

Lecture 5

An improved matrix multiply

Occupancy

Thread divergence

Today's lecture

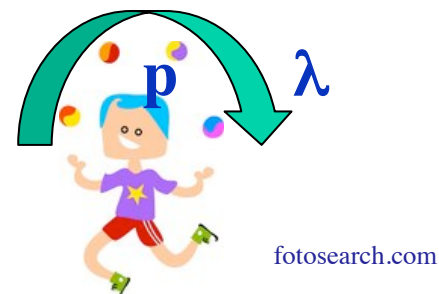
- Occupancy and latency
- Further Improvements to Matrix Multiply
- Thread divergence

Projects

- Counts for 65% of your grade
- Complete in 7 weeks
 - ♦ Proposal: in class presentation + writeup (1/27)
 - ♦ Progress report #1: presentation (2/10)
 - ♦ Progress report #2: writeup (2/26)
 - ♦ Final Presentation: 3/13*
 - ♦ Final Report: 3/16 (Monday) at 5pm

Occupancy

- A minimum number of warps needed to hide memory latency
- **Occupancy: # active warps $\div$ max # warps supported by vector unit**
- Limited by vector unit resources
 - ♦ Amount of shared memory
 - ♦ Number of registers
 - ♦ Maximum number of threads
- Consider a kernel (16x16 block size)
 - ♦ Shared memory/block = 4096 bytes
 - ♦ Reg/thread=30
$$2048 \text{ max threads/SMX} = 8 \text{ blocks}$$
$$30 \text{ reg/blk} \times 256 \text{ thrds/blk} \times 8 \text{ blks} = 61440 < 64k$$
 - ♦ # available regs not limiting factor are close
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
 - ♦ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
 - ♦ Register consumption
- Maximizing the occupancy may not maximize performance



Occupancy calculation with 16 x 16 threads

$$\text{Occupancy} = \frac{\# \text{ active warps per SM}}{\text{Maximum possible } \# \text{ active warps}}$$

Physical Limits for GPU Compute Capability:

2.0

Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	64
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity	2
Maximum Thread Block Size	1024

CUDA GPU Occupancy Calculator

1.) Select Compute Capability (click):	2.0
1.b) Select Shared Memory Size Config (bytes)	49152

2.) Enter your resource usage:

Threads Per Block	256
Registers Per Thread	25
Shared Memory Per Block (bytes)	0

(Don't edit anything below this line)

3.) GPU Occupancy Data

Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	67%

= Allocatable
Blocks Per

Allocated Resources	Per Block	Limit Per SM	SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	38	4
Shared Memory (Bytes)	0	49152	8

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor

Blocks/SM * Warps/Block = Warps/SM

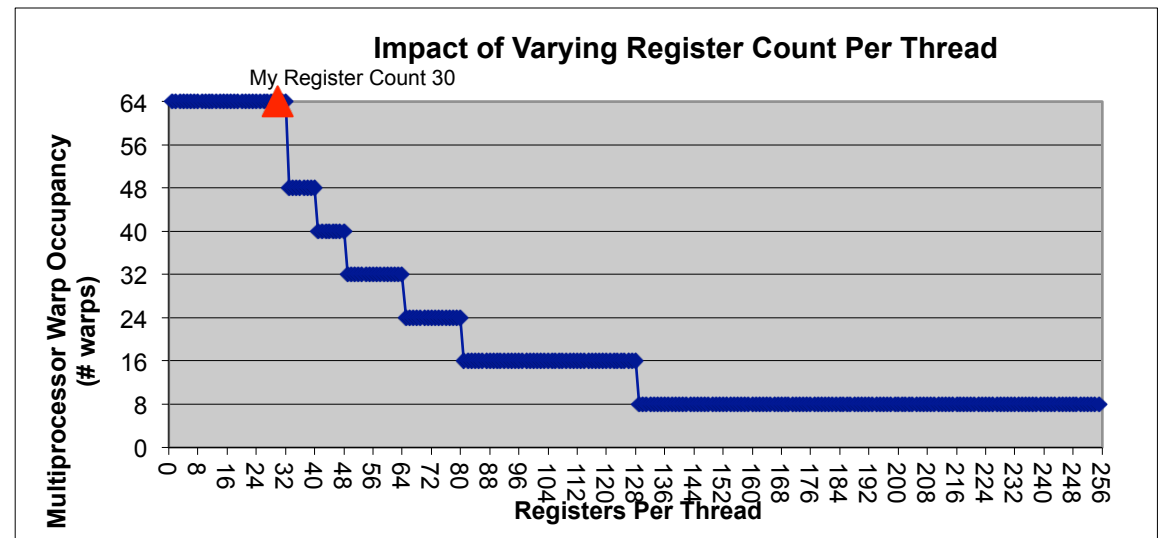
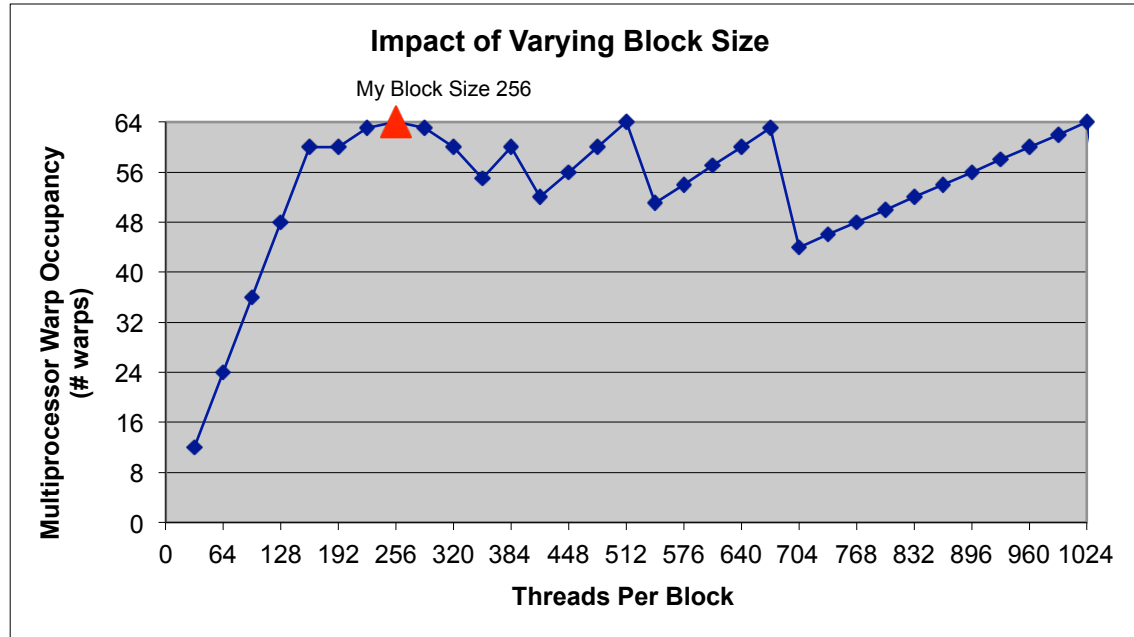
Limited by Max Warps or Max Blocks per Multiprocessor	6		
Limited by Registers per Multiprocessor	4	8	32
Limited by Shared Memory per Multiprocessor	8		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48

Occupancy = 32 / 48 = 67%

Full occupancy

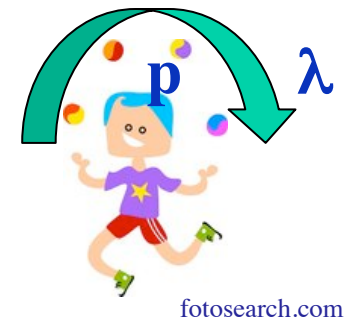


Latency

- Instructions wait on dependencies
 $x = a + b$; // ~ 20 for floating point, 500+ for memory
 $y = a + c$; // independent (stall)
 $z = x + d$; // **dependent, wait**

- How many warps are needed to hide latency (100% throughput)
if minimum latency is 4 cycles / instruction?

Parallelism (threads) = latency $\times$ throughput $T = \lambda \times p$
480 mul+adds/cycle; 32 memory ops /cycle [1.3 device]



- Required parallelism depends on op; for single precision
 - GT200 (C1060): **24** CP * 8 cores / SM = 192 ops/SM
 - GF104 (GTX 460, Cseclass03-07): **18** CP * 48 = 864
 - GK110?

Thread vs instruction level parallelism

- We are told to maximize the number of threads
- But we can also use instruction level parallelism to boost performance at a lower occupancy
 - ♦ See <http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf>
- On GT200, 100% peak with 25% occupancy
192 ops / cycle = 8 warps / 32 max possible warps
- On the GF104, we need ILP to go beyond 66% of peak
 - ♦ 48 cores/SM, half warp (16 cores) issues at a time
 - ♦ But we have only 2 schedulers
 - ♦ We must issue 2 independent instructions per warp in the same cycle
- GK110?

576 threads needed for 100% utilization

```
#pragma unroll UNROLL
for( i = 0; i < N_Iter; i++ ){
    a = a * b + c;
}
```

320 threads needed for 100% utilization

```
#pragma unroll UNROLL
for( i = 0; i < N_Iter; i++ ){
    a = a * b + c;
    d = d * b + c;
}
```

Hiding memory latency

- Parallelism = **latency** × **throughput**

Arithmetic: 576 ops/SM = 18CP x 32/SM/CP

Memory: 150KB = ~500CP (1100 nsec) × 150 GB/sec

- How can we keep 150KB in flight?

- Multiple threads: ~35,000 threads @ 4B/thread
- ILP (increase fetches per thread)
- Larger fetches (64 or 128 bit/thread)
- Higher occupancy



Copy 1 float /thread, need 100% occupancy

```
int indx = threadIdx.x + block * blockDim.x;  
float a0 = src[indx];  
dst[indx] = a0;
```

Copy 2 floats /thread, need 50% occ

```
float a0 = src[indx];  
float a1 = src[indx+blockDim.x];  
dst[indx] = a0;  
dst[indx+blockDim.x] = a1;
```

Copy 4 floats /thread, need 25% occ

```
int indx = threadIdx.x + 4 * block * blockDim.x;  
float a[4]; // in registers  
for(i=0;i<4;i++) a[i]=src[indx+i*blockDim.x];  
for(i=0;i<4;i++) dst[indx+i*blockDim.x]=a[i];
```

Today's lecture

- Occupancy and latency
- Further Improvements to Matrix Multiply
- Thread divergence

How to improve matrix multiply still further

- Volkov and Demmel, SC08
- Hide arithmetic latency using fewer threads
- Hide memory latency using fewer threads
- Improving performance using fewer threads
 - We can reduce number of threads through lower occupancy ...
 - ..by making better use of registers we can trade locality against parallelism
- Code was implemented on a 1.x device, so some details will be different (more registers on Kepler, for example)

Incremental improvements to matrix multiply

- Follows V. Volkov [GTC10]
- Gets 137 Gflops / sec

```
float Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    __syncthreads();

    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Two outputs / thread

- 2 outputs, double the loads

```
float Csub[2] = {0,0}; //array is allocated in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
}
```

Two outputs / thread, part 2

- $\times 2$ flops and stores
- 341 Gflops/sec

```
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+16, k) * BS(k, tx);
}
__syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+16) + tx] = Csub[1];
```

4 outputs / thread

```
float Csub[4] = {0,0,0,0}; //array is in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    AS(ty+24, tx) = A[a + wA * (ty+24) + tx];
    BS(ty+24, tx) = B[b + wB * (ty+24) + tx];
    __syncthreads();
}
```

4 outputs / thread

- 427 Gflops/sec [w/8 output/thread → 485 Gflops/s)
- ×2 # registers
- 50% occupancy

```
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+8, k) * BS(k, tx);
    Csub[2] += AS(ty+16, k) * BS(k, tx);
    Csub[3] += AS(ty+24, k) * BS(k, tx);
}
__syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
C[c + wB * (ty+16) + tx] = Csub[2];
C[c + wB * (ty+24) + tx] = Csub[3];
```

Volkov and Demmel's SGEMM

- Improve performance using fewer threads
 - ◆ Reducing concurrency frees up registers to trade locality against parallelism
 - ◆ ILP to increase processor utilization

Vector length: 64 //stripmined into two warps by GPU

Registers: **a**, **c**[1:16] //each is 64-element vector

Shared memory: **b**[16][16] //may include padding

Compute pointers in *A*, *B* and *C* using thread ID

c[1:16] = 0

do

b[1:16][1:16] = next 16×16 block in *B* or *B*^T

local barrier //wait until **b**[][] is written by all warps

unroll for *i* = 1 to 16 **do**

a = next 64×1 column of *A*

c[1] += **a*****b**[*i*][1] //rank-1 update of *C*'s block

c[2] += **a*****b**[*i*][2] //data parallelism = 1024

c[3] += **a*****b**[*i*][3] //stripmined in software

... //into 16 operations

c[16] += **a*****b**[*i*][16] //access to **b**[][] is stride-1

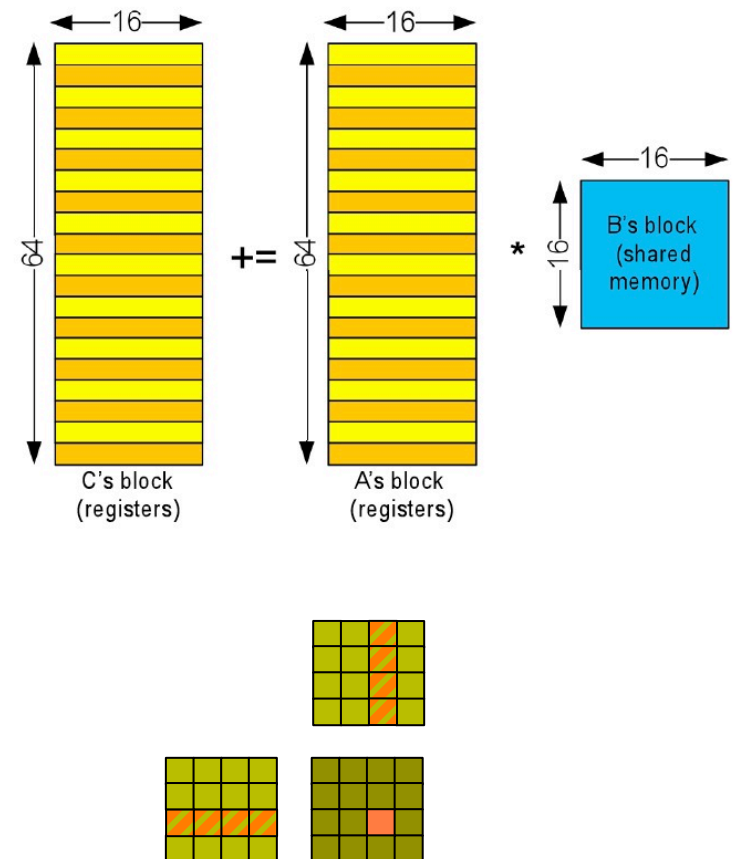
endfor

local barrier //wait until done using **b**[][]

update pointers in *A* and *B*

repeat until pointer in *B* is out of range

Merge **c**[1:16] with 64×16 block of *C* in memory



SGEMM Code

```

__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;
    __shared__ float bs[16][17];
    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    const float *Blast = B + k;
    do
    {
        #pragma unroll
        for( int i = 0; i < 16; i += 4 )
            bs[threadIdx.x][threadIdx.y+i] = B[i*ldb];
        B += 16;
        __syncthreads();

        #pragma unroll
        for( int i = 0; i < 16; i++, A += lda )
        {
            c[0] += A[0]*bs[i][0];  c[1] += A[0]*bs[i][1];  c[2] += A[0]*bs[i][2];  c[3] += A[0]*bs[i][3];
            c[4] += A[0]*bs[i][4];  c[5] += A[0]*bs[i][5];  c[6] += A[0]*bs[i][6];  c[7] += A[0]*bs[i][7];
            c[8] += A[0]*bs[i][8];  c[9] += A[0]*bs[i][9];  c[10] += A[0]*bs[i][10]; c[11] += A[0]*bs[i][11];
            c[12] += A[0]*bs[i][12]; c[13] += A[0]*bs[i][13]; c[14] += A[0]*bs[i][14]; c[15] += A[0]*bs[i][15];
        }
        __syncthreads();
    } while( B < Blast );
    for( int i = 0; i < 16; i++, C += ldc )
        C[i] = alpha*c[i] + beta*C[i];
}

```

Compute pointers to the data

Declare the on-chip storage

Read next B's block

The bottleneck:
Read A's columns
Do Rank-1 updates

Store C's block to memory

Volkov and Demmel

Data motion cost

- Communication performance is a major factor in determining the overall performance of an application

- The α - β model: $\alpha + \beta^{-1} n$

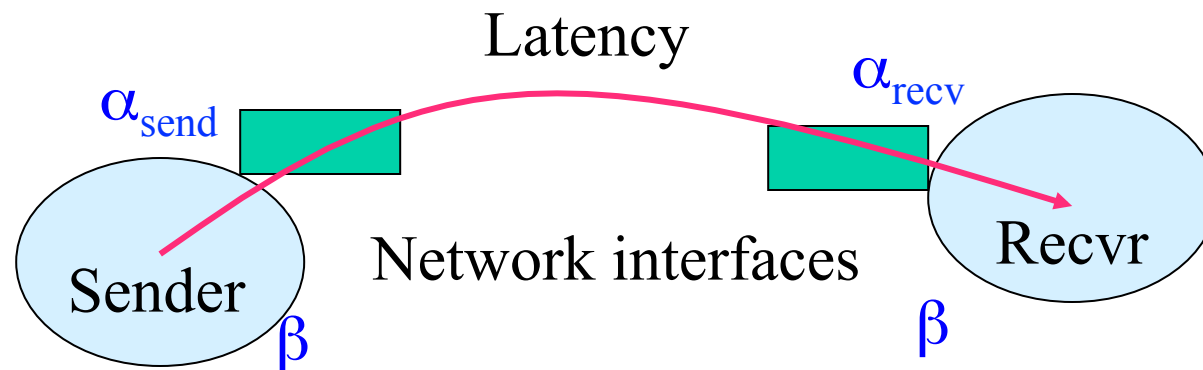
n = message length

α = message startup time

β_{∞} = peak bandwidth (bytes / second)

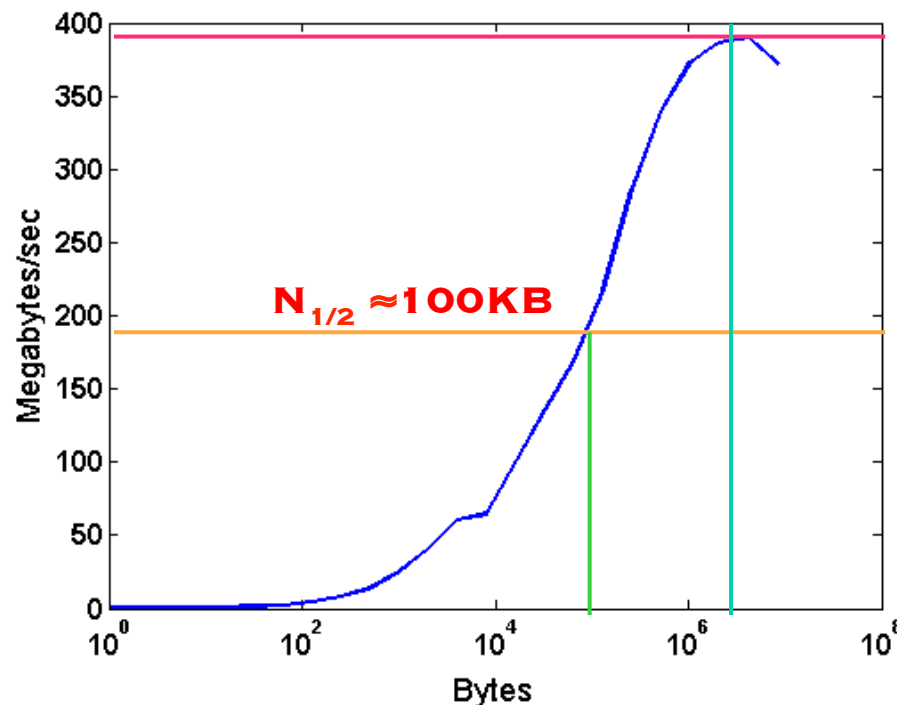
Machine	β_{∞} (Dev)	H-D	D-H
Stampede	144 GB/s	5.9	6.4
CseClass04	56.1	5.2	4.1

As reported by bandwidthTest



Half power point

- We define the *half power point* $n_{1/2}$ as the transfer size required to achieve $\frac{1}{2} \beta_{\infty}$
$$\frac{1}{2} \beta^{-1}_{\infty} = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_{\infty}$$
- In theory, this occurs when $\alpha = \beta^{-1}_{\infty} n_{1/2} \Rightarrow n_{1/2} = \alpha \beta_{\infty}$
- Formula may not be accurate



(SDSC Blue Horizon)

Consequences of data motion cost

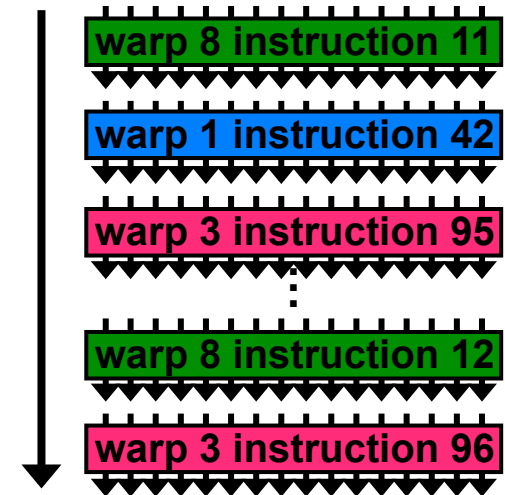
- Consider saxpy: $z[i] = a * x[i] + y[i]$
Performs 2 flops per 3 words of memory written and read
- This is a bandwidth bound kernel
Running time $\approx \alpha + (\text{Required Bandwidth}) \times \beta^{-1}_{\infty}$
 $\alpha = 4 \mu\text{s}$, $\beta_{\infty} = 127 \text{ GB/sec}$
- Flop rate bounded by
 $(2n \text{ flops} / 12n \text{ bytes}) \times 127 \text{ GB/sec} = 27 \text{ Gflops/sec}$
- $N_{1/2}$ Half bandwidth point: $N \approx 42,000$
Half the time is spent in the α term: $8 \mu\text{s}$
- Saxpy is used in matrix linear algebra, with matrices
- ♦ But matrix size N cannot exceed $\sqrt{M/24}$
 $M \sim 1 \text{ GB} \Rightarrow N \approx 16,000$
- ♦ Consequence: saxpy takes constant time to run for practical matrix sizes

Today's lecture

- Projects
- Further Improvements to Matrix Multiply
- **Thread divergence**

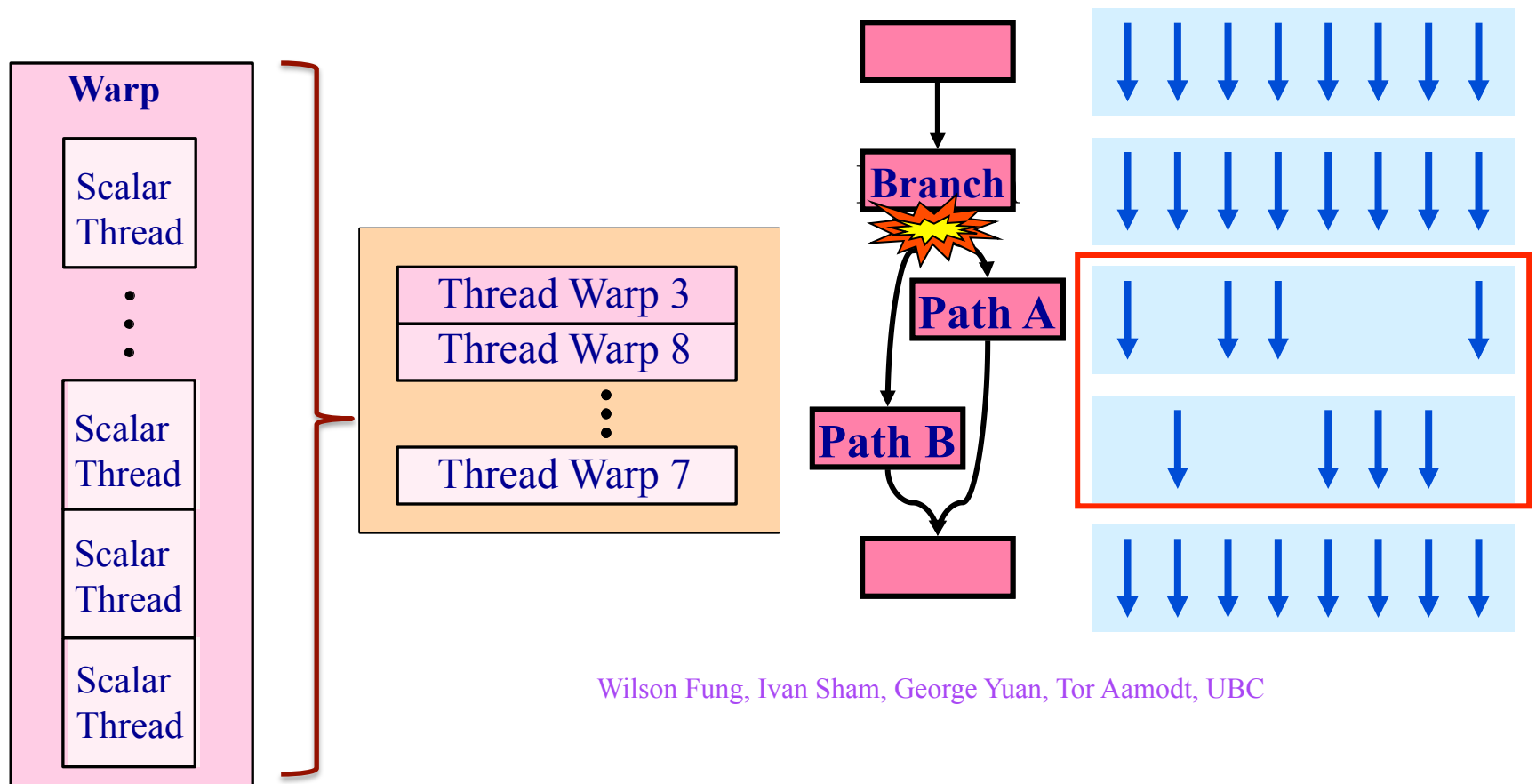
Thread Divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
- *Divergence* when a *predicate* is a function of the thread Id
`if (threadId < 2) { }`
- No divergence if all follow the same path
`if (threadId / WARP_SIZE < 2) { }`
- Consider reduction, e.g. summation $\sum_i x_i$



Thread divergence

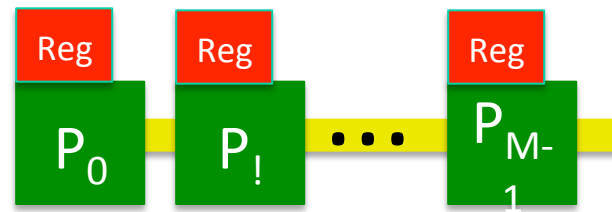
- All the threads in a warp execute the same instruction
- Different control paths are serialized



Wilson Fung, Ivan Sham, George Yuan, Tor Aamodt, UBC

Divergence example

```
if (threadIdx >= 2)
    a=100;
else
    a=-100;
```

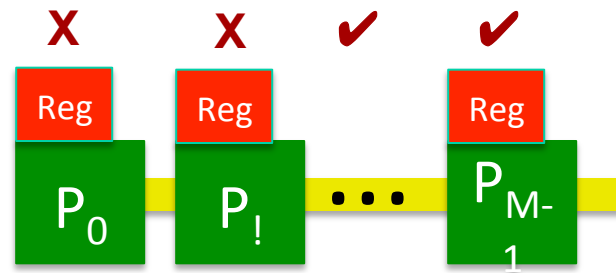


compare
threadIdx,2

Mary Hall

Divergence example

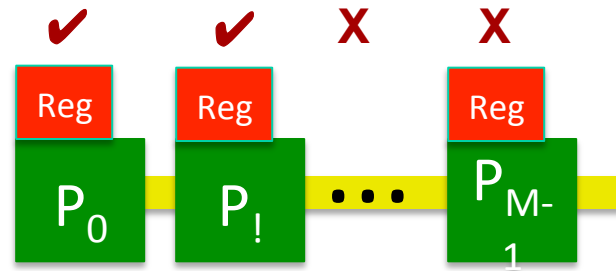
```
if (threadIdx >= 2)
    a=100;
else
    a=-100;
```



Mary Hall

Divergence example

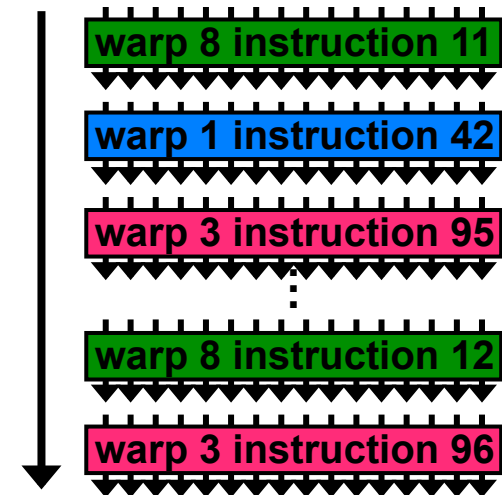
```
if (threadIdx >= 2)
    a=100;
else
    a=-100;
```



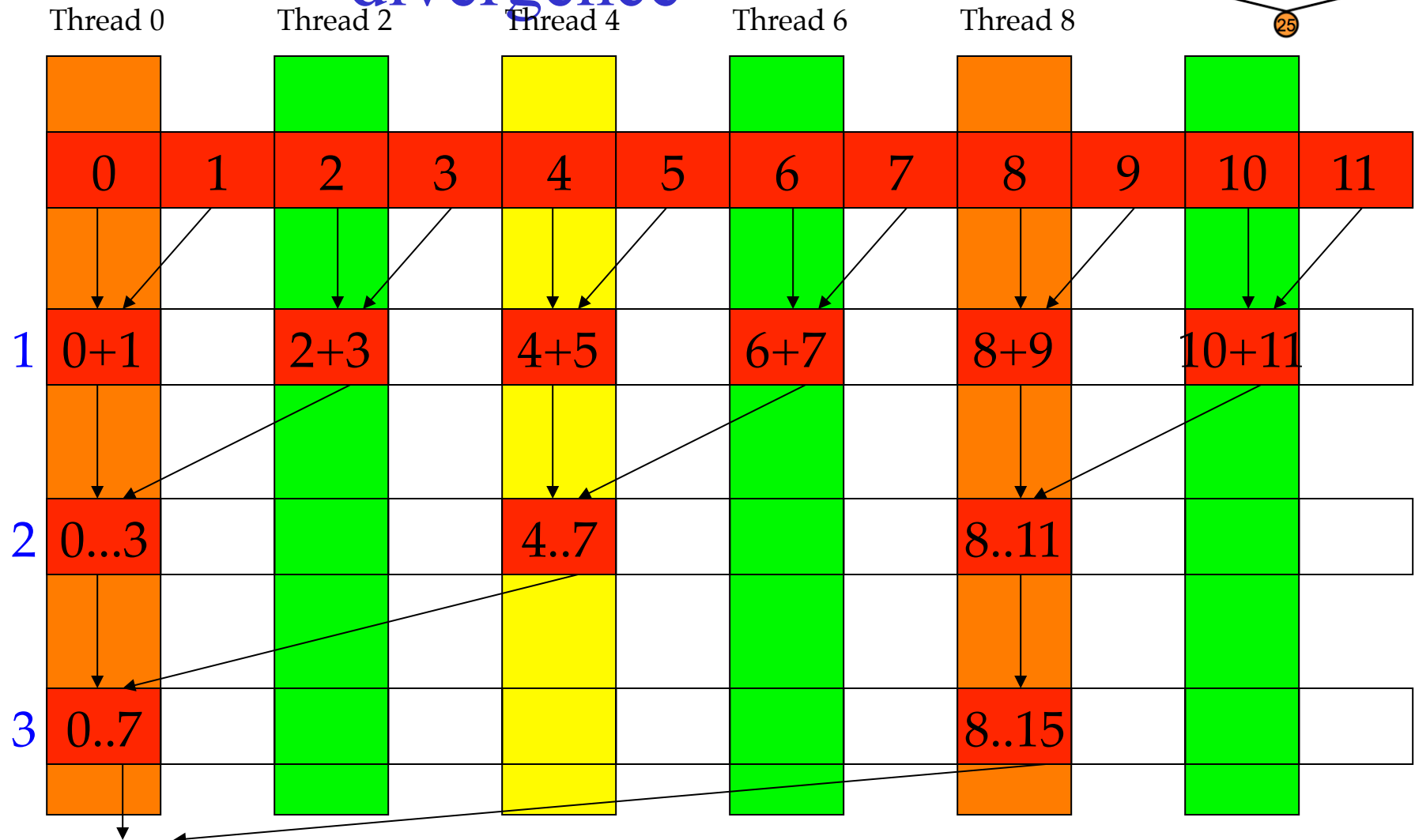
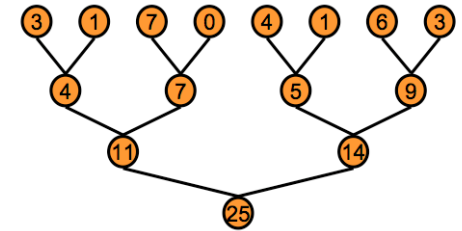
Mary Hall

Thread Divergence

- All the threads in a warp execute the same instruction
- Different control paths are serialized
- *Divergence* when a predicate is a function of the threadId
`if (threadId < 2) { }`
- No divergence if all follow the same path within a warp
`if (threadId / WARP_SIZE < 2) { }`
- We can have different control paths within the thread block



Example – reduction – thread divergence



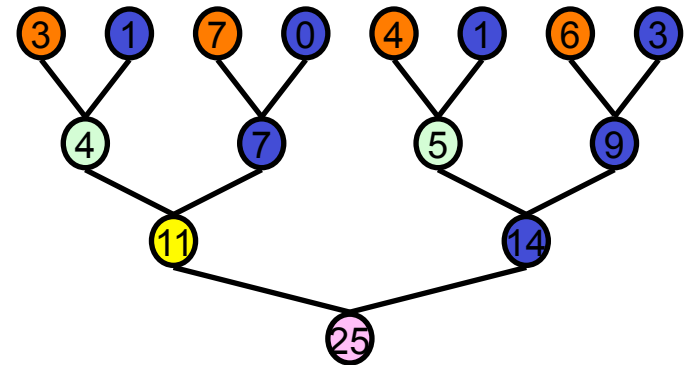
DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

The naïve code

```
__global__ void reduce(int *input, unsigned int N, int *total){
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

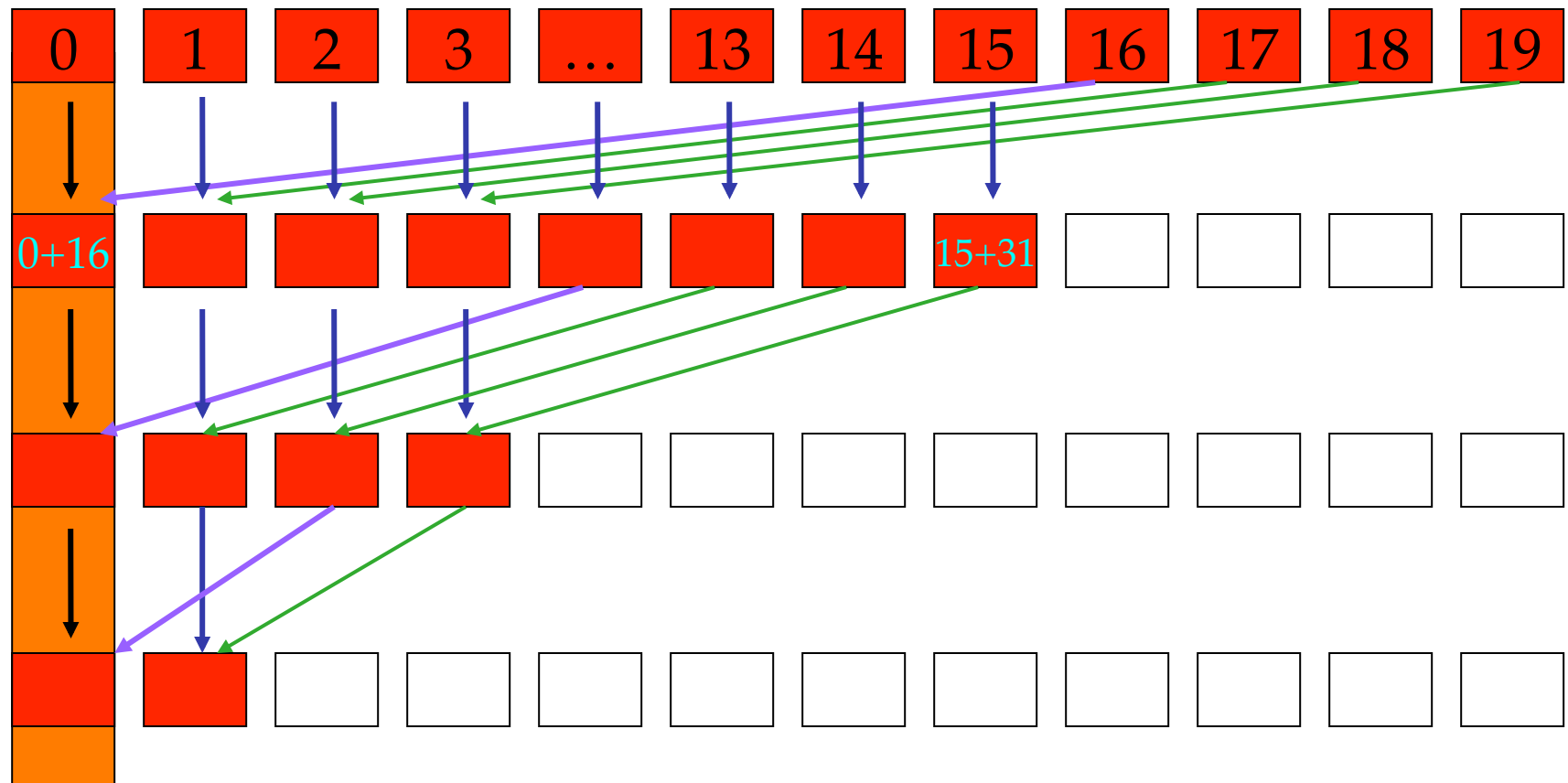
    __shared__ int x[BSIZE];
    x[tid] = (i < N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }
    if (tid == 0) atomicAdd(total, x[tid]);
}
```



Reducing divergence and avoiding bank conflicts

Thread 0



The improved code

- All threads in a warp execute the same instruction

`reduceSum <<<N/512,512>>> (x,N)`

- No divergence until stride < 32
- All warps active when stride ≥ 32

```
__shared__ int x[ ];  
    unsigned int tid = threadIdx.x;  
    unsigned int s;
```

```
for (s = blockDim.x/2;  
     s > 1;  
     s /= 2) {  
    __syncthreads();  
  
    if (tid < s )  
        x[tid] += x[tid + s ];  
}
```

```
for (stride = 1;  
     stride < blockDim.x;  
     stride *= 2) {  
  
    __syncthreads();  
  
    if (tid % (2*stride) == 0)  
        x[tid] += x[tid + stride];  
}
```

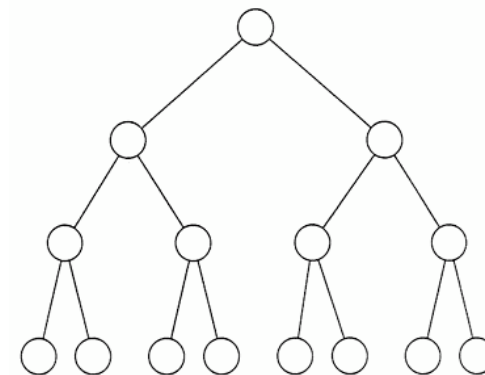
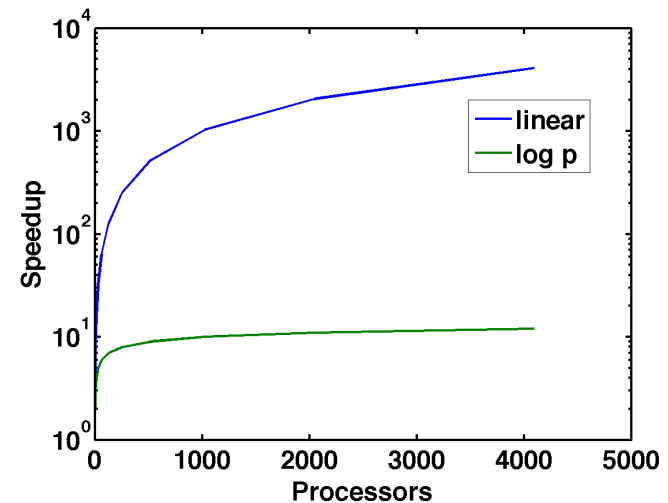
```
s = 256: threads 0:255  
s = 128: threads 0:127  
s =  64: threads 0:63  
s =  32: threads 0:31  
s =  16: threads 0:15  
...
```

Summary - Programming issues

- Branches serialize execution within a warp
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
 - ♦ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
 - ♦ Register consumption
 - ♦ Scheduling: hide latency
- Shared memory and registers do not persist across kernel invocations

Recapping: Maximize Performance on a GPU

- Avoid algorithms that present intrinsic barriers to utilizing the hardware
 - Avoid costly branches, or render harmless
 - Minimize serial sections
- **Cut data motion costs**
 - ◆ Hide latency of host $\leftrightarrow$ device memory transfers
 - ◆ Reduce global memory accesses $\rightarrow$ fast on-chip accesses
 - ◆ Coalesced memory transfers
- **Improve parallelism**
 - ◆ Vectorization, SSE
 - ◆ Pipelining
 - ◆ Instruction level parallelism



Branch Predication

- All instructions support predication
- Condition code or *predicate* per thread:
set to **true** or **false**
- Execute only if the predicate is true

```
if (x>1)  
    y = 7;
```

```
test = (x>1)  
test: y=7
```

- Compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by branch condition is not too large
- If the compiler predicts too many divergent warps....
threshold = 7, else 4

Concurrency – Host & Device

- Nonblocking operations
 - Asynchronous Device $\leftrightarrow$ {Host, Device}
 - Kernel launch
- Interleaved CPU and device calls: kernel calls run asynchronous with respect to host
- But a kernel call sequence runs sequentially on device
- Multiple kernel invocations running in separate CUDA streams: interleaved, independent computations

```
cudaStream_t st1, st2, st3, st4; cudaStreamCreate(&st1)...  
cudaMemcpyAsync(d1, h1, N, H2D, st1);  
kernel2<<<Grid,Block, 0, st2 >>> ( ...,d2,... );  
kernel3<<<Grid,Block, 0, st3 >>> ( ...,d3,... );  
cudaMemcpyAsync(h4, d4, N, D2H, st4);  
CPU_function(...);
```

[on-demand.gputechconf.com/gtc-express/2011/presentations/
StreamsAndConcurrencyWebinar.pdf](http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf)

Measuring performance

- Two ways
 - ♦ Use an ordinary timer, e.g. `gettimeofday()`
 - ♦ Use Cuda events/elapsed time (`#ifdef CUDA_TIMER`)
- See *incrArray*
- Note that kernel invocation is asynchronous

```
cudaThreadSynchronize();  
double t_device_compute = -getTime();  
    incr<<< nBlocks, bSize >>> (a_d, N);  
cudaThreadSynchronize();  
t_device_compute +=getTime();
```

- Nvprof profiler

CUDA Error Handling

Cuda error: Can't run kernel: invalid device function.

- Cuda can silently fail, you can observe misleading performance
- E.g. if you specify an invalid grid / thread block dimensions
- Note: the last error can be cleared by successive kernel calls, so check frequently

```
cudaMalloc((void **) &a_d, size);
```

```
checkCUDAError("Unable to allocate storage on the device");
```

- Consult `checkCUDAError()` in `utils.cu` (`incrArr`)
- What about asynchronous calls?
- cf CUDA Programming Guide, “Error Handling”

Getting information about the binary

- Compiler will report a kernel's register usage along with that of local, shared and constant memory

--ptxas-options=-v

incrementArrays (float *a, int N)

int idx = blockIdx.x*blockDim.x + threadIdx.x;

if (idx<N) a[idx] = a[idx]+1.f;

ptxas info : Compiling entry function

'_Z22incrementArrayOnDevicePfii' for 'sm_21'

ptxas info : Used 6 registers, 48 bytes cmem[0]

Hardware evolution

- Kepler
- Latest entry is Maxwell



	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	$2^{16}-1$	$2^{16}-1$	$2^{32}-1$	$2^{32}-1$
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes