

# Lecture 8

More on Coalescing

Avoiding Bank Conflicts

Further improvements to Matrix Multiply

## Announcements

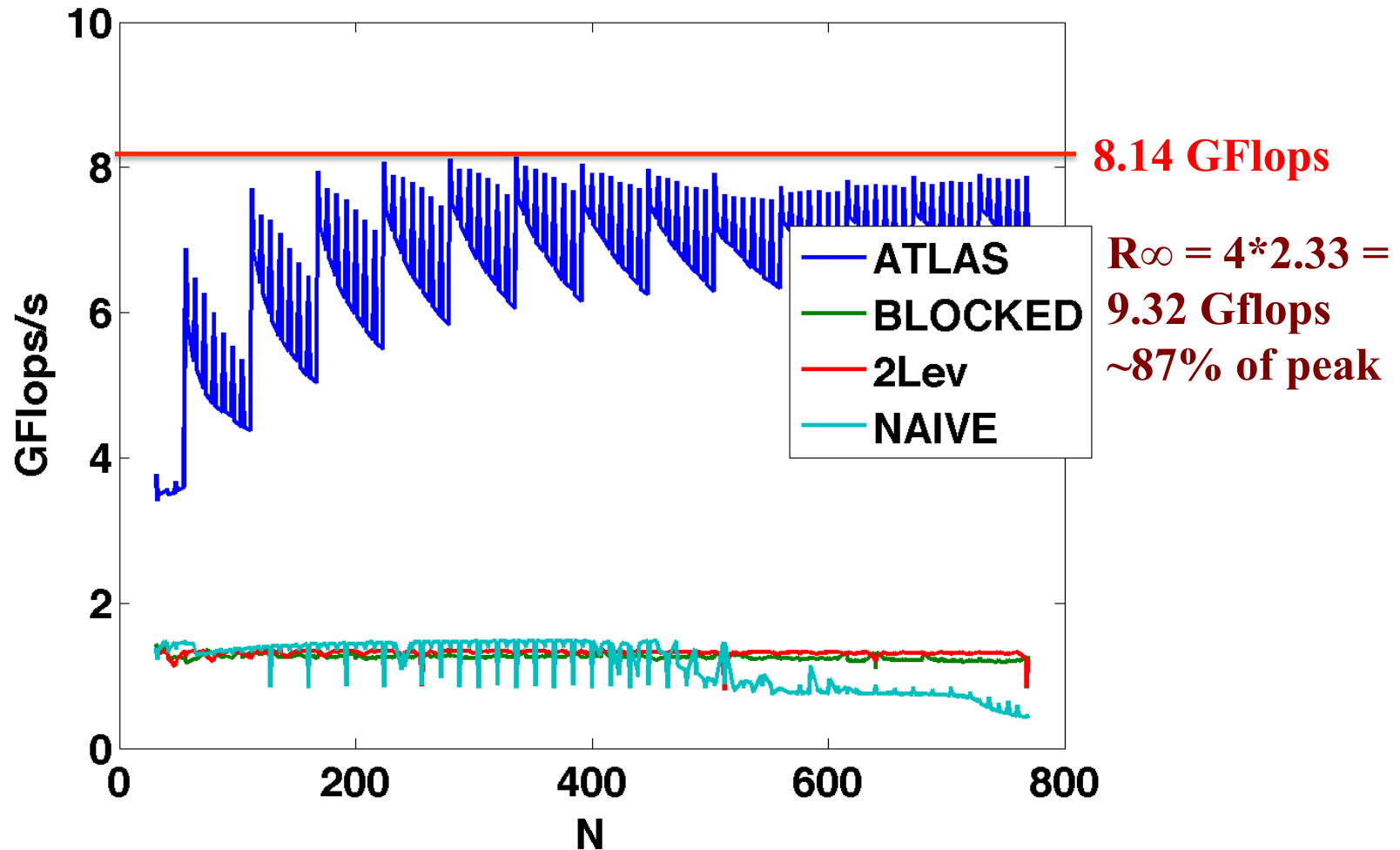
- Tuesday's lecture on 2/11 will be moved to room 4140 from 6.30 PM to 7.50 PM

# Results from A1

| GID           | Max  | Avg  |
|---------------|------|------|
| ATLAS         | 8.14 | 6.78 |
| G-520-802-976 | 7.87 | 6.38 |
| G-338-665     | 6.75 | 5.26 |
| G-507-689     | 8.63 | 4.61 |
| G-235-688     | 5.17 | 4.34 |
| G-126-180-722 | 4.72 | 4.24 |
| G-360-811-992 | 4.84 | 4.11 |
| G-157-526-538 | 4.74 | 4.11 |
| G-155-171-808 | 4.77 | 4.05 |
| G-543-876     | 4.25 | 3.97 |
| G-205-578     | 4.48 | 3.96 |
| G-183-229-535 | 3.95 | 3.68 |
| G-672         | 2.54 | 2.19 |
| G-279-389-589 | 1.87 | 1.82 |
| G-418-960     | 3.34 | 1.72 |
| G-163-197     | 1.83 | 1.62 |

# Assignment #1

- Blocking for cache will boost performance but a lot more is needed to approach ATLAS' performance



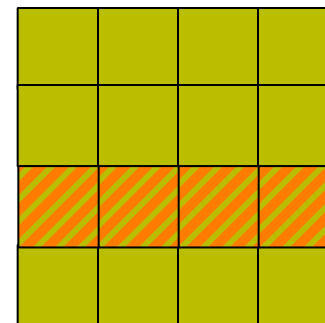
# Today's lecture

- Memory coalescing
- Avoiding bank conflicts
- Further Improvements to Matrix Multiply

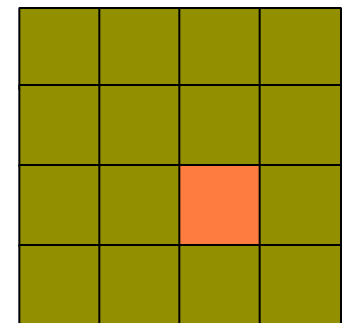
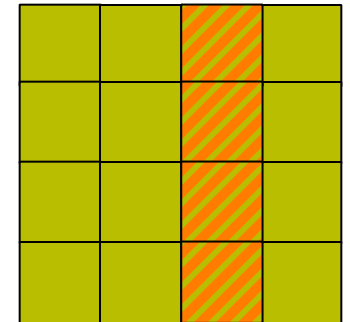
# Structure of blocked algorithm

- Threads cooperate to load blocks of A & B into shared memory
- Each thread in the block performs the *ijk* loop within shared memory

```
for (int k=0; k < N/BLK; k++){  
    Load blocks of A & B  
    __syncthreads();  
    for (int kk=0; kk< BLK; kk++){  
        c += a[kk][tx]*b[kk][ty];  
        __syncthreads();  
    }  
    C[I*N+J] = c;  
}
```



←→  
**N/BLK**



## Using shared memory (uncoalesced)

```
__global__ void matMul( float* C, float* A, float* B, int N) {  
    const unsigned int tx = threadIdx.x, ty = threadIdx.y;  
    const unsigned int I = blockIdx.x*bx + tx, J = blockIdx.y*by + ty;  
    __shared__ float a[BLK][BLK], b[BLK][BLK];  
    float c = 0.0f;  
  
    for (unsigned int k=0; k < N/BLK; k++){           // Sweep all blocks  
        a[tx][ty] = A[ I*N+k*BLK+ty];               // in block row/col  
        b[ty][tx] = B[J+N*(k*BLK+tx)];  
        __syncthreads();                             // Synchronizes all threads in a block  
        for (unsigned int kk=0; kk< BLK; kk++)  
            c += a[kk][tx]*b[kk][ty];  
        __syncthreads();                             // Avoids memory hazards  
    }  
    C[I*N+J] = c;  
}
```

## Results – shared memory

- N=512, double precision
- Dirac (C2050)
- Shared memory didn't improve performance

| Geometry    | 16 × 16 | 8 × 8 | 4 × 4 |
|-------------|---------|-------|-------|
| Uncoalesced | 40      | 39    | 14    |
| Coalesced   | 93      | 66    | 14    |

Global memory variant

|                   |       |       |       |
|-------------------|-------|-------|-------|
| Dirac (prefer SM) | 57    | 50    | 33    |
| (prefer L1\$)     | 64    | 58    | 42    |
| Geometries        | 1×512 | 2×256 | 4×128 |

# Coalescing with 2d arrays

- All warps in a block access consecutive elements within a row as they step through neighboring columns

```

I = blockIdx.y*BLK + ty;
J = blockIdx.x*BLK + tx;
int tx = threadIdx.x
a[ty][tx] = A[I*N+k*BLK+tx]
b[ty][tx] = B[J+N*(k*BLK+ty)]
    
```

|   |   |   |   |   |   |   |   |     |    |
|---|---|---|---|---|---|---|---|-----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |

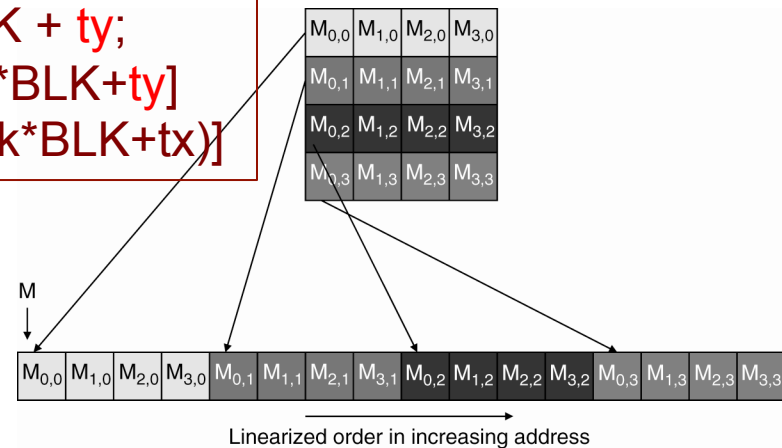
- Accesses by threads in a block along a column don't coalesce

|   |   |   |   |   |   |   |   |     |    |
|---|---|---|---|---|---|---|---|-----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |

```

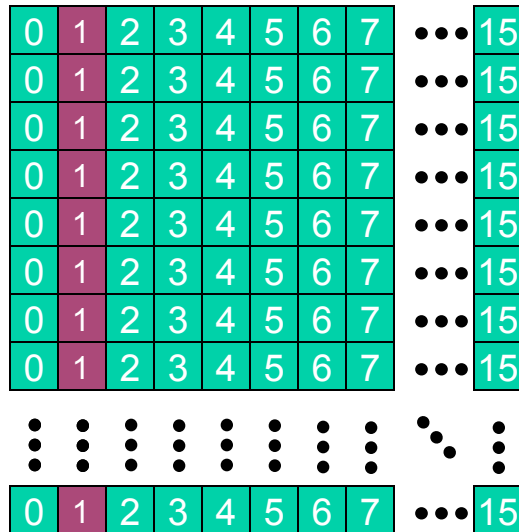
I = blockIdx.x*BLK + tx;
J = blockIdx.y*BLK + ty;
a[tx][ty] = A[I*N+k*BLK+ty]
b[ty][tx] = B[J+N*(k*BLK+tx)]
    
```

|   |   |   |   |   |   |   |   |     |    |
|---|---|---|---|---|---|---|---|-----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... | 15 |
|---|---|---|---|---|---|---|---|-----|----|



# Coalescing with 2d arrays

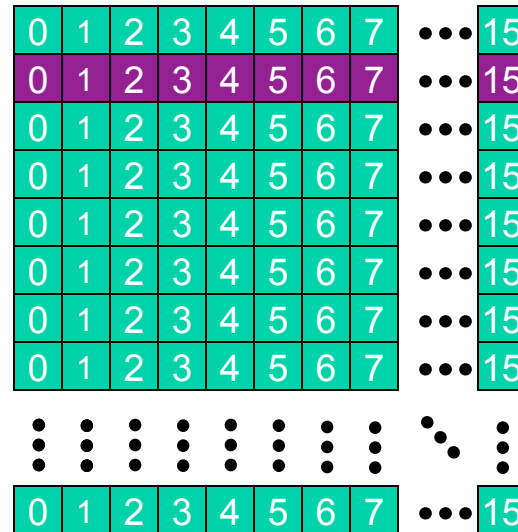
- Accesses by threads in a block along a column don't coalesce



```

I = blockIdx.x*BLK + tx;
J = blockIdx.y*BLK + ty;
a[tx][ty] = A[I*N+k*BLK+ty]
b[ty][tx] = B[J+N*(k*BLK+tx)]
    
```

- All warps in a block access consecutive elements within a row as they step through neighboring columns



```

I = blockIdx.y*BLK + ty;
J = blockIdx.x*BLK + tx;
int tx = threadIdx.x
a[ty][tx] = A[I*N+k*BLK+tx]
b[ty][tx] = B[J+N*(k*BLK+ty)]
    
```

# Coalesced access improves performance

```
I = blockIdx.y*by + ty;
J = blockIdx.x*bx + tx;
```

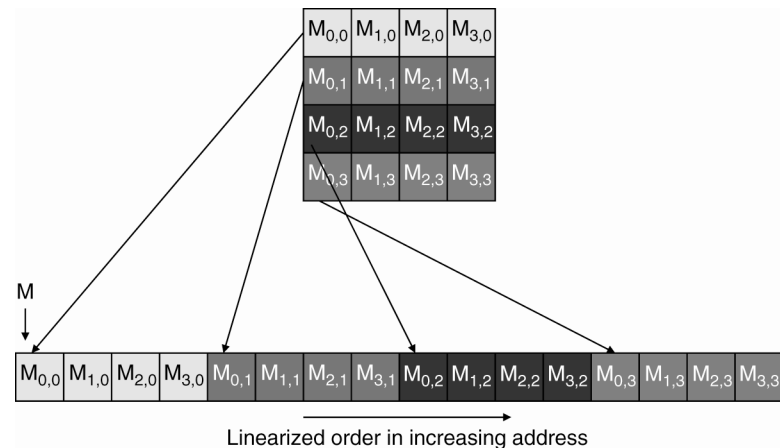
Uncoalesced:

```
I = blockIdx.x*BLK+ tx;
J = blockIdx.y*BLK+ ty;
```

```
__shared__ float a[BLK][BLK], b[BLK][BLK];
float c = 0.0f;
for (k=0; k < N/BLK; k++){
    a[ty][tx] = A[I*N+k*BLK+tx];
    b[ty][tx] = B[J+N*(k*BLK+ty)];
    __syncthreads();
    for (kk=0; kk< BLK; kk++){
        c += a[ty][kk]*b[kk][tx];
        __syncthreads();
    }
    C[I*N+J] = c;
```

Uncoalesced:

```
a[tx][ty] = A[I*N+k*BLK+ty];
b[ty][tx] = B[J+N*(k*BLK+tx)];
```



# Global memory coalescing

- Global memory accesses in units of 32, 64, 128 B
- Does not require sequential accesses by threads—only need to fall within the same 128B segment
- Accesses organized by warp
- Concurrent accesses by a warp's reads coalesce into  $K$  transactions = # different cache lines (128B) covered by the accesses
- Access not cached in L1: 32B at a time (x4) in L2

`shrd[threadIdx.x] = gbl[blockIdx.x*blockDim.x+threadIdx.x]` **YES!**  
`shrd[threadIdx.x] = gbl[blockDim.x+ blockIdx.x*threadIdx.x]` **NOT!**



32-byte segments

64-byte segments

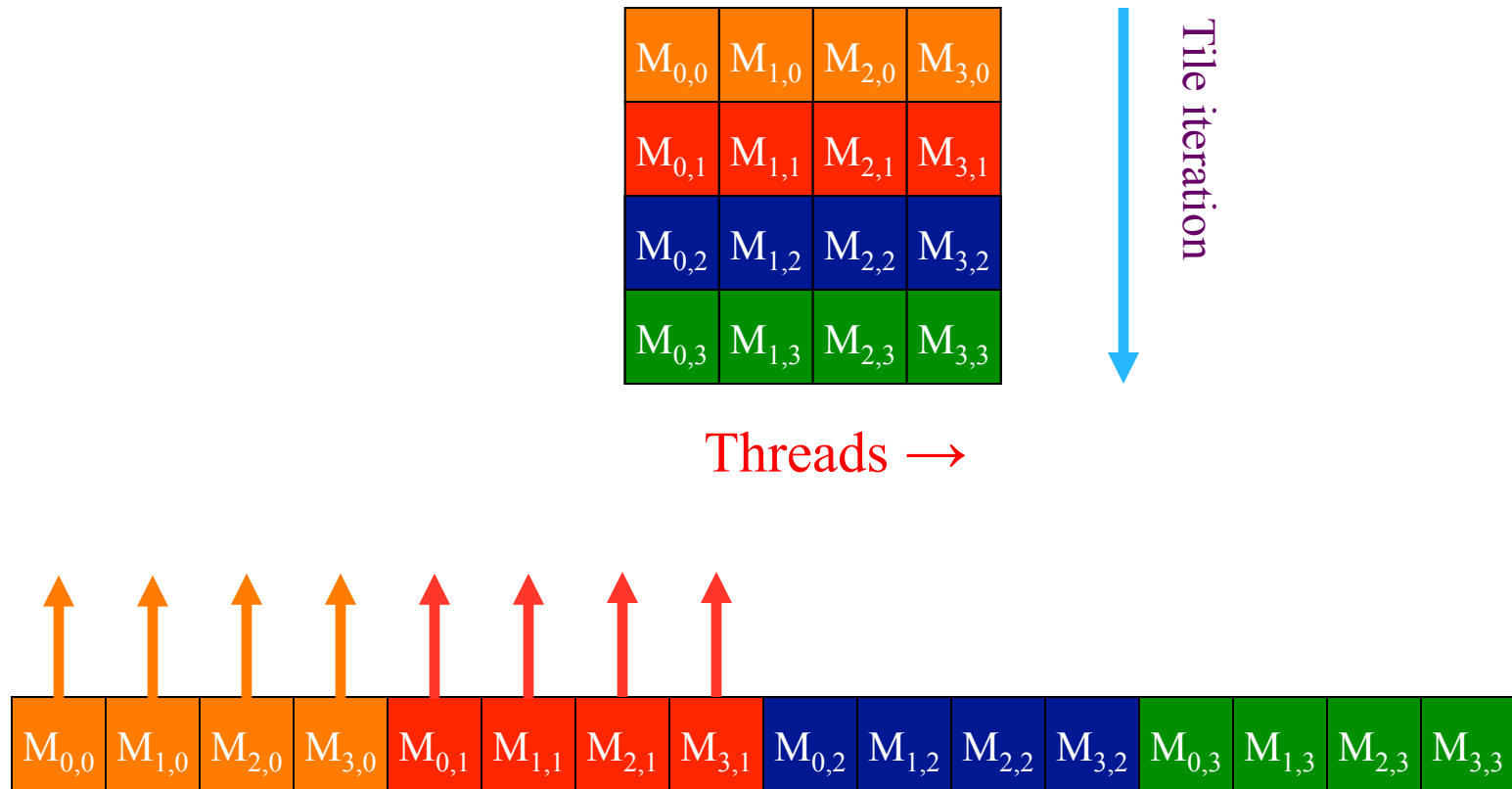
128-byte segments

# Global Memory

- If accessed word  $> 4$  bytes, warp's memory request split into separate, independently issued 128-byte memory requests
- Non-atomic, concurrent writes within a warp: writer not defined
- `cudaMalloc()` is guaranteed to be aligned to at least 256 bytes

# Memory coalescing

- Simplest: addresses fall within the same 128B segment
- Accesses organized by warps (32threads)



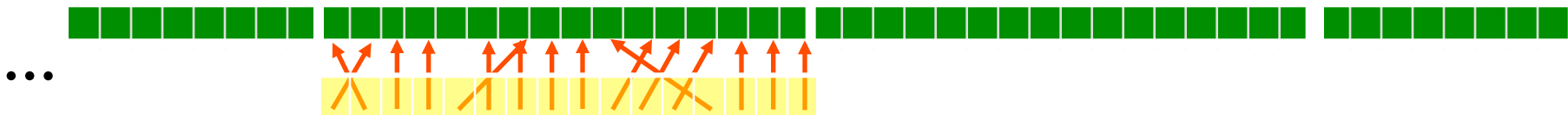
Nvidia Corp.

## Memory coalescing (compute capability $\geq 1.2$ )

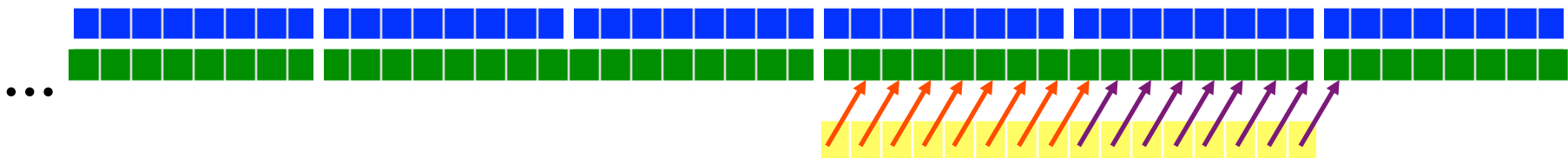
- Find the segment containing the address request of the lowest numbered active thread
- Find all other active threads requesting in same segment
- Reduce transaction size (if possible)
- Mark the serviced threads as inactive
- Repeat until all threads in the warp are complete
- Handles permutations, too

### 1 transaction – 128B segment

128-byte segments



### 2 transactions – 2 x 128B segments



Nvidia Cuda C Programming Guide: Appendix G.4.2

# Today's lecture

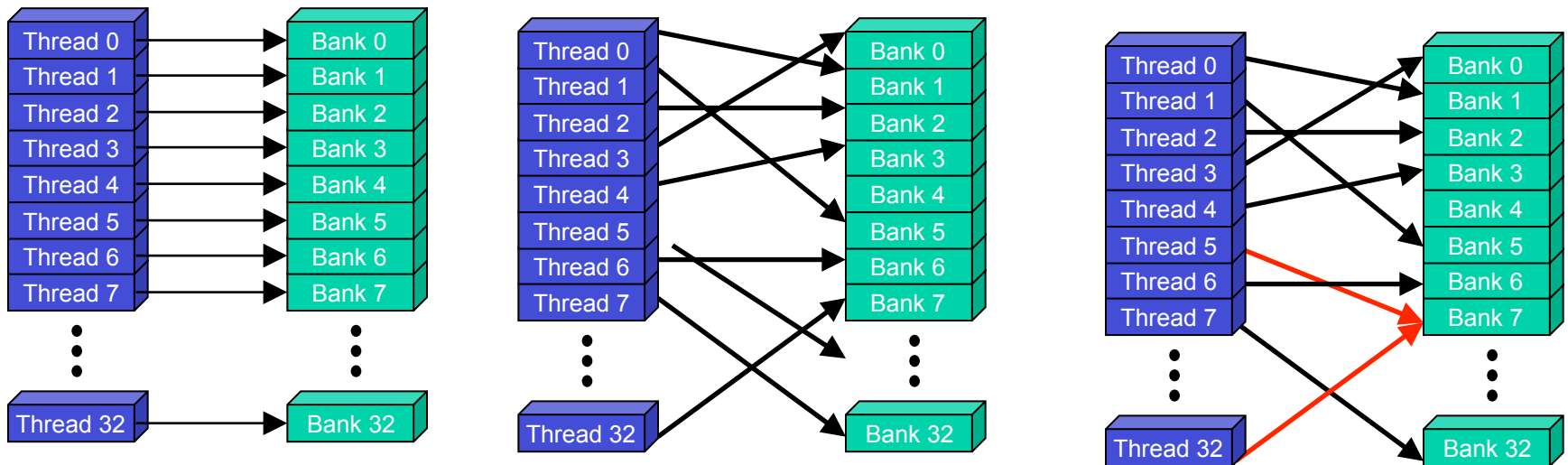
- Memory coalescing
- Avoiding bank conflicts
- Further Improvements to Matrix Multiply

## Shared memory banks

- A load or store of  $n$  addresses spanning  $n$  distinct memory banks can be serviced simultaneously, effective bandwidth  $n$  times than single bank bandwidth
- Multiple addresses map to same memory bank
  - Accesses are serialized
  - Hardware splits request into as many separate conflict-free requests as necessary  
Exception: if all access the same address: broadcast
- Devices of compute capability 2.x have the additional ability to multicast shared memory accesses
- See *CUDA C Best Practices Guide*

# Shared memory bank access

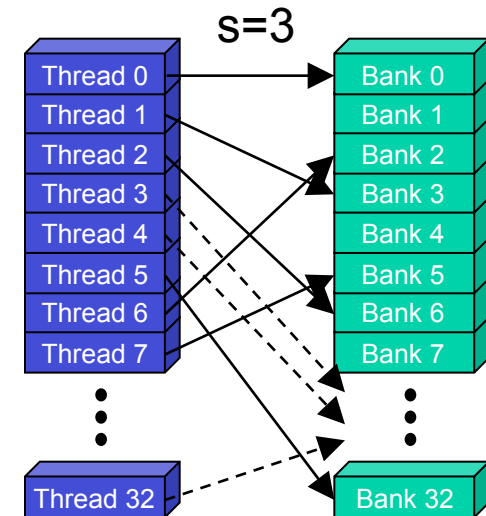
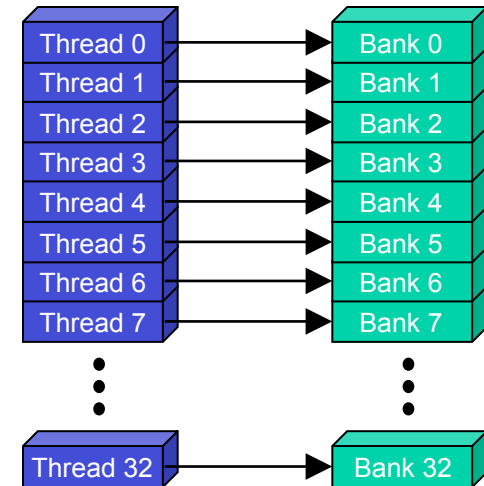
- Load/store of  $n$  addresses spanning  $n$  distinct memory banks can be serviced simultaneously, effective BW =  $\times n$  a single bank's
- Each bank can service 1 address / cycle (broadcast, too)
- Access to shared memory is fast unless...
  - 2 or more instructions in a warp access the same bank: we have a *conflict*
  - Exception: not if accesses to the same 32 bit word: broadcast
- For writes, only one thread writes, writer is undefined



```
int idx = blockIdx.x*blockDim.x + threadIdx.x;  
a[idx] = a[idx]+1.0f;
```

# Conflict free access

- Consider  
`__shared__ float shared[256];`  
`float foo = shared[base + s * threadIdx.x];`
- If **s** has no common factors with the number of banks (32), then there are no conflicts (**s** is odd)



# Identifying bank conflicts

- Traditional wisdom for exploiting cache locality can result in bank conflicts
- What if a thread loads 2 consecutive array elements?

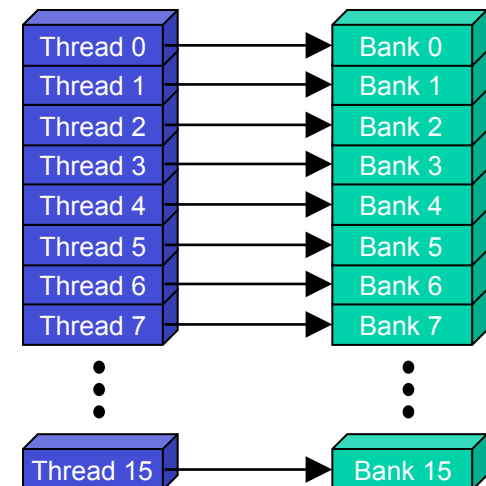
```
int tid = threadIdx.x;  
shared[2*tid] = global[2*tid];  
shared[2*tid+1] = global[2*tid+1];
```

- To avoid conflicts

```
shared[tid] = global[tid];  
shared[tid + blockDim.x] = global[tid + blockDim.x];
```

|    |    |    |    |
|----|----|----|----|
| 0  | 1  | 2  | 3  |
| 4  | 5  | 6  | 7  |
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |

A memory system with 4 banks



## Shared memory design

- Successive 32-bit words assigned to successive banks
- For devices of compute capability 2.x [Fermi]
  - Number of banks = 32
  - Bandwidth is 32 bits per bank per **2** clock cycles
  - Shared memory request for entire warp
  - No conflicts if access to bytes in same 32 bit word
- For devices of compute capability 1.x
  - Number of banks = 16
  - Bandwidth is 32 bits per bank per clock cycle
  - Shared memory request for a warp is split in two half-warps
  - Decreased susceptibility to bank conflicts
  - No conflict occurs if only one memory location per bank is accessed by a half warp of threads

# Coalesced access and no bank conflicts

```
I = blockIdx.y*BLK+ ty;  
J = blockIdx.x*BLK+ tx;
```

Slow:

```
I = blockIdx.x*BLK+ tx;  
J = blockIdx.y*BLK+ ty;
```

```
__shared__ float  a[BLK][BLK], b[BLK][BLK];  
if ((I < N) && (J < N)){  
    float c = 0.0f;  
    for (k=0; k < N/BLK; k++){  
        a[ty][tx] = A[I*N+k*BLK+tx];  
        b[ty][tx] = B[J+N*(k*BLK+ty)];  
        __syncthreads();  
        for (kk=0; kk< BLK; kk++){  
            c += a[ty][kk]*b[kk][tx]; all access same bank: broadcast  
        }  
        __syncthreads();  
    }  
    C[I*N+J] = c;
```

Slow:

```
a[tx][ty] = A[I*N+k*BLK+ty];  
b[ty][tx] = B[J+N*(k*BLK+tx)];
```

# Today's lecture

- Memory coalescing
- Avoiding bank conflicts
- Further Improvements to Matrix Multiply

## How to improve matrix multiply

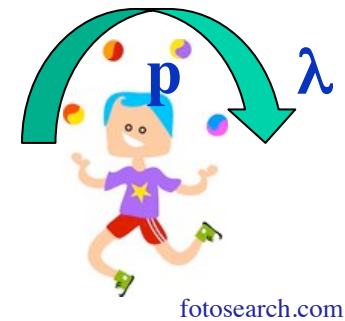
- Volkov and Demmel, SC08
- Hide arithmetic latency using fewer threads
- Hide memory latency using fewer threads
- Improving performance using fewer threads
  - We can reduce number of threads through lower occupancy
  - By making better use of registers we can trade locality against parallelism

# 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 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, Lilliput):  $24 \text{ CP} * 8 \text{ cores} / \text{SM} = 192 \text{ ops/cycle}$
  - GF100 (GTX-580, Cseclass01/02):  $18 \text{ CP} * 32 = 576$
  - GF104 (GTX 460, Cseclass03-07):  $18 \text{ CP} * 48 = 864$

## 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

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) x 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];
```

# Incremental improvements to matrix multiply

- Baseline code found in the SDK
- 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

- x2 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

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*<sup>T</sup>

**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

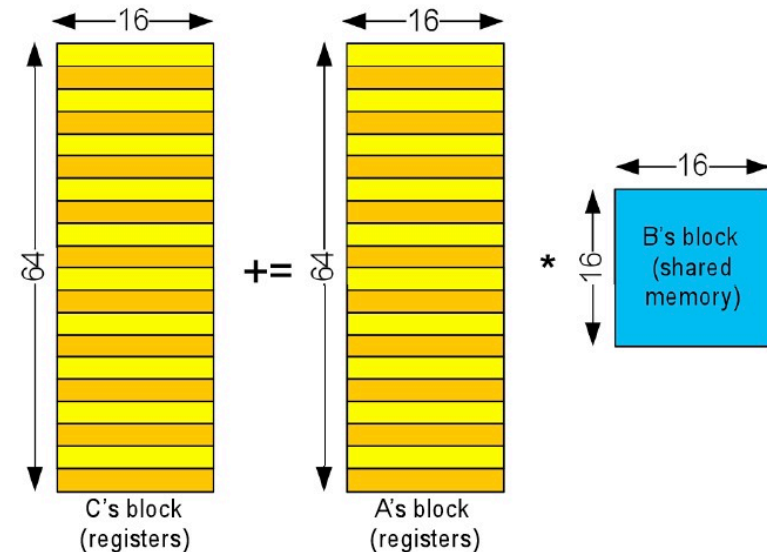


Figure 4: The structure of our matrix-matrix multiply routines.

# 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  $\alpha$ - $\beta$  model:  $\alpha + \beta^{-1} n$

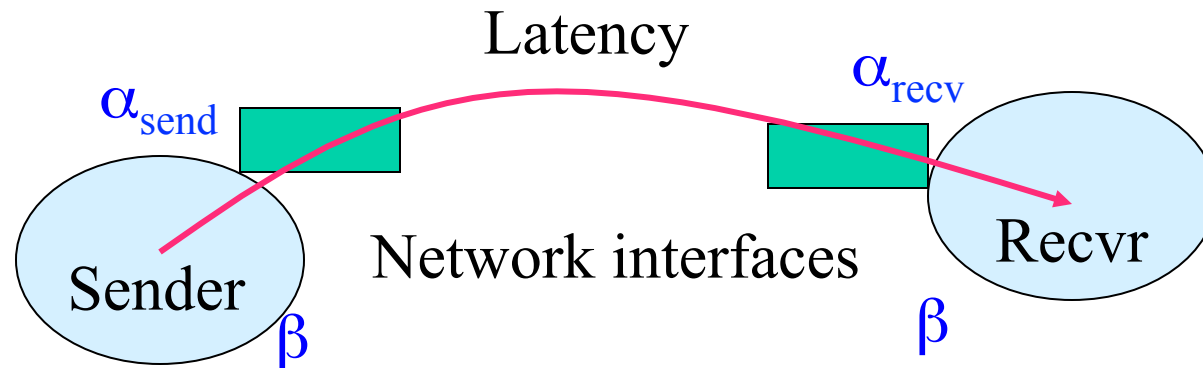
$n$  = message length

$\alpha$  = message startup time

$\beta_{\infty}$  = peak bandwidth (bytes / second)

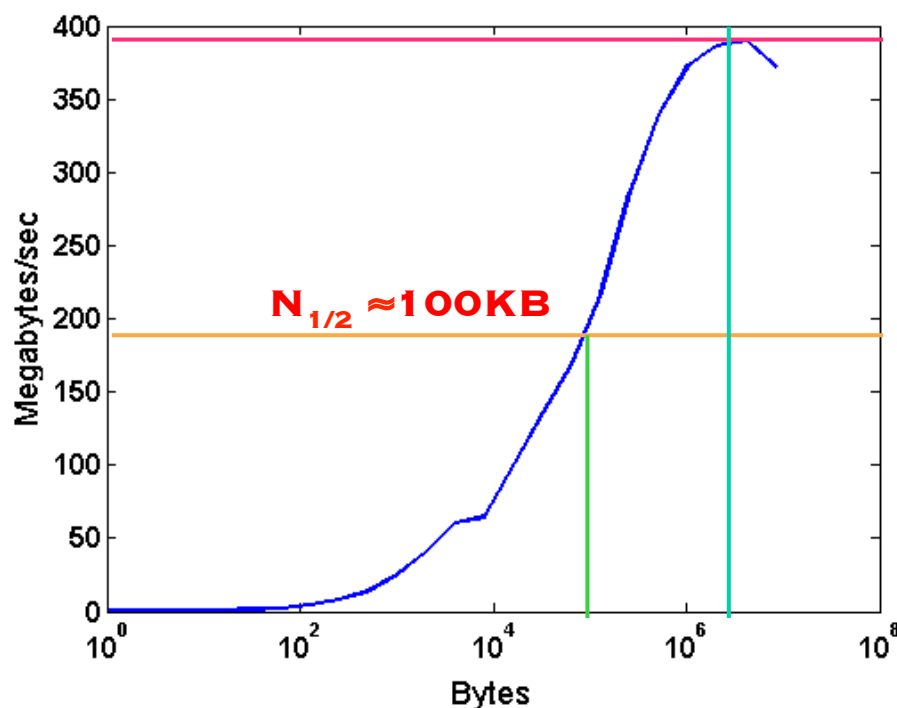
| Machine    | $\beta_{\infty}$ (Dev) | H-D | D-H |
|------------|------------------------|-----|-----|
| Dirac      | 104 GB/s               | 5.8 | 5.6 |
| CseClass01 | 122                    | 3.4 | 2.7 |
| CseClass04 | 56.1                   | 5.2 | 4.1 |

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  $\approx \alpha + (\text{Required Bandwidth}) \times \beta^{-1}_{\infty}$   
 $\alpha = 4 \mu\text{s}, \quad \beta_{\infty} = 127 \text{ GB/sec}$
- 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  $\alpha$  term:  $8 \mu\text{s}$
- Saxpy is used in matrix linear algebra, with matrices
- ♦ But matrix size  $N$  cannot exceed  $\text{sqrt}(M/24)$   
 $M \sim 1 \text{ GB} \Rightarrow N \approx 16,000$
- ♦ Consequence: saxpy takes constant time to run for practical matrix sizes

**Fin**