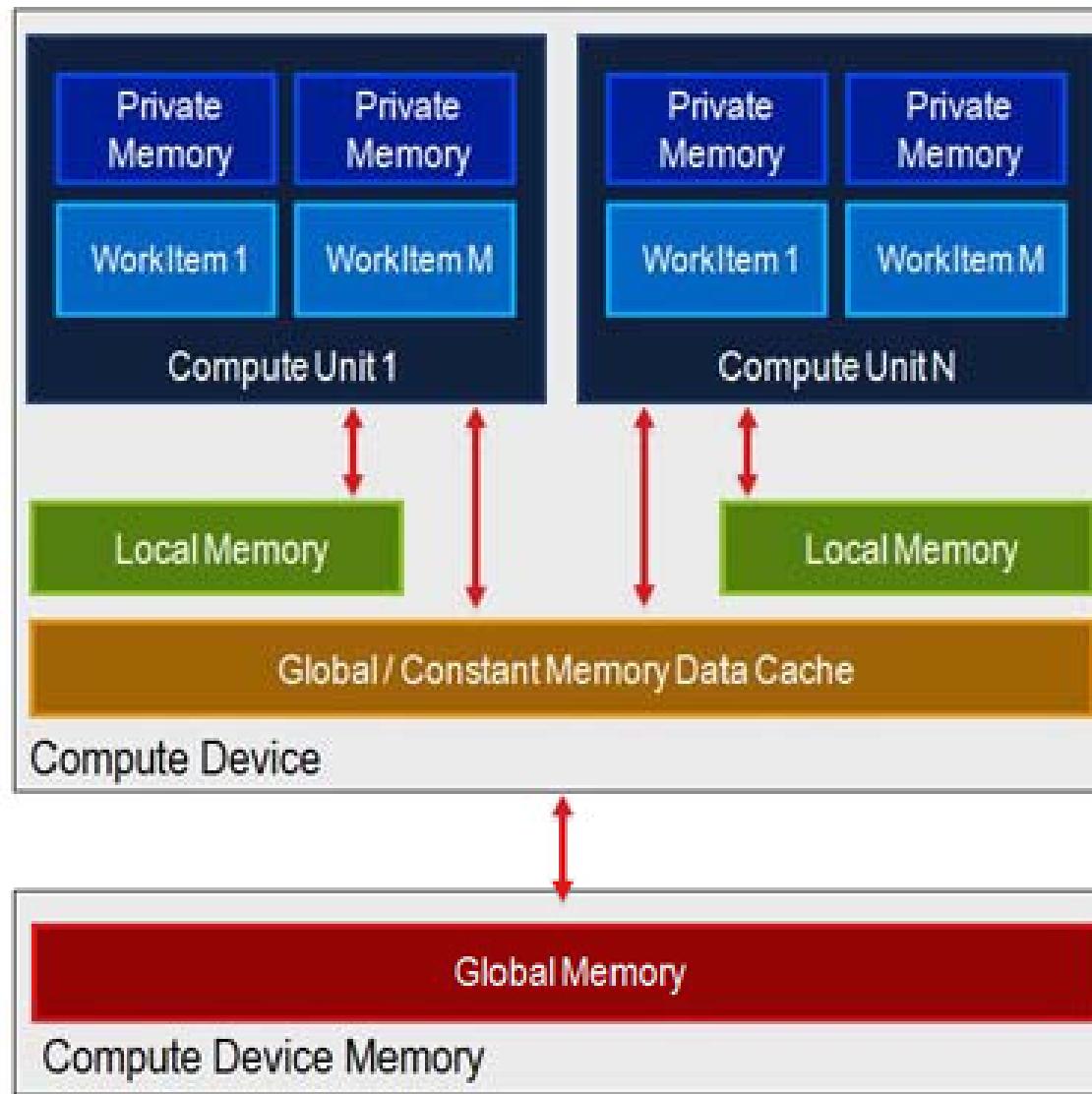


© NVIDIA Corporation

# GPU Memory Hierarchy (CUDA View)



# GPU Memory Hierarchy (OpenCL View)



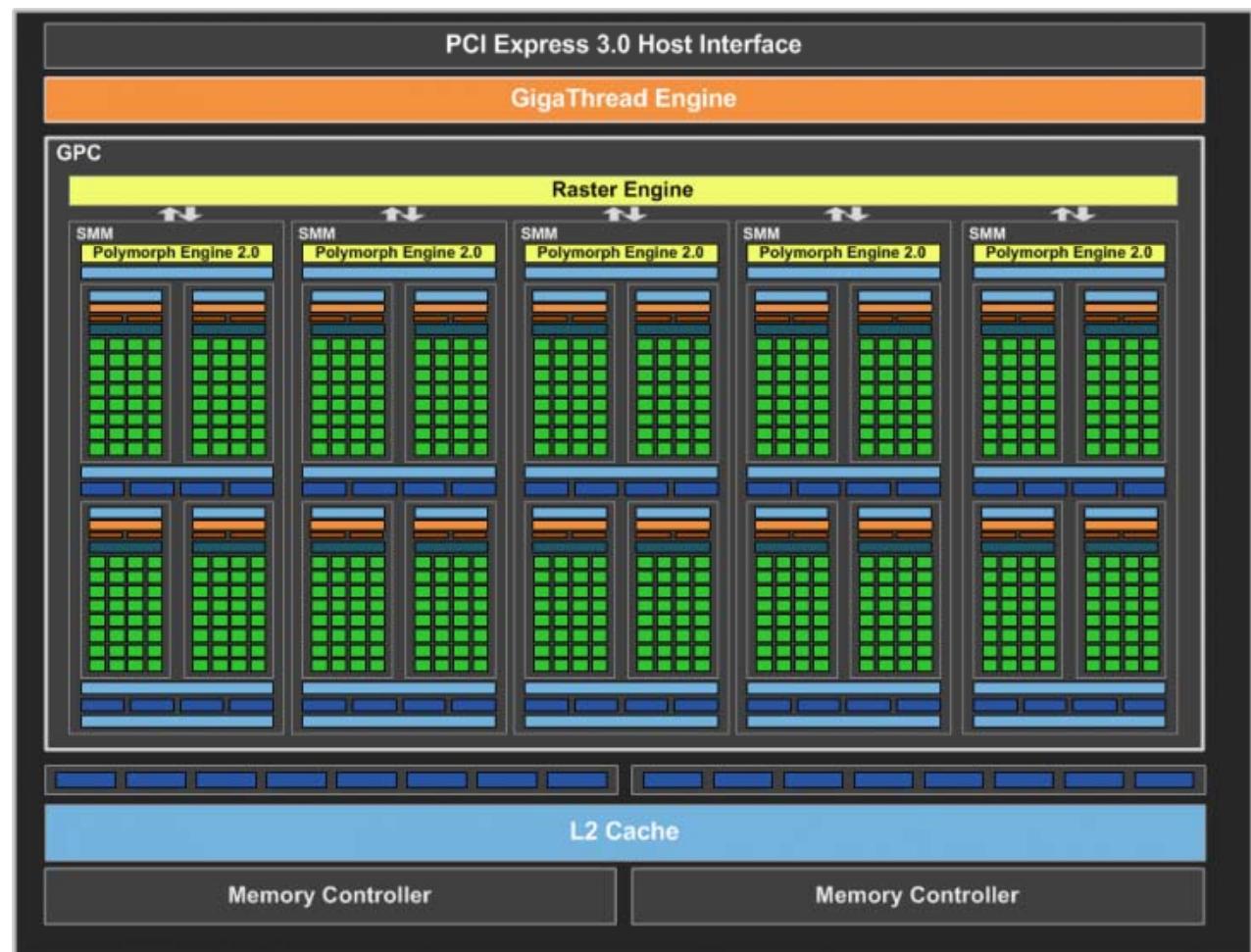
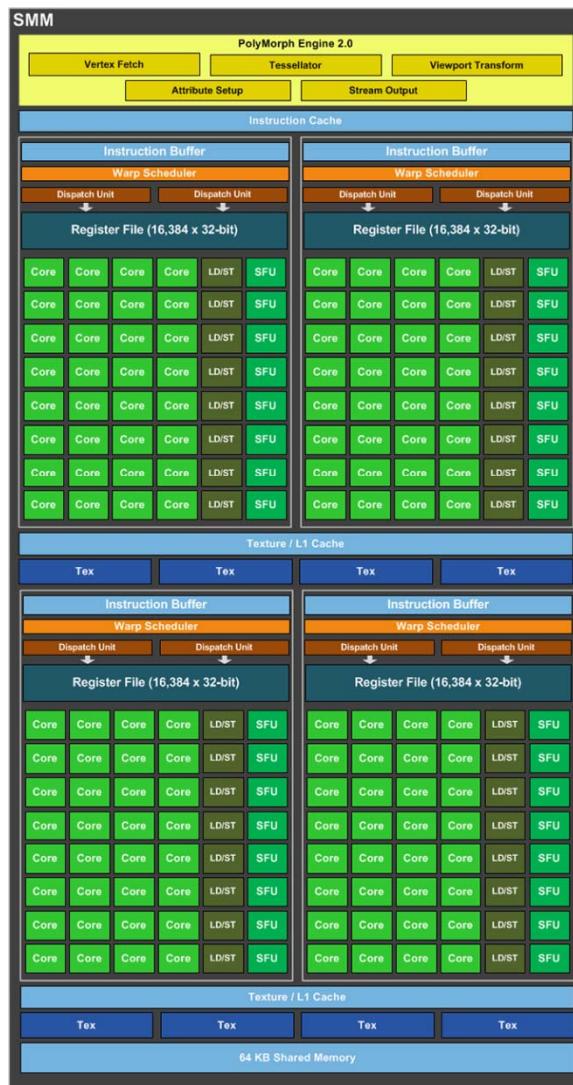
# Memory Access Speed

---

- ▶ 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
- ▶ Instruction Memory (invisible) - DRAM, cached



# GPU Architecture Review



# CUDA Programming Model

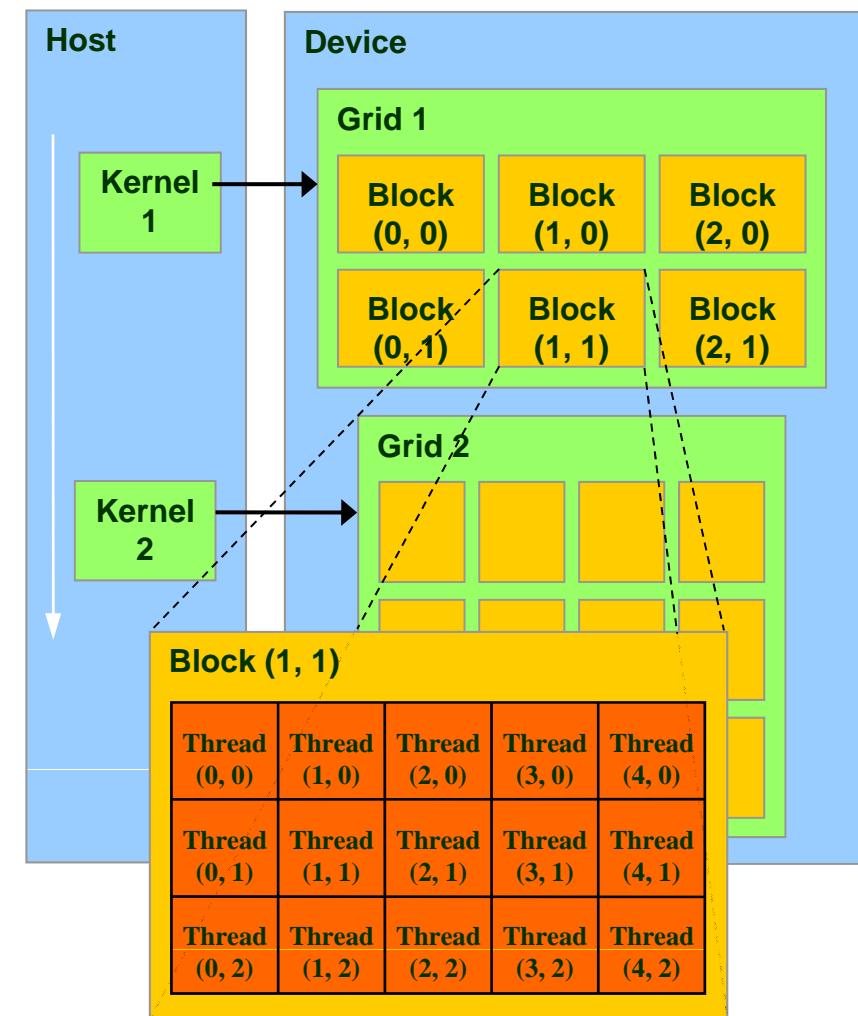
---

- ▶ The GPU is viewed as a compute device that:
    - ▶ Is a coprocessor to the CPU or host
    - ▶ Has its own DRAM (device memory)
    - ▶ Runs many threads in parallel
      - ▶ Hardware switching between threads (in 1 cycle) on long-latency memory reference
      - ▶ Overprovision (10000s of threads) → hide latencies
  - ▶ Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
  - ▶ Differences between GPU and CPU threads
    - ▶ GPU threads are extremely lightweight
      - ▶ Very little creation overhead
    - ▶ GPU needs 10000s of threads for full efficiency
      - ▶ Multi-core CPU needs only a few
- 

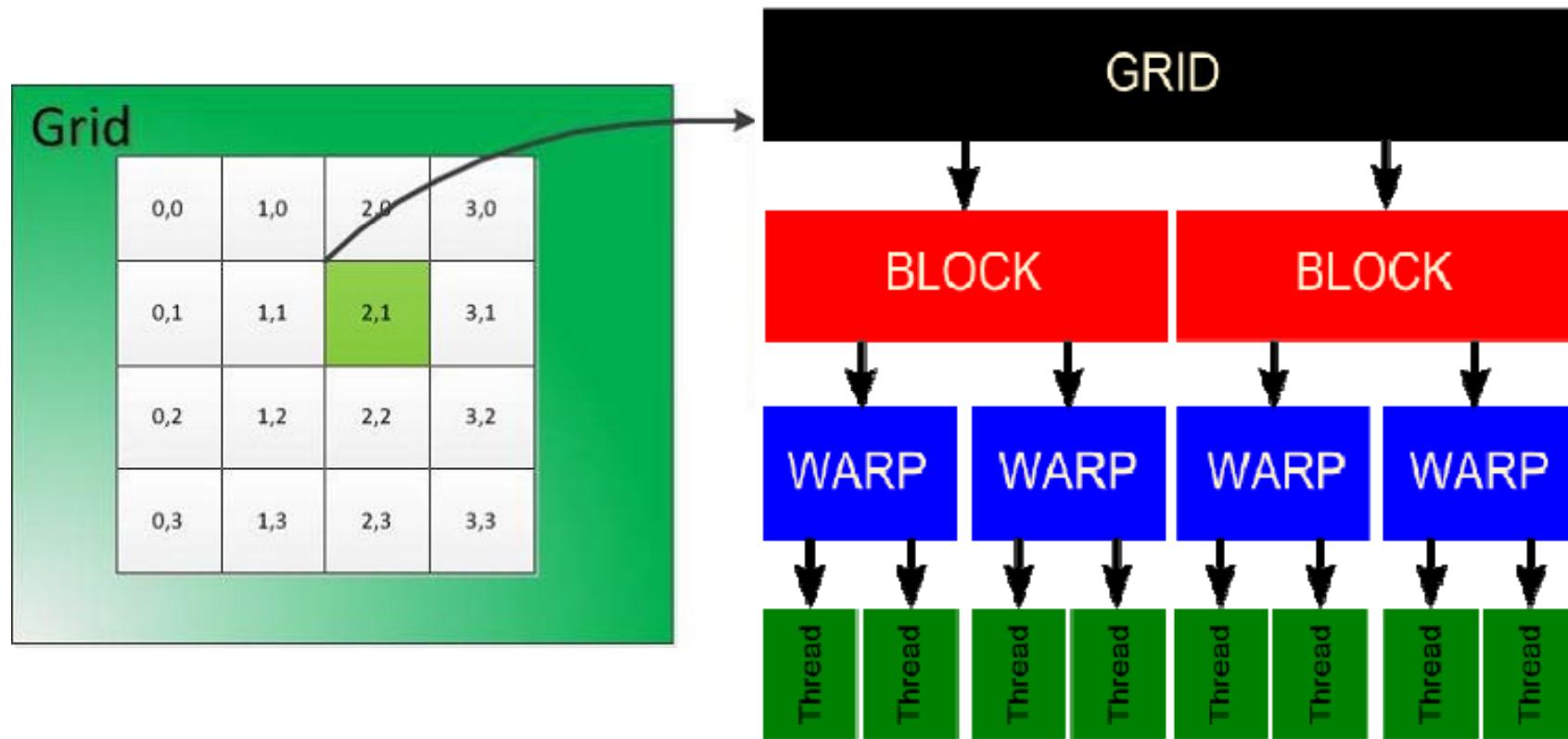


# Thread Batching: Grids and Blocks

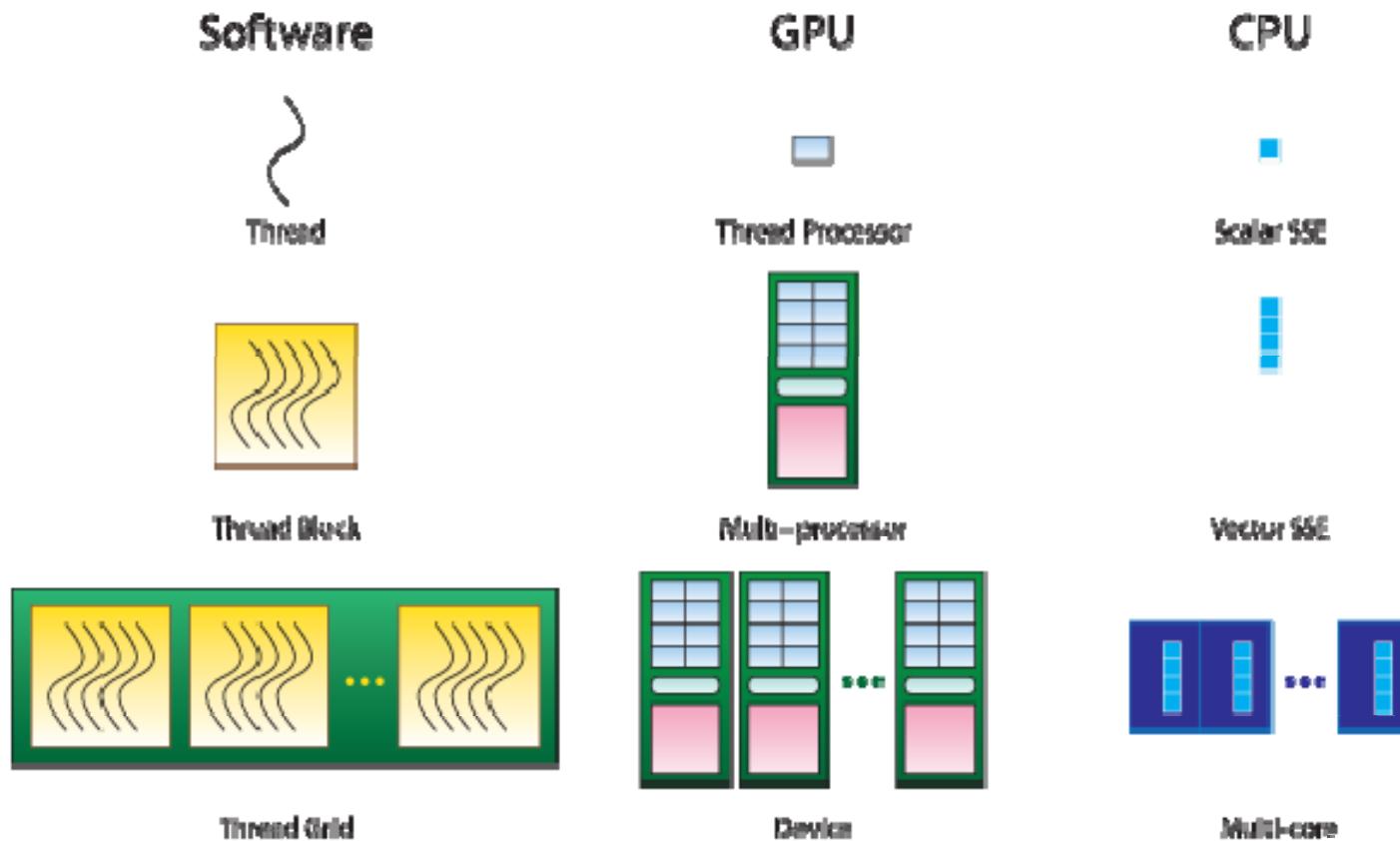
- ▶ Kernel executed as a grid of thread blocks
  - ▶ All threads share data memory space
- ▶ Thread block is a batch of threads, can cooperate with each other by:
  - ▶ Synchronizing their execution:  
For hazard-free shared memory accesses
  - ▶ Efficiently sharing data through a low latency shared memory
- ▶ Two threads from two different blocks cannot cooperate
  - ▶ (Unless thru slow global memory)



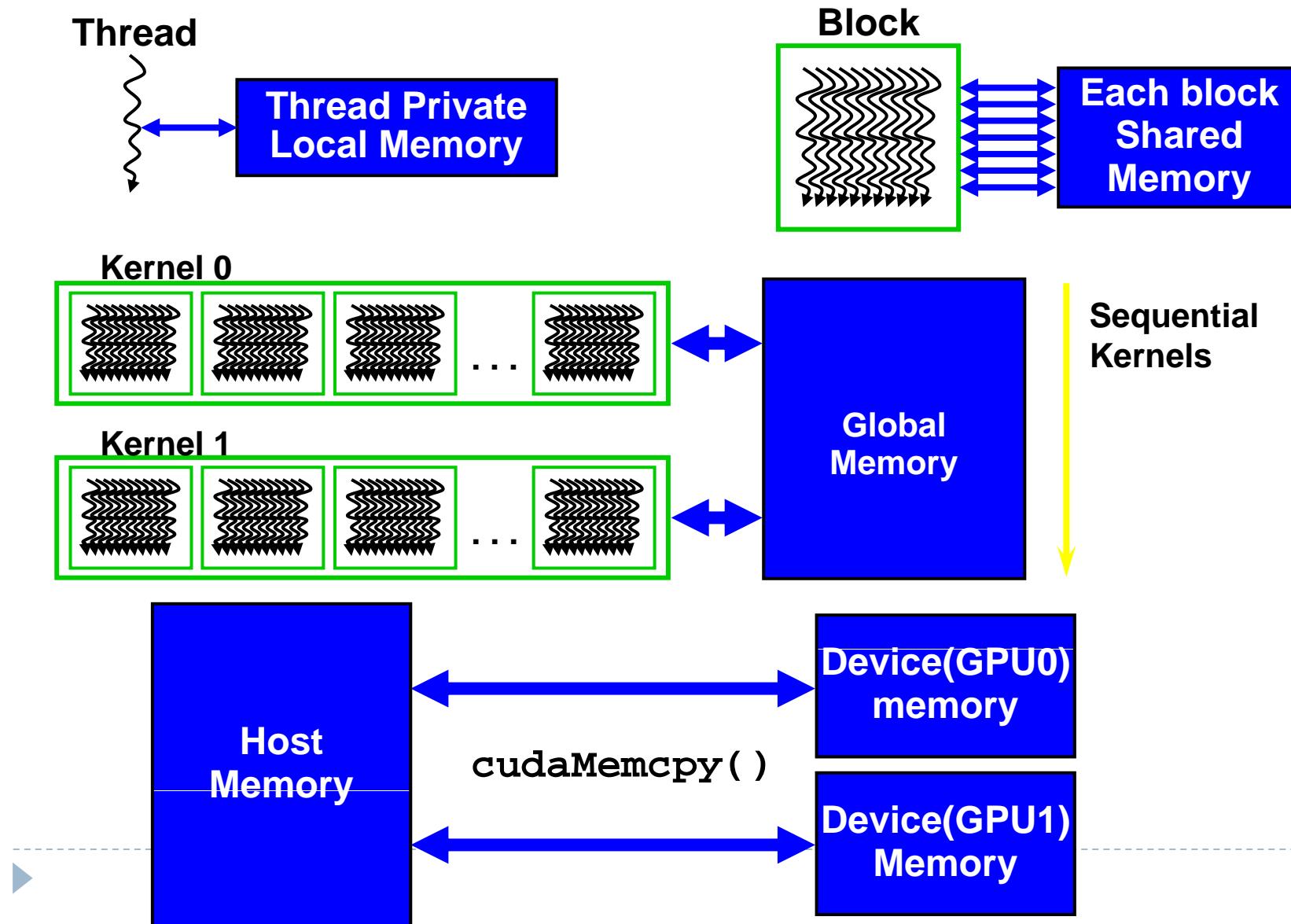
# GPU Threads Organization



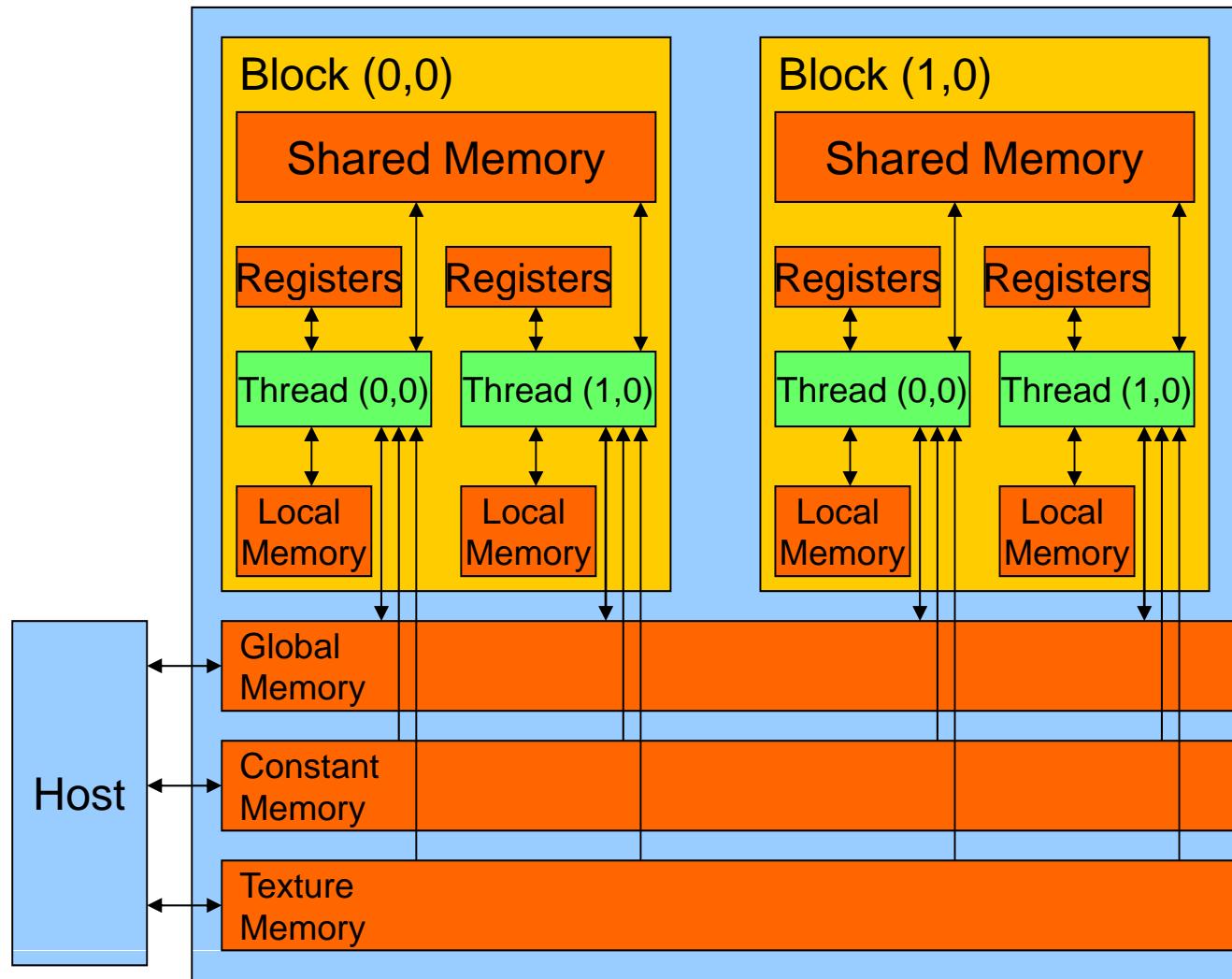
# GPU Threads Mapping to Hardware



# GPU Memory with Threads

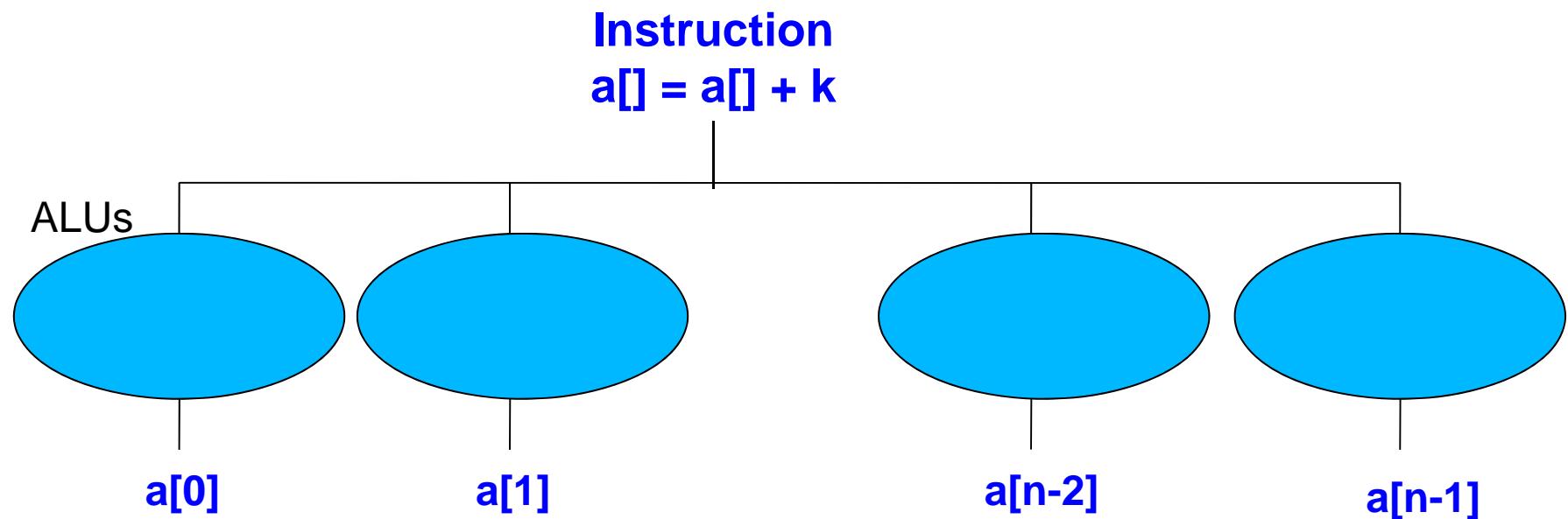


# GPU Memory Hierarchy Recall



# SIMD (Single Instruction Multiple Data)

Similar Idea with Data Partition/Different Level



# Extended C

---

- ▶ **Declspecs**
  - ▶ **global, device, shared, local, constant**
- ▶ **Keywords**
  - ▶ **threadIdx, blockIdx**
- ▶ **Intrinsics**
  - ▶ **\_syncthreads**
- ▶ **Runtime API**
  - ▶ **Memory, symbol, execution management**
- ▶ **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image) {

 __shared__ float region[M];
 ...

 region[threadIdx] = image[i];

 __syncthreads();
 ...
 image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```



# CUDA Function Declarations

---

|                                            | Executed on<br>the: | Only callable<br>from the: |
|--------------------------------------------|---------------------|----------------------------|
| <code>__device__ float DeviceFunc()</code> | device              | device                     |
| <code>__global__ void KernelFunc()</code>  | device              | Host                       |
| <code>__host__ float HostFunc()</code>     | host                | Host                       |

- ▶ **`__global__` defines a kernel function**
  - ▶ Must return `void`
- ▶ `__device__` and `__host__` can be used together



# **GPU Architecture in detail and Performance Optimization (Part II)**

Bin ZHOU 2014/06

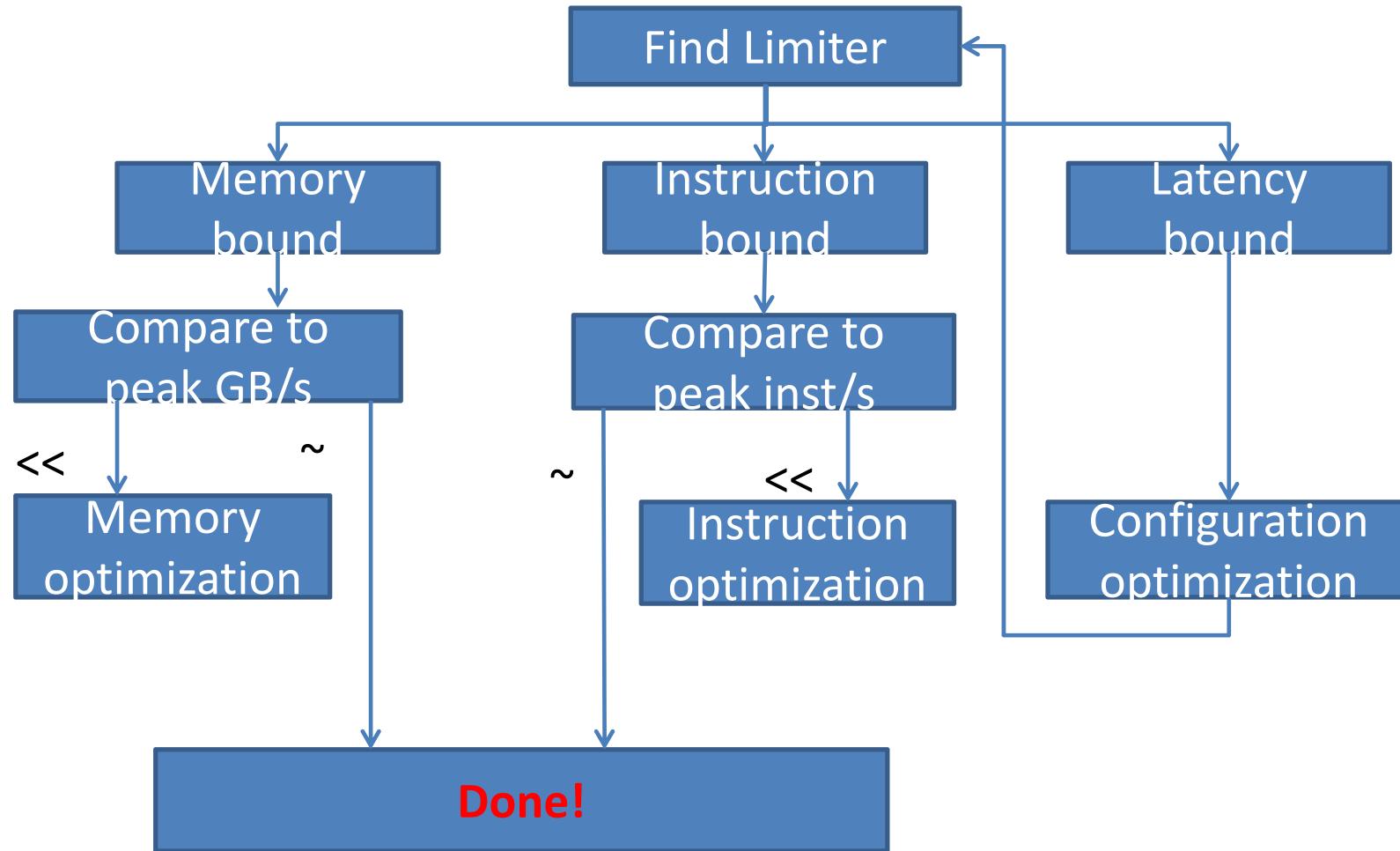
# Optimization

## Outline

- General guideline II
- CPU-GPU Interaction
- Kepler in detail

# **GENERAL GUIDELINE II**

# Kernel Optimization Workflow



# **General Optimization Strategies: Measurement**

- **Find out the limiting factor in kernel performance**
  - Memory bandwidth bound (memory optimization)
  - Instruction throughput bound (instruction optimization)
  - Latency bound (configuration optimization)
- **Measure effective memory/instruction throughput**

# **Memory Optimization**

- If the code is memory-bound and effective memory throughput is much lower than the peak
- Purpose: access only data that are absolutely necessary
- Major techniques
  - Improve access pattern to reduce wasted transactions
  - Reduce redundant access: read-only cache, shared memory

# Instruction Optimization

- **If you find out the code is instruction bound**
  - Compute-intensive algorithm can easily become memory-bound if not careful enough
  - Typically, worry about instruction optimization after memory and execution configuration optimizations
- **Purpose: reduce instruction count**
  - Use less instructions to get the same job done
- **Major techniques**
  - Use high throughput instructions (ex. wider load)
  - Reduce wasted instructions: branch divergence, reduce replay (conflict), etc.

# Latency Optimization

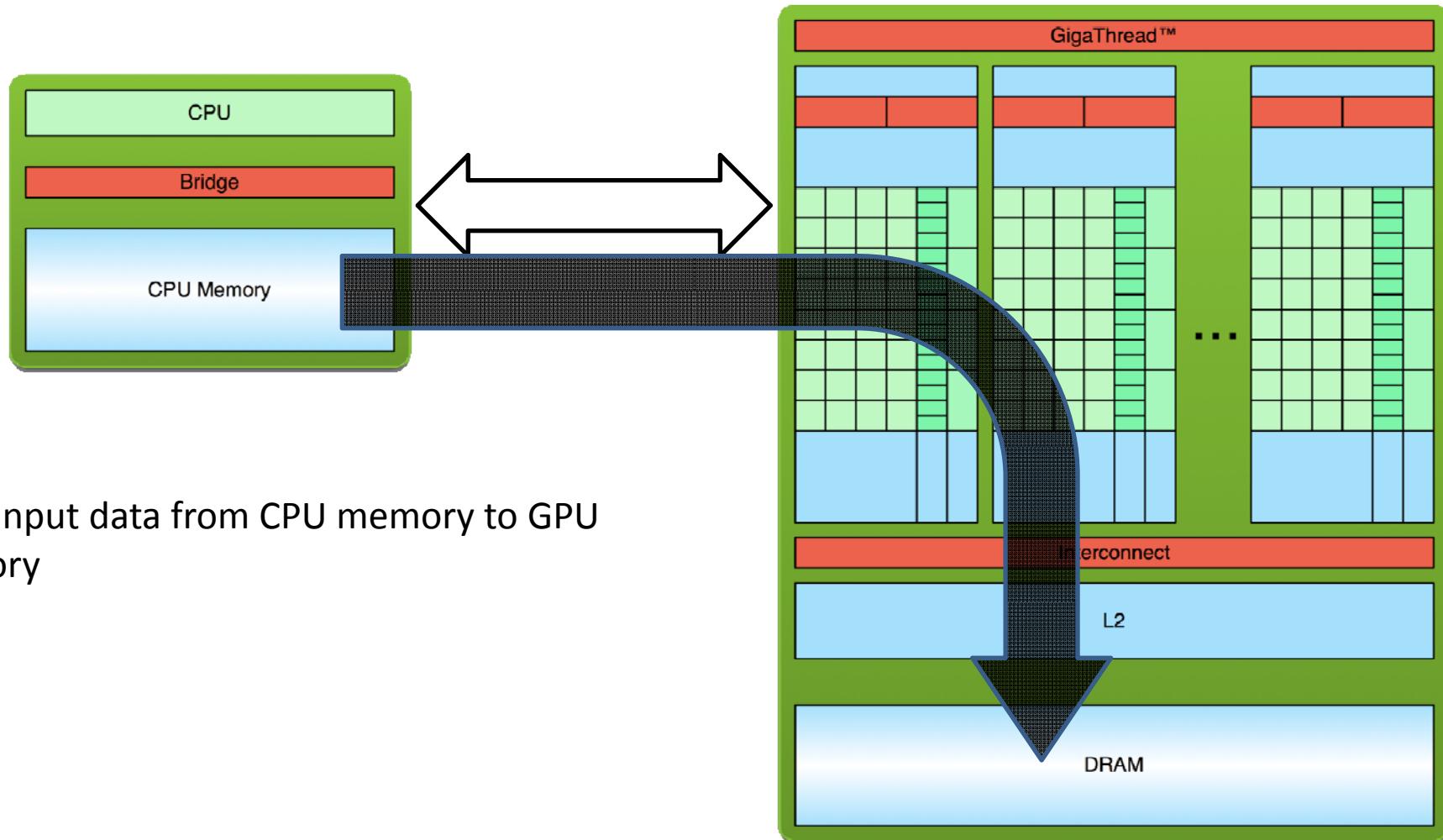
- **When the code is latency bound**
  - Both the memory and instruction throughputs are far from the peak
- **Latency hiding: switching threads**
  - A thread blocks when one of the operands isn't ready
- **Purpose: have enough warps to hide latency**
- **Major techniques: increase active warps, increase ILP**

# CPU-GPU INTERACTION

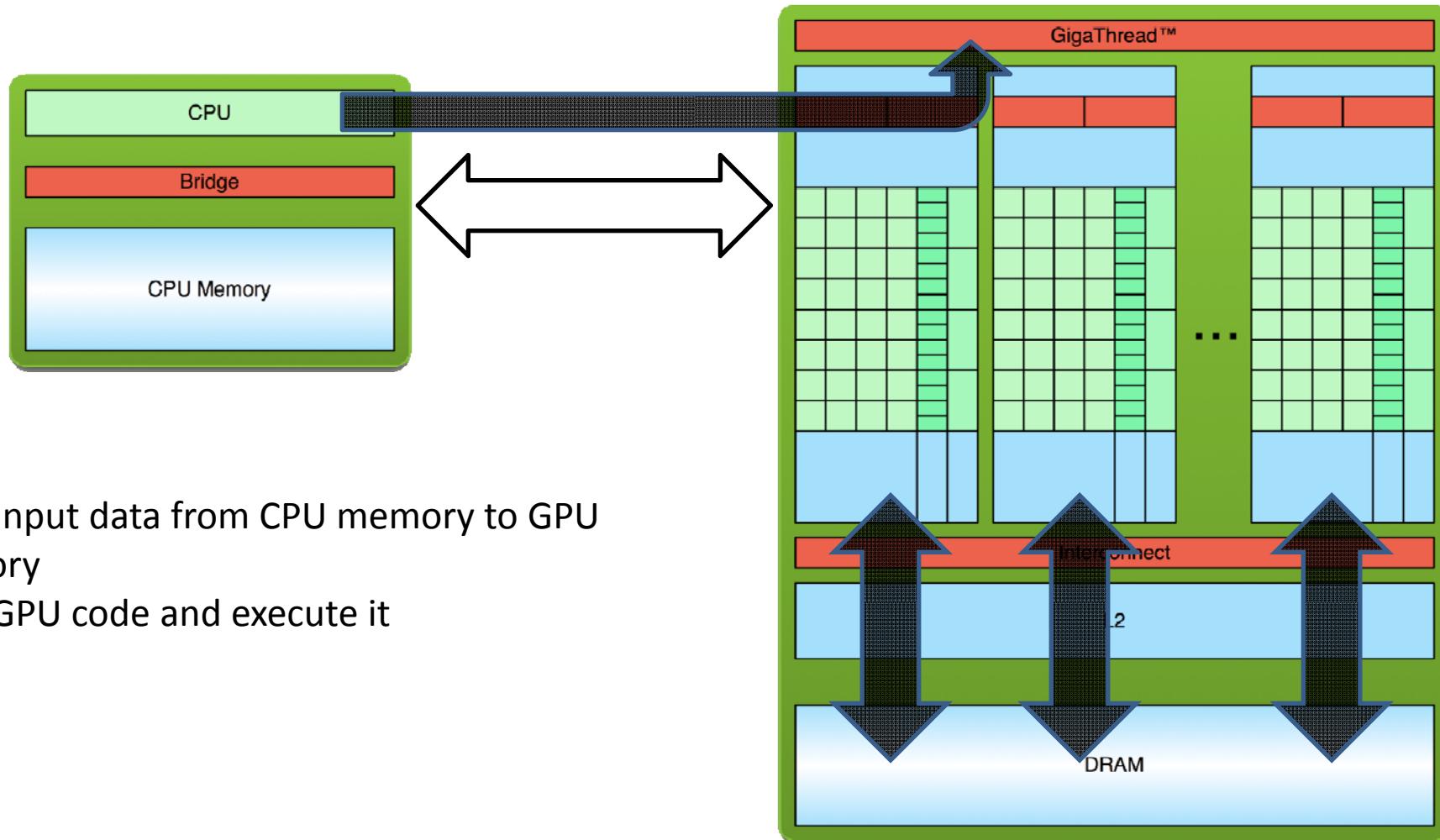
# **Minimize CPU-GPU data transfer**

- Host<->device data transfer has much lower bandwidth than global memory access.
  - 16 GB/s (PCIe x16 Gen3) vs 250 GB/s & 3.95 Tinst/s (GK110)
- **Minimize transfer**
  - Intermediate data can be allocated, operated, de-allocated directly on GPU
  - Sometimes it's even better to recompute on GPU
  - Move CPU codes to GPU that do not have performance gains if it can reduce data transfer
- **Group transfer**
  - One large transfer much better than many small ones
  - Overlap memory transfer with computation

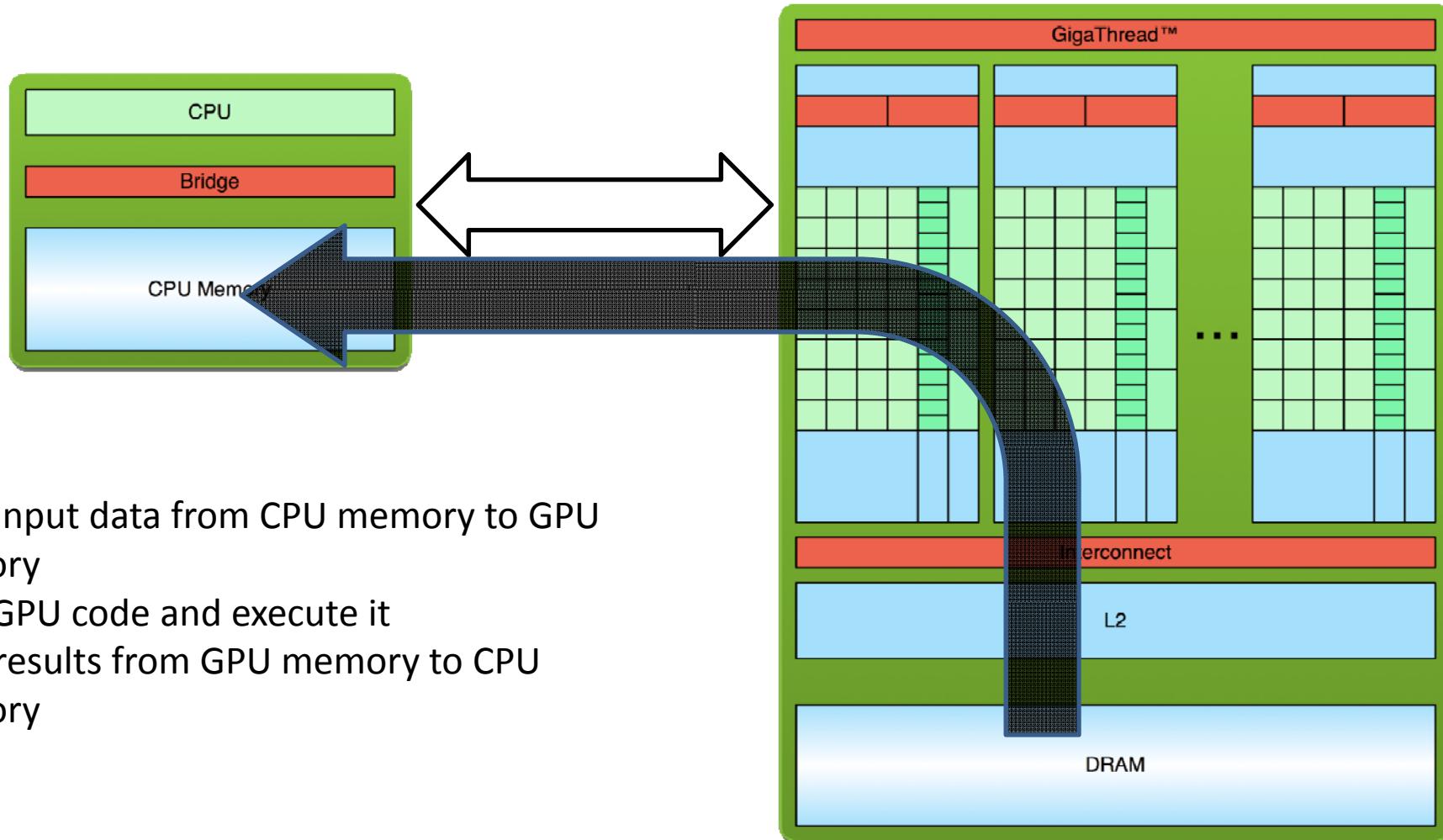
# Revisit GPU Processing Flow



# Revisit GPU Processing Flow



# Revisit GPU Processing Flow



# CUDA

- $T_{total} = T_{HtoD} + T_{Exec} + T_{DtoH}$
- More Overlap?

# CUDA



Stream 1



Stream 2

# Stream Example

```
cudaStreamCreate(&stream1);

cudaMemcpyAsync(dst1, src1, size, cudaMemcpyHostToDevice, stream1); <img alt="Blue curved arrow pointing from cudaMemcpyAsync to stream1" data-bbox="950 425 1000 550"

kernel<<<grid, block, 0, stream1>>>(...);

cudaMemcpyAsync(dst1, src1, size, stream1); <img alt="Blue curved arrow pointing from cudaMemcpyAsync to stream1" data-bbox="700 515 750 625

cudaStreamSynchronize(stream1);</pre>
```

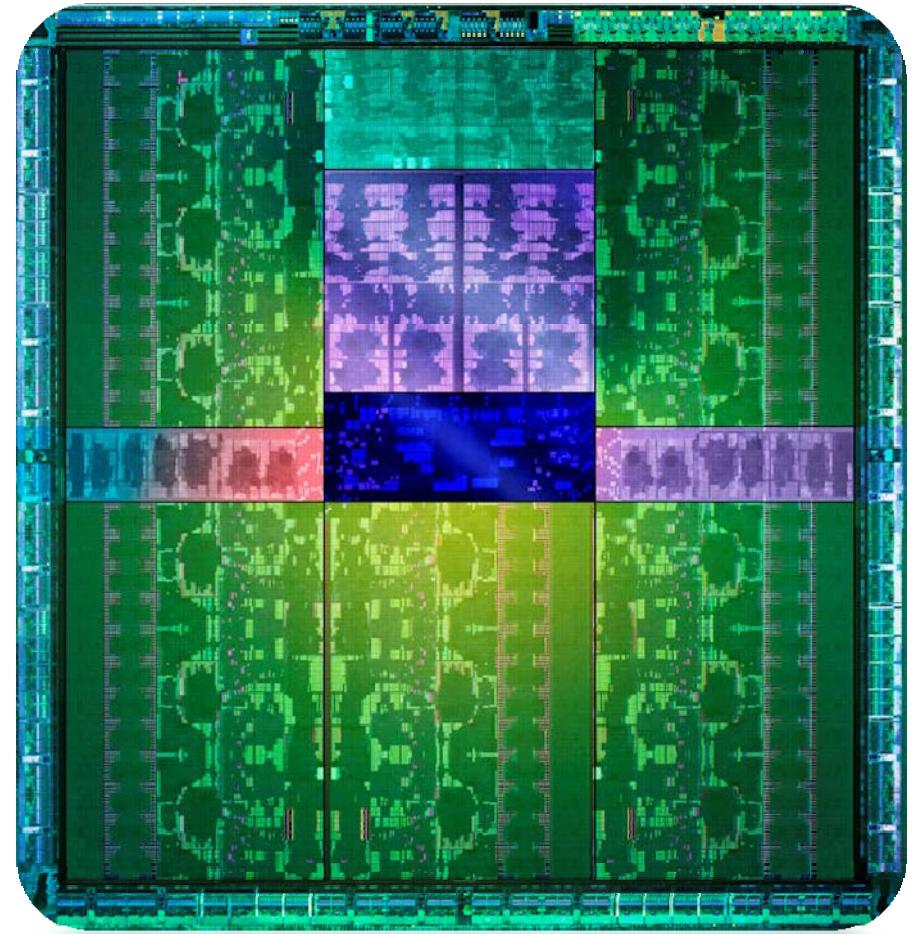
# Stream Example

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst1, src1, size, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(dst2, src2, size, cudaMemcpyHostToDevice, stream2);
kernel<<<grid, block, 0, stream1>>>(...);
kernel<<<grid, block, 0, stream2>>>(...);
cudaMemcpyAsync(dst1, src1, size, cudaMemcpyDeviceToHost, stream1);
cudaMemcpyAsync(dst2, src2, size, cudaMemcpyDeviceToHost, stream2);
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

# **KEPLER IN DETAIL**

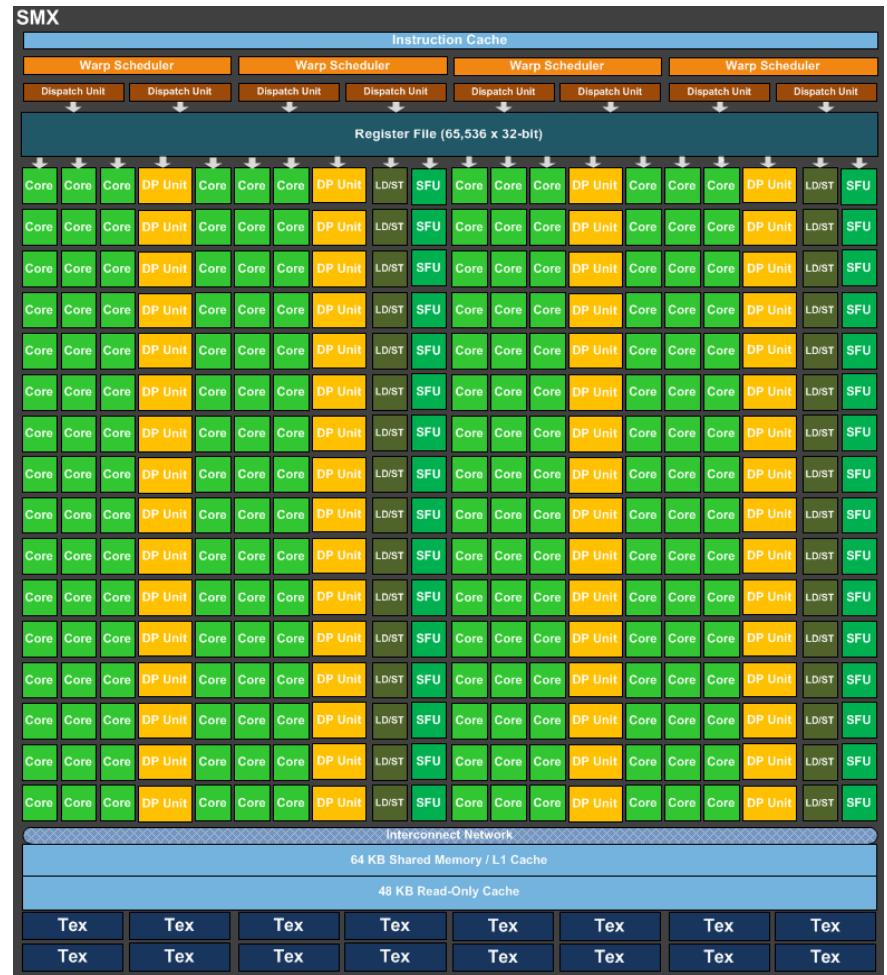
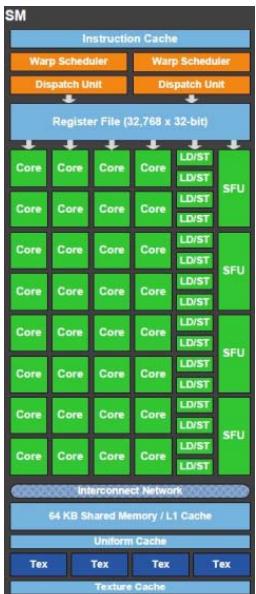
# Kepler

- **NVIDIA Kepler**
  - 1.31 tflops double precision
  - 3.95 tflops single precision
  - 250 gb/sec memory bandwidth
  - 2,688 Functional Units (cores)
- **≈ #1 on Top500 in 1997**



NVIDIA GK110 - Kepler

# Kepler GK110 SMX vs Fermi SM



3x perf  
Power goes down!

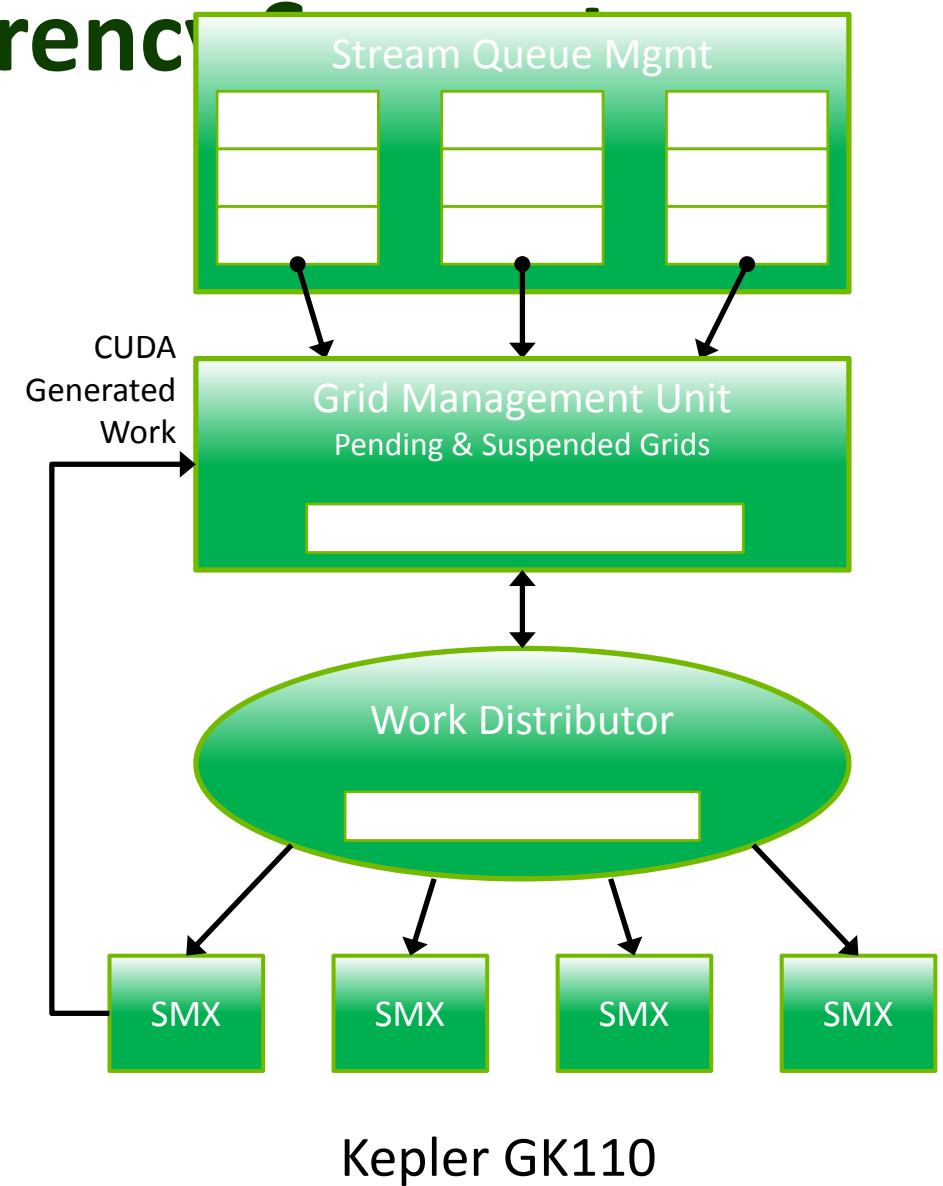
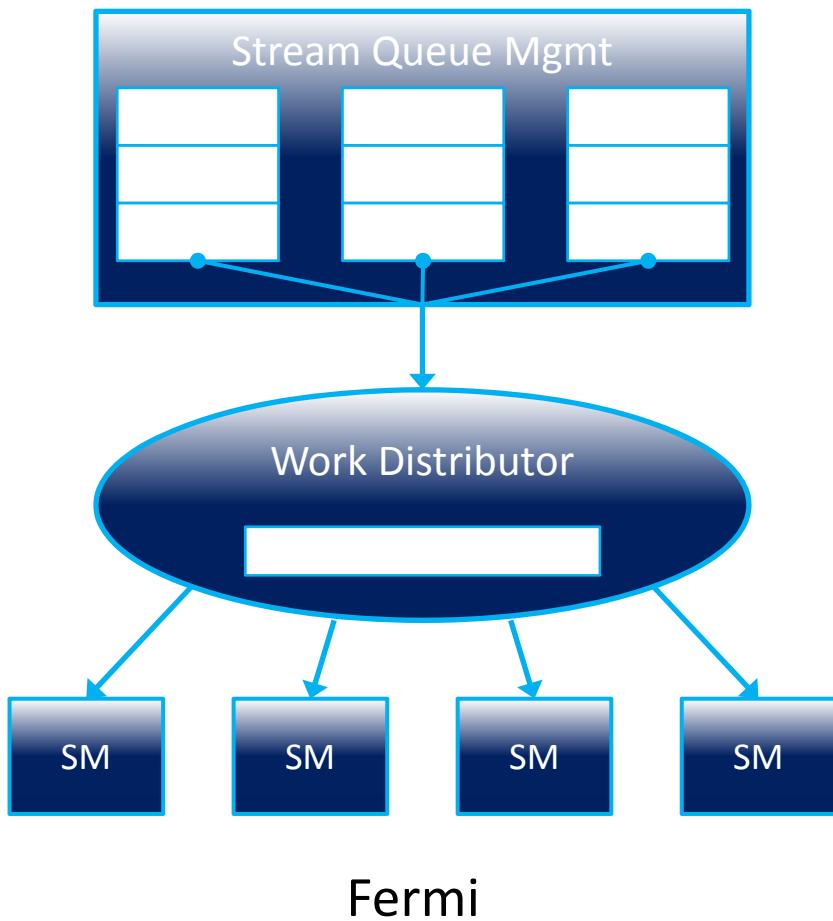
# New ISA Encoding: 255 Registers per Thread

- **Fermi limit: 63 registers per thread**
  - A common Fermi performance limiter
  - Leads to excessive spilling
- **Kepler : Up to 255 registers per thread**
  - Especially helpful for FP64 apps

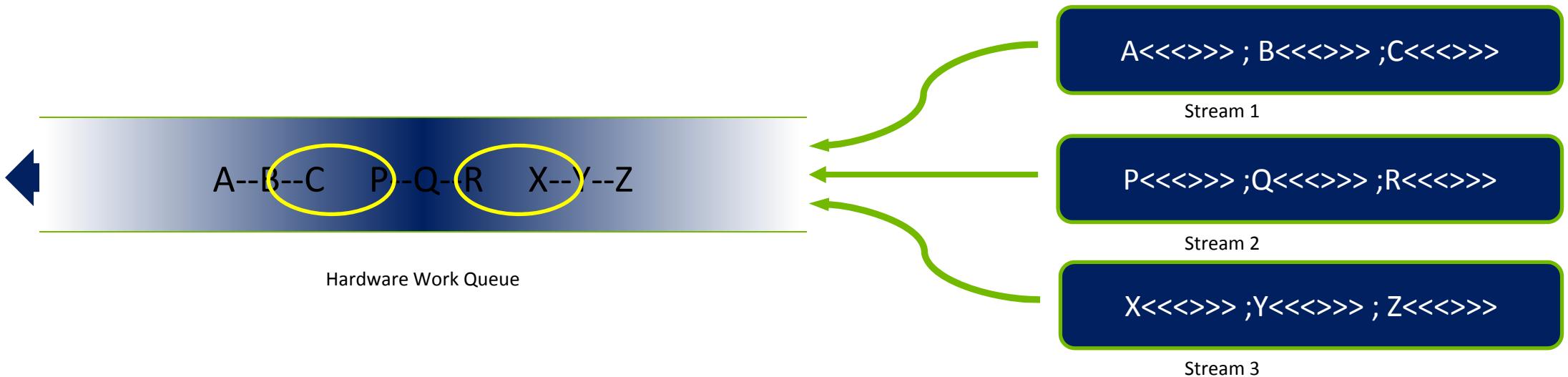
# Hyper-Q

- *Feature of Kepler K20 GPUs to increase application throughput by enabling work to be scheduled onto the GPU in parallel*
- **Two ways to take advantage**
  - CUDA Streams – now they really are concurrent
  - CUDA Proxy for MPI – concurrent CUDA MPI processes on one GPU

# Better Concurrency



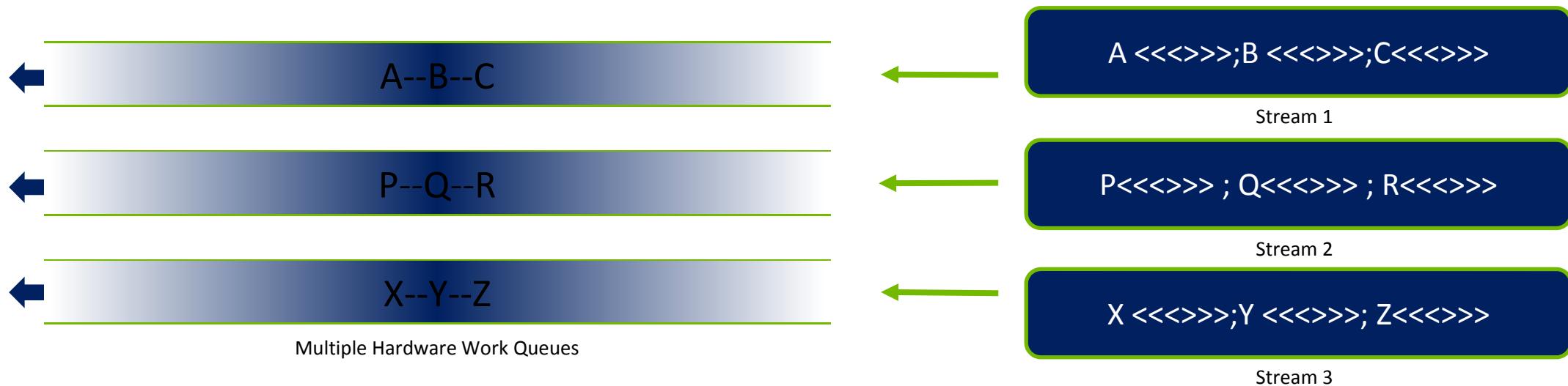
# Fermi Concurrency



## Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But CUDA streams multiplex into a single queue
- Overlap only at stream edges

# Kepler Improved Concurrency



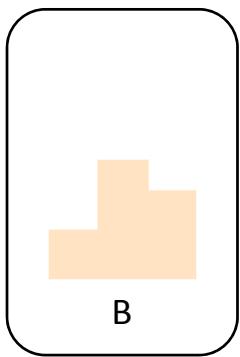
**Kepler allows 32-way concurrency**

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

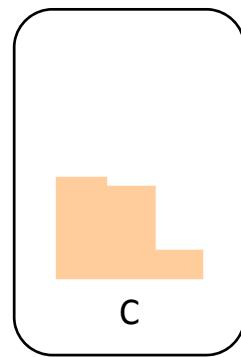
# Fermi: Time-Division Multiprocess



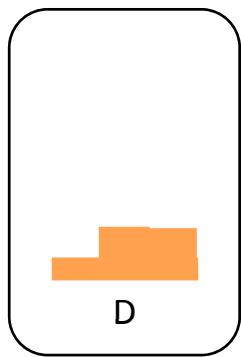
A



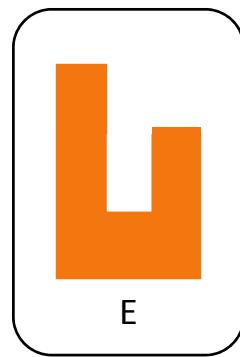
B



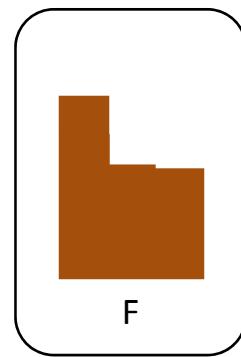
C



D



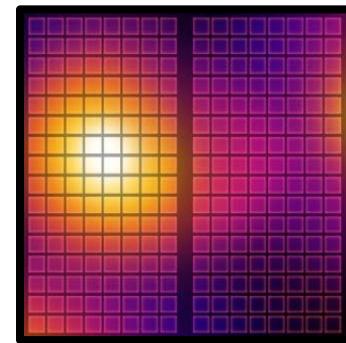
E



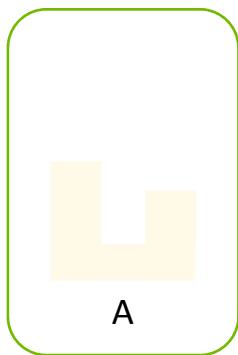
F

CPU Processes

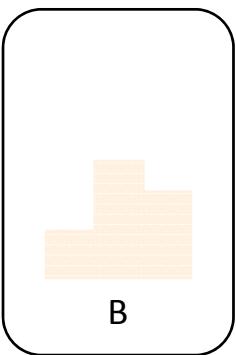
Shared GPU



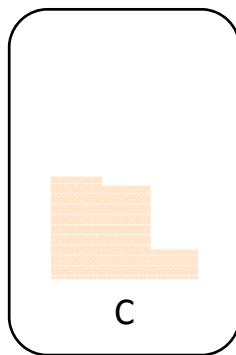
# Fermi: Time-Division Multiprocess



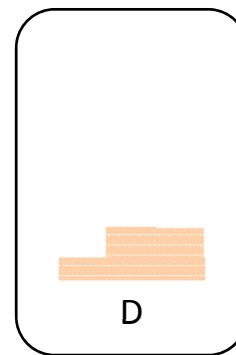
A



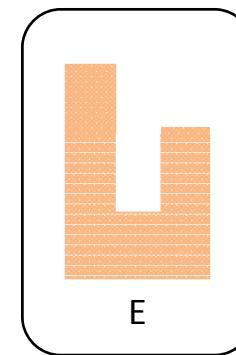
B



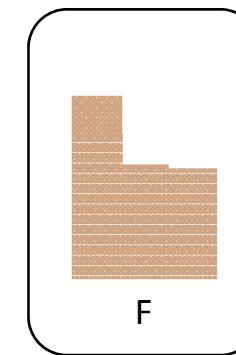
C



D



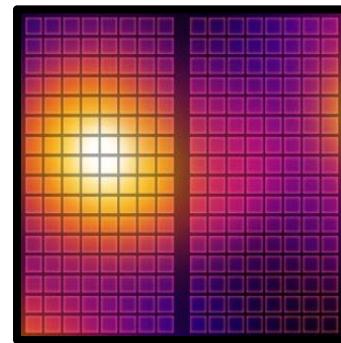
E



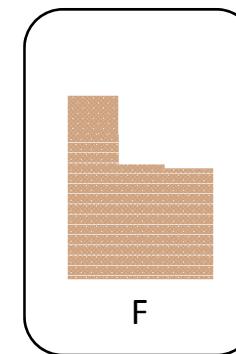
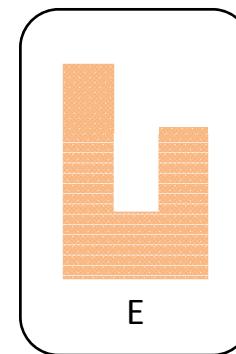
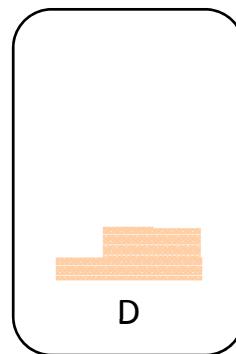
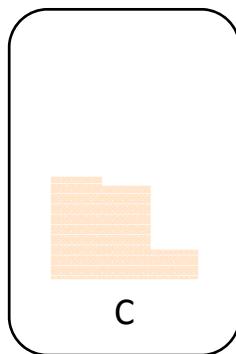
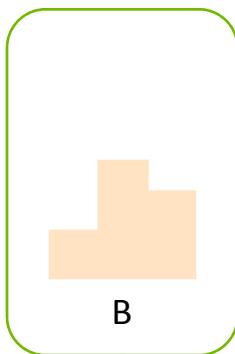
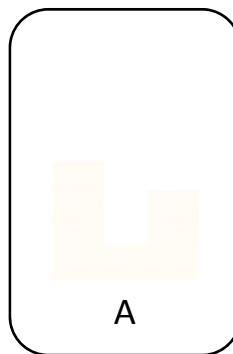
F

CPU Processes

Shared GPU

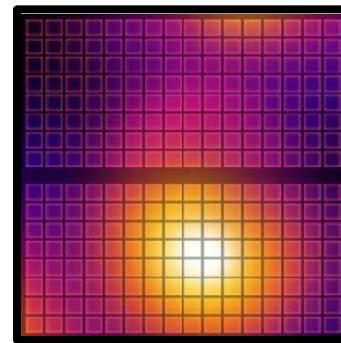


# Fermi: Time-Division Multiprocess

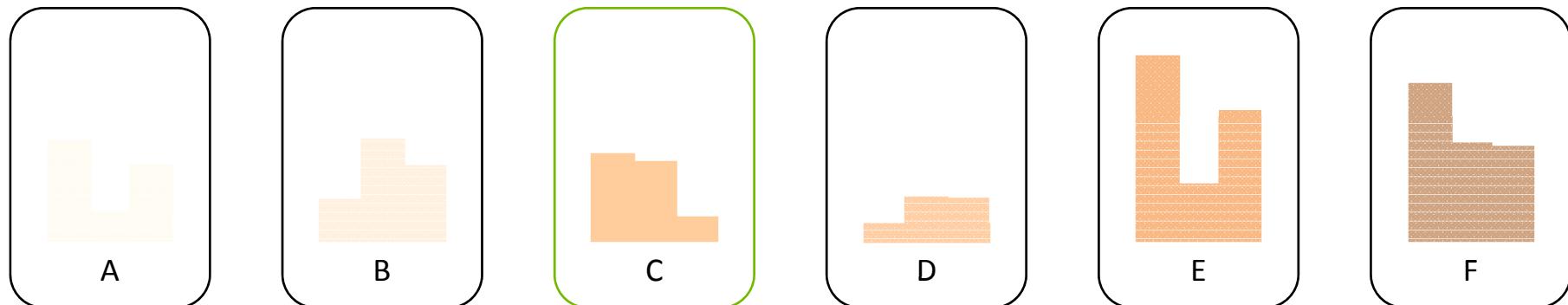


CPU Processes

Shared GPU

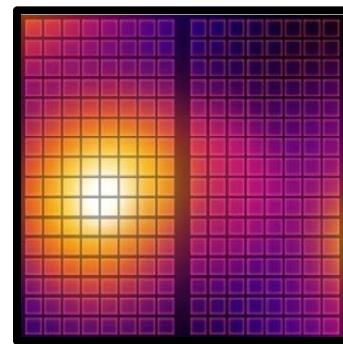


# Fermi: Time-Division Multiprocess

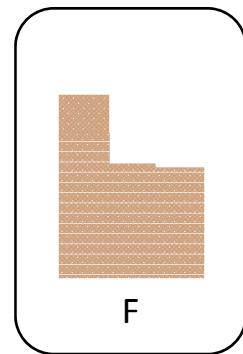
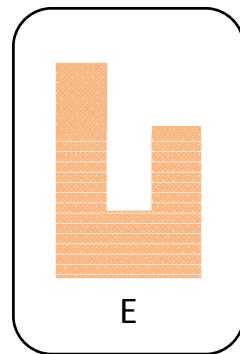
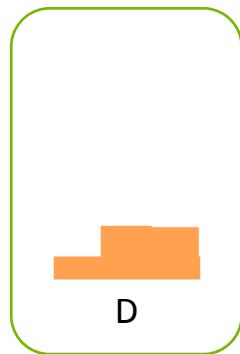
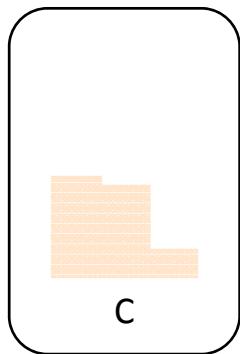
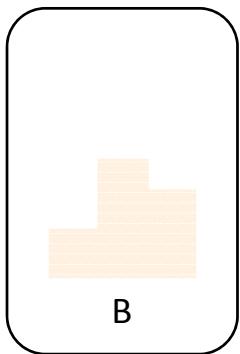
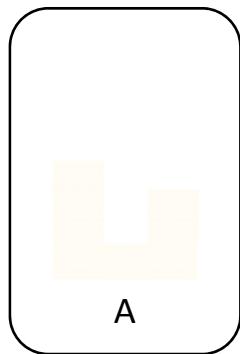


CPU Processes

Shared GPU

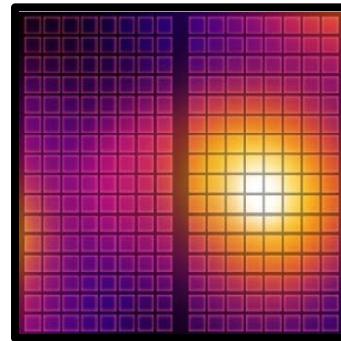


# Fermi: Time-Division Multiprocess

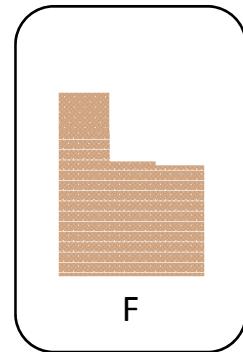
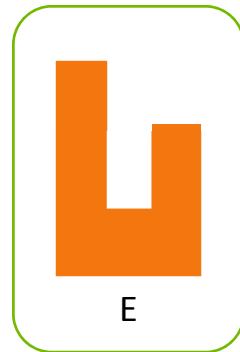
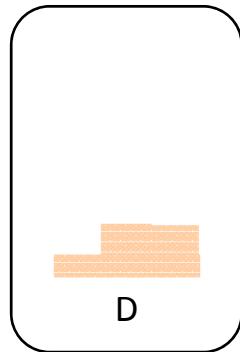
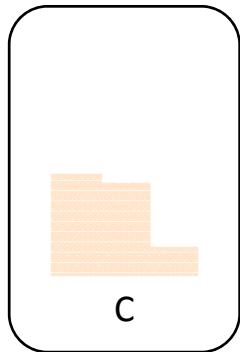
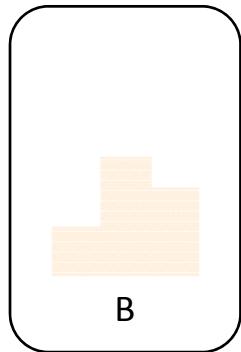
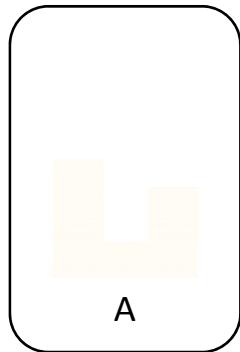


CPU Processes

Shared GPU

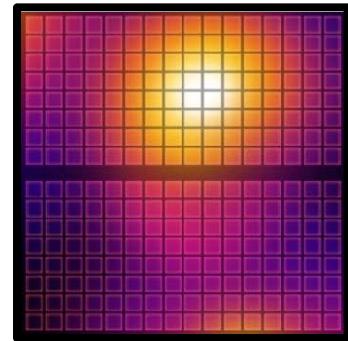


# Fermi: Time-Division Multiprocess

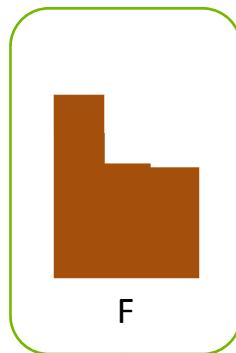
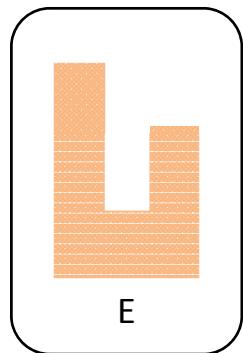
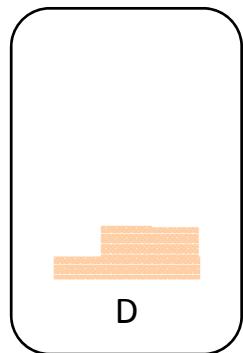
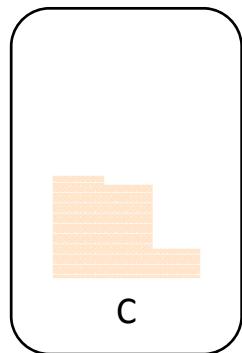
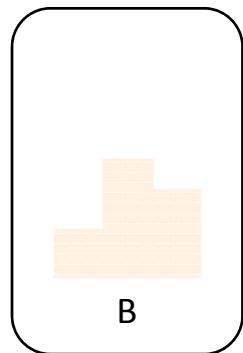
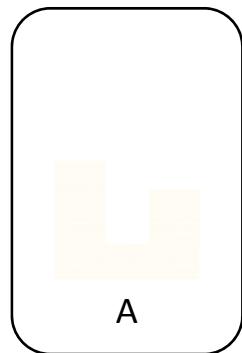


CPU Processes

Shared GPU

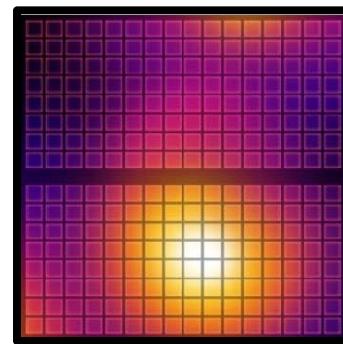


# Fermi: Time-Division Multiprocess

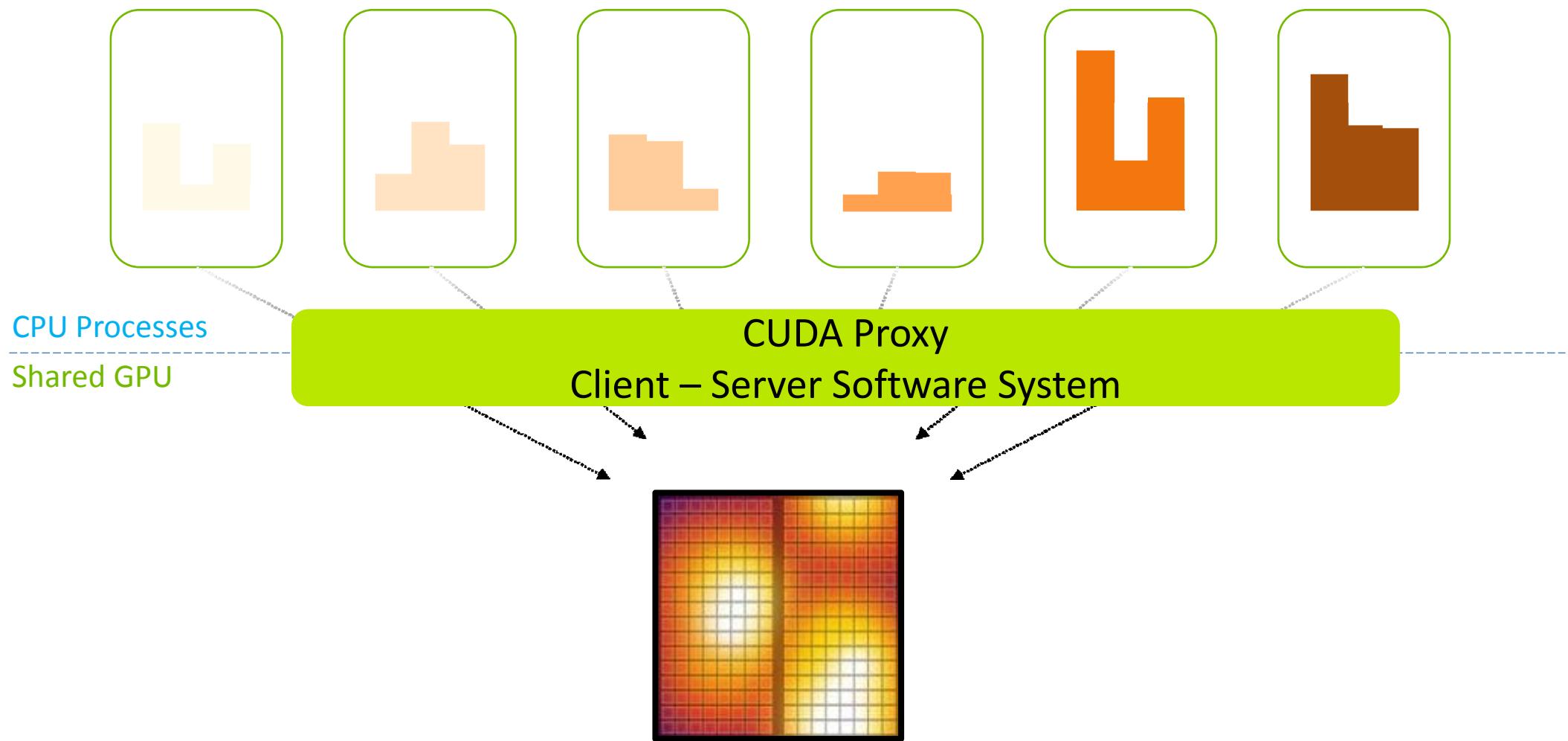


CPU Processes

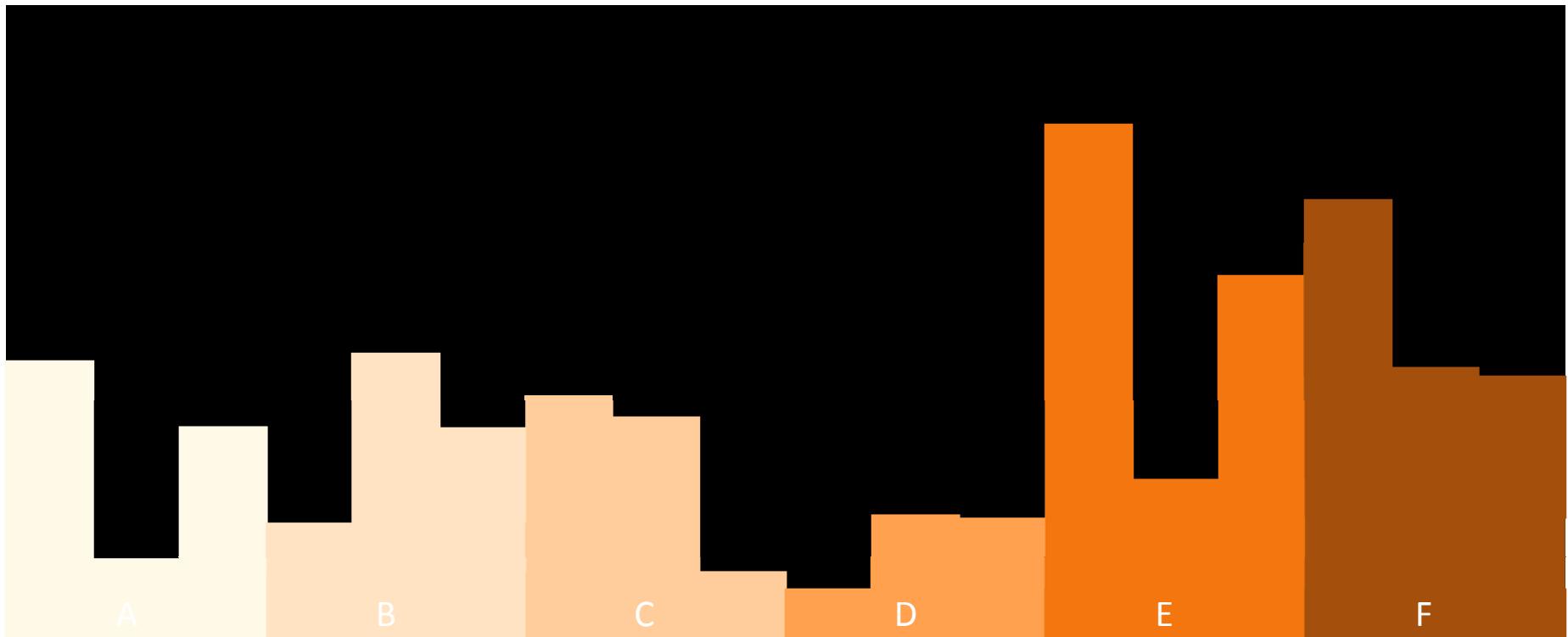
Shared GPU



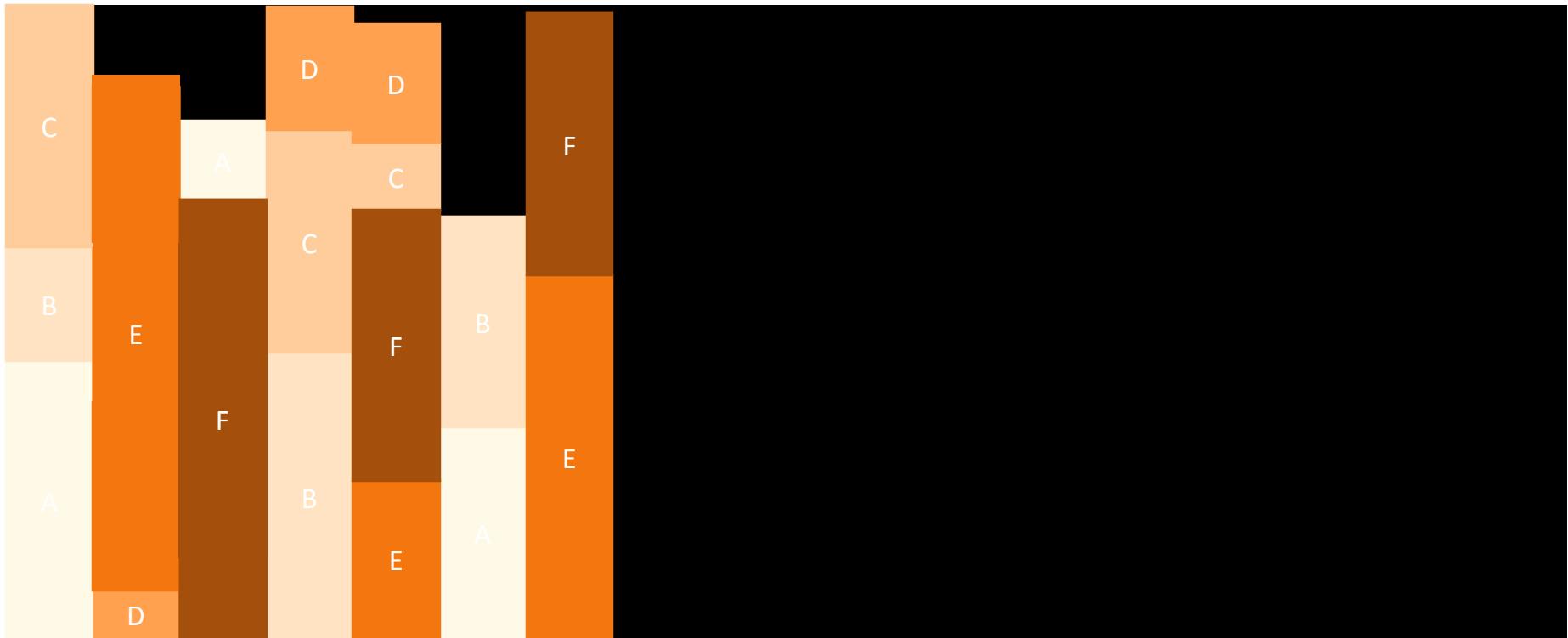
# Hyper-Q: Simultaneous Multiprocess



# Without Hyper-Q



# With Hyper-Q



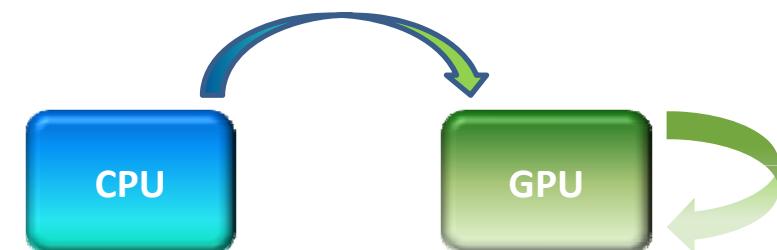
# What is Dynamic Parallelism?

**The ability to launch new kernels from the GPU**

- Dynamically - based on run-time data
- Simultaneously - from multiple threads at once
- Independently - each thread can launch a different grid



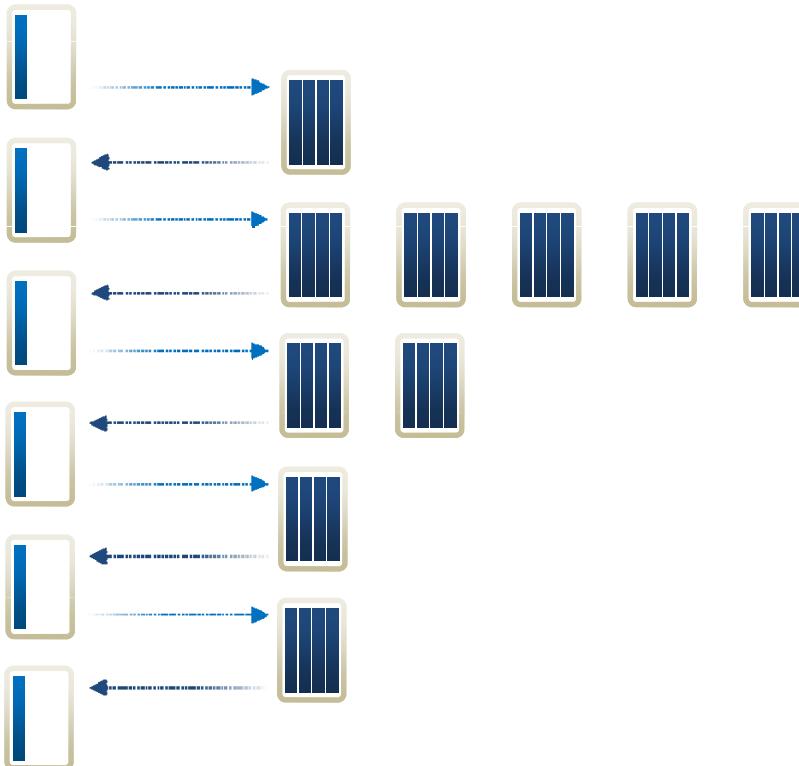
*Fermi: Only CPU can generate GPU work*



*Kepler: GPU can generate work for itself*

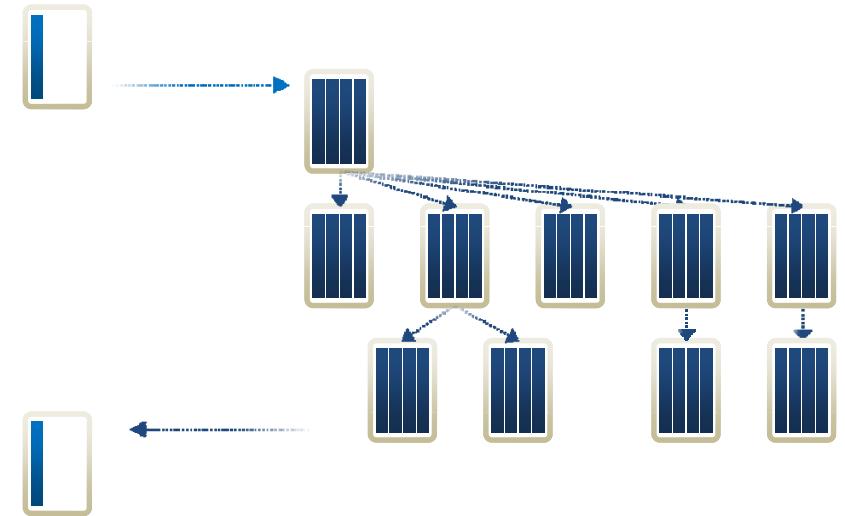
# What Does It Mean?

CPU      GPU



*GPU as Co-Processor*

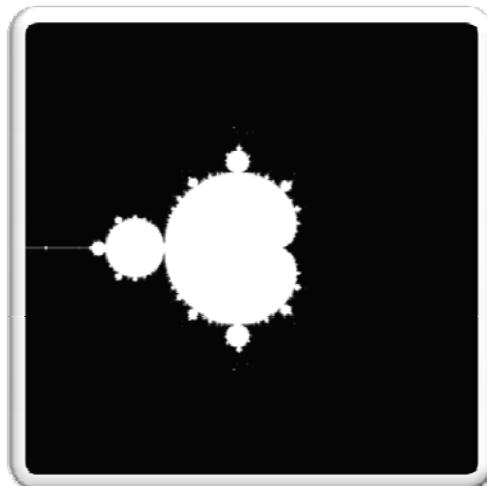
CPU      GPU



*Autonomous, Dynamic Parallelism*

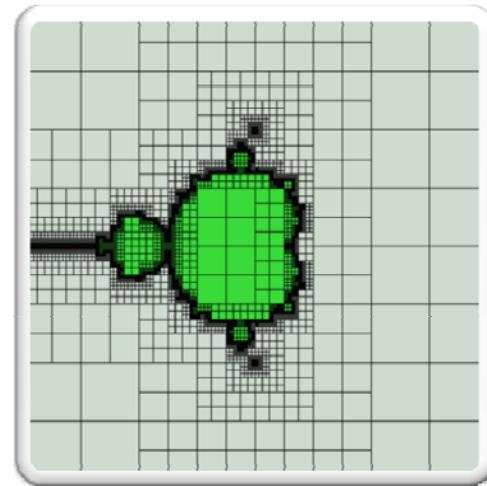
# New Types of Algorithms

- Recursive Parallel Algorithms like Quick sort
- Adaptive Mesh Algorithms like Mandelbrot



CUDA Today

Computational Power  
allocated to regions of  
interest



CUDA on Kepler

# Familiar Programming Model

```
int main() {
 float *data;
 setup(data);

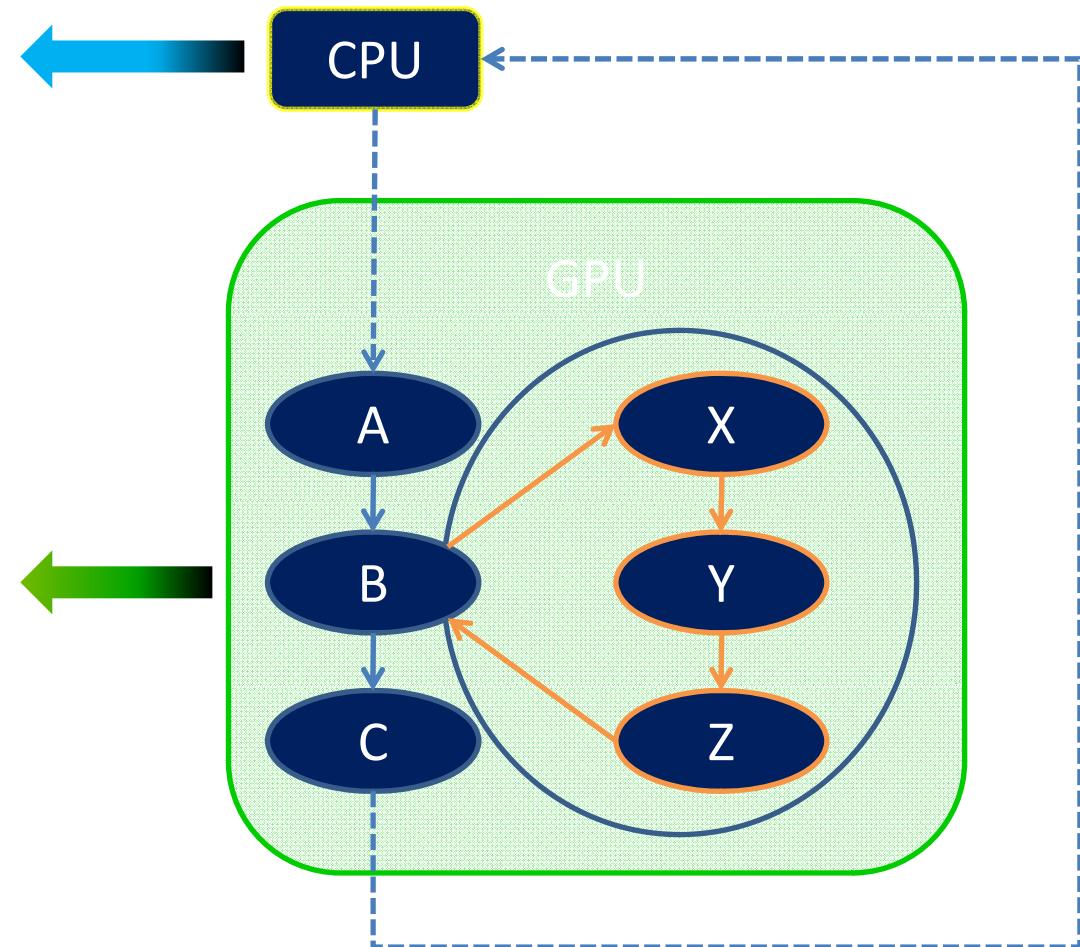
 A <<< ... >>> (data);
 B <<< ... >>> (data);
 C <<< ... >>> (data);

 cudaDeviceSynchronize();
 return 0;
}
```

```
__global__ void B(float *data)
{
 do_stuff(data);

 X <<< ... >>> (data);
 Y <<< ... >>> (data);
 Z <<< ... >>> (data);
 cudaDeviceSynchronize();

 do_more_stuff(data);
}
```



# Programming Model

- Launch is per-thread and asynchronous

## Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
 int tid = threadIdx.x;
 if(tid % 2)
 buf[tid/2] = data[tid]+data[tid+1];

 __syncthreads();

 if(tid == 0) {
 launch<<< 128, 256 >>>(buf);
 cudaDeviceSynchronize();
 }
 __syncthreads();

 cudaMemcpyAsync(data, buf, 1024);
 cudaDeviceSynchronize();
}
```

# Programming Model

- Launch is per-thread and asynchronous
- CUDA primitives are per-block
- launched kernels and CUDA objects like streams are visible to all threads in a thread block
- cannot be passed to child kernel

## Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
 int tid = threadIdx.x;
 if(tid % 2)
 buf[tid/2] = data[tid]+data[tid+1];

 __syncthreads();

 if(tid == 0) {
 launch<<< 128, 256 >>>(buf);
 cudaDeviceSynchronize();
 }
 __syncthreads();

 cudaMemcpyAsync(data, buf, 1024);
 cudaDeviceSynchronize();
}
```

# Programming Model

- Launch is per-thread and asynchronous
- CUDA primitives are per-block
- Sync includes all launches by any thread in the block

## Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
 int tid = threadIdx.x;
 if(tid % 2)
 buf[tid/2] = data[tid]+data[tid+1];

 __syncthreads();

 if(tid == 0) {
 launch<<< 128, 256 >>>(buf);
 cudaDeviceSynchronize();
 }
 __syncthreads();

 cudaMemcpyAsync(data, buf, 1024);
 cudaDeviceSynchronize();
}
```

# Programming Model

- Launch is per-thread and asynchronous
- CUDA primitives are per-block
- Sync includes all launches by any thread in the block
- `cudaDeviceSynchronize()` does not imply `syncthreads()`

## Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
 int tid = threadIdx.x;
 if(tid % 2)
 buf[tid/2] = data[tid]+data[tid+1];

 __syncthreads();

 if(tid == 0) {
 launch<<< 128, 256 >>>(buf);
 cudaDeviceSynchronize();
 }
 __syncthreads();

 cudaMemcpyAsync(data, buf, 1024);
 cudaDeviceSynchronize();
}
```

# Memory Model

- Launch implies membar  
(child sees parent state at time of launch)

## Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
 int tid = threadIdx.x;
 if(tid % 2)
 buf[tid/2] = data[tid]+data[tid+1];

 __syncthreads();

 if(tid == 0) {
 launch<<< 128, 256 >>>(buf);
 cudaDeviceSynchronize();
 }
 __syncthreads();

 cudaMemcpyAsync(data, buf, 1024);
 cudaDeviceSynchronize();
}
```

# Memory Model

- Launch implies membar  
(child sees parent state at time of launch)
- Sync implies invalidate  
(parent sees child writes after sync)

## Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
 int tid = threadIdx.x;
 if(tid % 2)
 buf[tid/2] = data[tid]+data[tid+1];

 __syncthreads();

 if(tid == 0) {
 Launch<<< 128, 256 >>>(buf);
 cudaDeviceSynchronize();
 }
 __syncthreads();

 cudaMemcpyAsync(data, buf, 1024);
 cudaDeviceSynchronize();
}
```

# Memory Model

- Launch implies membar  
(child sees parent state at time of launch)
- Sync implies invalidate  
(parent sees child writes after sync)
- Local & shared memory are private
- Constants are immutable

## Code Example

```
__device__ float buf[1024];
__global__ void cnp(float *data)
{
 int tid = threadIdx.x;
 if(tid % 2)
 buf[tid/2] = data[tid]+data[tid+1];

 __syncthreads();

 if(tid == 0) {
 launch<<< 128, 256 >>>(buf);
 cudaDeviceSynchronize();
 }
 __syncthreads();
 if (tid == 0) {
 cudaMemcpyAsync(data, buf, 1024);
 cudaDeviceSynchronize();
 }
}
```

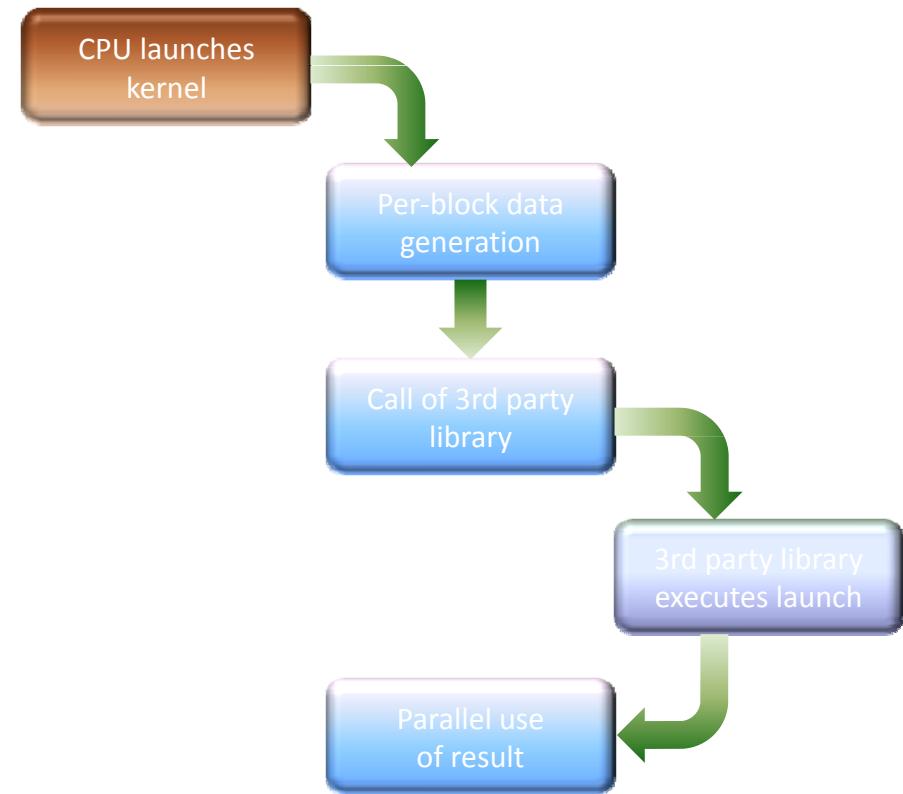
# Dynamic Parallelism and GPU Callable Libraries

```
__global__ void libraryCall (float *a,
 float *b,
 float *c)
{
 // All threads generate data
 createData(a, b);
 __syncthreads();

 // Only one thread calls library
 if(threadIdx.x == 0) {
 cublasDgemm(a, b, c);
 cudaDeviceSynchronize();
 }

 // All threads wait for dtrsm
 __syncthreads();

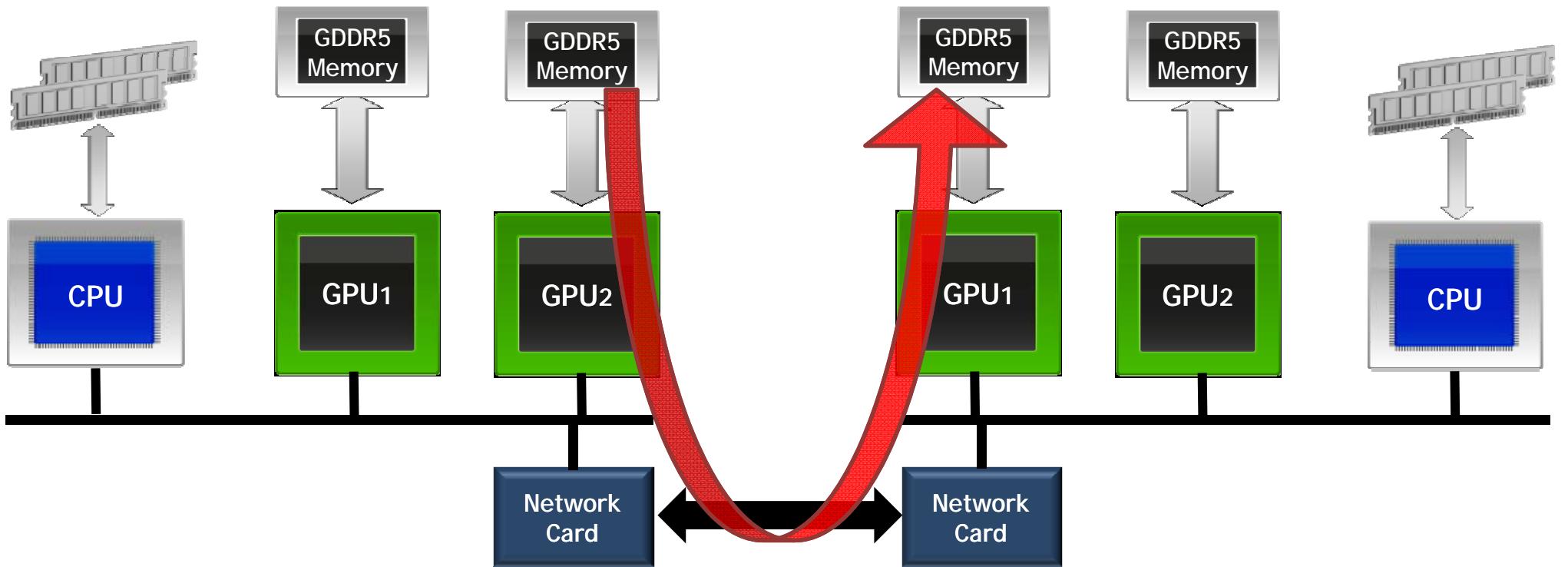
 // Now continue
 consumeData(c);
}
```



# NVIDIA GPUDirect™ RDMA

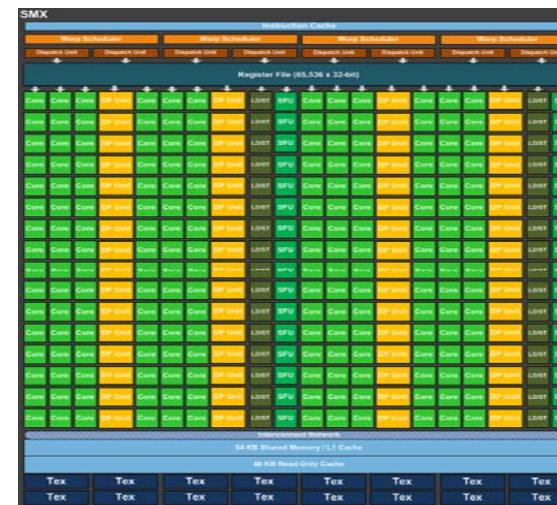
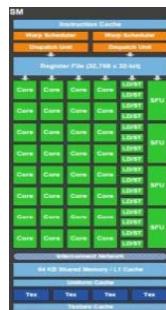
- *Provides technology necessary to enable lower latency memory transfers between GPU and other PCIe devices without requiring custom hardware.*
- API and documentation for device driver developers
- Available on Linux only
- Supported on Kepler Quadro and Tesla GPUs

# NVIDIA GPUDirect™ Now Supports RDMA



# More threads are needed

- 2-3x throughput per clock per SM
- Memory bandwidth increasing
- Bigger SM have bigger stomach!



# More thread are needed

- If you already launched enough threads, the following enhancement on kepler will ensure enough active warps on SMs.
- **2x register file on each SM**
  - E.g. 63 registers per thread, blockDim 256
  - In Fermi 16 active warps
  - In Kepler 32 active warps
- **2x simultaneous blocks per SM**
  - E.g. 16 registers per thread, blockDim 96
  - In Fermi  $96*8/32 = 24$  active warps
  - In Kepler  $96*16/32 = 46$  active warps
- **More flexible for shared memory configuration 16/32/48KB**

# If one kernel can't launch enough threads

- **Concurrent Kernels**
  - GK110 allows up to 32 concurrent kernels to execute.
- **Hyper-Q**
  - Using MPI, Different processes can use the device at the same time.
  - Using Stream, there's no inter-stream dependencies any more.
- **Dynamic Parallelism**
  - Threads can launch kernels

## Two GPUs on K10

- K10 is a dual-GK104 Gemini board.
- Appear as two separate CUDA devices.
- Need multi-GPU paradigm.

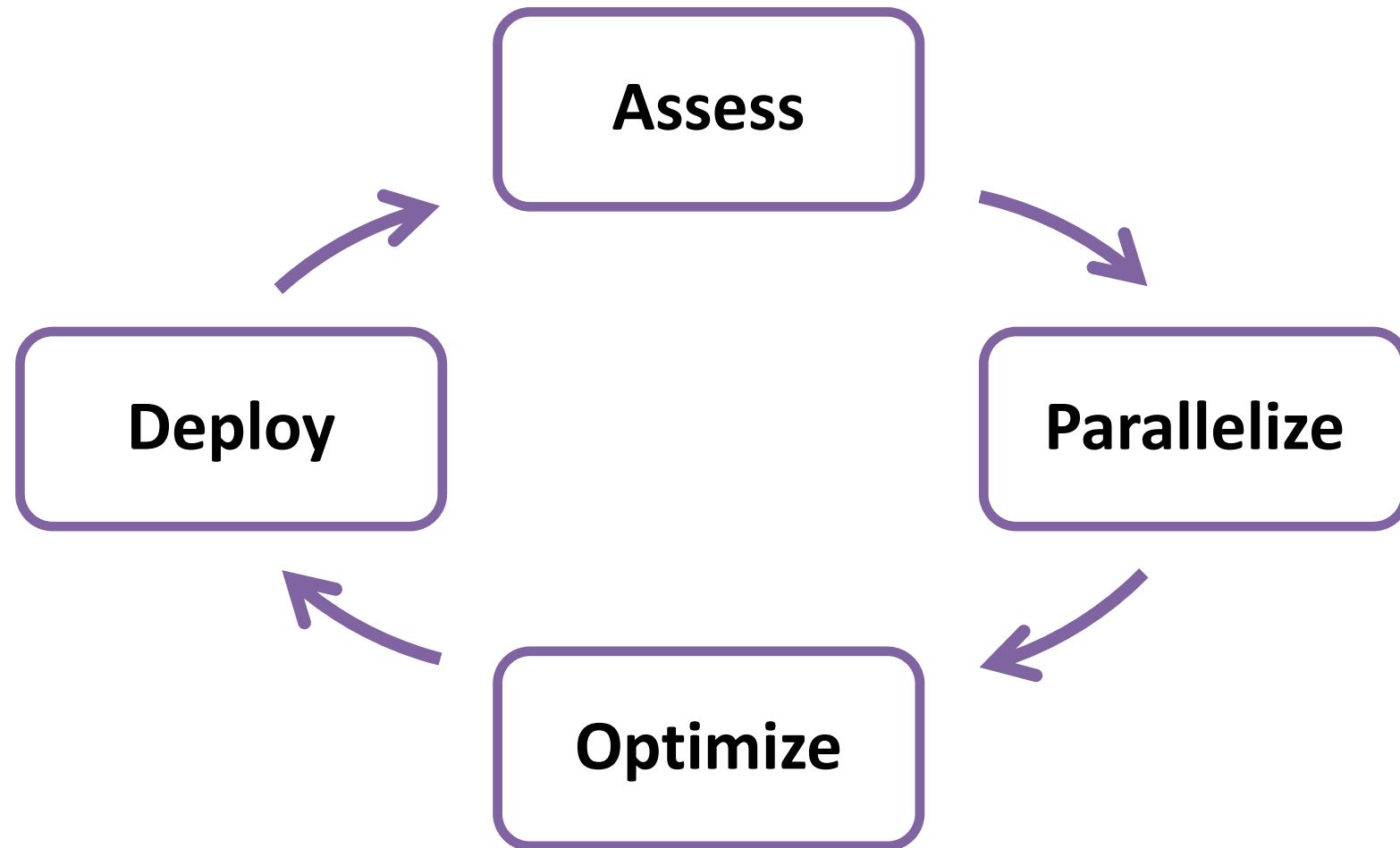


# New instructions for replacement

- **Communication in Shared memory -> shuffle**
  - Don't need Shared memory
  - Lower latency
- **Complex reduction -> Fast Global Memory Atomics**
  - More easy way
- **L1 cache read -> Read-Only Data Cache**
  - L1 is reserved only for register spills and stack data
  - A separate pipe, relaxed memory coalescing rules

# **SUMMARY**

# APOD: A Systematic Path to Performance



# Tools For Project

- **Linux SSH client**
  - Putty
- **cuda-gdb**
  - Along with GPU
- **Profiler**
  - Visual Profiler

# cuda-gdb

- 编译程序时，使用 **-g -G**选项； Linux下停止 **x-server**,启动命令行
  - 常用的调试命令列表
  - **breakpoint (b)**：设置断点，使代码在指定位置暂停执行。其参数可以是方法名，也可以是行号。
  - **run (r)**：在调试器内执行程序。
  - **next (n)**：单步执行到下一行代码。
  - **continue (c)**：继续执行已暂停的程序至下一个断点或程序结尾处。
  - **backtrace (bt)**：显示当前方法调用的栈中的内容。
  - **thread**: 列出当前的主机线程。
  - **cuda thread**: 列出当前活跃的**GPU**线程（若有的话）。
  - **cuda kernel**: 列出当前活跃的**GPU Kernel**，并允许将“焦点”转移到指定的**GPU**线程。

# GPU Architecture Overview

Bin ZHOU 2015 USTC

# Acknowledgement

---

- ▶ Patrick Cozzi, University of Pennsylvania, CIS 565 - Fall 2013
- ▶ Some CPU slides – Varun Sampath, NVIDIA
- ▶ Some GPU slides Kayvon Fatahalian, CMU



# Contents

---

- ▶ Why do we need GPU?
- ▶ 3 Ideas behind GPU to improve Performance
- ▶ Some Real-GPU design review
  - ▶ --NVIDIA GTX 480: Fermi
  - ▶ --NVIDIA GTX 680: Kepler
- ▶ GPU Memory design



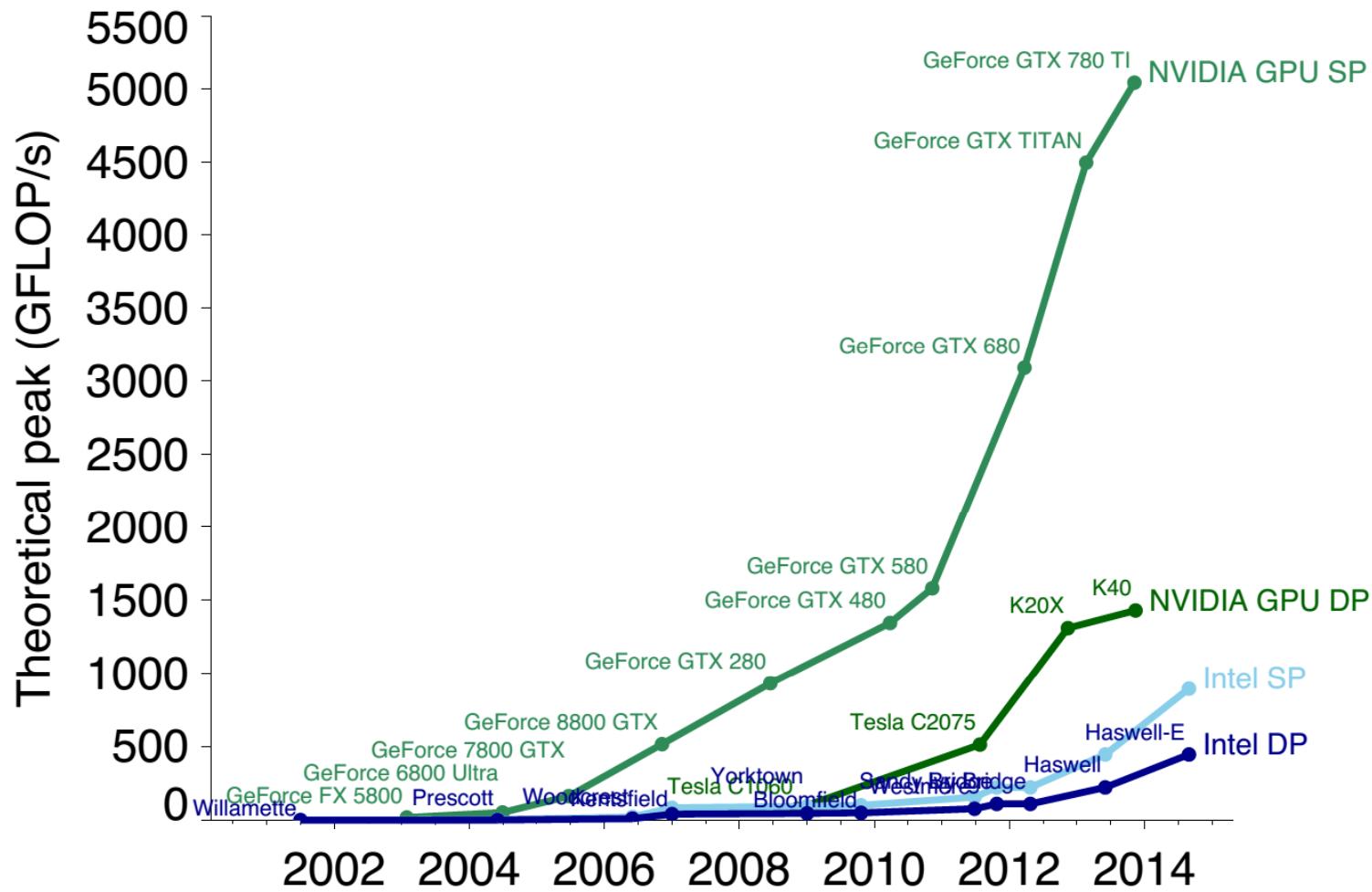
# Terms

---

- ▶ FLOPS - Floating-point Operations per Second
- ▶ GFLOPS - One billion ( $10^9$ ) FLOPS
- ▶ TFLOPS - 1,000 GFLOPS



# CPU and GPU Performance Trends



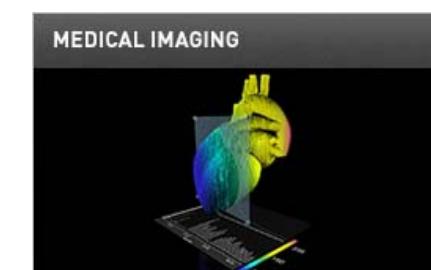
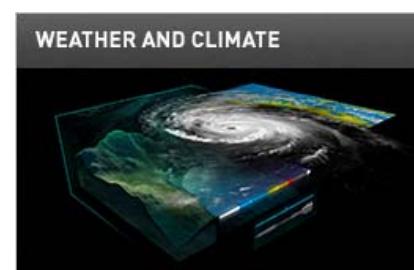
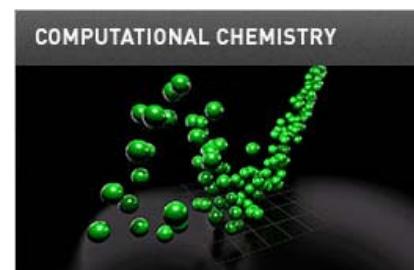
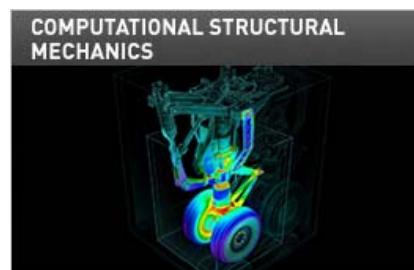
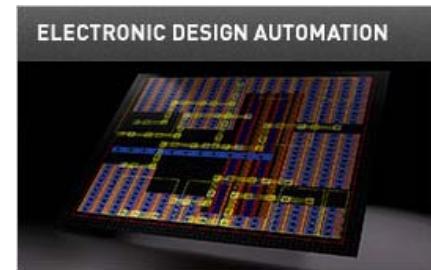
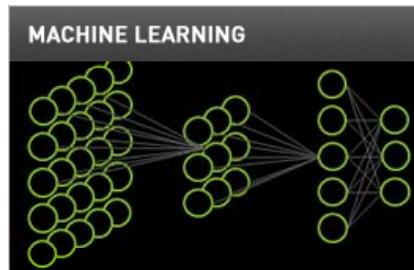
# Why do we need GPU?

---

- ▶ More and More Complex Problems
- ▶ (Application Driven)



# Applications Drives



---

Any More Applications Hunger for  
Computing Power?

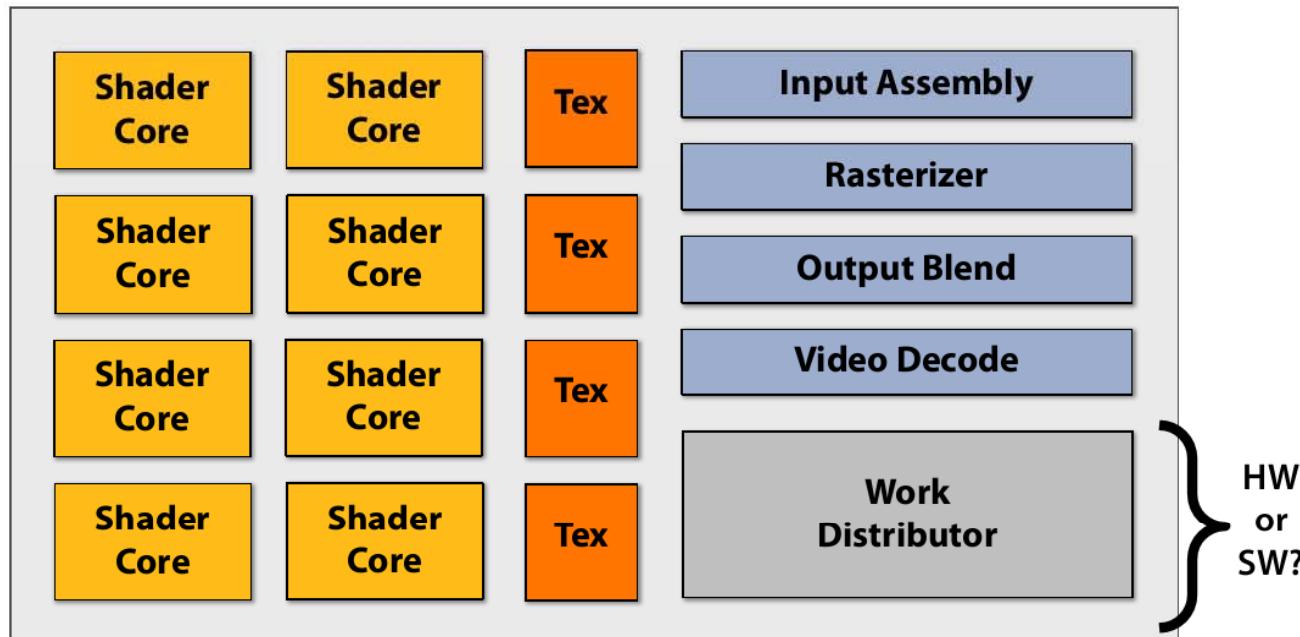
Or your research related problems?



# GPU (Graphic Processing Unit) Architecture Diagram

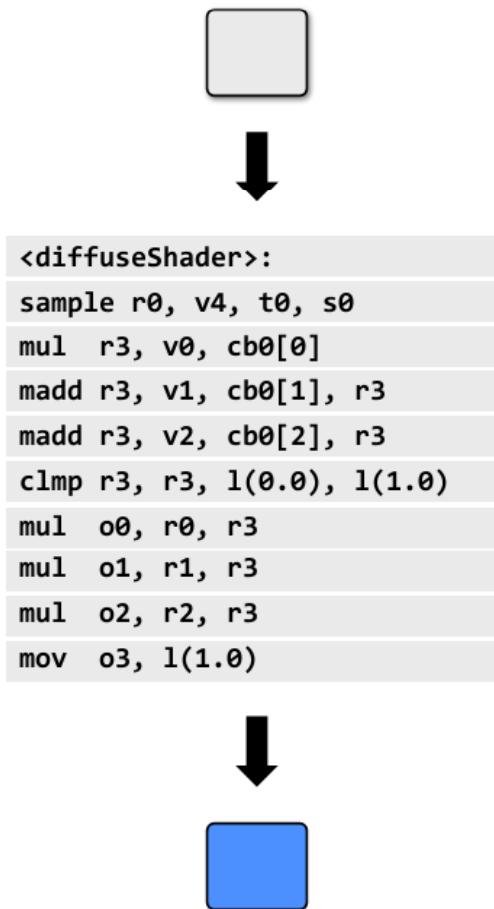
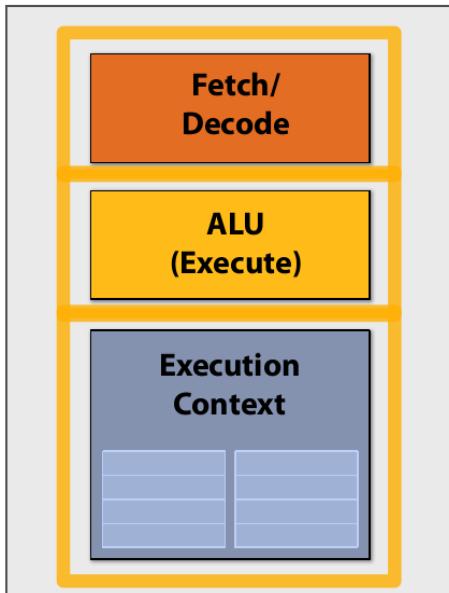
GPU is designed for embarrassingly parallel workloads

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



# Basic Exe Elements

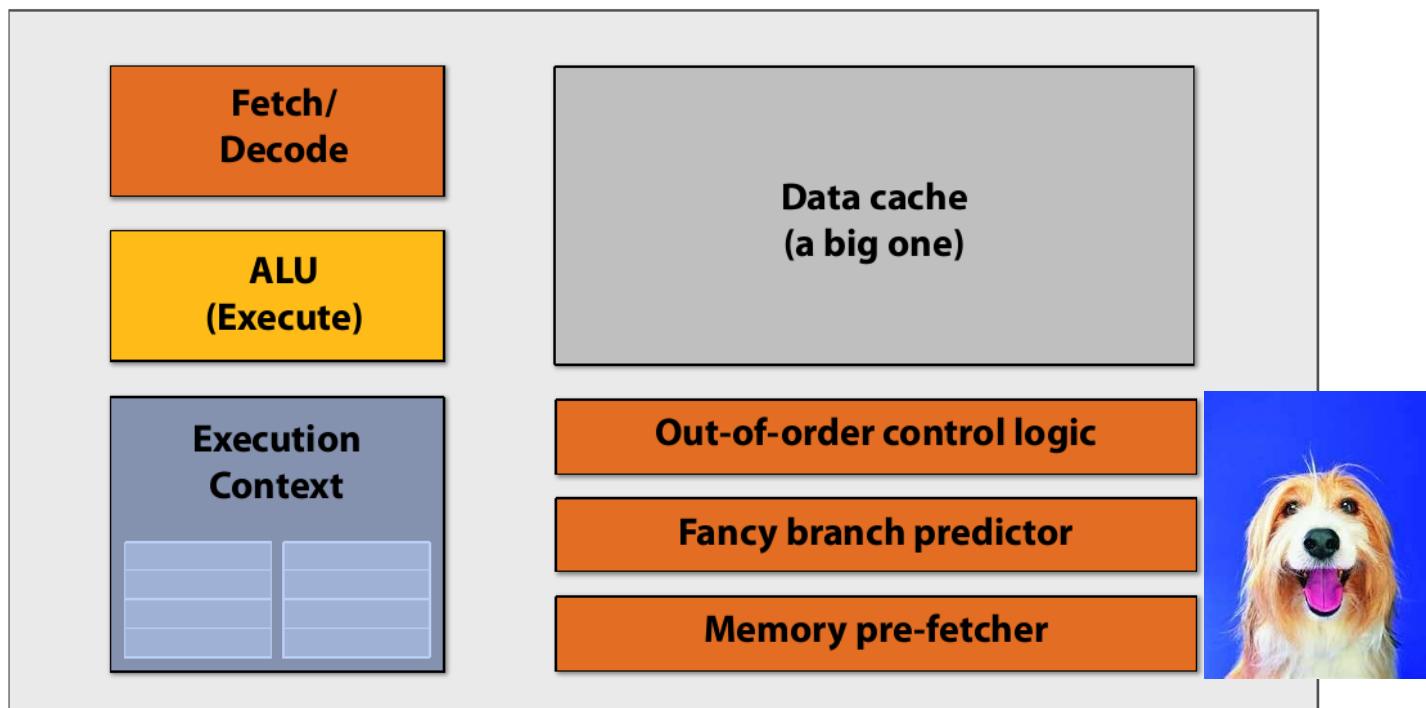
## Execute shader



# Recall CPU Architecture

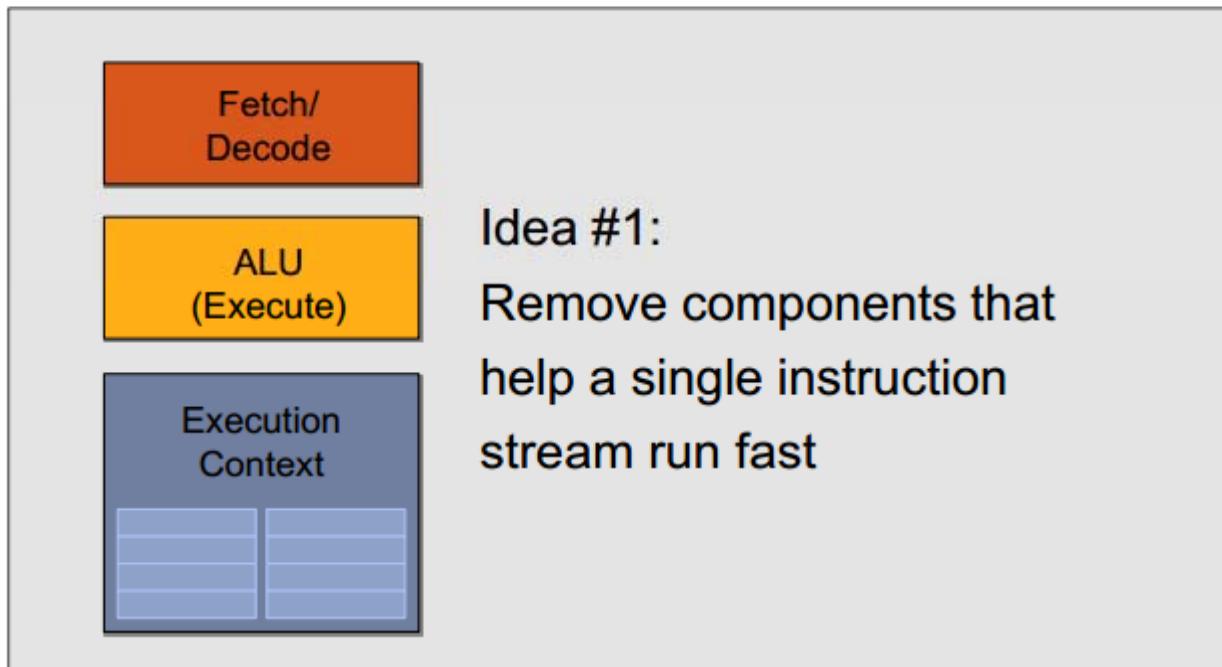
---

## “CPU-style” cores



Idea 1:

## Slimming down



# Copy & Paste

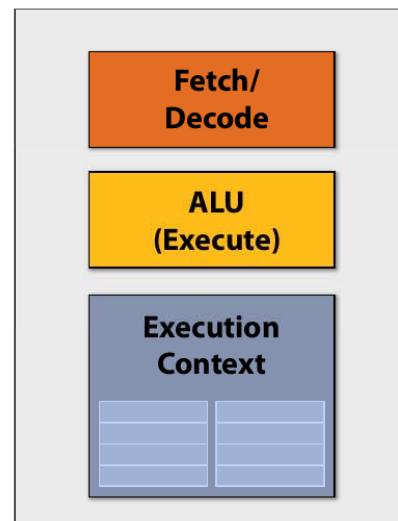
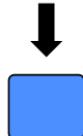
## Two cores (two fragments in parallel)



fragment 1



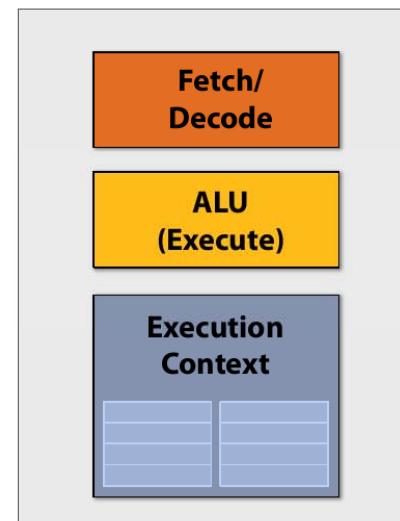
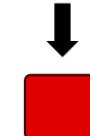
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```



fragment 2

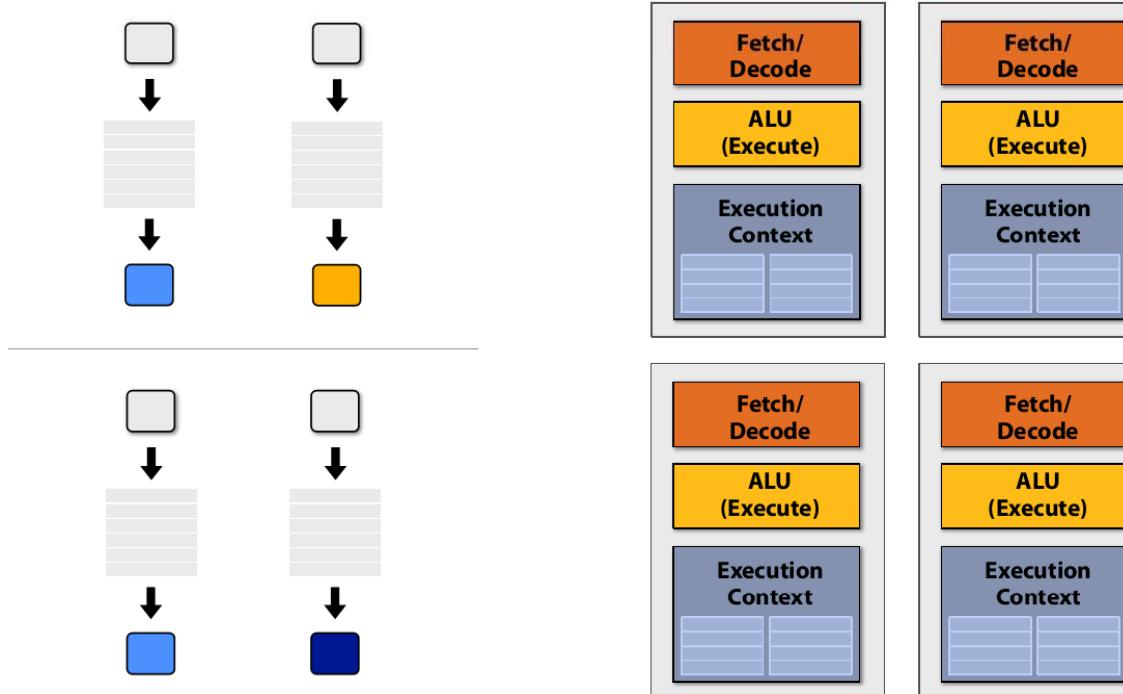


```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```



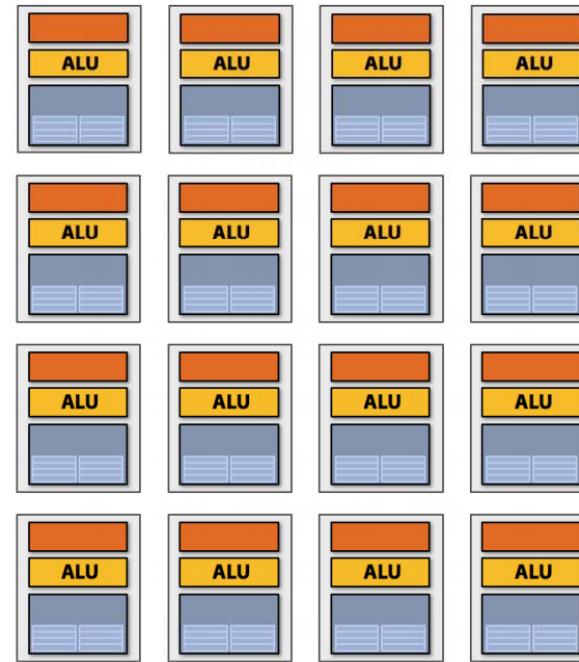
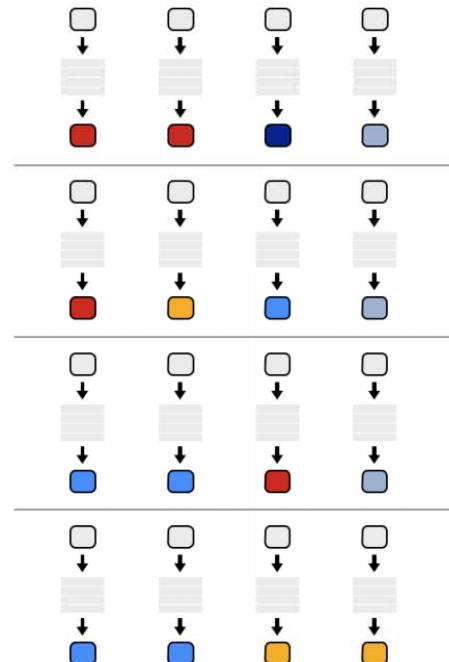
# Copy and Paste Again!

## Four cores (four fragments in parallel)



Copy and Paste 4 Times:

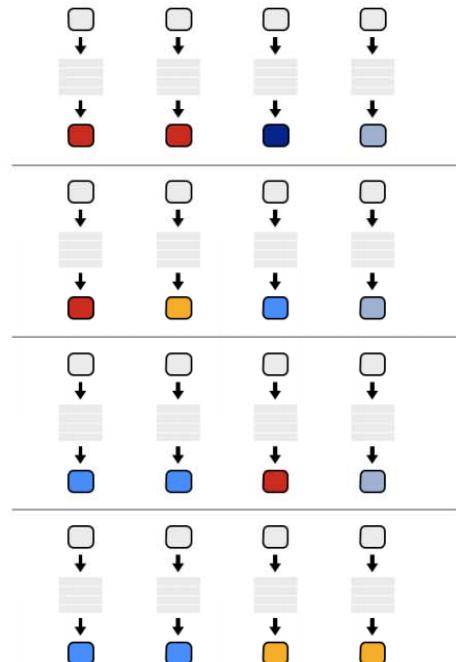
## Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

# Sharing for Saving

## Instruction stream sharing

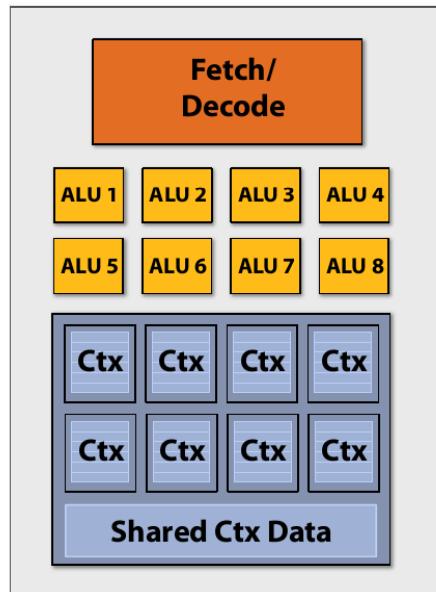


**But ... many fragments  
should be able to share an  
instruction stream!**

```
<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, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```

Idea 2:

## Add ALUs

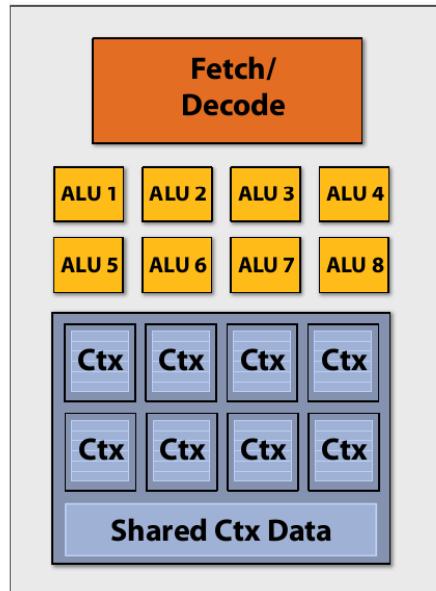


**Idea #2:  
Amortize cost/complexity of  
managing an instruction  
stream across many ALUs**

## SIMD processing

# Improve the Design

## Modifying the shader



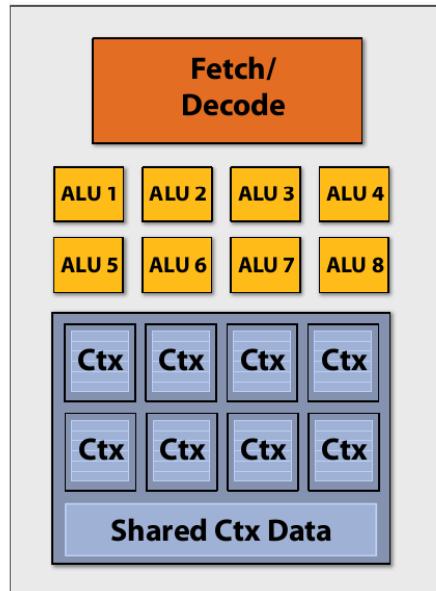
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```

Original compiled shader:

Processes one fragment using  
scalar ops on scalar registers

# Vector Instructions:

## Modifying the shader



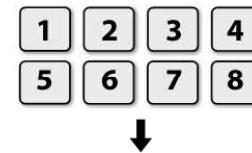
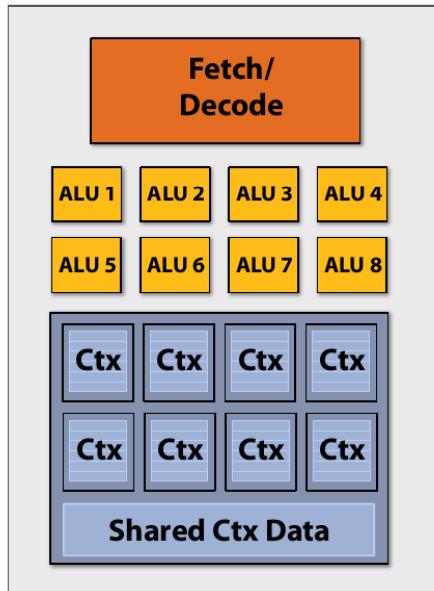
```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)
VEC8_mul vec_o0, vec_r0, vec_r3
VEC8_mul vec_o1, vec_r1, vec_r3
VEC8_mul vec_o2, vec_r2, vec_r3
VEC8_mov o3, 1(1.0)
```

New compiled shader:

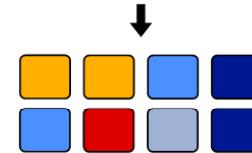
Processes eight fragments using  
vector ops on vector registers

# Multiple Operands

## Modifying the shader

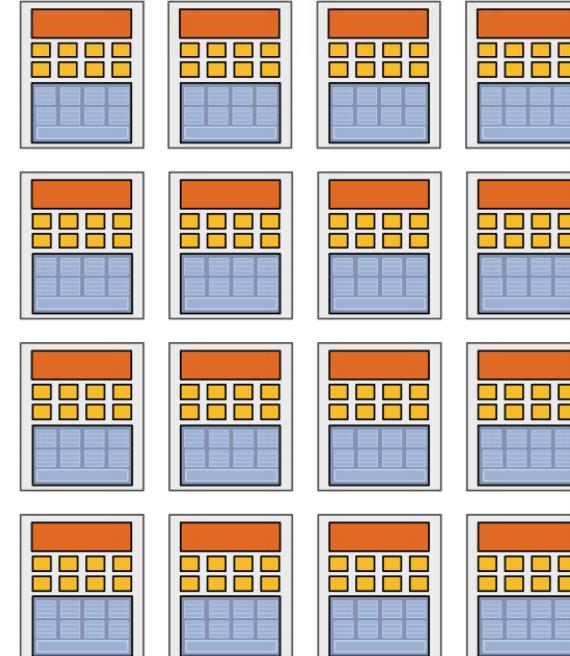
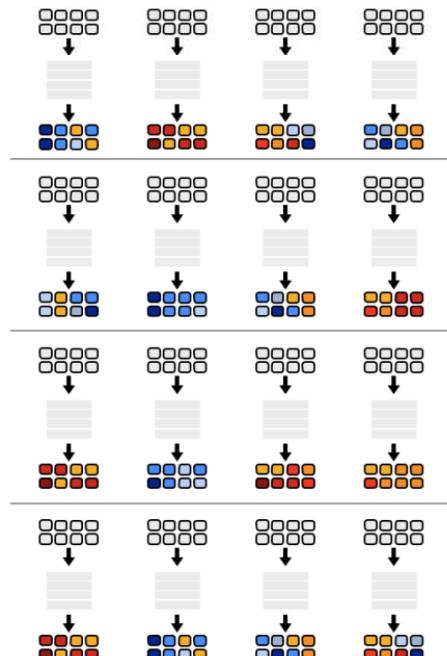


```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)
VEC8_mul vec_o0, vec_r0, vec_r3
VEC8_mul vec_o1, vec_r1, vec_r3
VEC8_mul vec_o2, vec_r2, vec_r3
VEC8_mov o3, 1(1.0)
```



# More fragments, More Simultaneous Instructions

**128 fragments in parallel**



**16 cores = 128 ALUs, 16 simultaneous instruction streams**

07/29/10

Thursday, July 29, 2010

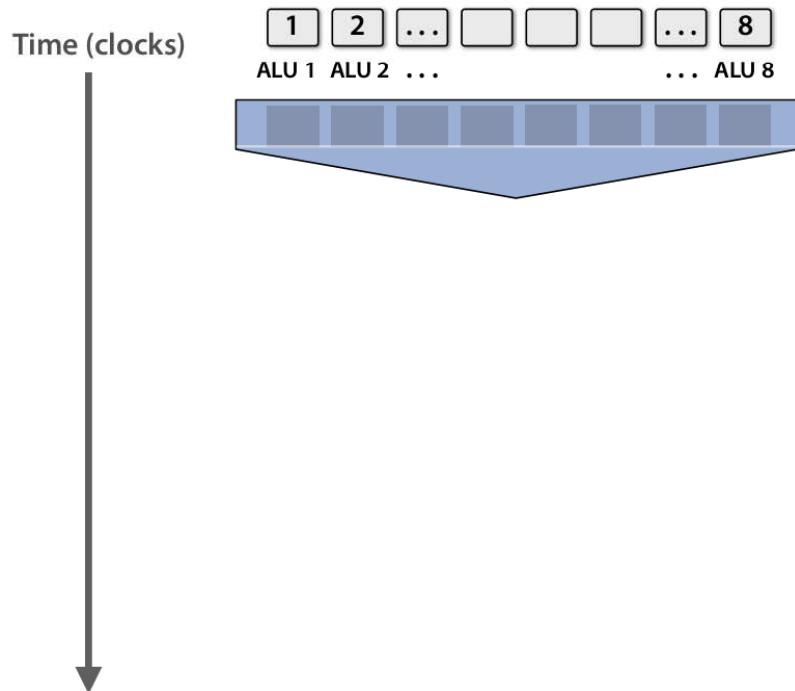
Beyond Programmable Shading Course, ACM SIGGRAPH 2010

25

Could be different

# Sharing Dispatcher

## But what about branches?



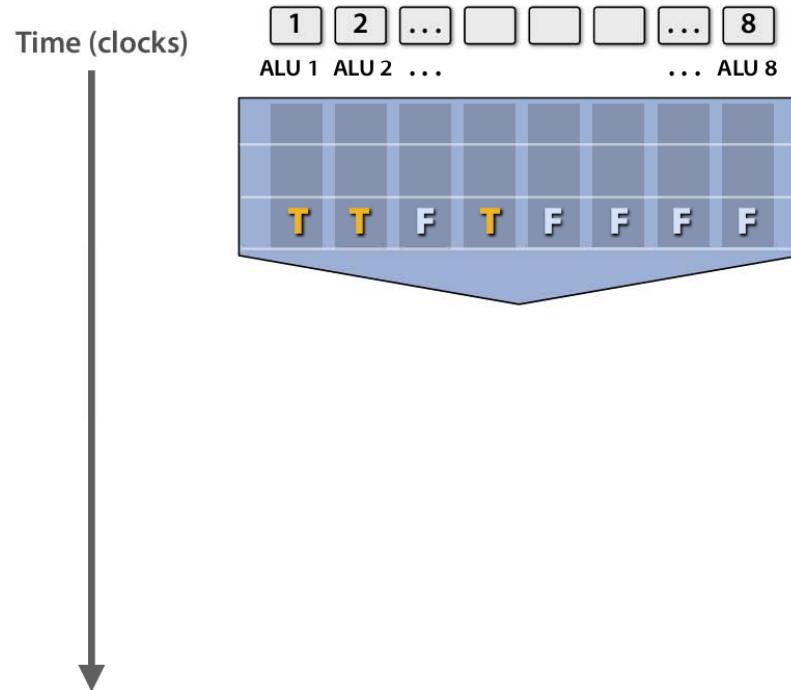
```
<unconditional
 shader code>

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?



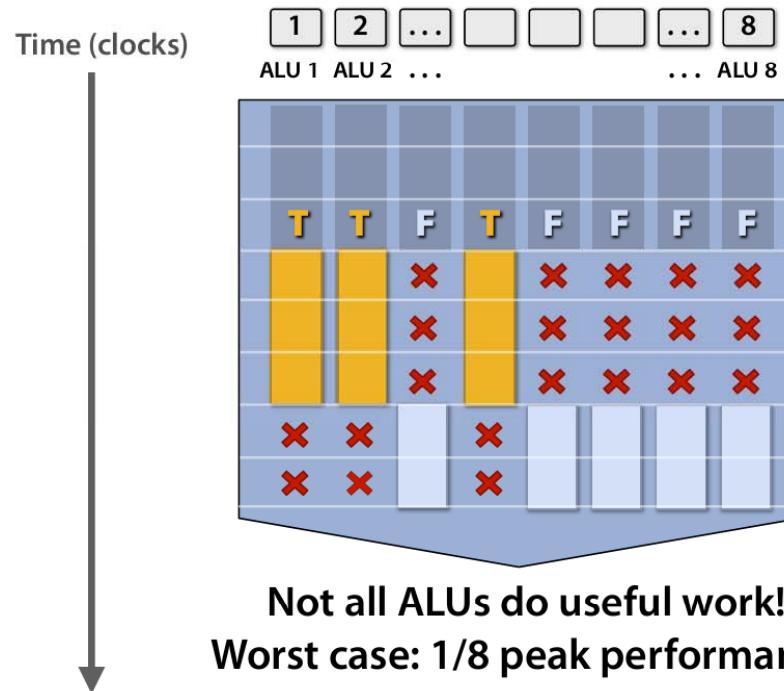
```
<unconditional
 shader code>

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?

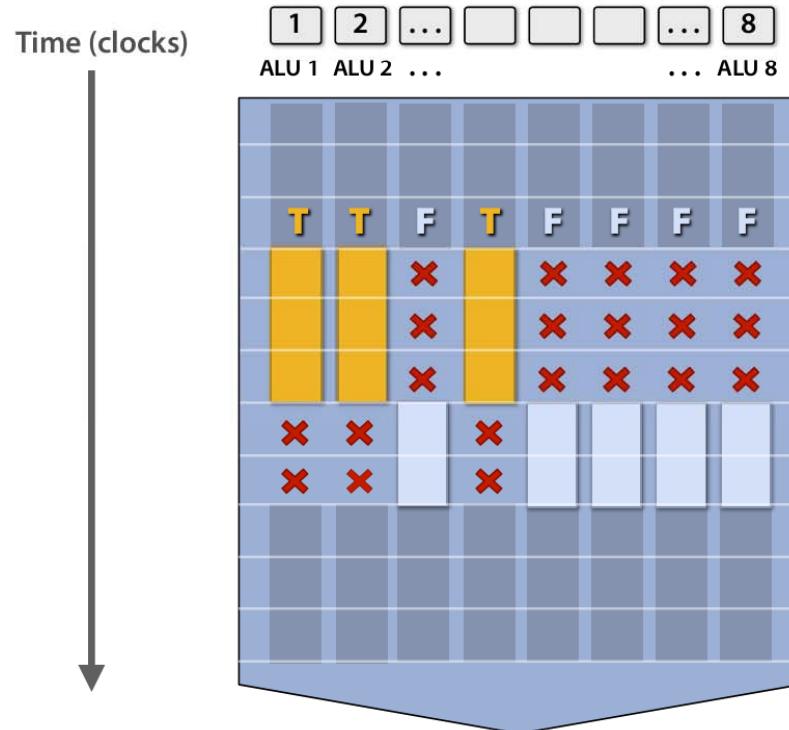


```
<unconditional
 shader code>

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>
```

## Clarification

---

- ▶ SIMD Processing doesn't imply SIMD instructions
- ▶ Option 1: Explicit Vector Instructions:
  - ▶ SSE, AVX and etc.
- ▶ Option 2: Scalar instructions, implicit HW vectorization
  - ▶ Hardware decides instruction stream sharing across scalar ALUs
  - ▶ NVIDIA, AMD GPUs



# Problems and Challenges

---



## **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.**



## Solution: Idea 3

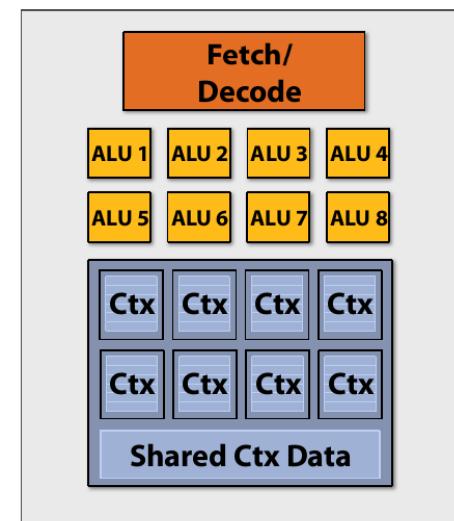
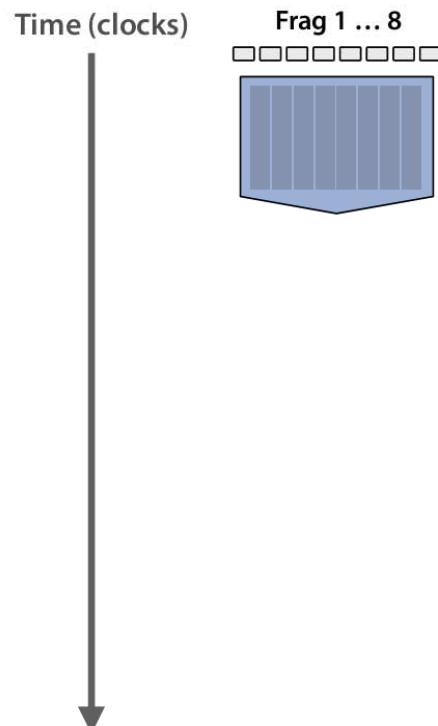
---

- ▶ Lots of independent fragments switching
- ▶ Interleave processing of many program slices on ALUs to avoid the stalls caused by high latency operations—like “Fill in the blanks”



# Latency Hiding

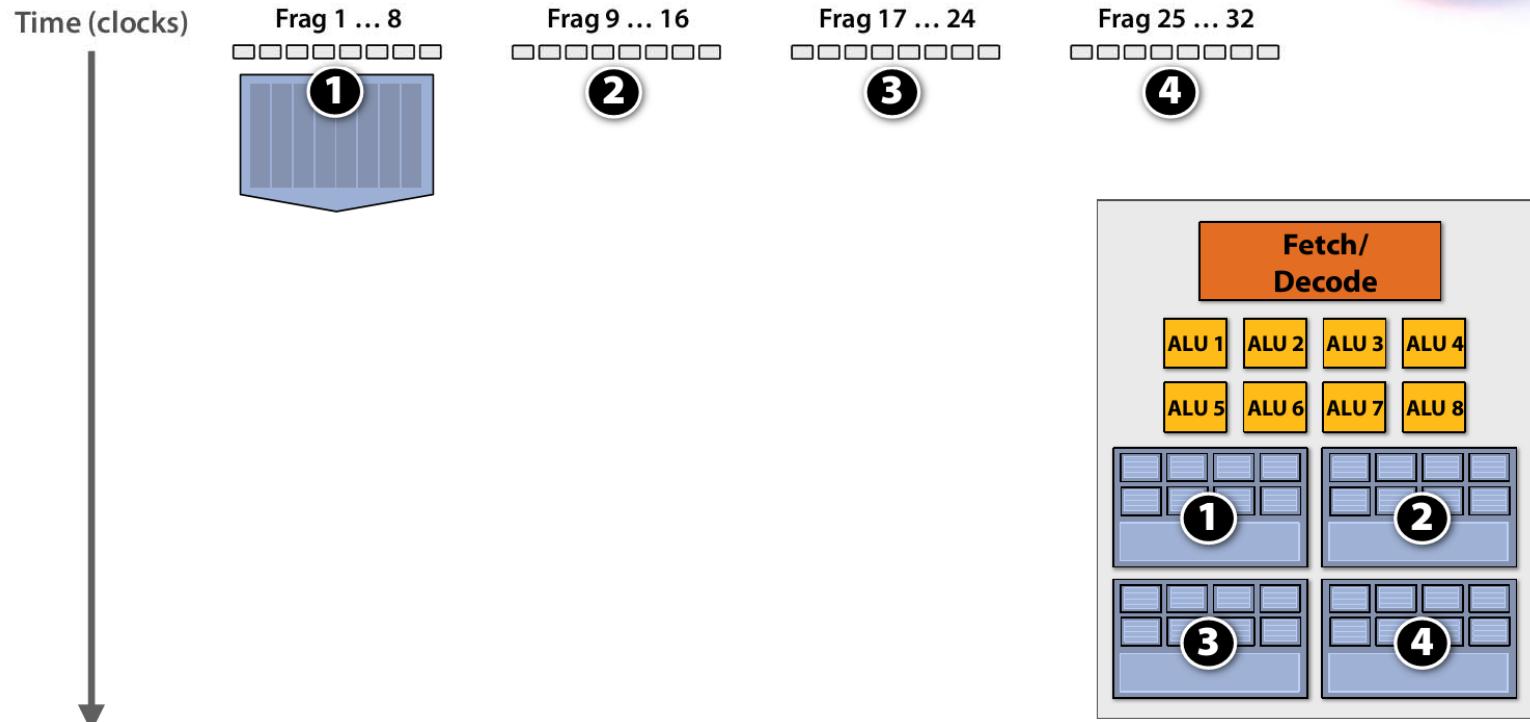
## Hiding shader stalls





# Hiding shader stalls

Time (clocks)



07/29/10

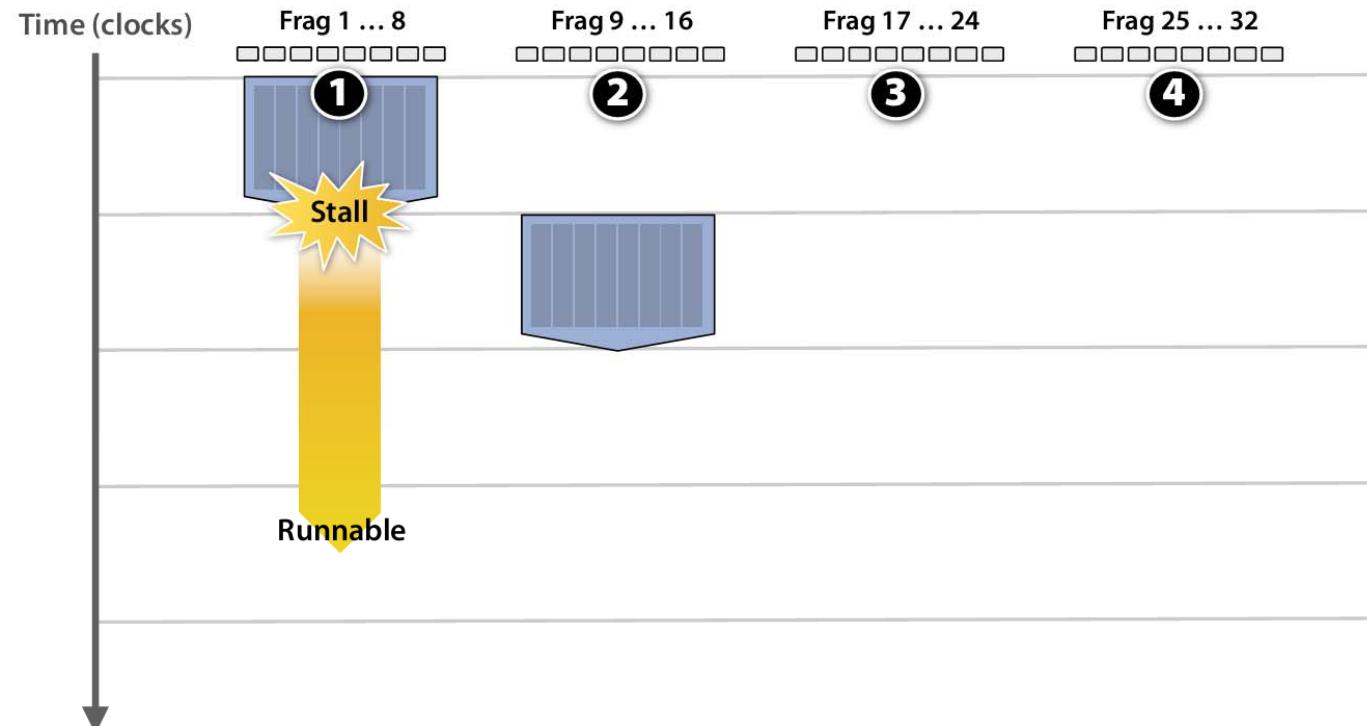
Thursday, July 29, 2010

Beyond Programmable Shading Course, ACM SIGGRAPH 2010

35

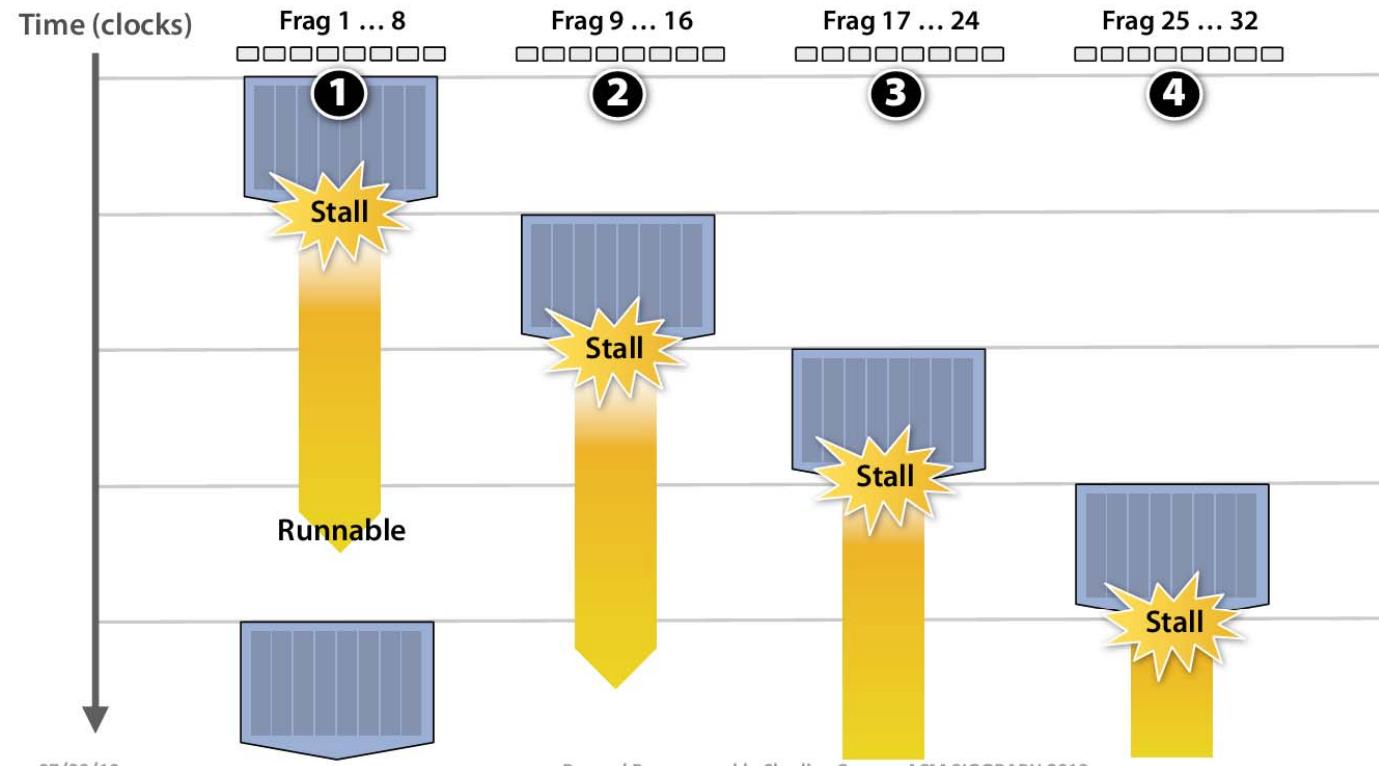


# Hiding shader stalls



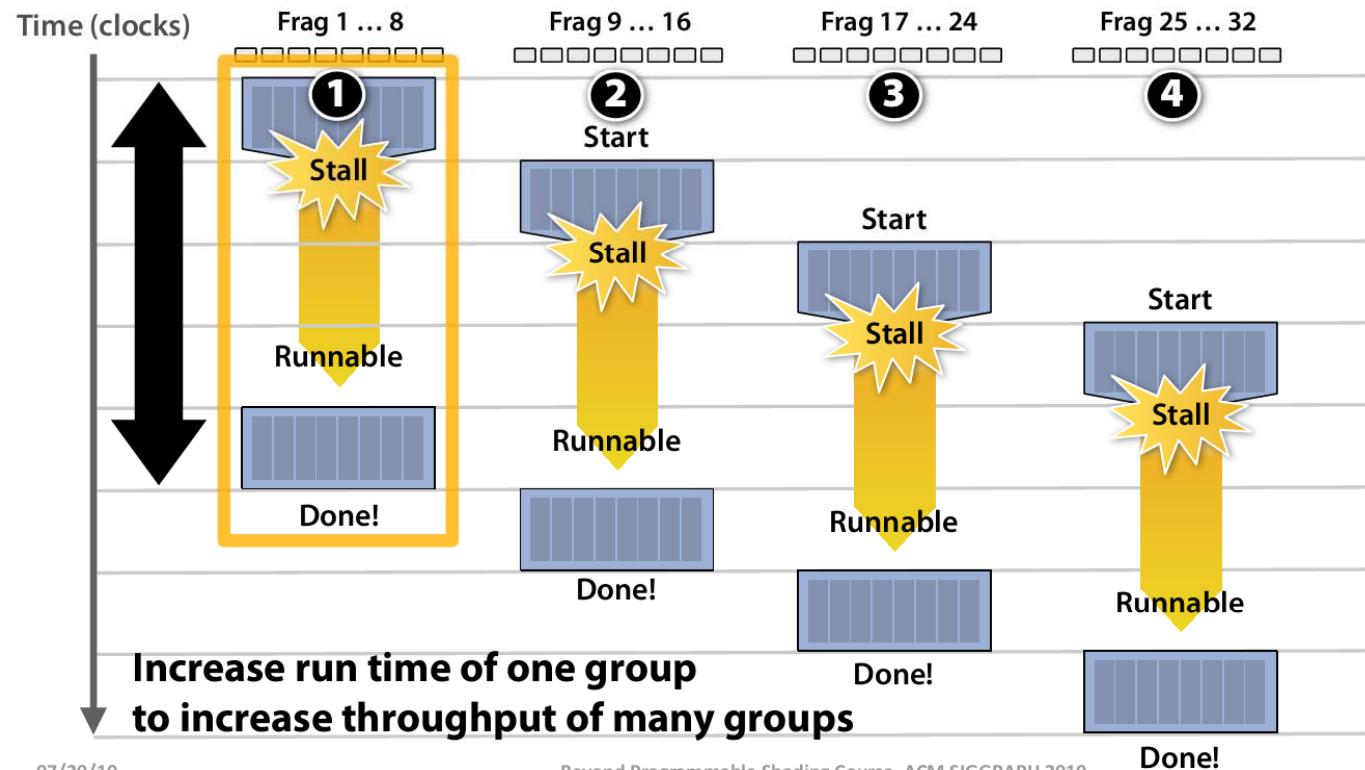


# Hiding shader stalls



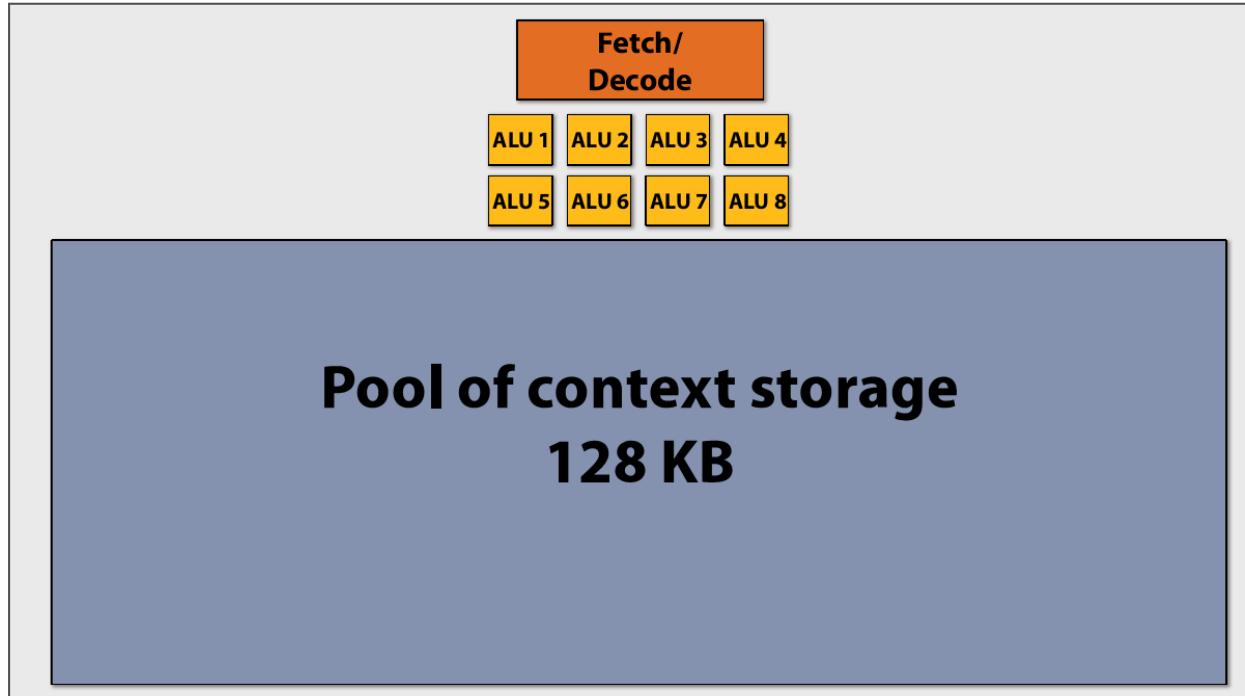
# Gain a high throughput

## Throughput!



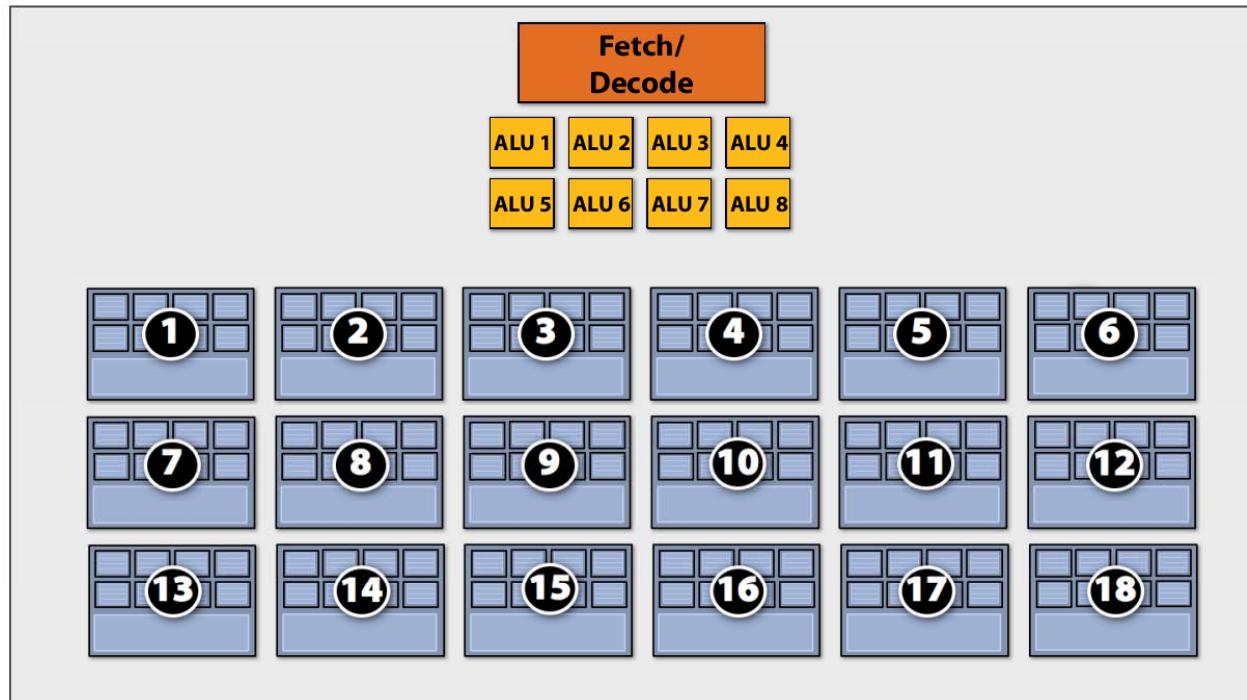
# Dealing with the Context

## Storing contexts



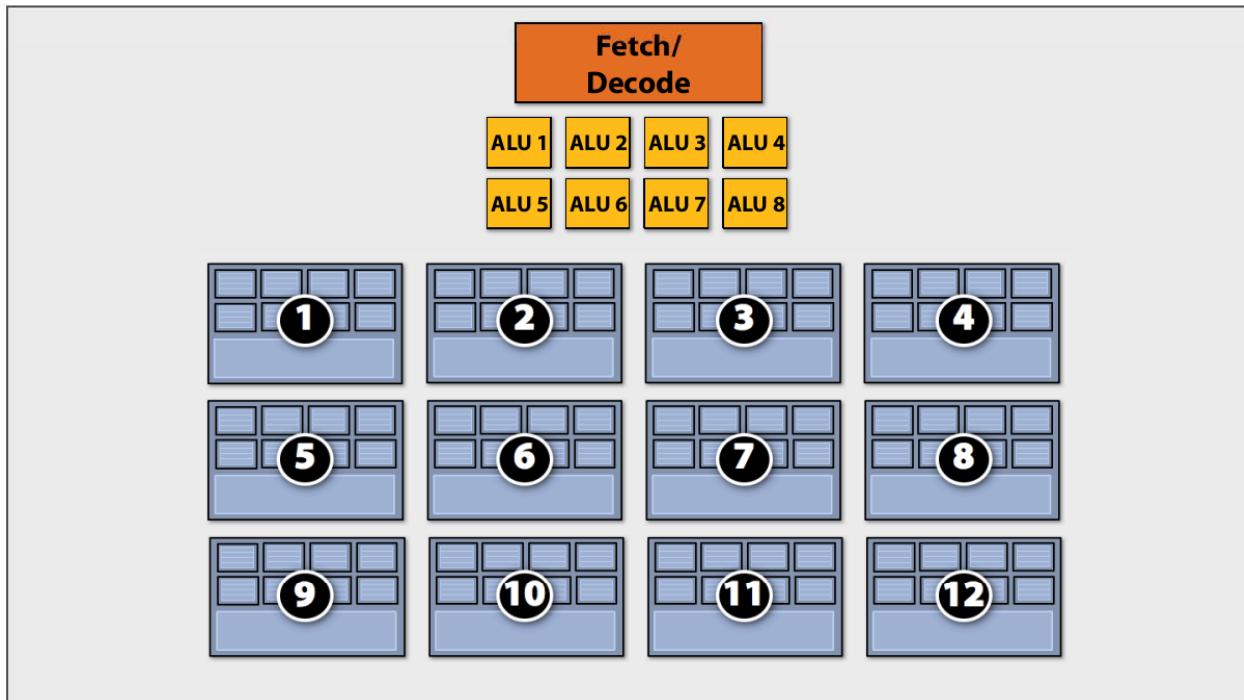
Smaller contexts: Pros: Better latency hiding;  
Cons: ?

## Eighteen small contexts (maximal latency hiding)



# Medium Contexts: Pros:? Cons:?

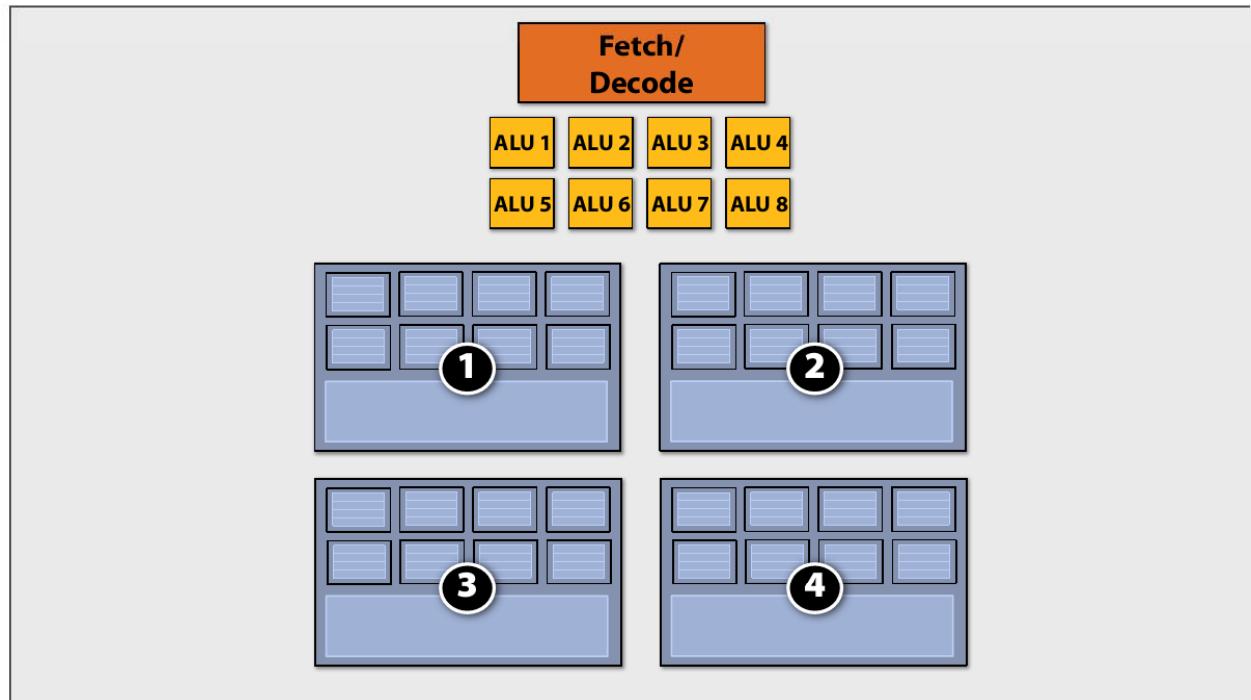
## Twelve medium contexts



# Large Contexts: Pros? Cons?

## Four large contexts

(low latency hiding ability)



# The Tradeoff in Architecture Design

## 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**

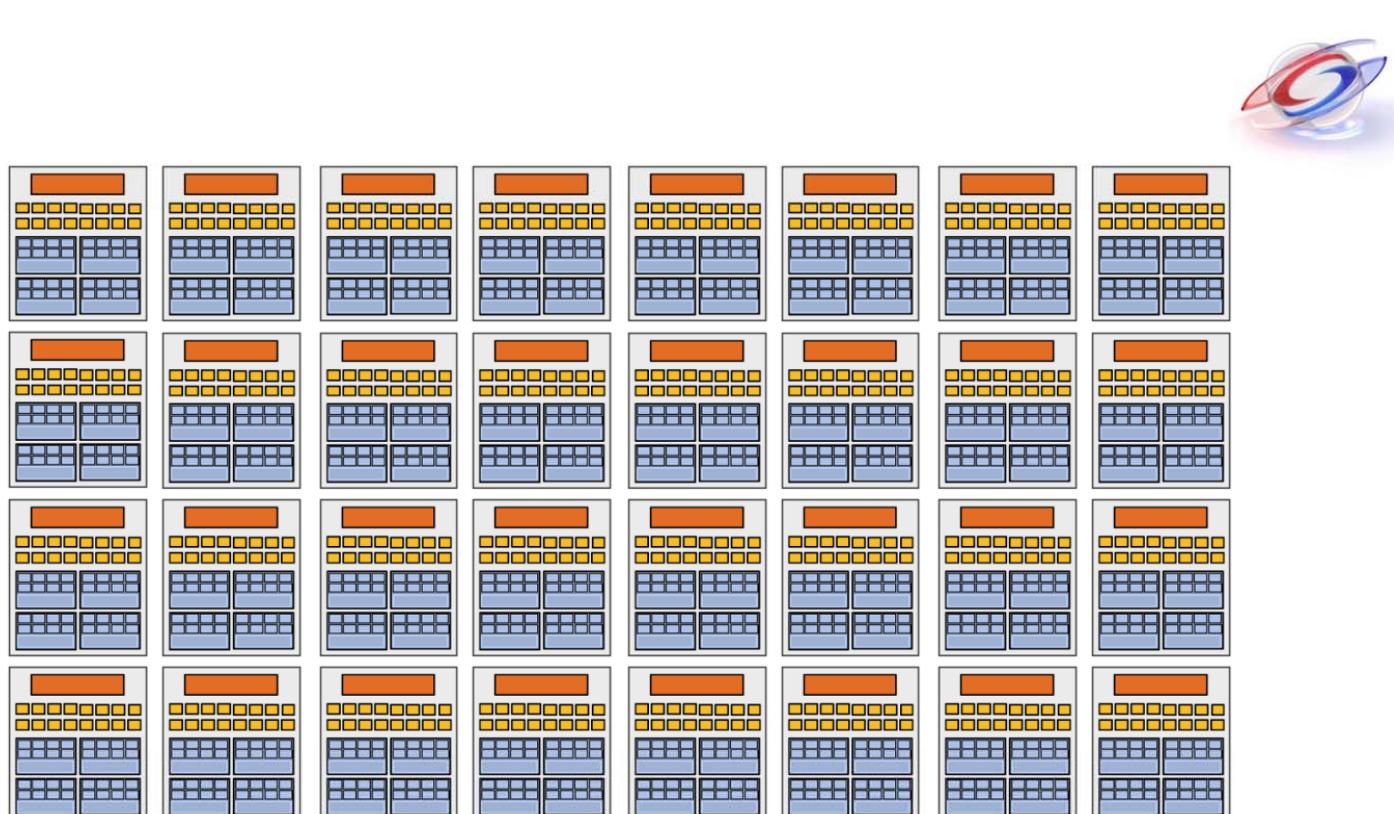
**64 concurrent (but interleaved)  
instruction streams**

**512 concurrent fragments**

**= 256 GFLOPs (@ 1GHz)**



# Bigger Chip with higher performance



**32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)**

## **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**



# Fermi Architecture

## NVIDIA GeForce GTX 480 (Fermi)

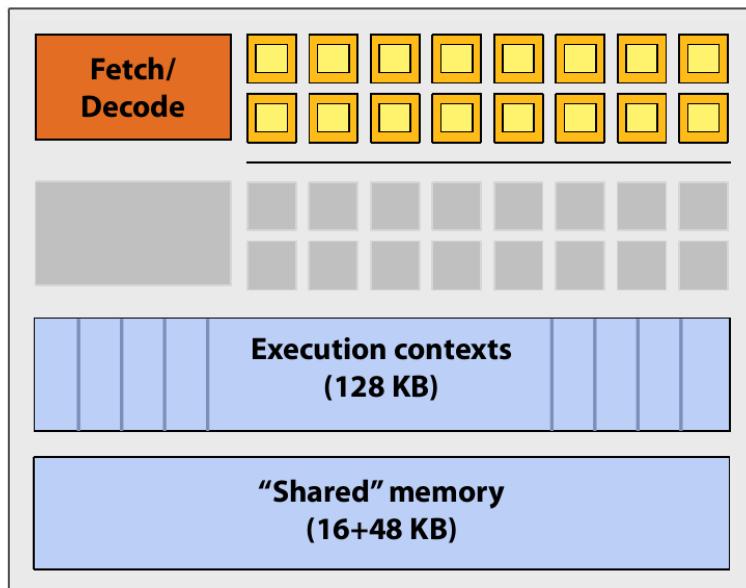


- NVIDIA-speak:
  - 480 stream processors (“CUDA cores”)
  - “SIMT execution”
- Generic speak:
  - 15 cores
  - 2 groups of 16 SIMD functional units per core

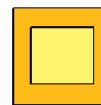




# NVIDIA GeForce GTX 480 "core"



Source: Fermi Compute Architecture Whitepaper  
CUDA Programming Guide 3.1, Appendix G



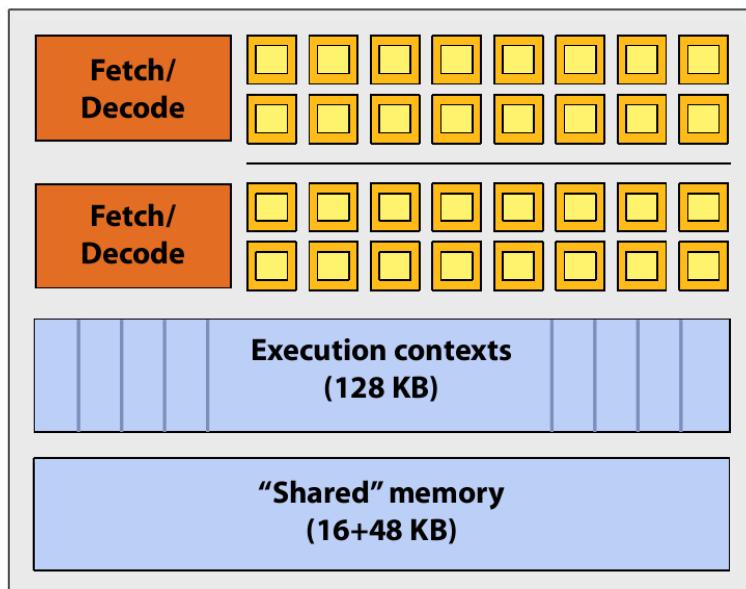
= SIMD function unit,  
control shared across 16 units  
(1 MUL-ADD per clock)

- Groups of 32 [fragments/vertices/CUDA threads] share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

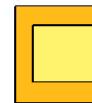




# NVIDIA GeForce GTX 480 "core"



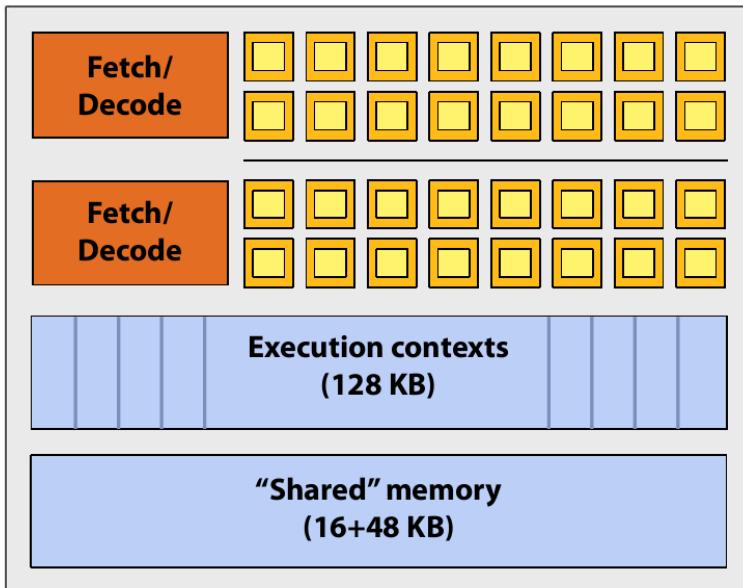
Source: Fermi Compute Architecture Whitepaper  
CUDA Programming Guide 3.1, Appendix G



= SIMD function unit,  
control shared across 16 units  
(1 MUL-ADD per clock)

- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

# NVIDIA GeForce GTX 480 "SM"



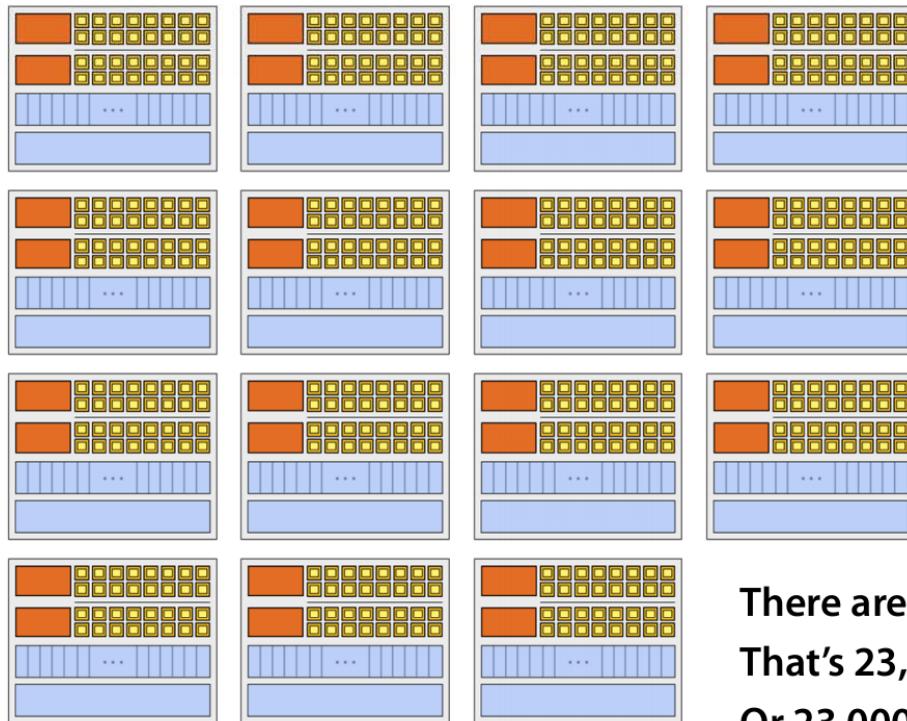
= **CUDA core**  
(1 MUL-ADD per clock)

- The **SM** contains 32 **CUDA cores**
- Two **warps** are selected each clock (decode, fetch, and execute two **warps** in parallel)
- Up to 48 warps are interleaved, totaling 1536 **CUDA threads**

Source: Fermi Compute Architecture Whitepaper  
CUDA Programming Guide 3.1, Appendix G

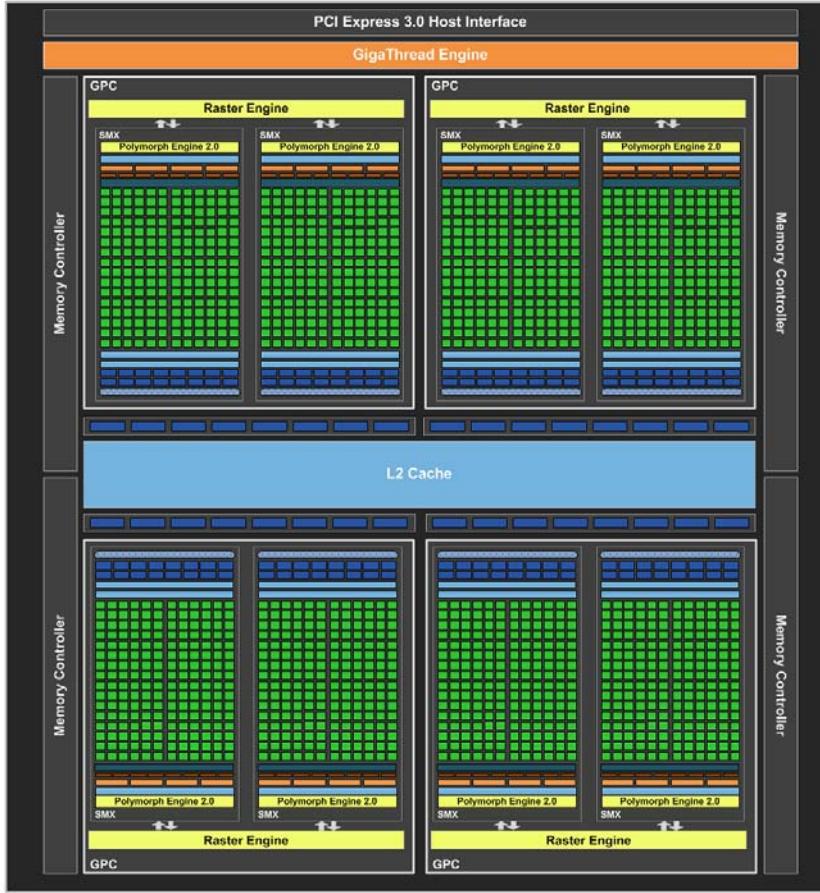


# NVIDIA GeForce GTX 480



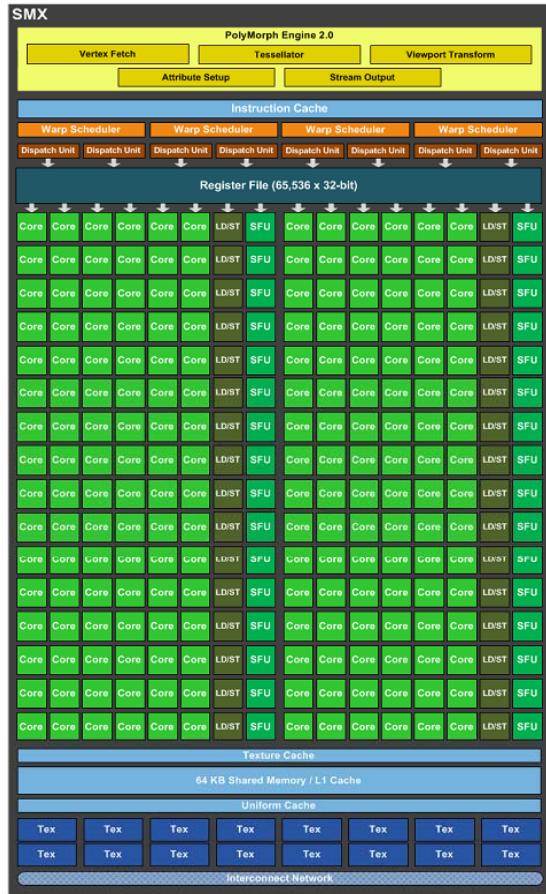
**There are 15 of these things on the GTX 480:  
That's 23,000 fragments!  
Or 23,000 CUDA threads!**

# Kepler Architecture: GTX 680



GPU	GF110 (Fermi)	GK104 (Kepler)	Ratio	Ratio (w/ clk freq)
<b>Total unit counts :</b>				
CUDA Cores	512	1536	3.0x	
SFU	64	256	4.0x	
LD/ST	256	256	1.0x	
Tex	64	128	2.0x	
Polymorph	16	8	0.5x	
Warp schedulers	32	32	1.0x	
<b>Throughput per graphics clock :</b>				
FMA32	1024	1536	1.5x	2.0x
SFU	128	256	2.0x	2.6x
LD/ST (64b operations)	256	256	1.0x	1.3x
Tex	64	128	2.0x	2.6x
Polygon/clk	4	4	1.0x	1.3x
Inst/clk	32*32	64*32	2.0x	2.6x

# SMX

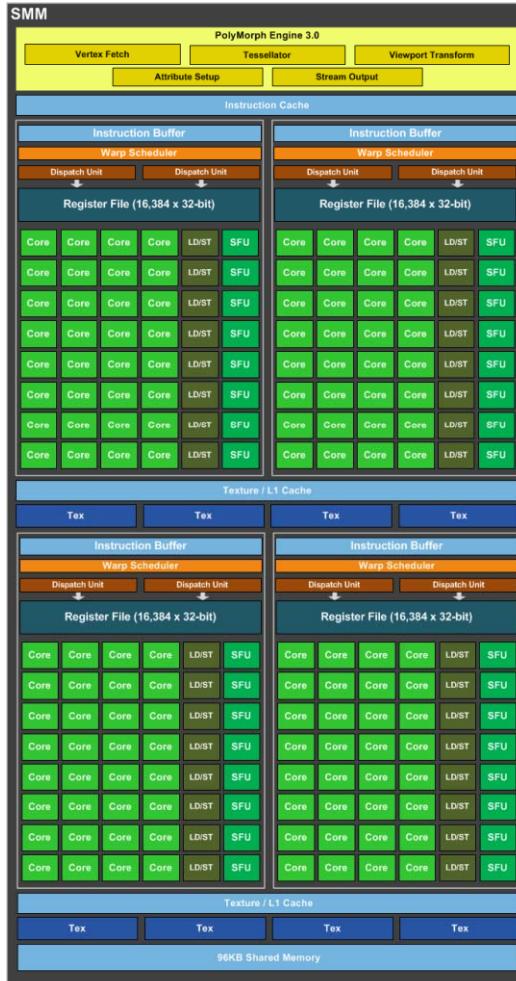


GPU	GF110 (Fermi)	GK104 (Kepler)	Ratio	Ratio (w/ clk freq)
<b>Per SM unit counts :</b>				
CUDA Cores	32	192	6.0x	
SFU	4	32	8.0x	
LD/ST	16	32	2.0x	
Tex	4	16	4.0x	
Polymorph	1	1	1.0x	
Warp schedulers	2	4	2.0x	
<b>Throughput per graphics clock :</b>				
FMA32	64	192	3.0x	3.9x
SFU	8	32	4.0x	5.2x
LD/ST (64b operations)	16	32	2.0x	2.6x
Tex	4	16	4.0x	5.2x
Polygon/clk	0.25	0.5	2.0x	2.6x
Inst/clk	32*2	32*8	4.0x	5.2x

# NVIDIA GK110



# Maxwell Architecture SMM:



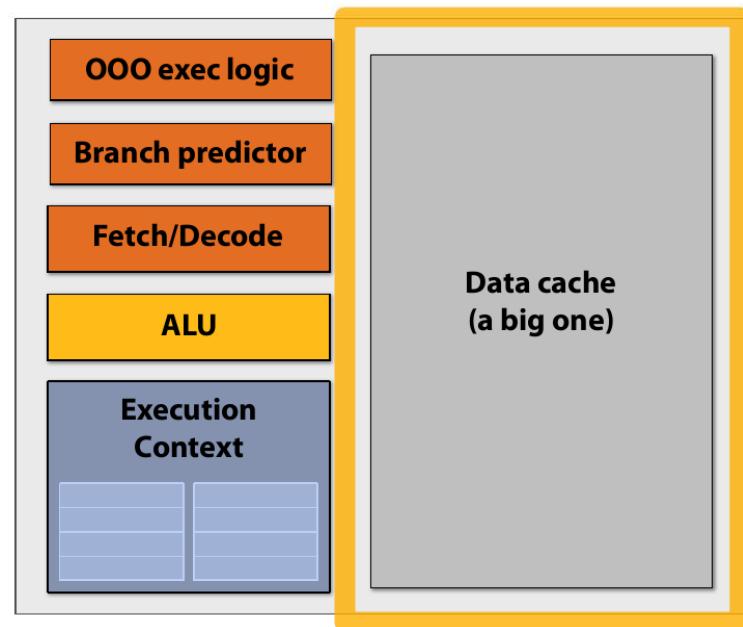
**MAXWELL  
“GM204”  
Top Level**

- ▶ 5.2 Billion Transistors
- ▶ 2x performance vs GK104
- ▶ 16 SMM
- ▶ 2048 CUDA Cores
- ▶ 16 Geometry Units
- ▶ 128 Texture Units
- ▶ 64 ROP Units
- ▶ 256-bit GDDR5



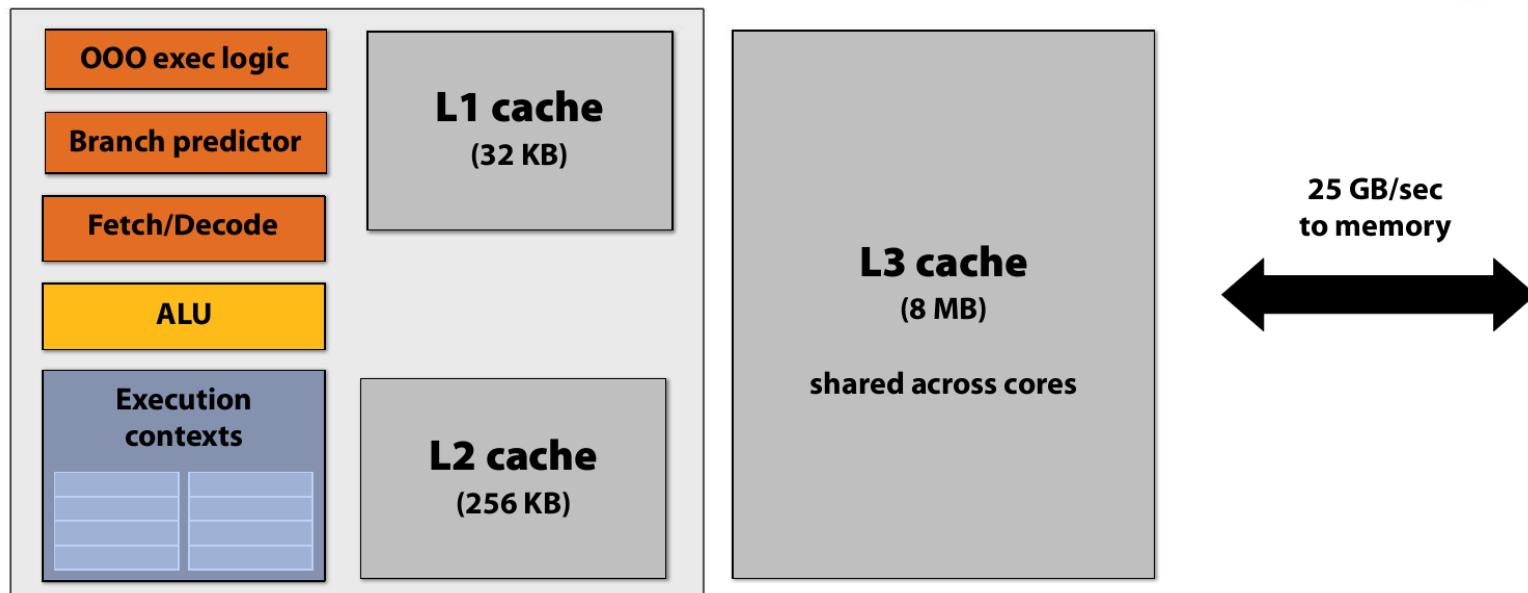
# Memory and Data Access

Recall: “CPU-style” core



# CPU Style Memory

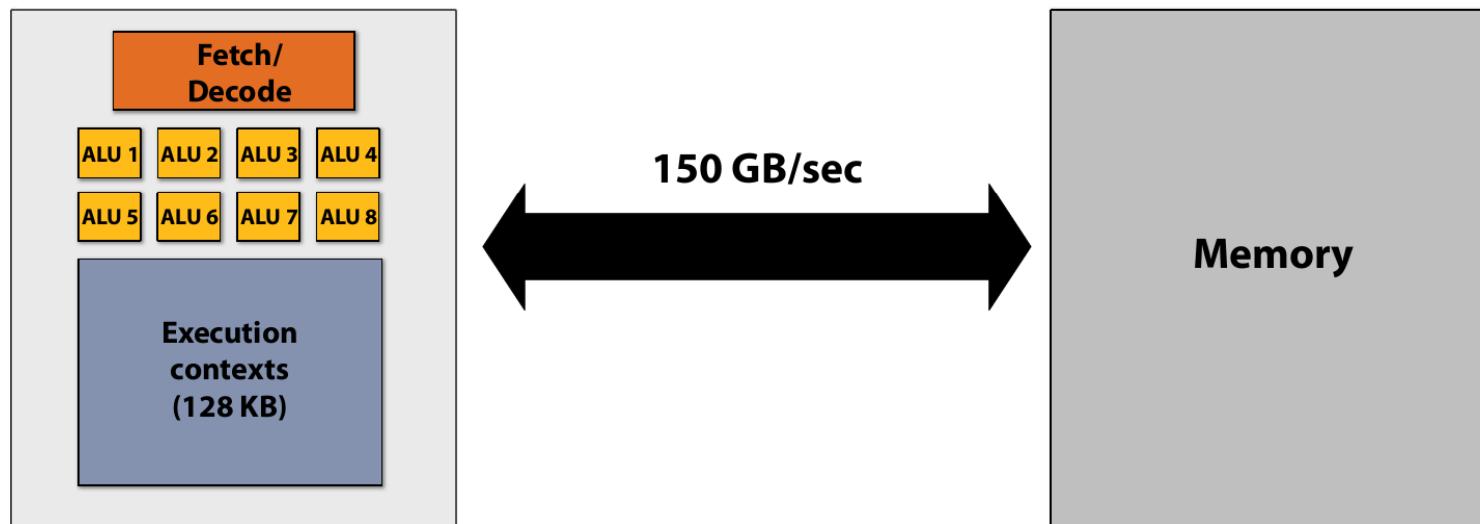
## “CPU-style” memory hierarchy



CPU cores run efficiently when data is resident in cache  
(caches reduce latency, provide high bandwidth)

# GPU Style : Throughput !

## Throughput core (GPU-style)



**More ALUs, no large traditional cache hierarchy:  
Need high-bandwidth connection to memory**



# Memory Bound !

## Bandwidth is a critical resource



- A high-end GPU (e.g. Radeon HD 5870) has...
  - Over **twenty times** (2.7 TFLOPS) the compute performance of quad-core CPU
  - No large cache hierarchy to absorb memory requests
- GPU memory system is designed for throughput
  - Wide bus (150 GB/sec)
  - Repack/reorder/interleave memory requests to maximize use of memory bus
  - Still, this is only **six times** the bandwidth available to CPU

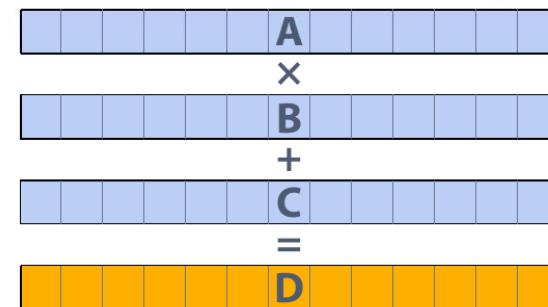
# A Simple Example

## Bandwidth thought experiment



Task: element-wise multiply two long vectors A and B

1. Load input A[i]
2. Load input B[i]
3. Load input C[i]
4. Compute  $A[i] \times B[i] + C[i]$
5. Store result into D[i]



Four memory operations (16 bytes) for every MUL-ADD

Radeon HD 5870 can do 1600 MUL-ADDS per clock

Need ~20 TB/sec of bandwidth to keep functional units busy

**Less than 1% efficiency... but 6x faster than CPU!**



# Memory Bound



## Bandwidth limited!

If processors request data at too high a rate,  
the memory system cannot keep up.

No amount of latency hiding helps this.

Overcoming bandwidth limits are a common challenge  
for GPU-compute application developers.



Try from the beginning

## Reducing bandwidth requirements



- Request data less often (instead, do more math)
  - “arithmetic intensity”
- Fetch data from memory less often (share/reuse data across fragments)
  - on-chip communication or storage

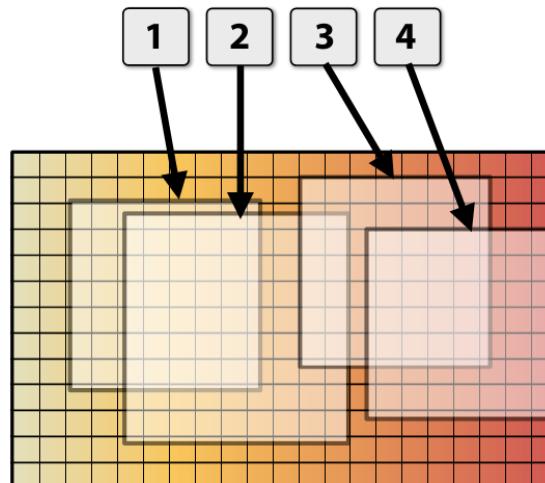




# Reducing bandwidth requirements

- Two examples of on-chip storage
  - Texture caches
  - OpenCL “local memory” (CUDA shared memory)

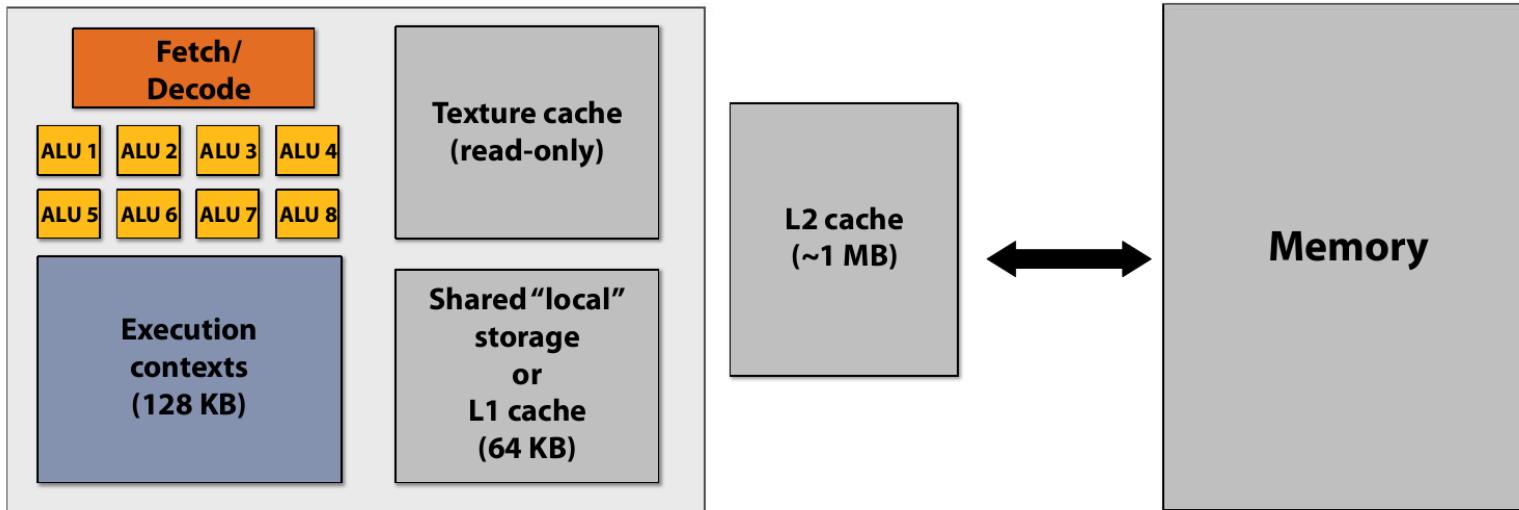
Texture data



**Texture caches:**  
**Capture reuse across**  
**fragments, not temporal**  
**reuse within a single**  
**shader program**



# Modern GPU memory hierarchy



On-chip storage takes load off memory system.  
Many developers calling for more cache-like storage  
(particularly GPU-compute applications)



---

**GPU is a heterogeneous  
many core computing  
system, highly-tuned for  
high throughput  
applications**

---



## Effective GPU Tasks Require

---

- ▶ Thousands of independent processing fragments
  - ▶ Utilize many ALUs
  - ▶ Hide the latency
- ▶ Better Sharing instruction streams
  - ▶ SIMD
- ▶ Computing Intensive
  - ▶ Nice computing and communication ratio
  - ▶ Not limited by bandwidth



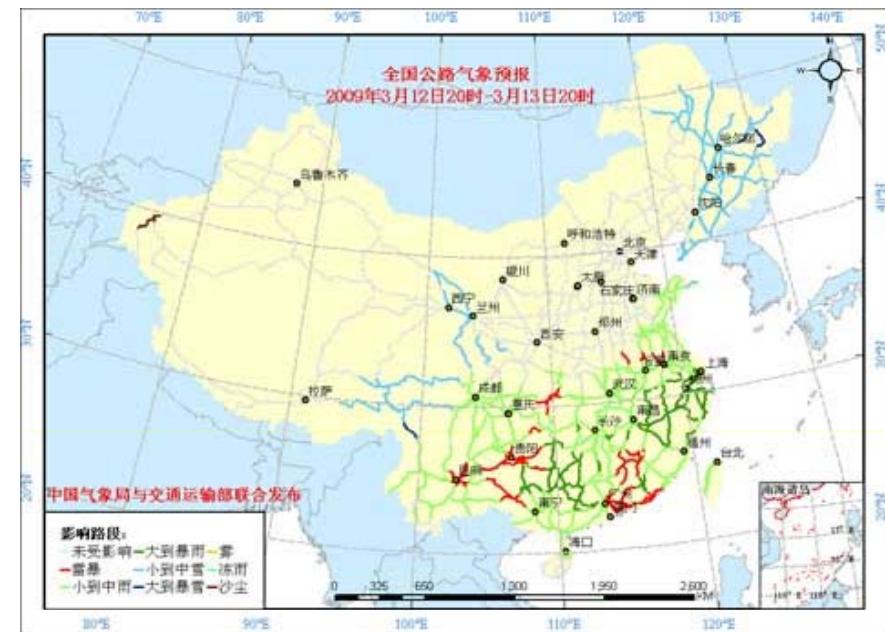
## 举例：石油勘探

- ▶ 目前的CPU只能满足石油勘探的普通处理技术，如解编、预处理、叠后偏移等
- ▶ 目前的CPU不能完全满足
  - ▶ 需要大量运算的处理技术，
  - ▶ 如叠前时间偏移、叠前深度偏移、波动方程偏移等
- ▶ 以叠前偏移为例，一般实现一道
  - ▶ 偏移需要  $1000000 \times 6000 \times 2$  次数学运算，计算量和需要处理的数据量极其巨大



## 举例：气象预报

- ▶ 目前，气象预测对计算资源的需求日益增长
- ▶ 对于24小时的短期预报，要求
  - ▶ 一般在0.5~1小时内得到结果
  - ▶ 对于中期预报（10天，15公里），大概5~6小时
- ▶ 精细化预报
  - ▶ 网格<3km，甚至<1km
  - ▶ <每半小时完成一次



# Simple NN with CUDA/GPU

Bin ZHOU @ USTC

Jan. 2015

# Very Simple digit recognition

---

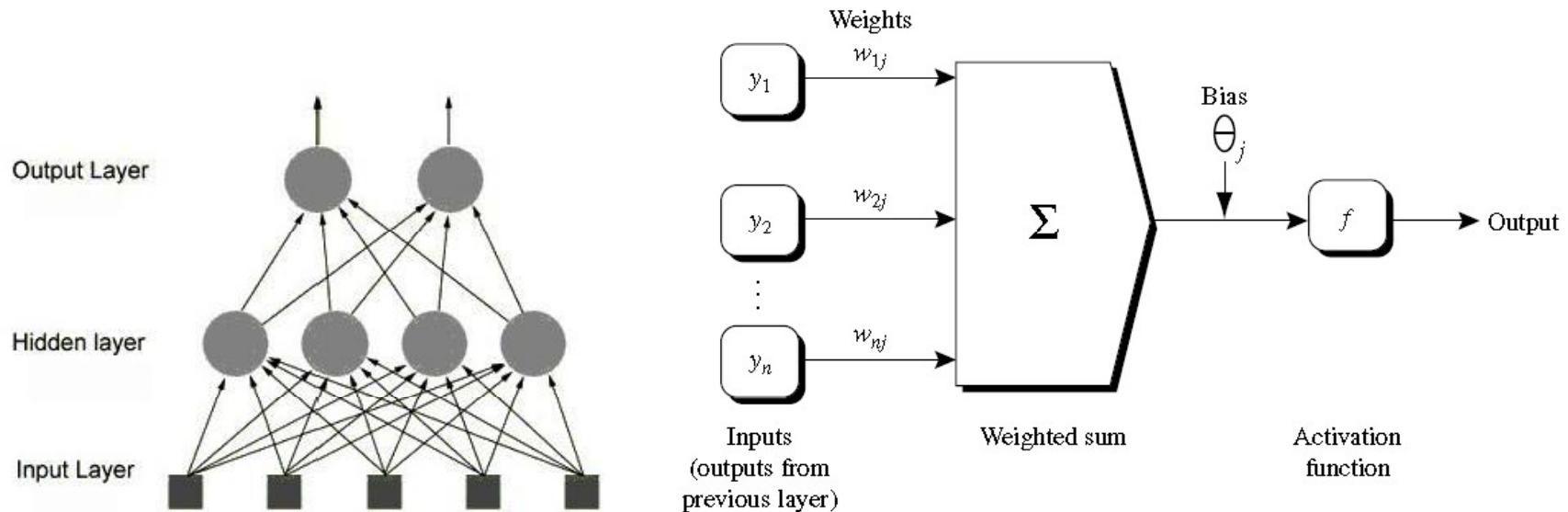
- 3 layers: Simple BP network
  - 1 Input Layer, 1 hidden layer, 1 output layer
- Several Neurons
  - 784 ( 28\*28) input, 100 hidden, 10 output
- Some configuration
  - Activation Function: Sigmoid function

$$f(x) = \frac{1}{1 + e^{-ax}} \quad (0 < f(x) < 1)$$



# You've already known this very well

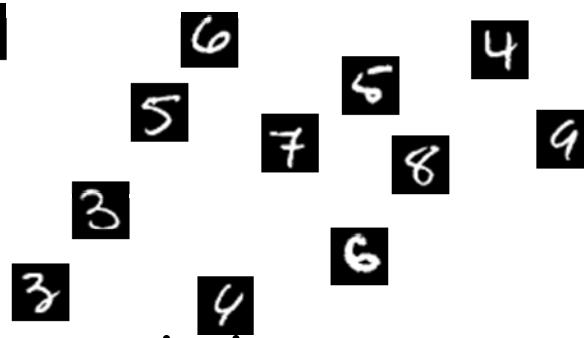
---

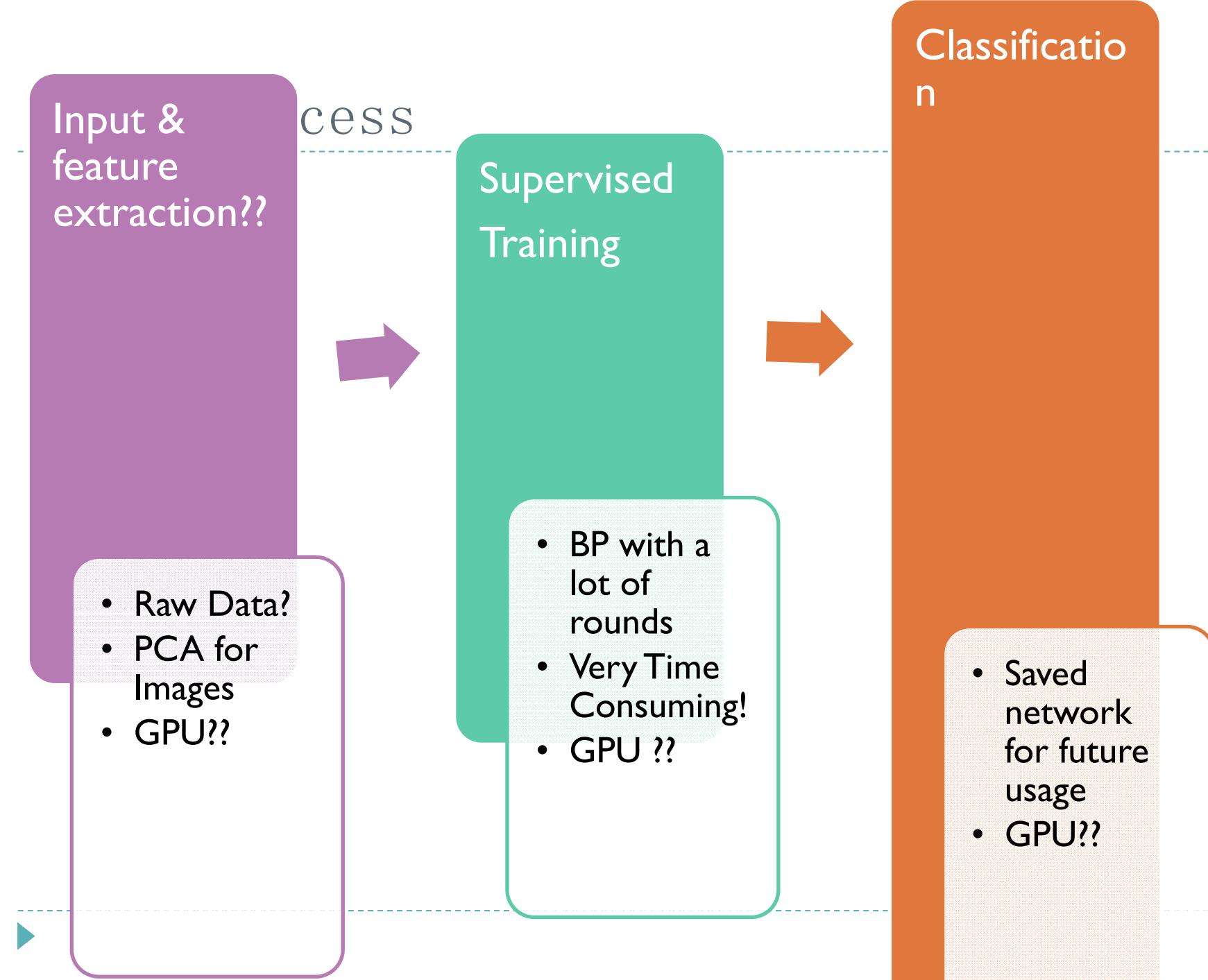


# 10 Digits to Recognize

---

- Training and testing samples are from
- MNIST , <http://yann.lecun.com/exdb/mnist/>
- Every pic is 28\*28
- Totally 10000 pics
- We use 600 of each as training set and 200 as test set.





# Training Process: GPU Accelerated

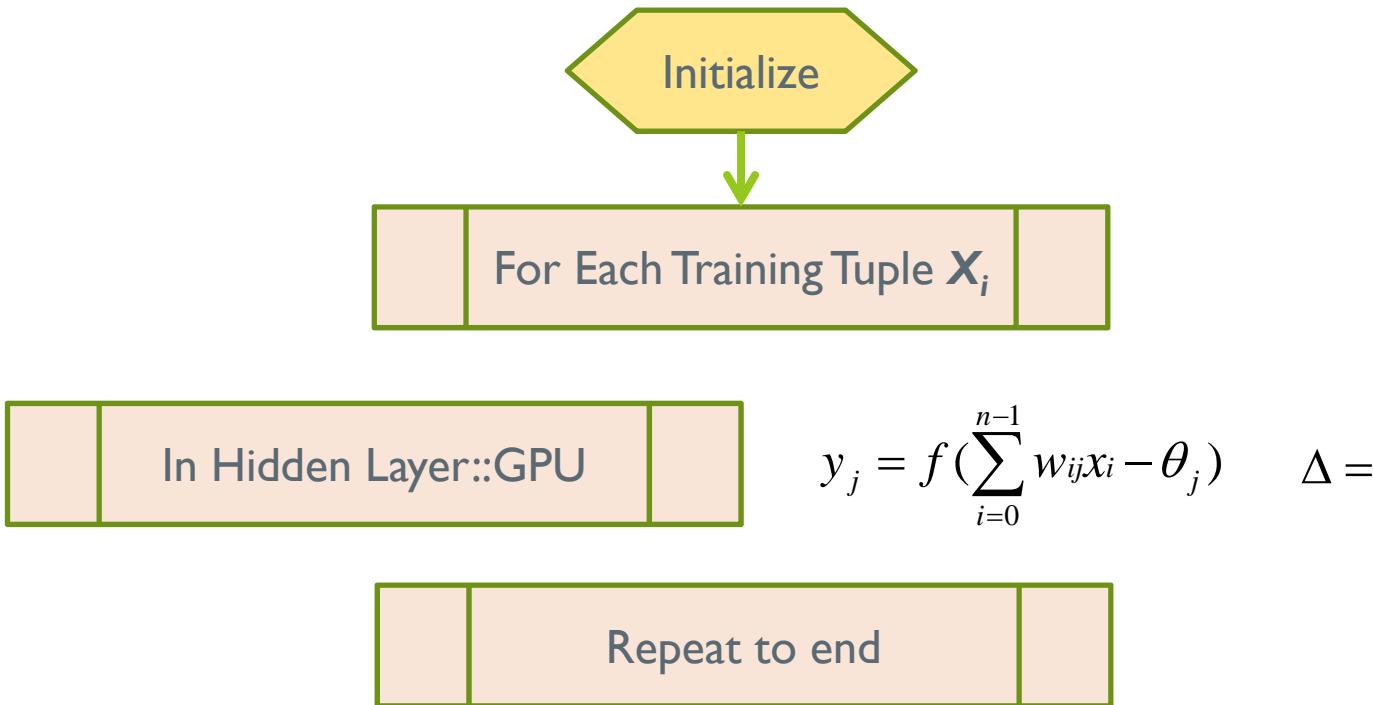
---

- ▶ Which Part?
- ▶ linear Algebra inside single iteration/Sample
- ▶ But not between iterations/Samples
- ▶ Dependency between iterations/Samples



# Single Step Computation

---



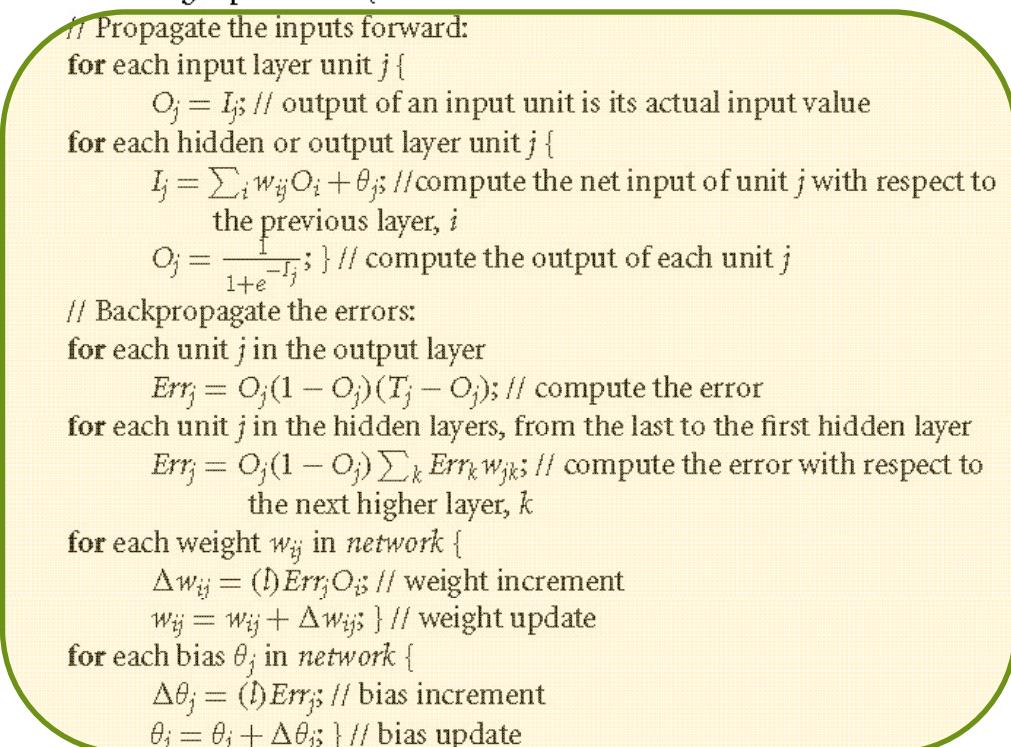
# Algorithm View

---

**Method:**

---

```
(1) Initialize all weights and biases in network;
(2) while terminating condition is not satisfied {
(3) for each training tuple X in D {
(4) // Propagate the inputs forward:
(5) for each input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) for each hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; // compute the net input of unit j with respect to
 the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10) // Backpropagate the errors:
(11) for each unit j in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13) for each unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to
 the next higher layer, k
(15) for each weight w_{ij} in network {
(16) $\Delta w_{ij} = (\eta) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) for each bias θ_j in network {
(19) $\Delta \theta_j = (\eta) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21) } }
```



# GPU Implementation

---

- ▶ Initialize the network on GPU
  - ▶ Hidden Layer Nodes, Weight and Bias
  - ▶ Output Layer Nodes, Weight and Bias
  - ▶ Input dataset
- ▶ Prepare the data to GPU
  - ▶ Pack the batched images in CPU and then
  - ▶ Remember to do it all at once
- ▶ Then start the training for each sample



# Parallelization Strategy

---

- ▶ Each thread is in charge of computing one output of the neuron
- ▶ Not limited by the thread number within a block
- ▶ Back propagation is also the same
- ▶ Very careful about the **Memory Access Pattern!**



## Close look at the code

---

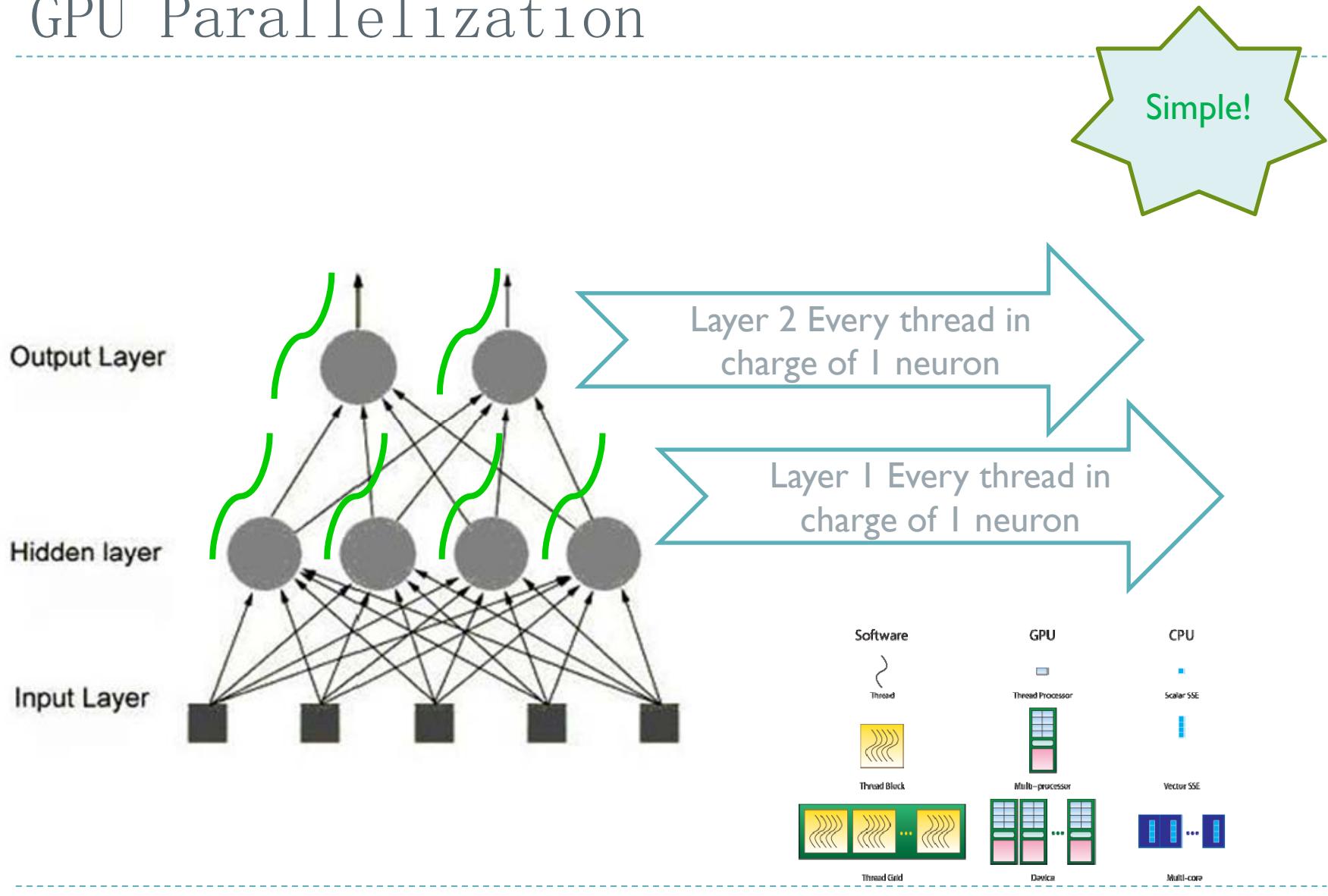
```
for(i=0;i<N0N;i++)
 node0[i].Output=pic[i];
```

```
for(j=0;j<N1N;j++)
{
 node1[j].Input=node1[j].bias;
 for(i=0;i<N0N;i++)
 {
 node1[j].Input+=w01[i][j]*node0[i].Output;
 }
 node1[j].Output=1.0/(1.0+exp(-node1[j].Input));
}
```

j is independent, which can be  
processed parallel



# GPU Parallelization



# Close look at CUDA/GPU code

---

```
__global__ void kL0toL1(float *input, float *output, float *w, float *b)
{
 int nodeNum = threadIdx.x;
 int i = 0;
 float aTmp=0;
 if (nodeNum < N1N)
 {
 aTmp=b[nodeNum];
 for (i = 0; i< N0N; i++)
 aTmp += *(w+i*100+nodeNum)*input[i];
 output[nodeNum] = 1.0/(1.0+exp(-aTmp));
 }
}
```

Every thread in charge of 1 neuron



# Performance Consideration

---

- ▶ Memory Limited ? Instruction Limited?
- ▶ Memory Access Pattern?
  - ▶ Every thread will access `w01[ ][ ]` in a continuous way; Not so good.

Training Perf	i5 2.0G CPU 1 core	Kepler GPU 1 SM
1 image	57ms	1ms



## How to get a Better Solution?

---

- ▶ Memory Access Pattern is the first thing to deal with
- ▶ Put W01 into shared memory is a simple try
- ▶ Redesign the Memory Storage structure
- ▶ Or redesign the Algorithm to avoid the F function

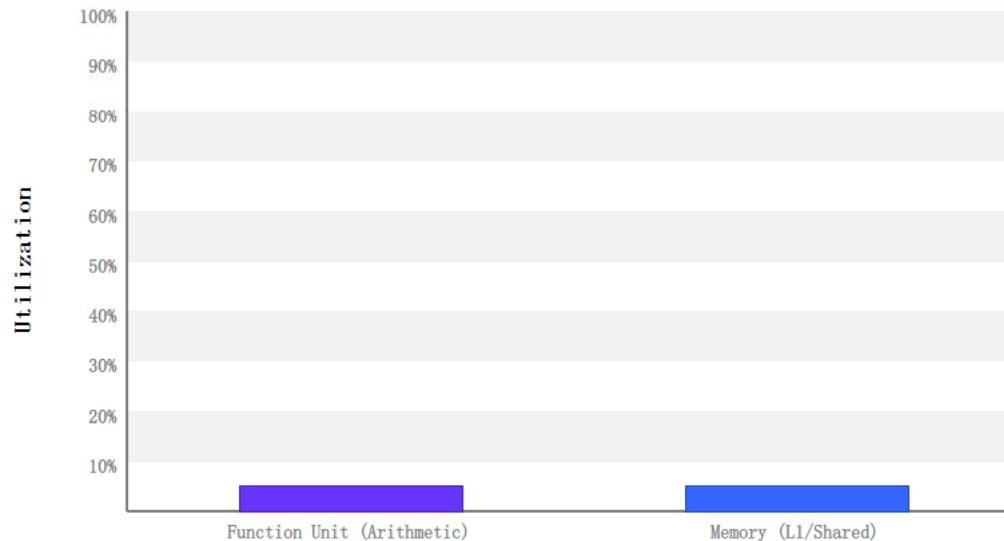


## More Detailed Analysis

---

Performance is bounded by both Arithmetic and Memory latency. Too bad.

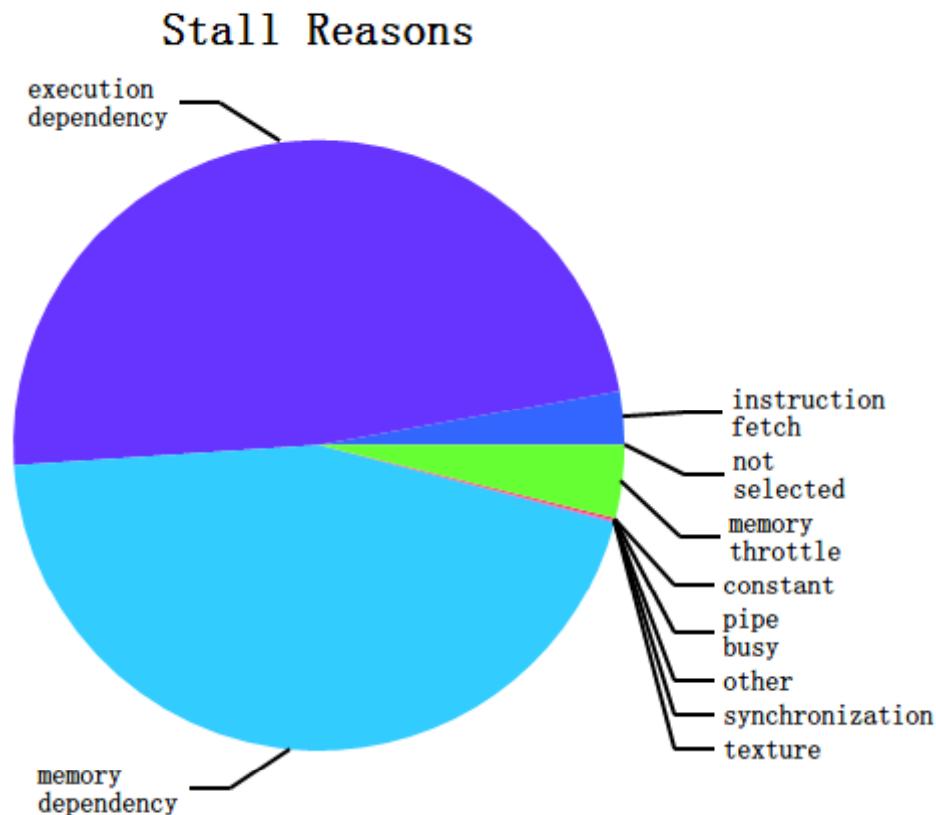
We have only 1 block, far away from filling the SM.



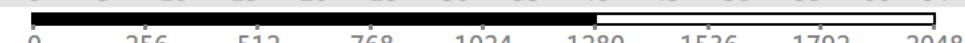
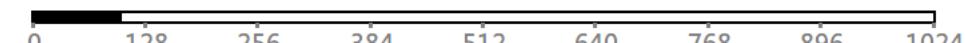
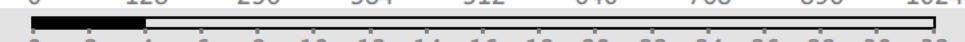
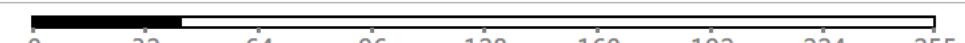
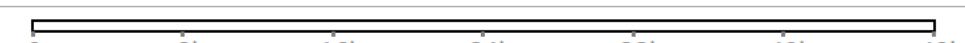
# Kernel Latency

---

- ▶ Grid Size is too small to hide the latency



# Register Analysis

Variable	Achieved	Theoretical	Device Limit	Grid Size: [ 1,1,1 ] (1 block) Block Size: [ 100,1,1 ] (100 threads)
<b>Occupancy Per SM</b>				
Active Blocks		10	16	
Active Warps	3.93	40	64	
Active Threads		1280	2048	
Occupancy	6.1%	62.5%	100%	
<b>Warp Metrics</b>				
Threads/Block		100	1024	
Warps/Block		4	32	
Block Limit		16	16	
<b>Registers</b>				
Registers/Thread		42	255	
Registers/Block		6144	65536	
Block Limit		10	16	
<b>Shared Memory</b>				
Shared Memory/Block		0	49152	
Block Limit			16	



# Kernel Memory

	Transactions	Bandwidth	Utilization						
<b>L1/Shared Memory</b>									
Local Loads	0	0 B/s							
Local Stores	0	0 B/s							
Shared Loads	0	0 B/s							
Shared Stores	0	0 B/s							
Global Loads	8334	857.238 MB/s							
Global Stores	4	767.657 kB/s							
Atomic	0	0 B/s							
<b>L1/Shared Total</b>	<b>8338</b>	<b>858.005 MB/s</b>	Idle	Low	Medium	High	Max		
<b>L2 Cache</b>									
L1 Reads	14517	857.238 MB/s							
L1 Writes	13	767.657 kB/s							
Texture Reads	0	0 B/s							
Atomic	0	0 B/s							
Noncoherent Reads	0	0 B/s							
<b>Total</b>	<b>14530</b>	<b>858.005 MB/s</b>	Idle	Low	Medium	High	Max		
<b>Texture Cache</b>									
Reads	0	0 B/s	Idle	Low	Medium	High	Max		
<b>Device Memory</b>									
Reads	8930	527.322 MB/s							
Writes	14	826.708 kB/s							
<b>Total</b>	<b>8944</b>	<b>528.149 MB/s</b>	Idle	Low	Medium	High	Max		
<b>System Memory</b> [ PCIe configuration: Gen2 x4, 5 Gbit/s ]									
Reads	0	0 B/s	Idle	Low	Medium	High	Max		
Writes	1	59.05 kB/s	Idle	Low	Medium	High	Max		



# Target

Line	Exec Count	File - /C:/Users/zhoubin/Documents/Visual Studio 201
186		1
187	12	int nodeNum = threadIdx.x;
188		int i = 0;
189		float aTmp=0;
190		
191	24	if (nodeNum < N1N)
192		{
193	24	aTmp=b[nodeNum];
194		
195	18840	for (i = 0; i < N0N; i++)
196	40768	aTmp += *(w+i*100+nodeNum)*input[i];
197		
198	80	output[nodeNum] = 1.0/(1.0+exp(-aTmp))
199		}
200	16	}
201		
202		
203		_global_ void kL1toL2(float *input, float *ou
204		{
205		int nodeNum = threadIdx.x;
206		int i = 0;

Line	Exec Count	Disassembly
186		.L_30:
187	3140	ISETP. LT. AND P0, PT, R5, 0x310, PT;
188	3140	PSETP. AND. AND P0, PT, !P0, PT, PT;
189	3140	@P0 BRK;
190	3136	BRA `(.L_29);
191		.L_29:
192	3136	IMUL R6, R5, 0x64;
193	3136	SHL R6, R6, 0x2;
194	3136	IADD R6, R3, R6;
195	3136	SHL R7, R16, 0x2;
196	3136	IADD R6, R6, R7;
197	3136	MOV R6, R6;
198	3136	LD R7, [R6];
199	3136	SHL R6, R5, 0x2;
200	3136	IADD R6, R0, R6;
201	3136	MOV R6, R6;
202	3136	LD R6, [R6];
203	3136	FMUL R6, R7, R6;
204	3136	FADD R4, R4, R6;
205		.L_40:
206	3136	IADD R5, R5, 0x1;

# Very Brief Review of Parallel Computing

Bin ZHOU @ NVIDIA & USTC  
Jan. 2015

# Acknowledgements

---

- ▶ Florent NOLOT, Univ-reims, france, HPC
- ▶ Introduction to Parallel Processing. Shantanu Dutt. University of Illinois at Chicago



# Contents

---

- ▶ Why do we need parallel computing?
- ▶ How?
- ▶ Some concepts and ideas



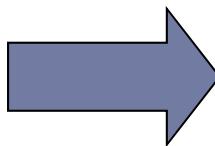
# Parallel Processing is common in real life

---

- ▶ Task:
- ▶ Moving
- ▶ A Pile of
- ▶ Bricks



# Serial Processing Strategy



► CPU way of processing?

# Parallel Processing Strategy



# Moore's Law

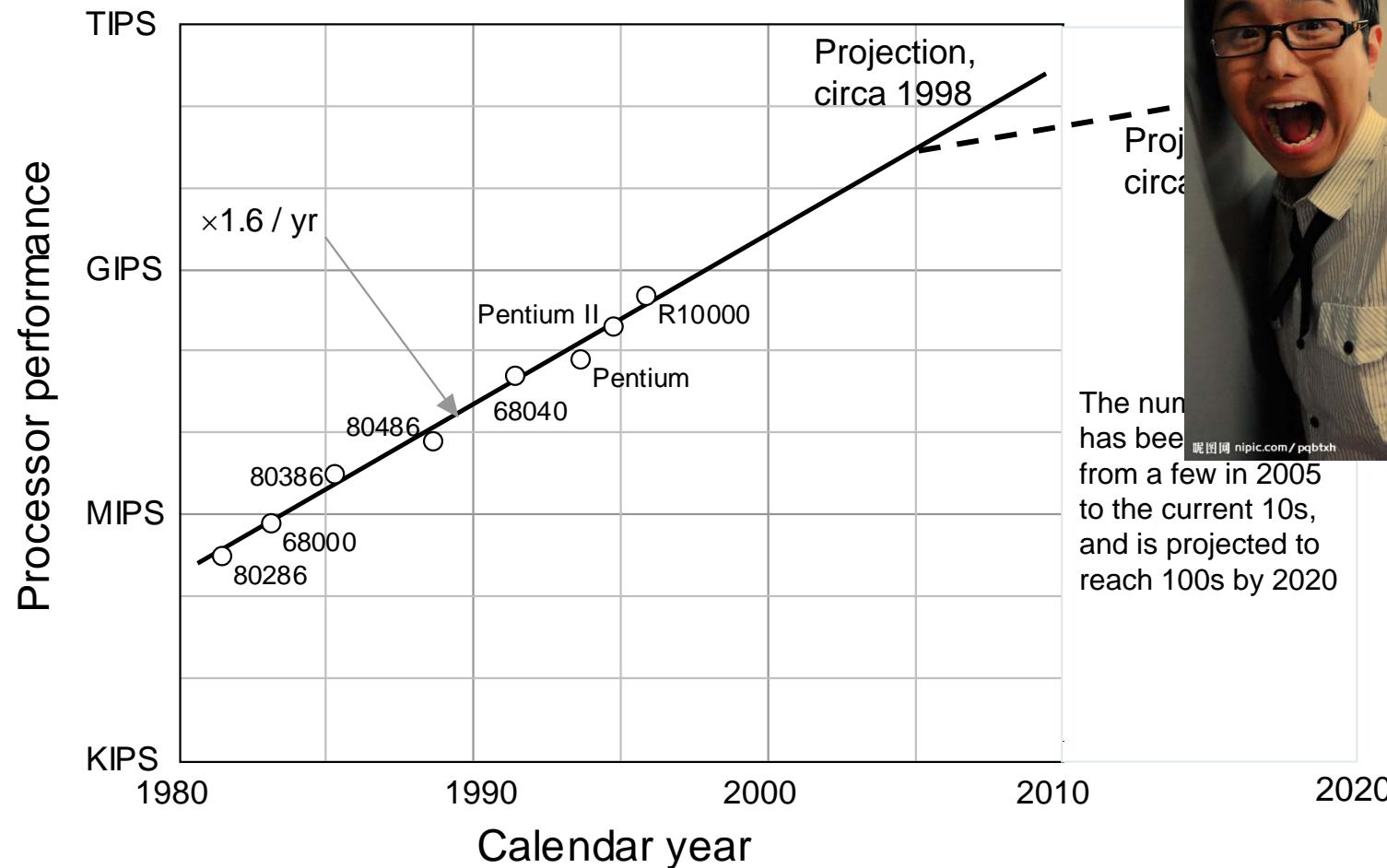
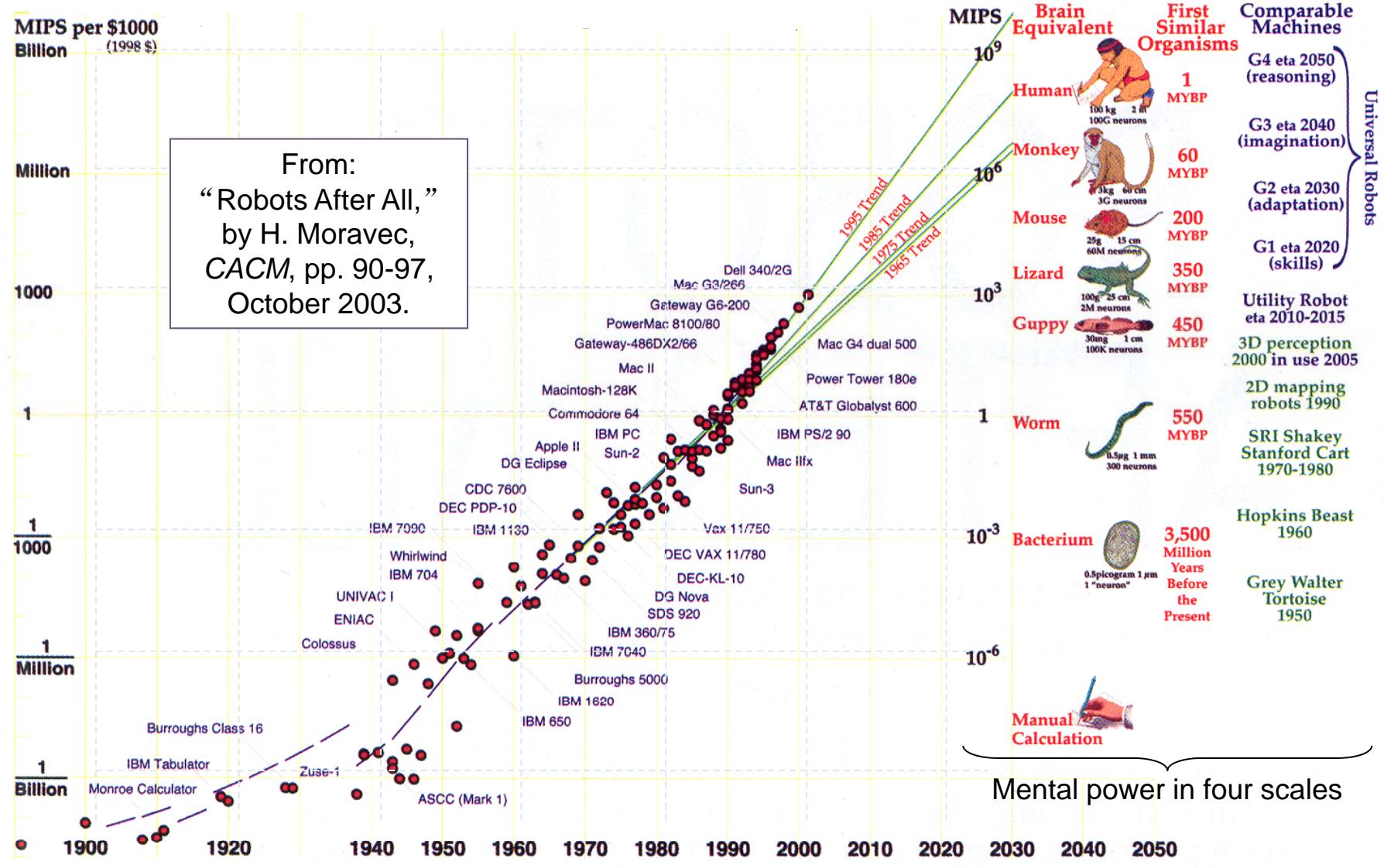


Fig. 1.1 Moore's Law again(extrapolated).

# Performance/Cost



# Semi-conductor Industry Roadmap

---

Year →	2001	2004	2007	2010	2013	2016	2015	2020	2025
width (nm)	140	90	65	45	32	22	19	12	8
frequency. (GHz)	2	<del>4</del>	<del>7</del>	<del>3.6</del>	<del>12</del>	<del>4.1</del>	<del>20</del>	<del>4.6</del>	<del>30</del>
Wiring levels	7	8	9	10	10	10			
Voltage (V)	1.1	1.0	0.8	0.7	0.6	0.5			0.6
Power (W)	130	160	190	220	250	290			

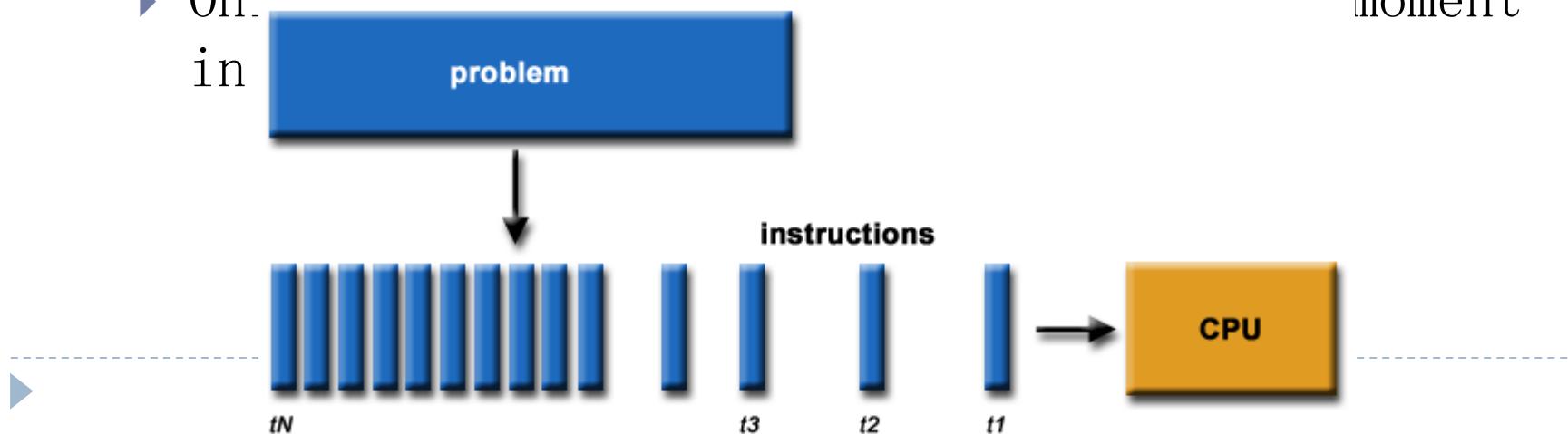
From the 2001 edition of the roadmap [Alla02]

From the 2011 edition  
(Executive Summary)



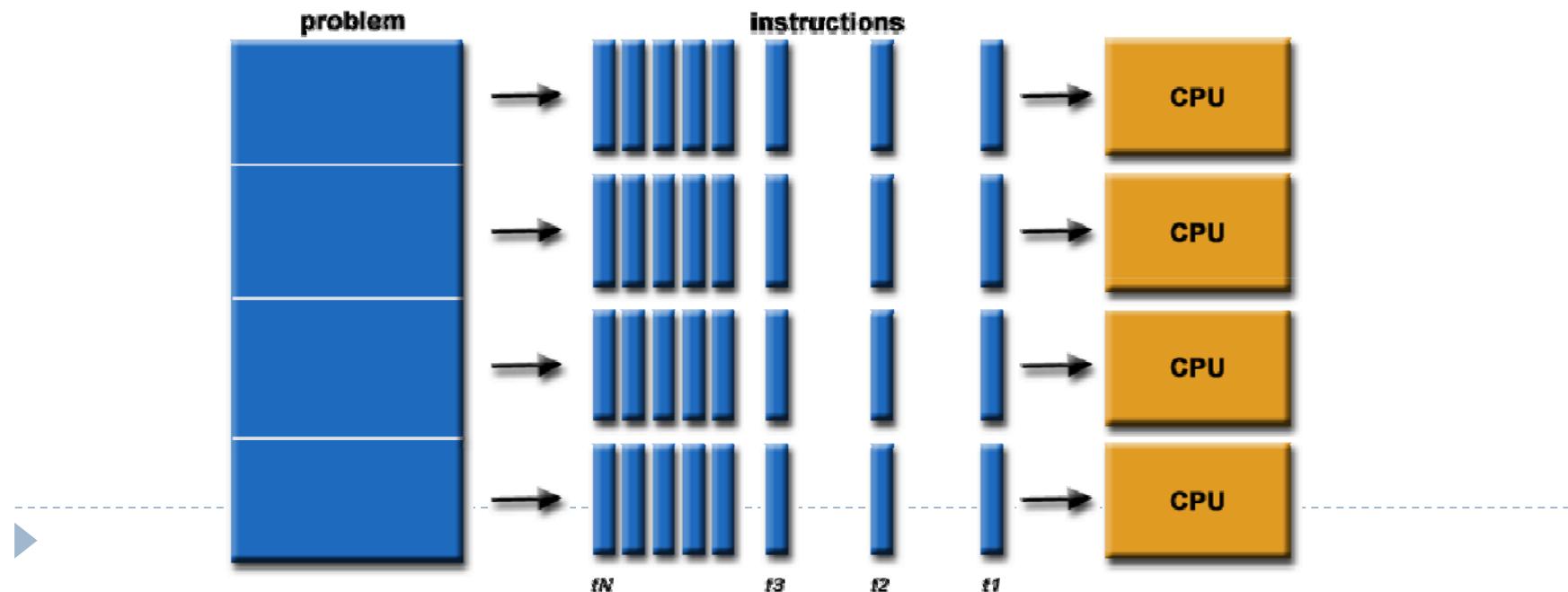
# Serial Processing Model

- ▶ Traditionally, software has been written for serial computation:
  - ▶ To be run on a single computer having a single Central Processing Unit (CPU);
  - ▶ A problem is broken into a discrete series of instructions.
  - ▶ Instructions are executed one after another.
  - ▶ Only one instruction may execute at any moment in time.



# Parallel Processing Model

- ▶ In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve **one** computational problem.
  - ▶ To be run using multiple CPUs
  - ▶ A problem is broken into discrete parts that can be solved concurrently
  - ▶ Each part is further broken down to a series of instructions
- ▶ Instructions from each part execute simultaneously on different CPUs



# What problems can parallel computing solve?

---

- ▶ So many.....
  
- ▶ X
- ▶ XX
- ▶ XXX
- ▶ XXXX

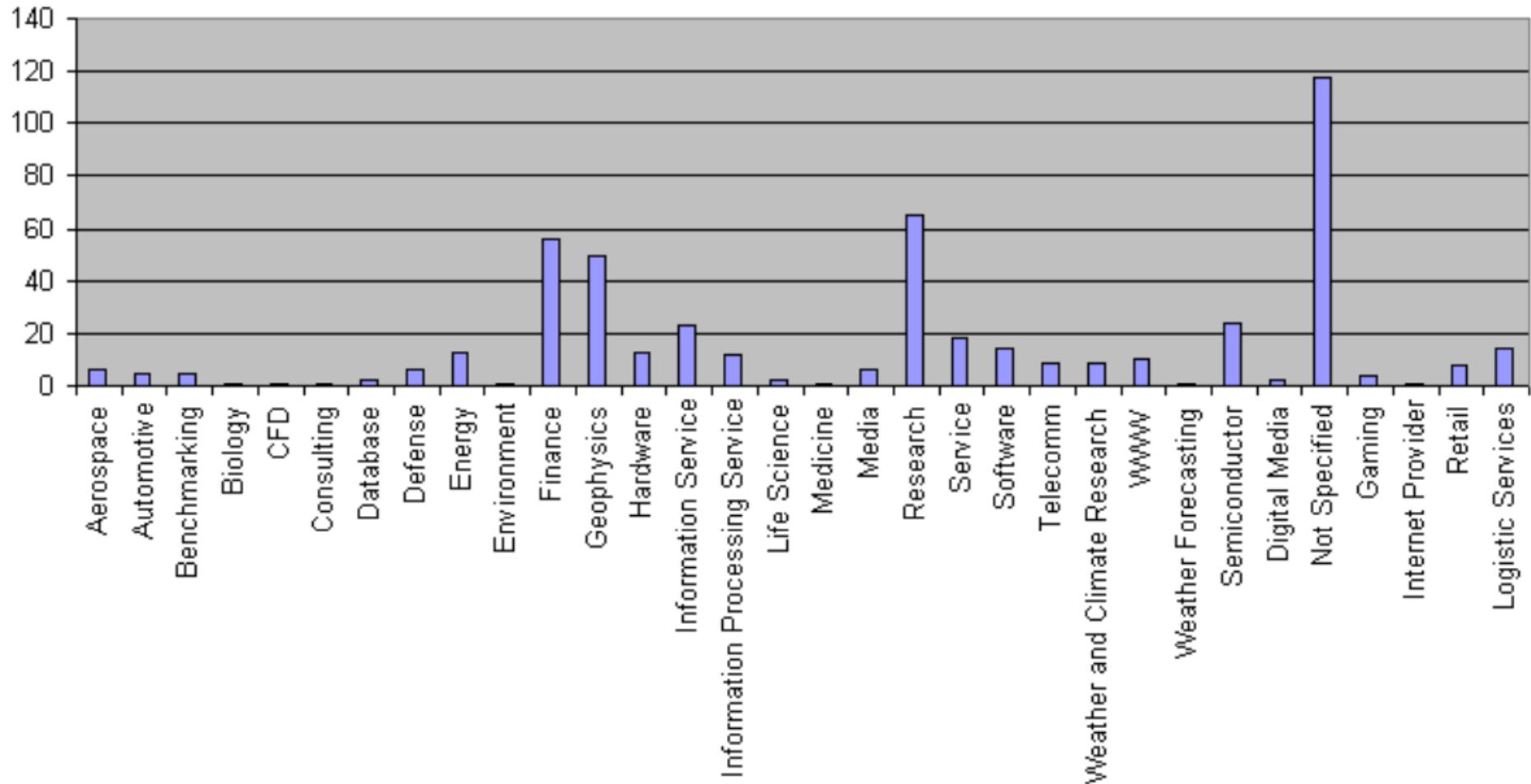


# The Need for Speed: Complex Problems

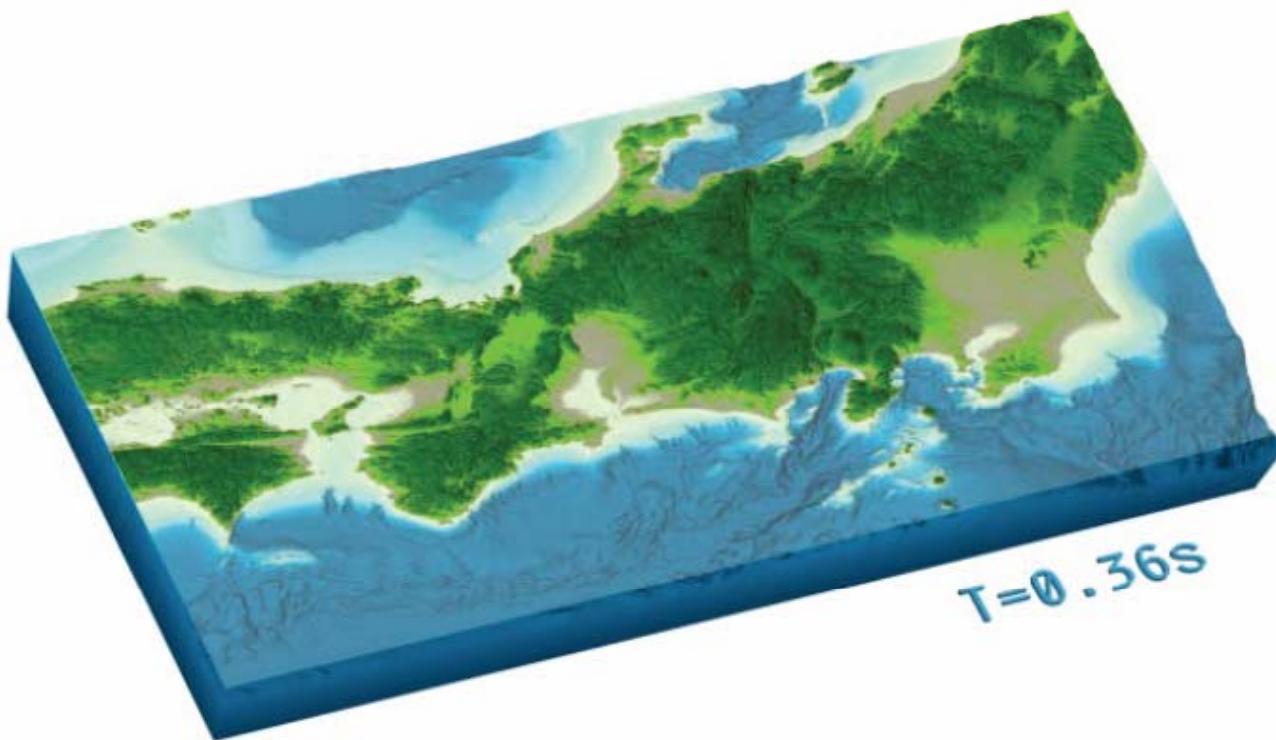
---

- **Science**
  - understanding matter from elementary particles to cosmology
  - storm forecasting and climate prediction
  - understanding biochemical processes of living organisms
- **Engineering**
  - combustion and engine design
  - computational fluid dynamics and airplane design
  - earthquake and structural modeling
  - pollution modeling and remediation planning
  - molecular nanotechnology
- **Business**
  - computational finance - high frequency trading
  - information retrieval
  - data mining
- **Defense**
  - nuclear weapons stewardship
  - cryptology

# Applications of Parallel Processing



# Earthquake Simulation

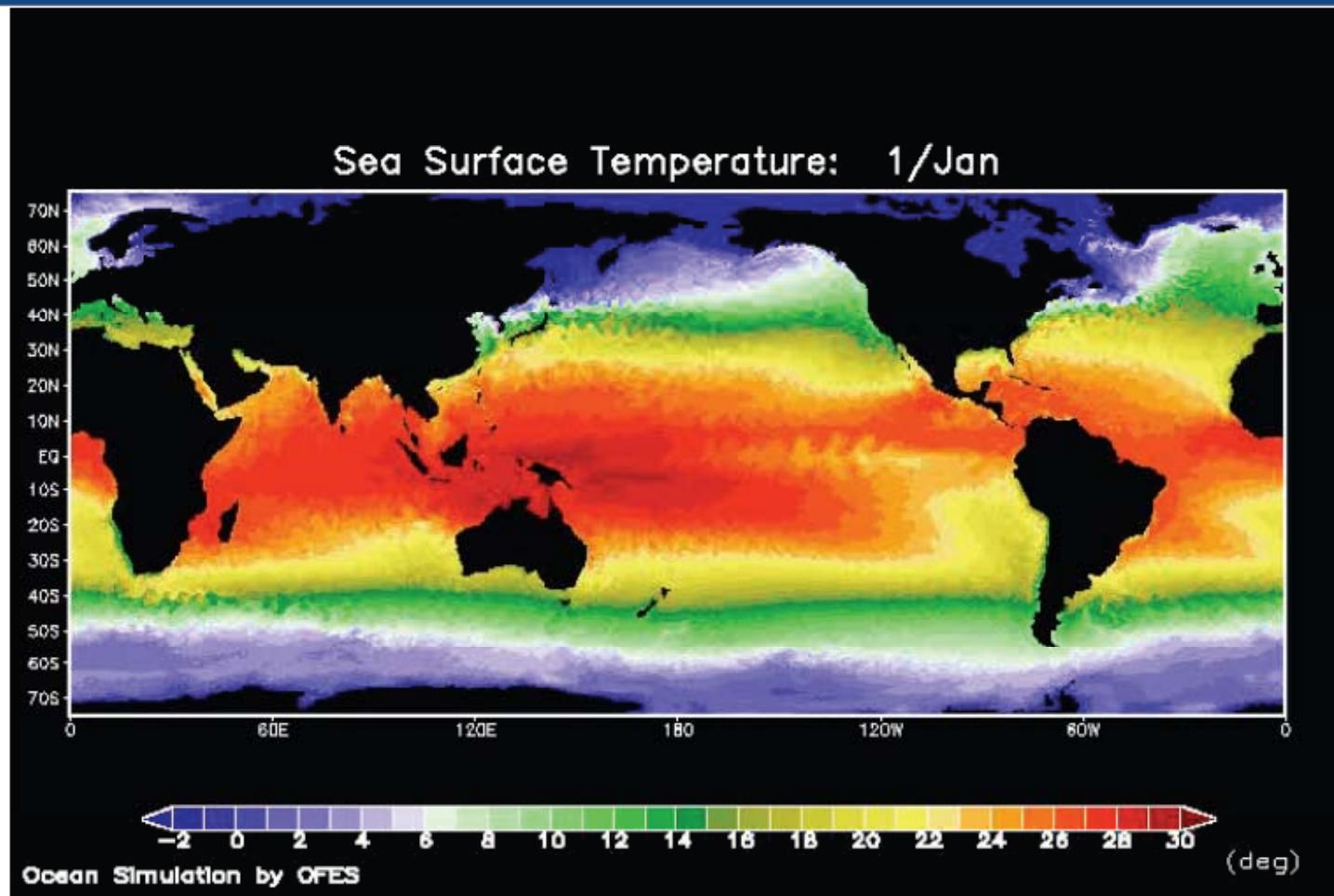


**Earthquake Research Institute, University of Tokyo**

**Tonankai-Tokai Earthquake Scenario**

**Photo Credit: The Earth Simulator Art Gallery, CD-ROM, March 2004**

# Ocean Circulation Simulation



**Ocean Global Circulation Model for the Earth Simulator  
Seasonal Variation of Ocean Temperature  
Photo Credit: The Earth Simulator Art Gallery, CD-ROM, March 2004**

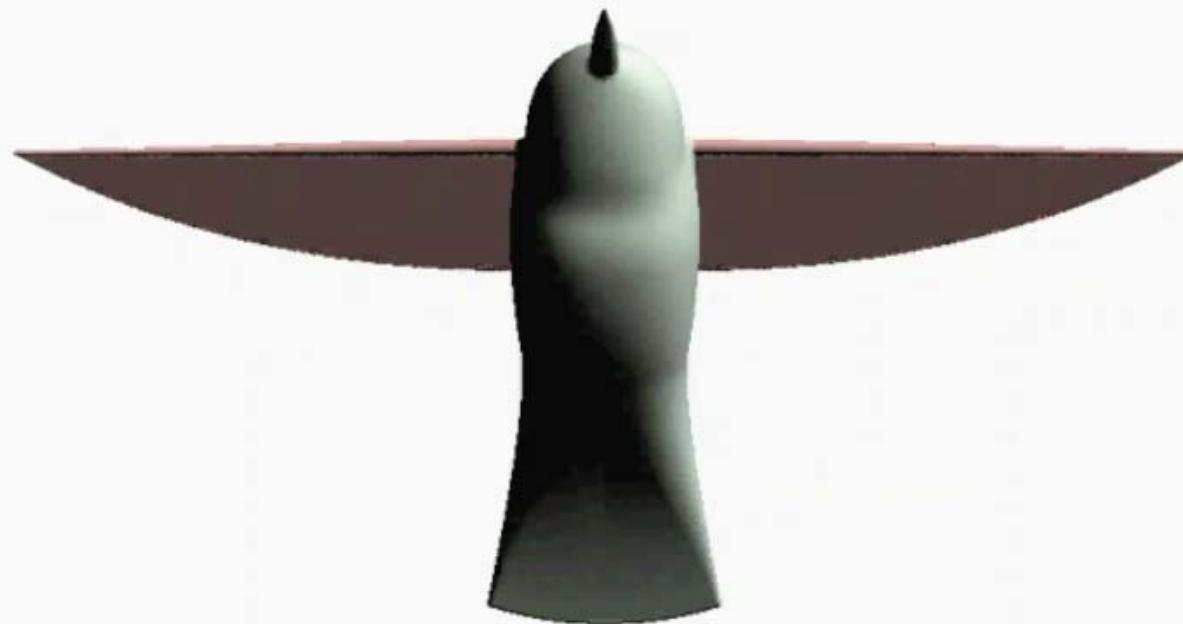
# Fluid-Structure Interactions

---

- Simulate ...
  - rotational geometries (e.g. engines, pumps), flapping wings
- Traditionally, such simulations have used a fixed mesh
  - drawback: solution quality is only as good as initial mesh
- Dynamic mesh computational fluid dynamics
  - integrate automatic mesh generation within parallel flow solver
    - nodes added in response to user-specified refinement criteria
    - nodes deleted when no longer needed
    - element connectivity changes to maintain minimum energy mesh
  - mesh changes continuously as geometry + solution changes
- Example: 3D simulation of a hummingbird's flight

[Andrew Johnson, AHPCRC 2005]

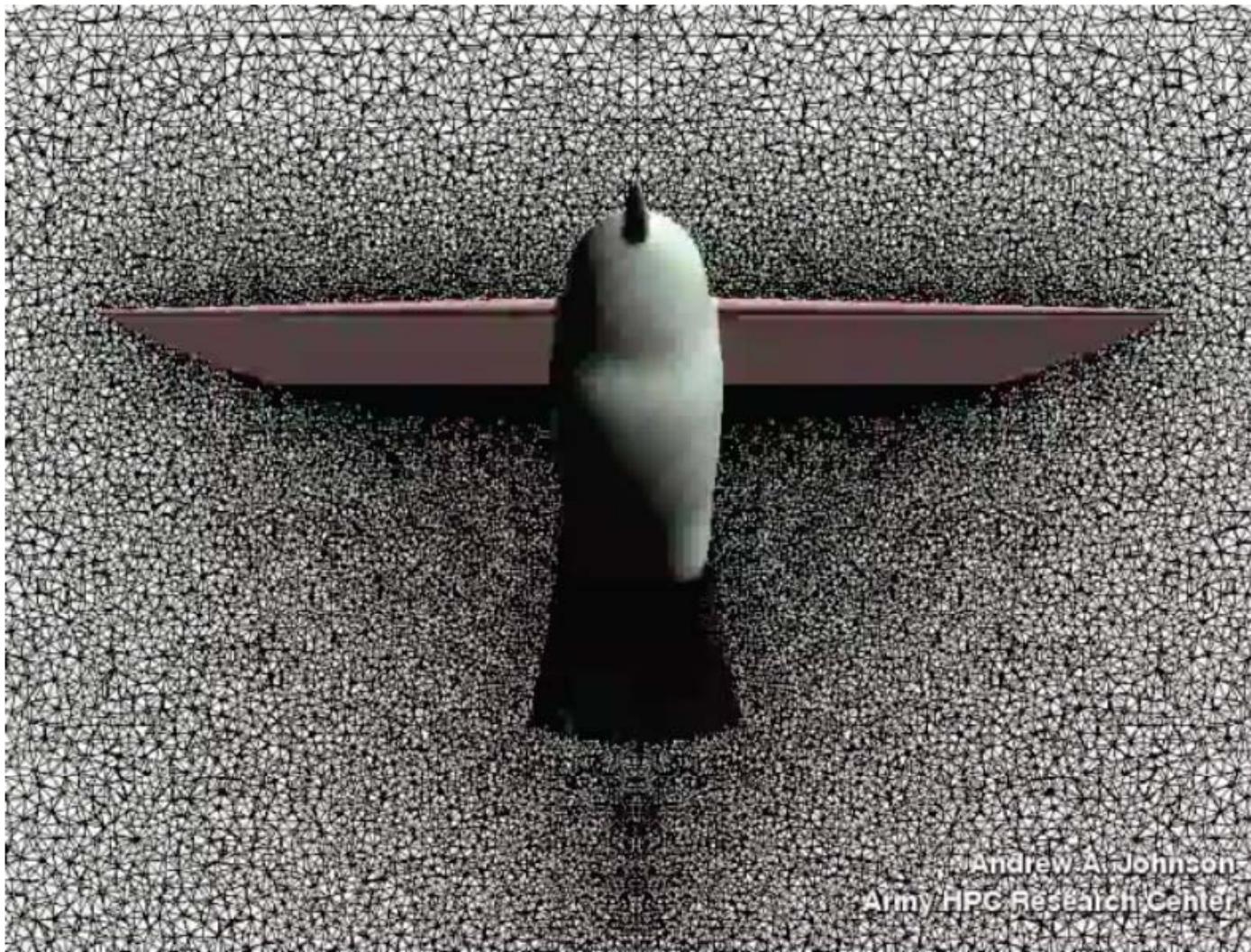
## Air Velocity (Front)



Andrew A. Johnson  
Army HPC Research Center



# Mesh Adaptation (front)



Andrew A. Johnson  
Army HPC Research Center

# Some lists

---

- ▶ weather and climate
- ▶ chemical and nuclear reactions
- ▶ biological, human genome
- ▶ geological, seismic activity
- ▶ mechanical devices – from prosthetics to spacecraft
- ▶ electronic circuits
- ▶ manufacturing processes



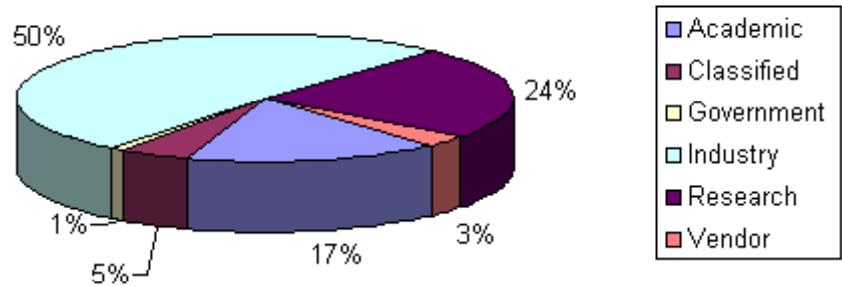
# Daily life related

---

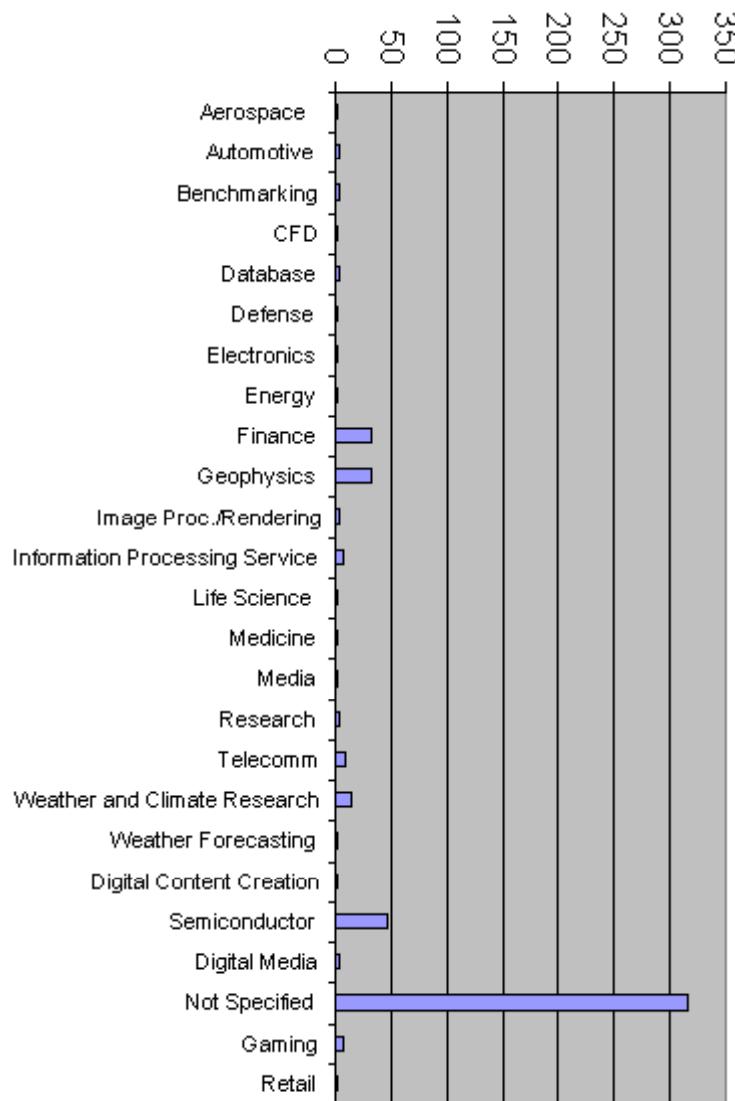
- ▶ Data Mining
  - ▶ Search Engine,
  - ▶ Online Shopping
  - ▶ Medicine
  - ▶ Game
  - ▶ Video, Virtual Reality
  - ▶ Even Cellphones
  - ▶ Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time.
-

# Parallel Computing Usage

## Who's Doing Parallel Computing?



*Data obtained from top500.org, June 2006*



**What Are They Using it For?**

---

# Basic Concepts and Terms



# Flynn Matrix

---

- ▶ The matrix below defines the 4 possible classifications according to Flynn

<b>S I S D</b>	<b>S I M D</b>
<b>Single Instruction, Single Data</b>	<b>Single Instruction, Multiple Data</b>
<b>M I S D</b>	<b>M I M D</b>
<b>Multiple Instruction, Single Data</b>	<b>Multiple Instruction, Multiple Data</b>



# Terms in Parallel Computing

---

- ▶ Task
- ▶ Parallel Task
- ▶ Serial Execution
- ▶ Parallel Execution
- ▶ Shared Memory
- ▶ Distributed Memory
- ▶ Communications
- ▶ Synchronization
- ▶ Granularity
- ▶ Observed Speedup
- ▶ Parallel Overhead
- ▶ Scalability



# Parallel Computer Memory Architectures

---

- ▶ Shared Memory
- ▶ Distributed Memory
- ▶ Hybrid Distributed–Shared Memory



# Parallel Programming Models

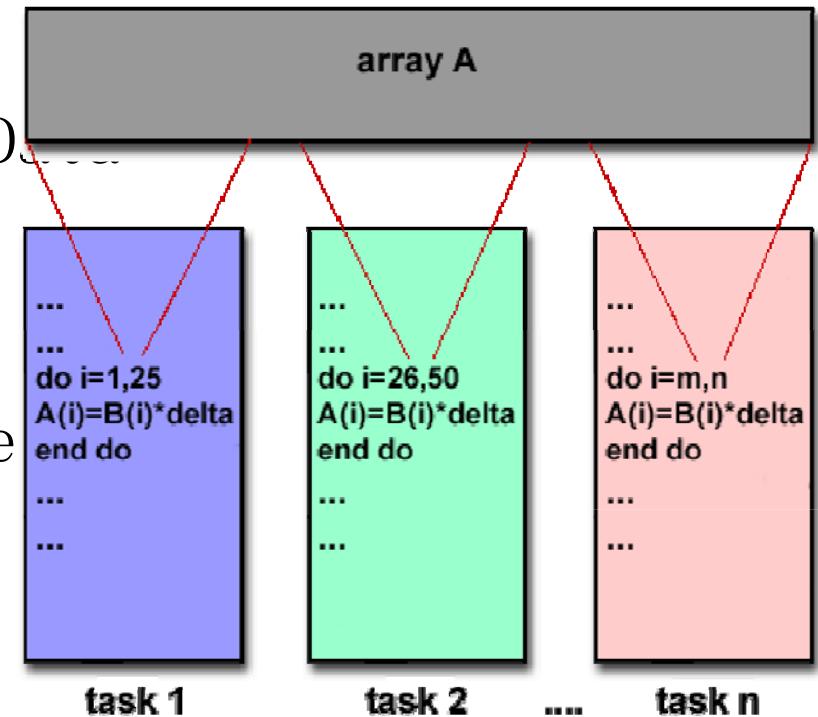
---

- ▶ Shared Memory Model
- ▶ Threads Model
- ▶ Message Passing Model
- ▶ Data Parallel Model



# Some Implementations

- ▶ OpenMP
- ▶ MPI
- ▶ Single Program Multiple Data (SPMD)
- ▶ (SPMD)
  - ▶ Multiple Program Multiple Data (MPMD)
  - ▶ (MPMD)



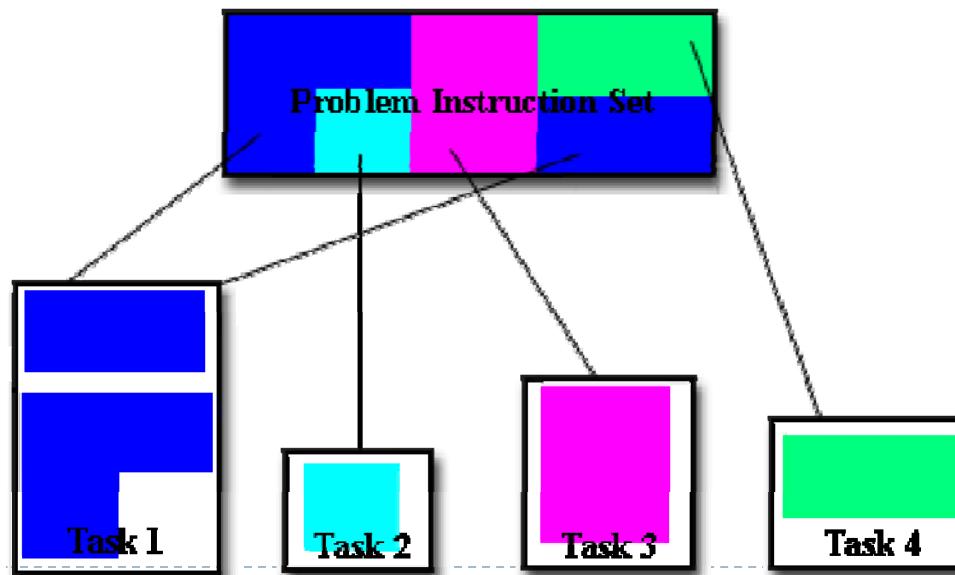
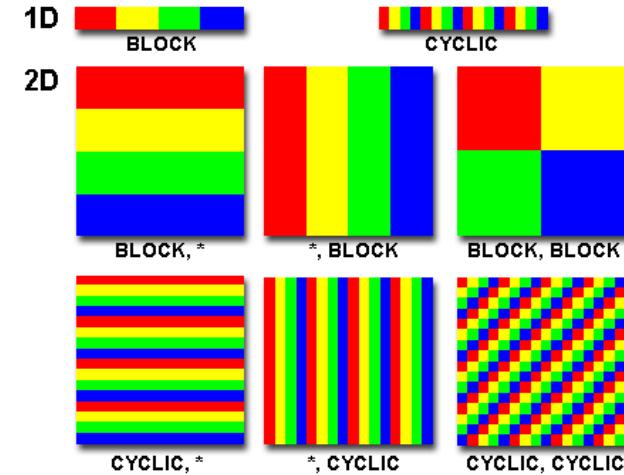
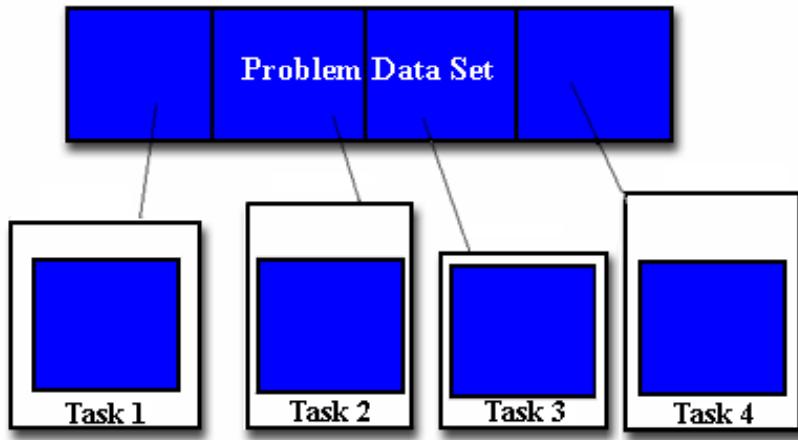
# Designing Parallel Programs

---

- ▶ Automatic vs. Manual Parallelization
- ▶ Understand the Problem and the Program
- ▶ Partitioning
- ▶ Communications
- ▶ Synchronization
- ▶ Data Dependencies
- ▶ Load Balancing
- ▶ Granularity
- ▶ I/O
- ▶ Limits and Costs of Parallel Programming
- ▶ Performance Analysis and Tuning

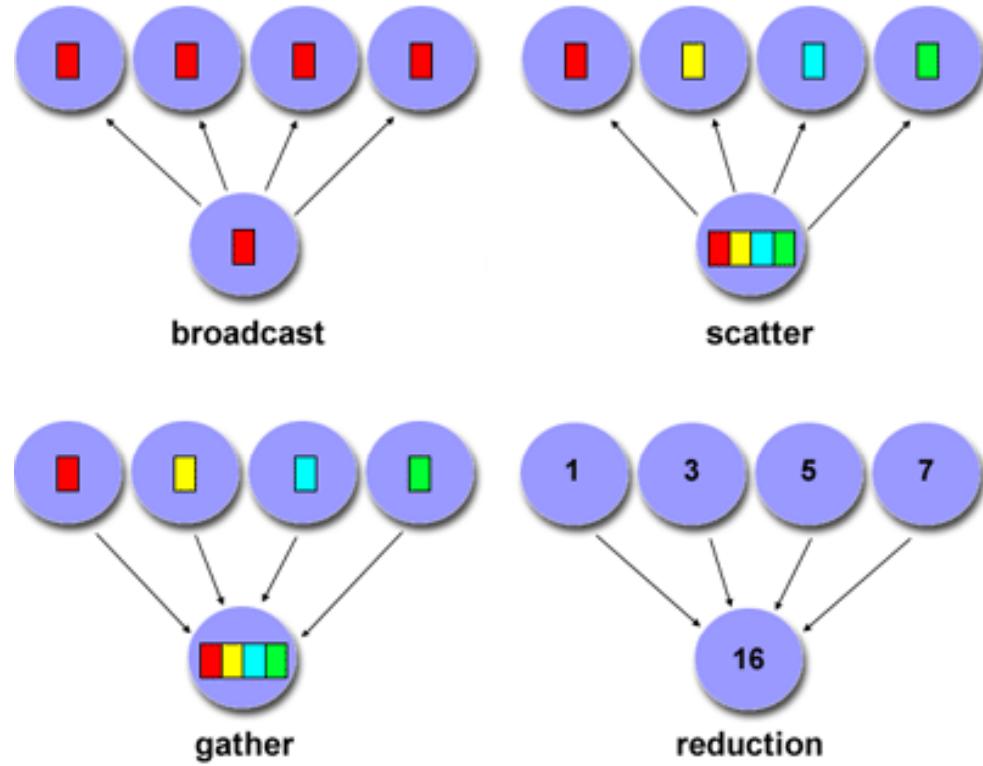


# Domain Decomposition



# Communication and Synchronization

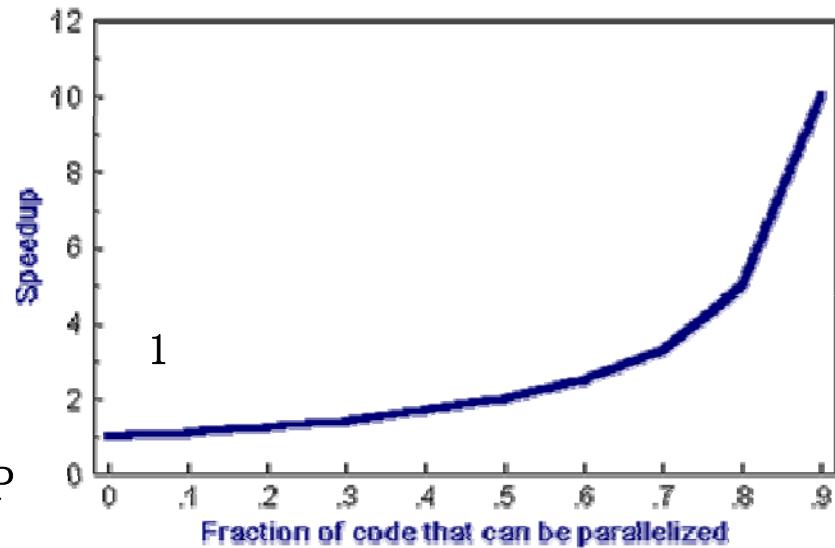
- ▶ Considerations:
  - ▶ Barrier
  - ▶ Lock / semaphore
  - ▶ Synchronous communication



# Amdahl's Law

■ Amdahl's Law states that potential program speedup is defined by the fraction of code ( $P$ ) that can be parallelized:

$$\text{speedup} = \frac{1}{1 - P}$$



- ▶ If none of the code can be parallelized,  $P = 0$  and the speedup = 1 (no speedup). If all of the code is parallelized,  $P = 1$  and the speedup is infinite (in theory).
- ▶ If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast

## Amdahl's Law

---

- ▶ Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by

$$\text{speedup} = \frac{1}{\frac{P + S}{N}}$$

- ▶ where  $P$  = parallel fraction,  $N$  = number of processors and  $S$  = serial fraction



## Amdahl's Law

---

- ▶ It soon becomes obvious that there are limits to the scalability of parallelism. For example, at  $P = .50$ ,  $.90$  and  $.99$  (50%, 90% and 99% of the code is parallelizable)

**speedup**

N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

