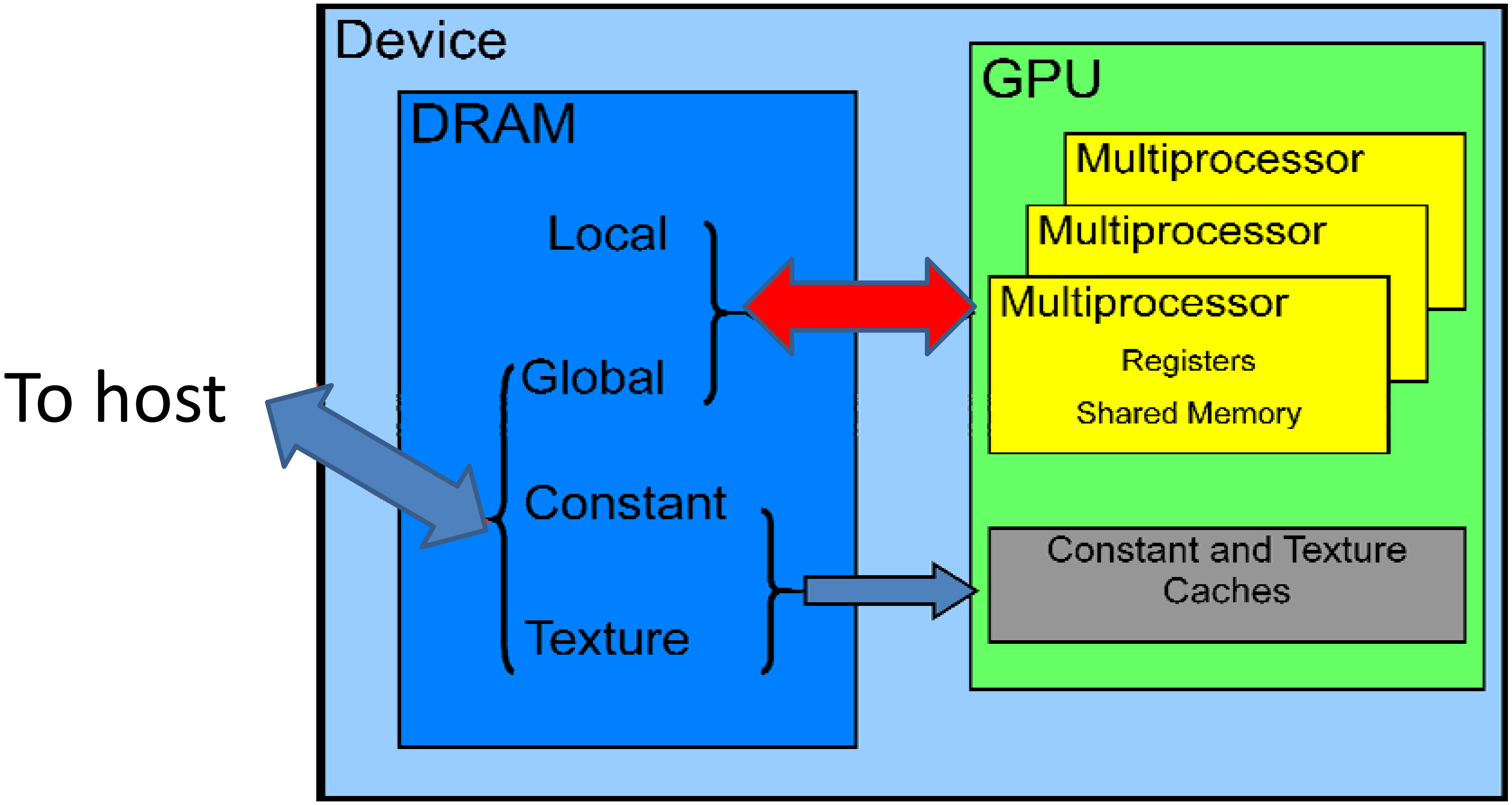
CUDA Programming

Week 2. CUDA Memory

Outline

- Memory review
- Data alignment
- Global memory coalesced
- Memory Padding
- Tiling
- Homework assignment

Device memory



Salient features of device memory

Memory	Location	Cached	Access	Scope	Lifetime
Register	On chip	N	R/W	1 thread	Thread
Local	RAM	N	R/W	1 thread	Thread
Shared	On chip	N	R/W	Threads in a block	Block
Global	RAM	N	R/W	All thread + host	Host allocation
Constant	RAM	Y	R	All thread + host	Host allocation
Texture	RAM	Y	R	All thread + host	Host allocation

Size and speed

- Size
  - Global and texture is limited by the size of RAM
  - Local memory: limited 16 KB per thread
  - Shared memory: limited 16KB
  - Constant memory: 64 KB in total
  - 8,192 (or 16,384) 32-bit registers per SM
- Speed:
  - Global, local, texture << constant << shared, register

Host-Device Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI) vs. 76 GB/s peak (Tesla C870)
- Method 1: Group transfers
  - One large transfer is much better than many small ones (memory coalescing)
- Method 2: Minimize transfers
  - Increase computation-communication ratio (tiling)

DATA ALIGNMENT

Data alignment

- Device can read 4-byte, 8-byte, or 16-byte words from global memory into registers in a single instruction.
  - The following code is in single instruction

```
__device__ type device[32];
type data = device[tid];
```

- Reading mis-aligned 8-byte or 16-byte words produces incorrect results

Data alignment

- A data of size 4-byte(8-byte, 16 byte) must aligned to 4-byte(8-byte, 16 byte).
  - Built-in types, like float2 or float4, fulfill this requirement automatically.
  - Structures need __align__(8) or __align__(16)

```
struct
__align__(8){
    float a;
    float b;
};
```

```
struct
__align__(16){
    float a;
    float b;
    Float c;
};
```

```
struct
__align__(16) {
    float a;
    float b;
    float c;
    float d;
    float e;
};
```

Two load instructions

Build-in data type

- The alignment requirement is automatically fulfilled for built-in types, like float2 or float4.

Data type	Size	Alignment
float2	8 byte	8
float3	12 byte	4
float4	16 byte	16

MEMORY COALESCING

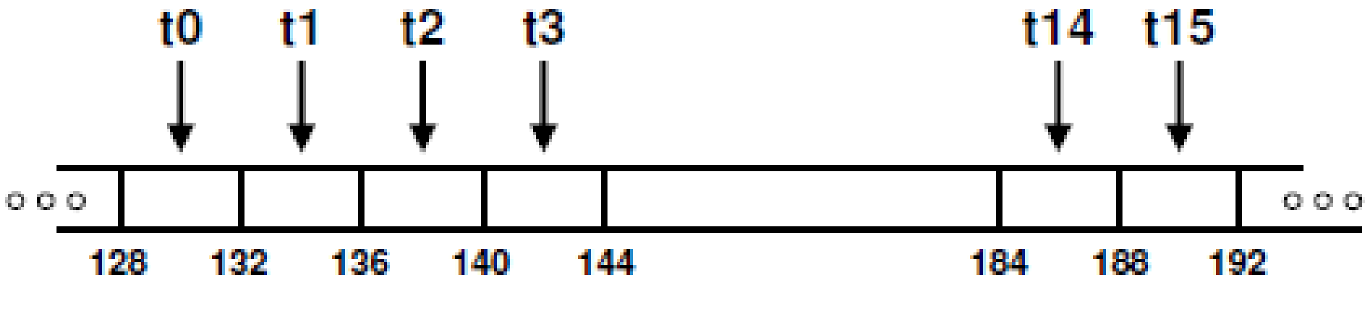
Global memory coalescing

- Global memory bandwidth is used most efficiently when the simultaneous memory accesses by 16 threads
- A contiguous region of global memory:
  - 64 bytes - each thread reads a word: int, float, …
  - 128 bytes - each thread reads a double-word: int2, float2, …
  - 256 bytes – each thread reads a quad-word: int4, float4, …

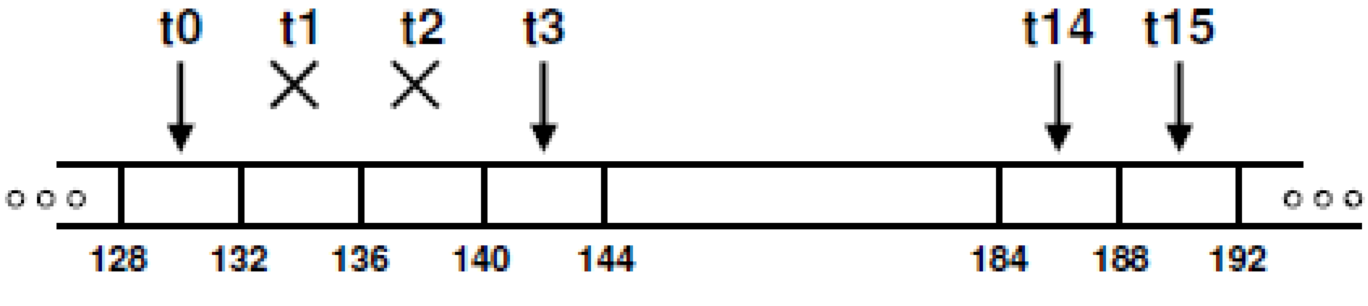
Memory coalescing for cuda 1.1

- The global memory access by 16 threads is coalesced into one or two memory transactions if all 3 conditions are satisfied
  1. Threads must access
     - Either 4-byte words: one 64-byte transaction,
     - Or 8-byte words: one 128-byte transaction,
     - Or 16-byte words: two 128-byte transactions;
  2. All 16 words must lie in the same segment
  3. Threads must access words sequentially.

Coalesced Access (Cuda 1.0-1.1)

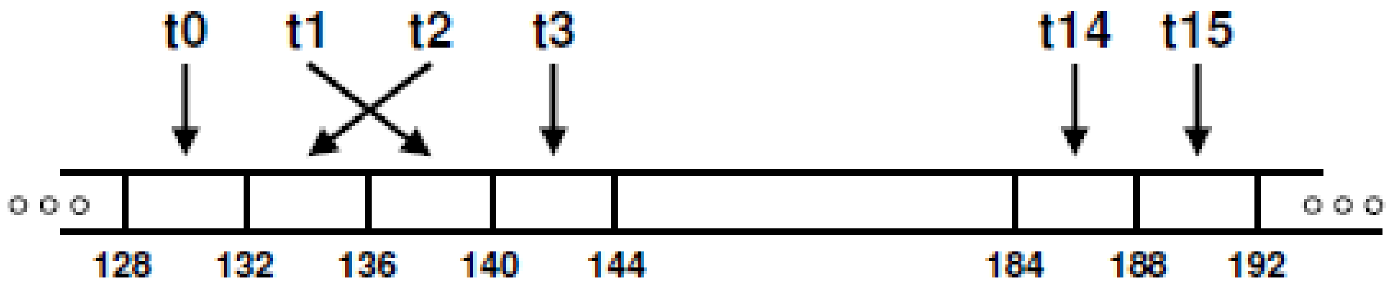


All threads participate



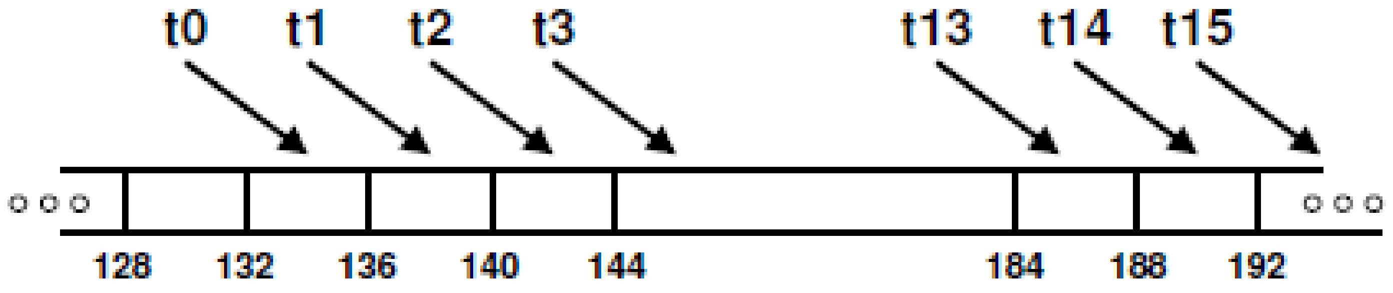
Some Threads Do Not Participate

Uncoalesced Access (Cuda 1.0-1.1)



Permuted Access by Threads

non-sequential **float memory access, resulting in 16 memory transactions.**

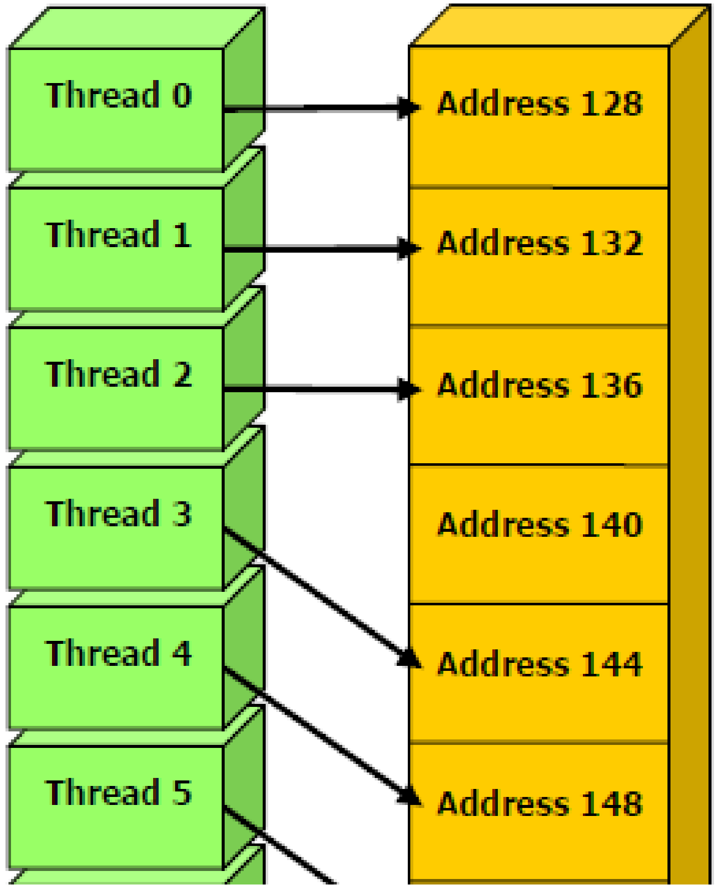


Misaligned Starting Address (not a multiple of 64)

access with a misaligned starting address, resulting in 16 memory transactions.

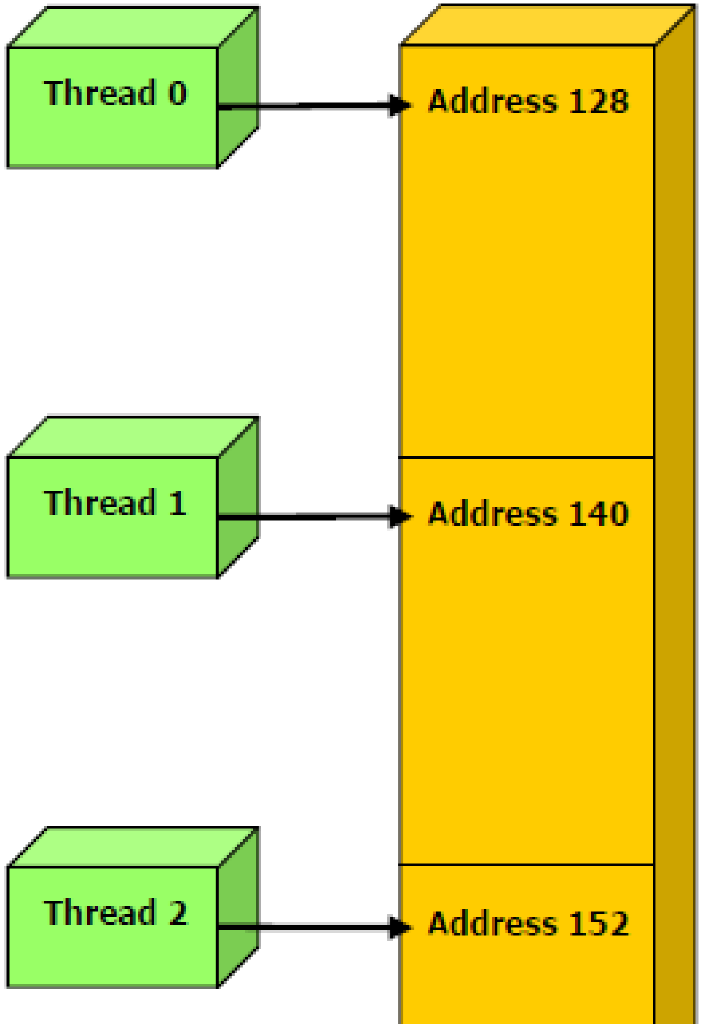
Uncoalesced Access (Cuda 1.0-1.1)

- non-contiguous float memory access, resulting in 16 memory transactions.



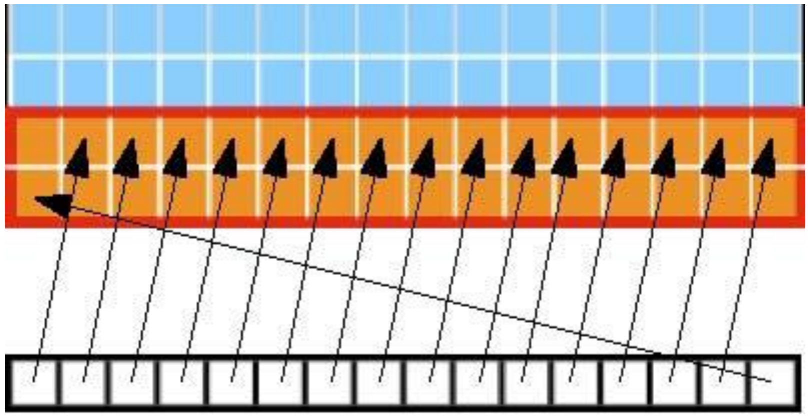
Uncoalesced Access (Cuda 1.0-1.1)

- non-coalesced float3 memory access, resulting in 16 memory transactions.



Things changed

- In cuda 1.2 and later version, the restrictions are relaxed
  - For Cuda 1.1 or lower versions, misaligned access pattern is split into 16 transactions
  - For Cuda 1.2 or higher versions, misaligned access pattern, like the figure, only has in one transactions

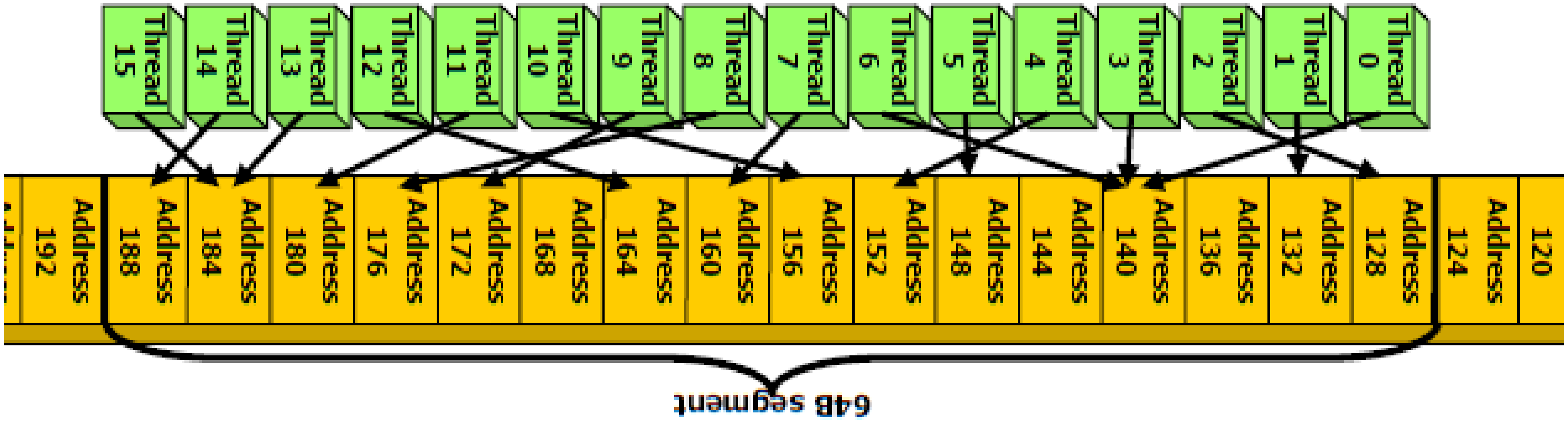


Memory coalescing for cuda 1.2

- The global memory access by 16 threads is coalesced into a single memory transaction as soon as the words accessed by all threads lie in the same segment of size equal to:
  - 32 bytes if all threads access 1-byte words,
  - 64 bytes if all threads access 2-byte words,
  - 128 bytes if all threads access 4-byte or 8-byte words.

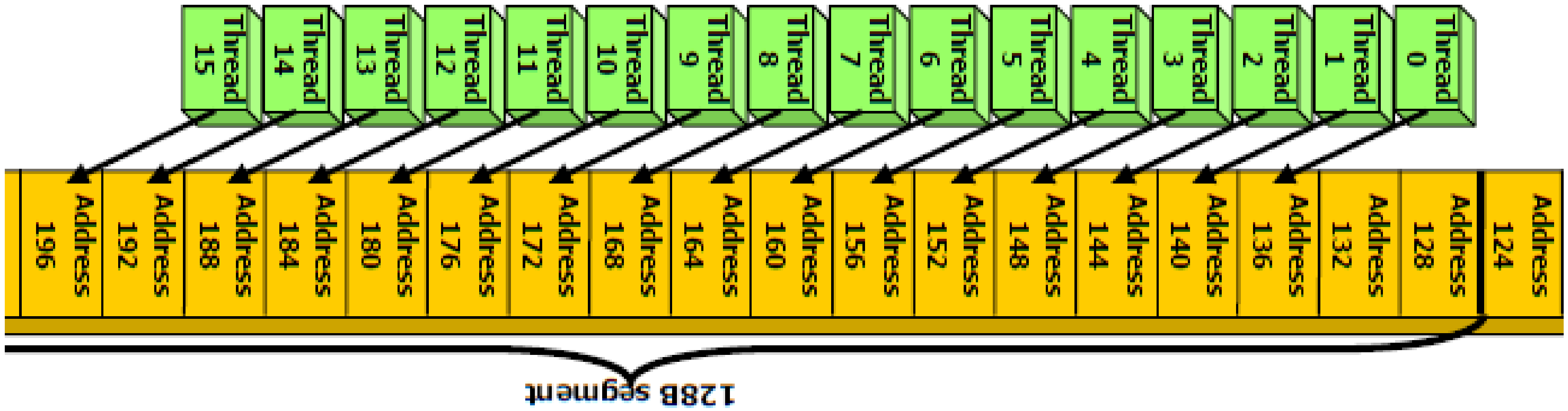
Coalesced Access (Cuda 1.2 later)

- Random float memory access within a 64B segment, resulting in one memory transaction.



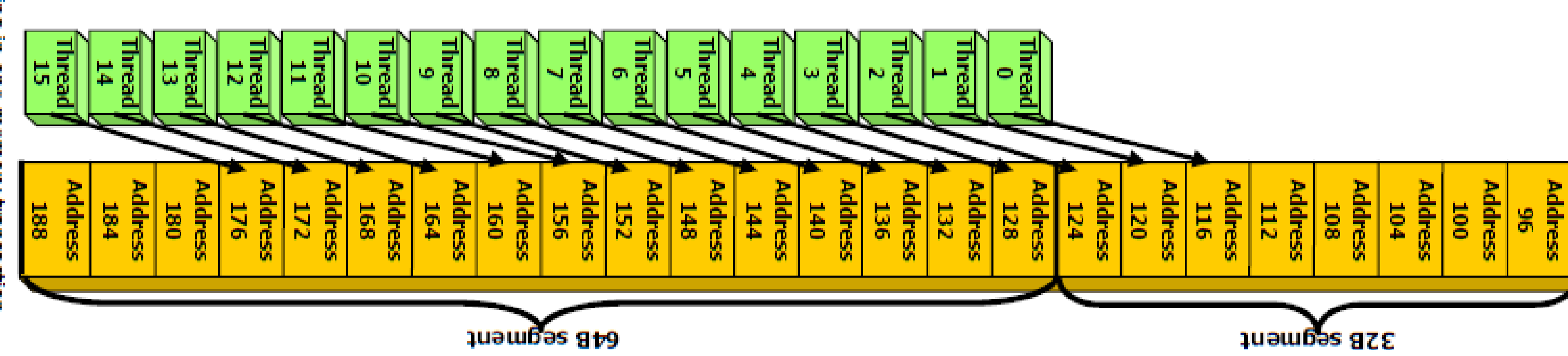
Coalesced Access (Cuda 1.2 later)

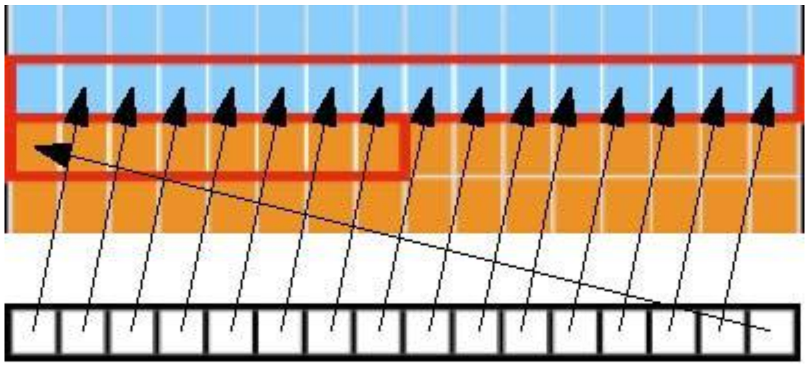
- misaligned float memory access, resulting in one transaction.



Coalesced Access (Cuda 1.2 later)

- misaligned float memory access, resulting in two transactions.

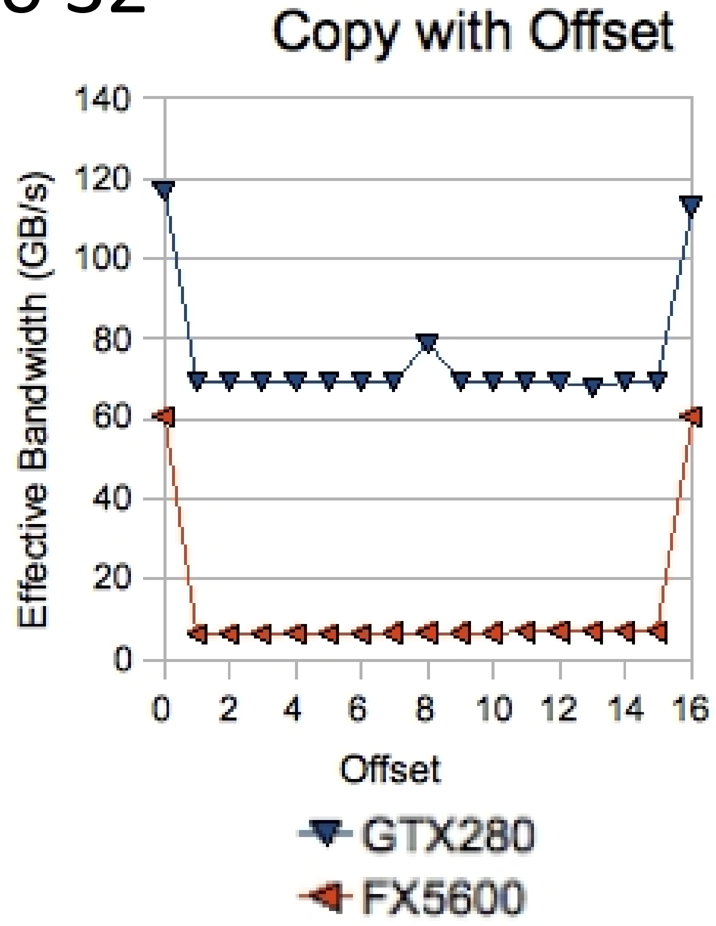




How important it is?

- EX1: Let offset run from 1 to 32

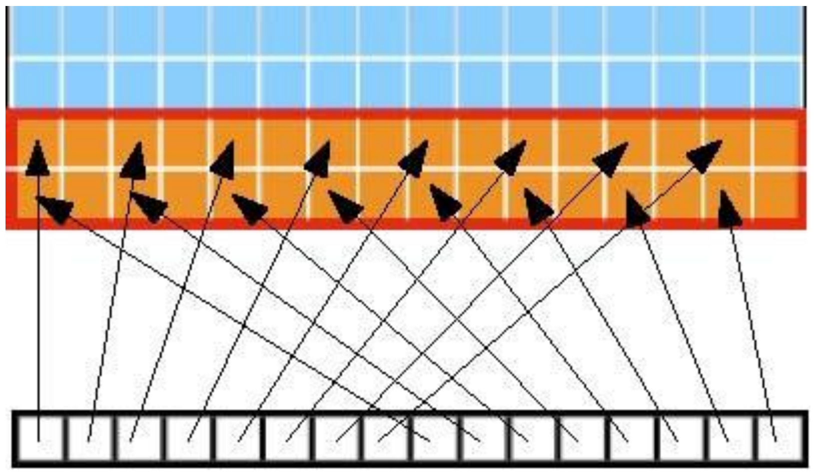
```
__global__ void offsetCopy
               (float *odata,
                float* idata,
                int offset)
{
    int xid = blockIdx.x*
              blockDim.x+
              threadIdx.x+
              offset;
    odata[xid] = idata[xid];
}
```

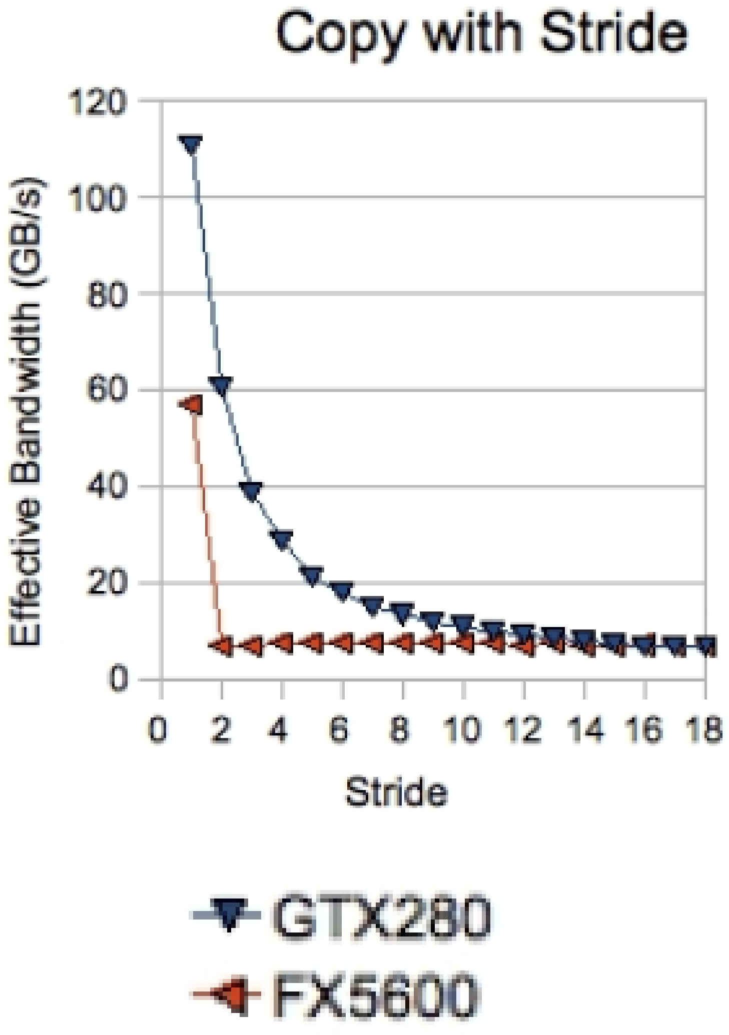


EX2, Strided Accesses

- stride changes from 1 to 18

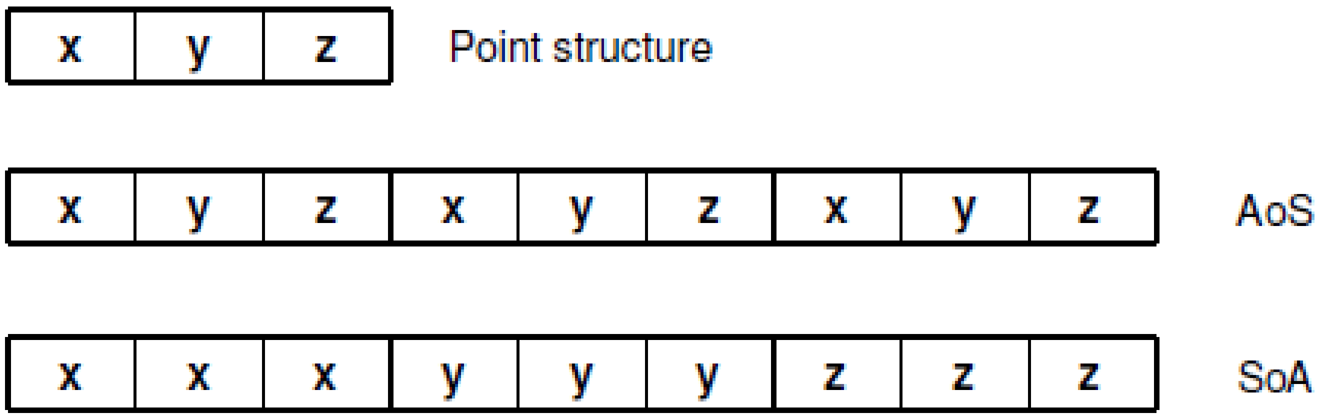
```
__global__ void strideCopy(float
                  *odata, float*
                  idata, int stride){
  int xid =(blockIdx.x* blockDim.x+
              threadIdx.x)*stride;
  odata[xid]=idata[xid];
}
```





Make memory access coalesced

1. Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)



2. Use shared memory to achieve coalescing
   – Example in the following slices

Example: float3 Code

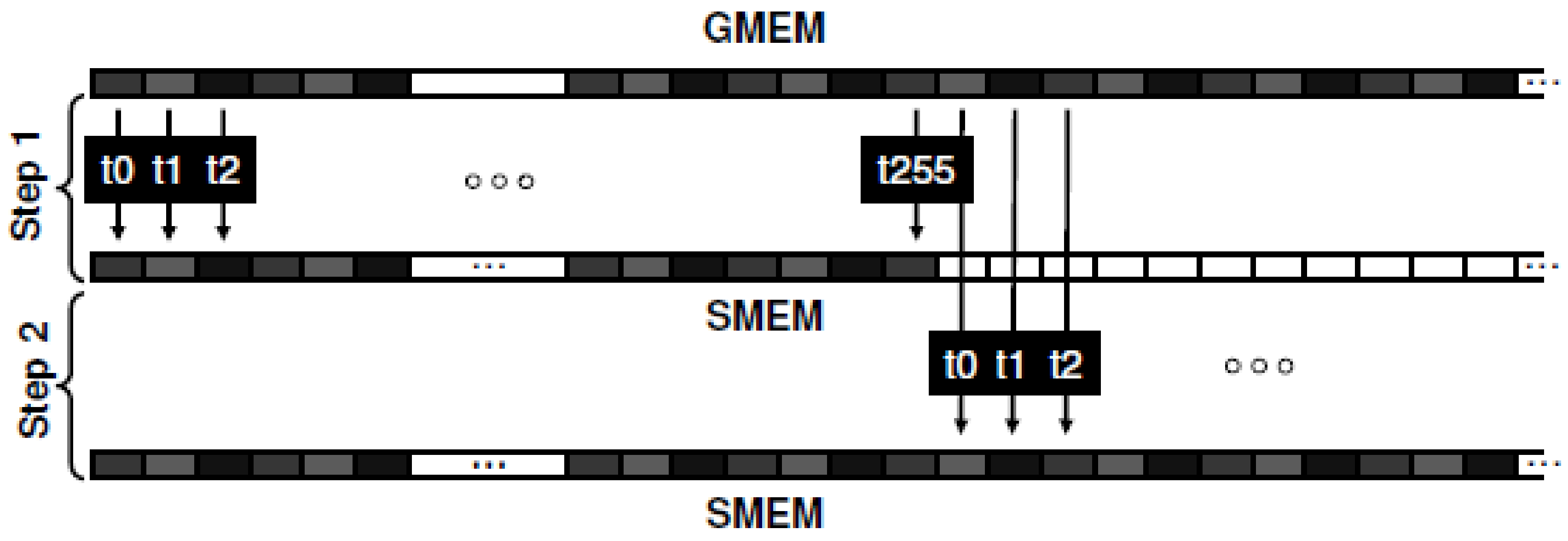
- Read an array of float 3, add 2 to each element
- float3 is of 12 bytes, not 4, 8 or 16.

```
__global__ void accessFloat3(float3 *d_in,
float3 d_out){
   int index = blockIdx.x * blockDim.x +
   threadIdx.x;
   float3 a = d_in[index];
   a.x += 2;
   a.y += 2;
   a.z += 2;
   d_out[index] = a;
}
```

Coalesced Access: float3 Case

- Use shared memory to allow coalescing
  - Need sizeof(float3)*(threads/block) bytes
  - Each thread reads 3 scalar floats:
  - Offsets: 0, (threads/block), 2*(threads/block)
  - These is likely processed by other threads, so sync
- Processing
  - Each thread retrieves its float3 from SMEM array
  - Cast the SMEM pointer to (float3*)
  - Use thread ID as index

```
__global__ void accessInt3Shared(float *g_in,float
*g_out){
   int index = 3*blockIdx.x*blockDim.x+threadIdx.x;
   __shared__ float s_data[256*3];
   s_data[threadIdx.x] = g_in[index];
   s_data[threadIdx.x+256] = g_in[index+256];
   s_data[threadIdx.x+512] = g_in[index+512];
   __syncthreads();
   float3 a = ((float3*)s_data)[threadIdx.x];
   a.x += 2;
   a.y += 2;
   a.z += 2;
   ((float3*)s_data)[threadIdx.x] = a;
   __syncthreads();
   g_out[index] = s_data[threadIdx.x];
   g_out[index+256] = s_data[threadIdx.x+256];
   g_out[index+512] = s_data[threadIdx.x+512];
}
```


GMEM
Step 1
t0 t1 t2
t255
SMEM
Step 2
t0 t1 t2
SMEM

Coalescing: Timing Results

- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB), Times averaged over 10K runs
- 12K blocks x 256 threads reading floats:
  - 356µs – coalesced
  - 3,494µs – permuted/misaligned thread access
- 4K blocks x 256 threads reading float3s:
  - 3,302µs – float3 uncoalesced
  - 359µs – float3 coalesced through shared memory

MEMORY PADDING

Common Access Pattern: 2D Array

- Each thread of index (tx,ty) accesses one element of a 2D array located at address `BaseAddress` of type `type*` and of width `N` using the following address

```
BaseAddress + N*ty + tx
```

- How to ensure the memory access is coalesced?
  - `blockDim.x = 16x and N=16x`
  - Recall EX1 (offset) and EX2 (stride)

Memory padding

- We can control blockDim.x, but the array size is not always 16x
- Memory padding: create an array of width=16x, and fill the unused part by 0
- pitch(投擲,音高,間距): the **l**eading **d**imension of an array **A** (called **lda**)
  - Since C/C++ is row major, the leading dimension is the row-width (number of elements in a row)

CUDA supporting API

- Cuda provides functions to allocate memory and copy data for 2D array

```
cudaMallocPitch((void**)&devPtr, size_t* &pitch,
           size_t width*sizeof(type), //in bytes
                 size_t height);
cudaMemcpy2D(void *  dst,
             size_t  dpitch,
             const void *  src,
             size_t  spitch,
             size_t  width,
             size_t  height,
             enum cudaMemcpyKind  kind)
```

- Similar functions also available for 3D array

TILING

Computation/communication ratio

- Let f be the number of flops, m be number of memory access. Then $q = f/m$ is the computation/communication ratio
- Let t_c be the time per flops, t_m be the time per memory access. The running time is

$$ft_c + mt_m = ft_c\left(1 + \frac{mt_m}{ft_c}\right) = ft_c\left(1 + \frac{1}{q}\frac{t_m}{t_c}\right)$$

  - t_c improves 60% per year, t_m improves 20% per year

Some examples

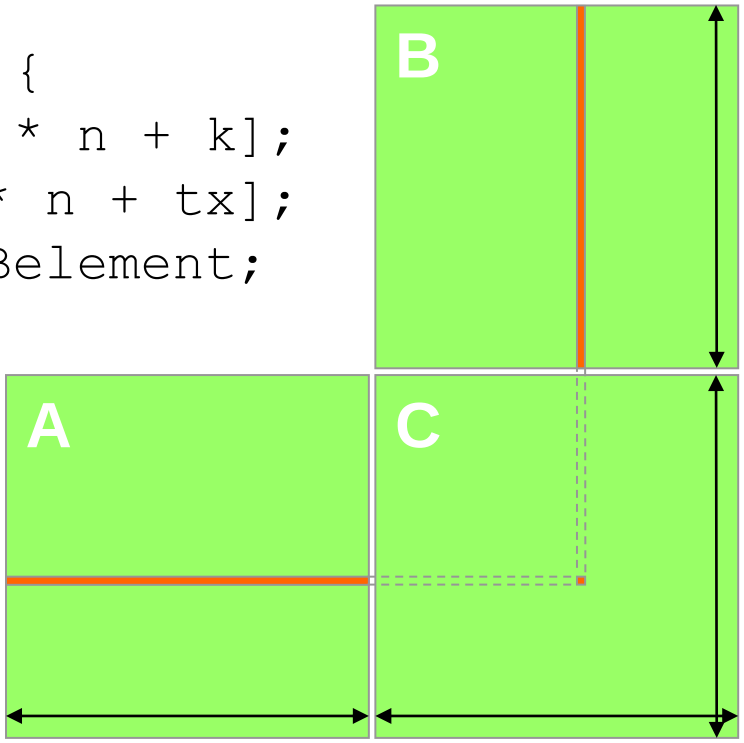
- Vector addition: z = x+y
  - f=n, m=3n, q = 1/3
- Matrix-vector multiplication: y=Ax
  - f = $2n^2$, m = n^2+2n, q = 2
- Matrix-matrix multiplication: C= AB
  - f = $2n^3$, m = $3n^2$, q = 2n/3
  - Therefore, the larger n, the better utilization
  - But, can we really achieve that

MMM on CPU

```
void MatrixMulOnHost(float* A, float* B,
float* C, int n){
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            double sum = 0;
            for (int k = 0; k < n; ++k) {
                double a = A[i * n + k];
                double b = B[k * n + j];
                sum += a * b;
            }
            C[i * n + j] = sum;
        }
}
```

MMM on GPU

```
__global__ void MatrixMulKernel(float* A,float*
B,float* C,int n){
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Cvalue = 0;
    for (int k = 0; k < n; ++k){
        float Aelement = A[ty * n + k];
        float Belement = B[k * n + tx];
        Cvalue += Aelement * Belement;
    }
    // Write to device memory;
    // each thread writes 1
    // element
    C[ty * n + tx] = Cvalue;
}
```

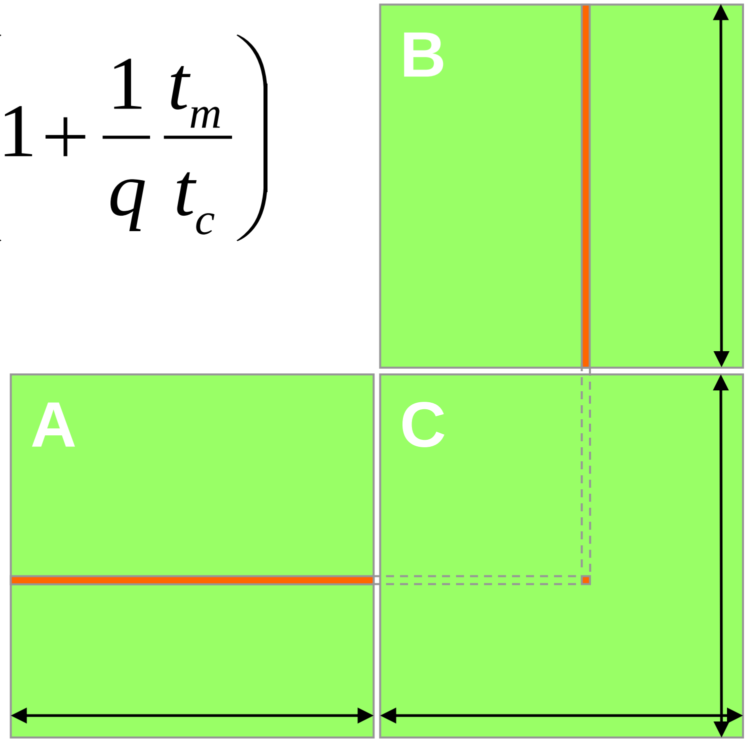


The C-C ratio

- One thread computes an element
- A and B are read n times from global memory: $2n^3$.

$$ft_c + mt_m = ft_c\left(1 + \frac{mt_m}{ft_c}\right) = ft_c\left(1 + \frac{1}{q}\frac{t_m}{t_c}\right)$$

- The c-c ratio $q=2n^3/2n^3=1$.



How to improve it?

- Use Shared Memory to reuse global memory data (**Hundreds of times faster**)
- The bandwidth from host memory to device memory is 8GB/s (PCI expressx2 GEN2)
  - Higher bandwidth is for pinned memory (later)
- In G80, the bandwidth from device memory to GPU is 86.4GB/s   0.9*384/8*2=86.4
  - 900MHZ memory clock, 384 bit interface, 2 issues (DDR RAM: double data rate)

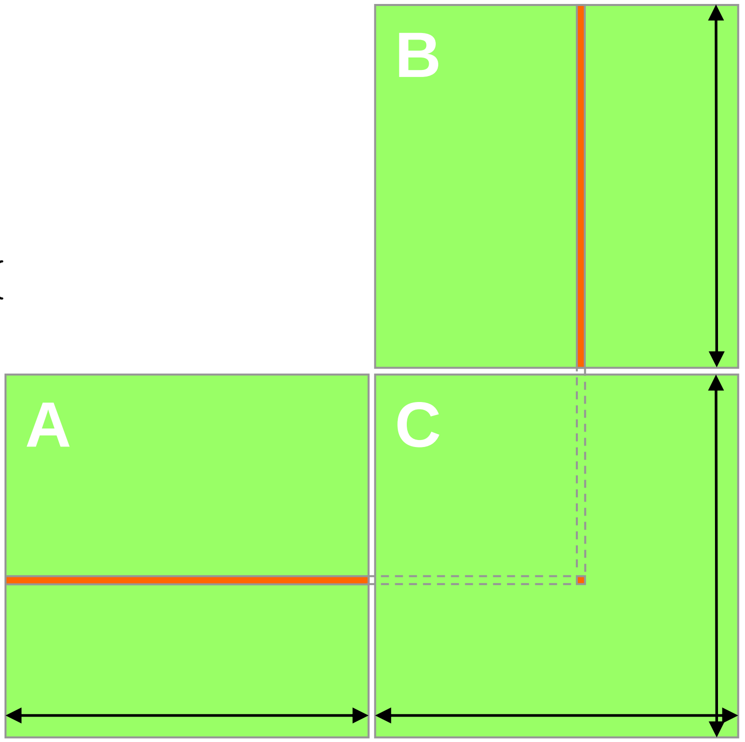
Basic Idea

- Load A,B into shared memory and have several threads use the local version
  - Shared memory has limited size
  - Suppose the size of shared memory can store 1 column and 1 row of A and B
  - Elements of C can be stored in registers
- Memory read can be parallelized too. (how?)
  - Recall the float3 Code

MMM on GPU v2

```
__global__ void MatrixMulKernel(float* A,
                         float* B,float* C,int n){
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    extern __shared__ float SA[];
    SA[ty*n+k] = A[ty*n+k];
    __syncthreads();

    float Cvalue = 0;
    for (int k = 0; k < n; ++k){
       float Bval = B[k*n+tx];
       Cvalue+=SA[ty*n+k]*Bval;
    }
    // Write to device memory;
    C[ty*n+tx] = Cvalue;
}
```



The C-C ratio

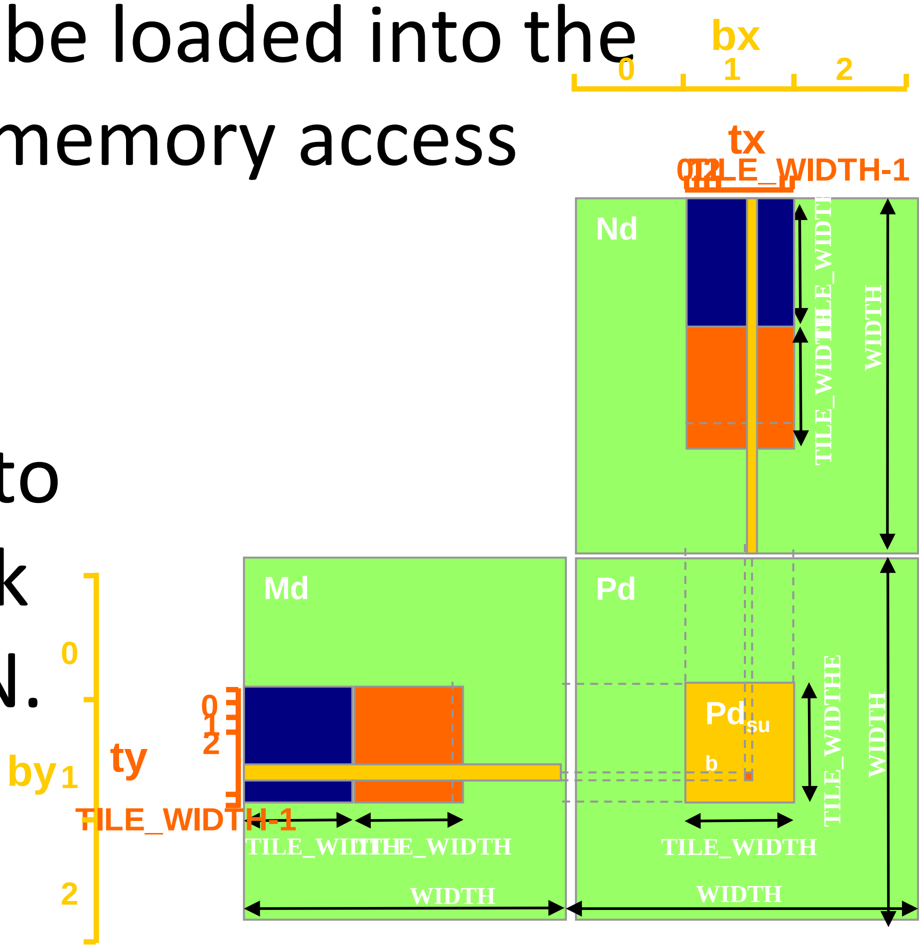
- One block of threads compute a row of C
  - A is read n times from global memory: n^2.
  - B is read n times from global memory: n^3.
  - The c-c ratio $q=2n^3/(n^3+n^2) \sim 2$.

$$ft_c + mt_m = ft_c\left(1+\frac{mt_m}{ft_c}\right) = ft_c\left(1+\frac{1}{q}\frac{t_m}{t_c}\right)$$

- Another problem: the matrix size is limited by
  - Number of threads per block
  - The size of shared memory

Another try

- If the matrix size is small enough, say 16x16, then A and B can both be loaded into the shared memory → n^2 memory access
  - 16x16x4x3 =3K < 16K
  - The c-c ratio is n/2
- Partition A, B, and C into N×N blocks. Each block submatrix is of size n/N.



Block matrix-matrix multiplication

- Partition A, B, and C into N×N blocks. Each submatrix is of size n/N. Suppose $M>3(n/N)^2$.
- Denote A[I, J] the I, J block submatrix of A.

```
for I = 1:N %
    for J = 1:N % read C[I,J] into fast memory
        for K = 1:N % read A[I,K] and B[K,J] into fast memory
                C[I,J]=C[I,J] + A[I,K]*B[K,J]
        end
    end
end
```

The C-C ratio

- Memory access counts.
  - Read B N times: Nn^2.
  - Read A N time: Nn^2.
  - Write C 1 time: n^2.
- Total memory access is $(2N+1)n^2 \sim 2Nn^2$.
- The ratio $q=2n^3/(2Nn^2) = n/N$
- Which N can maximize the performance?

Performance of G80 (8800GTX)

- Peak performance of G80 is 345.6GFLOPS
  - 128 MP; each runs 1.35GHZ;
  - One mult-and-add per cycle for floating point operations (more on this later)
- Need two floating numbers (8 bytes) for one mult-and-add➔4 bytes for one operation
  - For peak performance, need 4*345.6=1386GB/s
- Memory bandwidth of G80 is 86.4GB/s
  - Need c-c ratio 1386/86.4 > 16.04

Homework

- Read chap3 and chap4 from UIUC's class
  - http://courses.ece.illinois.edu/ece498/al/Syllabus.html
- Implement matrix-matrix multiplication and using memory padding and tiling
  - See this webpage for reference
  - http://heresy.spaces.live.com/blog/cns!E0070FB8ECF9015F!3435.entry
  - Write different versions and compare their performance to learn the effectiveness of each techniques