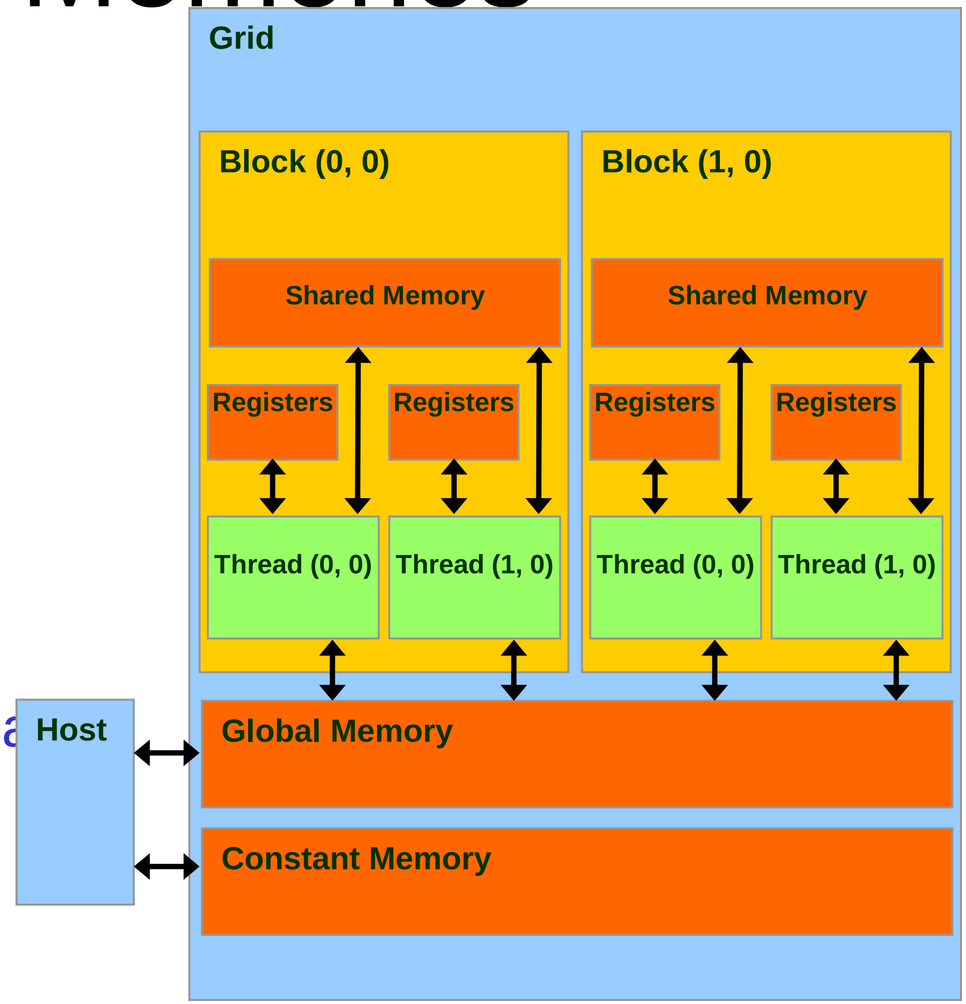
CUDA Memory Hierarchy and Synchronization

COEN 296 Spring 2012

Dr. Maria Pantoja

Hardware Implementation of CUDA Memories

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory



CUDA Variable Type Qualifiers

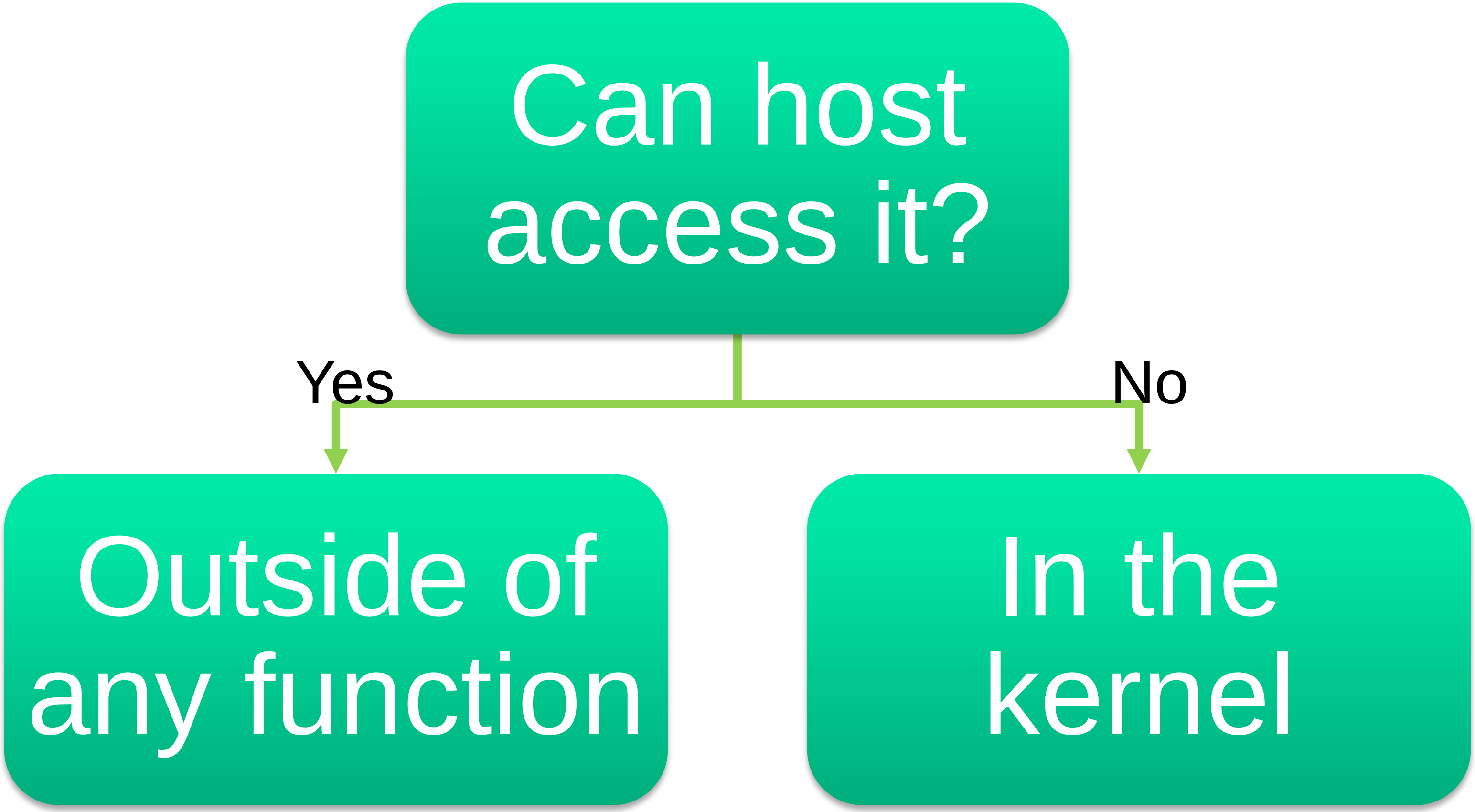
Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

Where to declare variables?




```
__constant__ int constant_var;

__device__ int global_var;
```

```
int var;

int array_var[10];

__shared__ int shared_var;
```

Shared Memory

Shared Memory

CUDA C makes available a region in memory that we call shared memory.

A programmer can declare a variable as __shared__ to make this variable resident in shared memory.

Shared Memory

For what?

- CUDA compiler treats variables on shared memory differently
- Creates a copy of the variable for each block launched on GPU
- Every thread in the block shares the memory, but threads cannot see or modify the copy of this variable on other blocks
- So this is the best way for threads to communicate within a block and collaborate on computations
- Shared memory reside on GPUs the latency to access shared memory is very low
- If we can communicate among threads that means we will need a mechanism for synchronization

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // each thread loads two elements from global memory
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i – x_i_minus_one;
  }
}
```

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // what are the bandwidth requirements of this kernel?
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i – x_i_minus_one;
  }
}
```

Two loads

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // How many times does this kernel load input[i]?
    int x_i = input[i]; // once by thread i
    int x_i_minus_one = input[i-1]; // again by thread i+1

    result[i] = x_i – x_i_minus_one;
  }
}
```

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // Idea: eliminate redundancy by sharing data
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i – x_i_minus_one;
  }
}
```

Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
  // shorthand for threadIdx.x
  int tx = threadIdx.x;
  // allocate a __shared__ array, one element per thread
  __shared__ int s_data[BLOCK_SIZE];
  // each thread reads one element to s_data
  unsigned int i = blockDim.x * blockIdx.x + tx;
  s_data[tx] = input[i];

  // avoid race condition: ensure all loads
  // complete before continuing
  __syncthreads();
  ...
}
```

Example – shared variables

```
// optimized version of adjacent difference

__global__ void adj_diff(int *result, int *input)

{

  ...

  if(tx > 0)

    result[i] = s_data[tx] - s_data[tx-1];

  else if(i > 0)

  {

    // handle thread block boundary

    result[i] = s_data[tx] - input[i-1];

  }

}
```

Example – shared variables

```
int main(void){
  …
   adj_diff<<<NUM_BLOCKS,BLOCK_SIZE>>>(r,i);
   …
}
```

Shared Memory Example
CPU Dot Product

{a1, a2, a2, a4}*{b1, b2, b3, b4}= a1*b1+a2*b2+a3*b3+a4*b4

Code on a CPU:

```
#define N 100
void dot( float *a, float *b, float *c){
    int i; *c=0;
    for( i=0; i<N;i++)
          *c+=a[i]*b[i];
}
```

Shared Memory Example
GPU Dot Product

```
//One solutuon will be to calculate one multiplication per thread
//and do the reduction on the cpu
///Kernel
 __global__ void dot( float *a, float *b, float *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    c[i]= a[tid] * b[tid];
}
//on CPU main
for(i=0;i<n;i++)
    cfinal+=c[i];
```

BuT this solution is very slow

Shared Memory Example. GPU Dot Product

```
//One solution will be to calculate one multiplication per thread
//and do the reduction on the GPU
 __global__ void dot( float *a, float *b, float *c ) {
    float temp[N]; //local array
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    temp[i]= a[tid] * b[tid];
    __syncthreads()
   int i = blockDim.x/2; //reduction
   while (i != 0) {
        if (threadIdx.x < i)
            temp[threadIdx.x] += temp[threadIdx.x + i];
        __syncthreads();
        i /= 2;
    }
    //threads finish computation and result is on first entry
    if (threadIdx.x== 0)
          c[blockIdx.x] = temp[0];
}    //problem is that local mem is x100 slower than shared mem
```

Shared Memory Example. GPU Dot Product

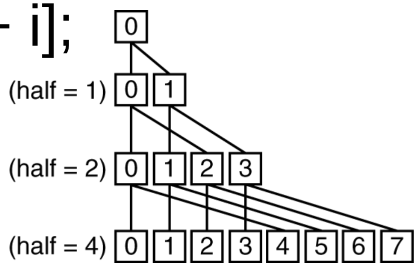
```
//One solution will be to calculate one multiplication per thread
//and do part of the reduction on the GPU
 __global__ void dot( float *a, float *b, float *c ) {
    __shared__ float temp[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[threadIdx.x]*b[threadIdx.x];
    _syncthreads();
   int i = blockDim.x/2; REDUCTION STEP:
   while (i != 0) {
        if (threadIdx.x < i)
            temp[threadIdx.x] += temp[threadIdx.x + i];
        __syncthreads();
        i /= 2;
    }
    //threads finish computation and result is on first entry
    if (threadIdx.x == 0)
          c[blockIdx.x] = temp[0];
}    //will work better if we use the shared mem more efficiently
```

Shared Memory Example
GPU Dot Product

```
__global__ void dot( float *a, float *b, float *c ) {
    //buffer to store each thread running sum
    //each block has its own copy of shared memory so no need to
    //allocate for block offset just for threadsperblock
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float   temp = 0;
    while (tid < N) { //each thread has to perform this operation
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    // save the cache values
    cache[cacheIndex] = temp;
    // synchronize threads before adding the values on the cache
    __syncthreads();
```

Shared Memory Example
GPU Dot Product

```
// At this point we need to sum all temporary values we have
//placed on the cache. REDUCTION STEP:
int i = blockDim.x/2; //threads with id less than this will do the work
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
//here all threads had finish computation and result is on first entry
//so we just need to store this value to global memory
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
}
```



Shared Memory Example
GPU Dot Product

REDUCTION

Taking an input array, performing computations that produce a smaller array of results.

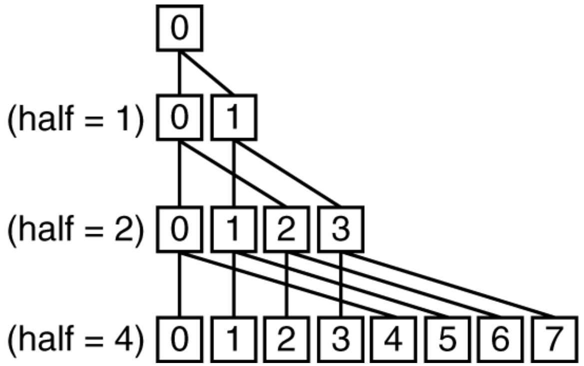
Parallel computations produce all the time reductions

The naïve way to perform reduction is by having one thread iterate over the shared memory and calculate a running sum.

But we have many threads that can work in parallel so the better algorithms is:

1. each thread adds two values in cache[] and stores the result back in cache[] (result is a cache half the size)

2. we continue in this fashion for log2(threadsperblock) steps until we have the sum of every entry in the cache



Shared Memory Example
GPU Dot Product

```
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
```

After the reduction is done each block has input the results of all threads in the block computations into an array cache[]

Then we copy this array into global memory

We need to sum the entries of c[] and we will use the CPU. WHY? Because the GPU is bad at performing the last step of the reduction since the size of the data is so small

Shared Memory Example
GPU Dot Product (main)

```
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
        imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

int main( void ) {
    float   *a, *b, c, *partial_c;
    float   *dev_a, *dev_b, *dev_partial_c;

    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
```

Shared Memory Example
GPU Dot Product (main)

```
// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                          N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                          N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                          blocksPerGrid*sizeof(float) ) );

// fill in the host memory with data
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}
```

Shared Memory Example
GPU Dot Product (main)

```
dot<<<blocksPerGrid, threadsPerBlock>>>( dev_a, dev_b,
                                          dev_partial_c );

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                          blocksPerGrid*sizeof(float),
                          cudaMemcpyDeviceToHost ) );

// finish up on the CPU side
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
```

Shared Memory Example
GPU Dot Product

```
    // free memory on the gpu side
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_partial_c ) );

    // free memory on the cpu side
    free( a );
    free( b );
    free( partial_c );
}
```

Dot Product Optimized Incorrectly

Take a look at the second loop __syncthreads()

```
while (i != 0) {
    if (cacheIndex < i){
        cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
    }
    i /= 2;
}
```

Seams like we should only wait for the threads that are going to write

But this will not work and will stop the GPU from running

Why?

- Not every thread will execute the instruction others don't (THREAD DIVERGENCE)
- Since _synchreads guarantees that every thread in the block will not continue to next instruction until done, so hw waits forever

Dot Product Optimized Incorrectly

The instruction that every thread has to execute is inside a conditional block like an if()

Some threads remain idle while others are working is call thread divergence
The problem is with __synchthread() , the CUDA architecture guarantees that no thread will advance to an instruction beyond __synchthread() until all the threads on the block execute synchronizethreads()

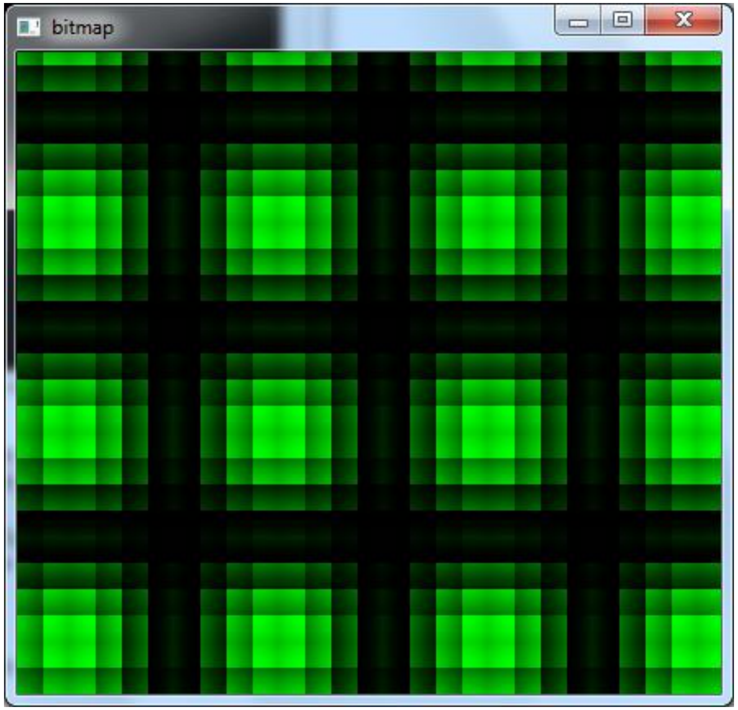
Shared Memory Bitmap

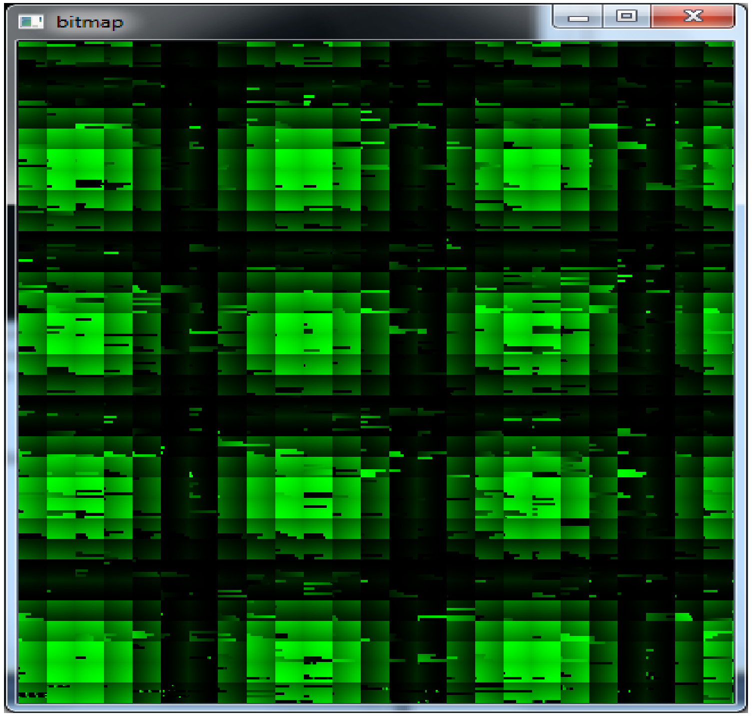
```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    __shared__ float    shared[16][16];

    // now calculate the value at that position
    const float period = 128.0f;

    shared[threadIdx.x][threadIdx.y] =
            255 * (sinf(x*2.0f*PI/ period) + 1.0f) *
                 (sinf(y*2.0f*PI/ period) + 1.0f) / 4.0f;
```

Shared Memory Bitmap

```
    // removing this syncthreads shows graphically what happens
    // This is an example of why we need it.
    __syncthreads();

    ptr[offset*4 + 0] = 0;
    ptr[offset*4 + 1] = shared[15-threadIdx.x][15-threadIdx.y];
    ptr[offset*4 + 2] = 0;
}
```

Shared Memory Bitmap

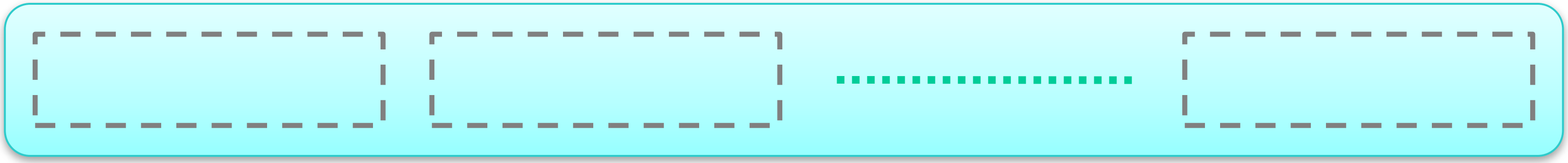




A Common Programming Strategy

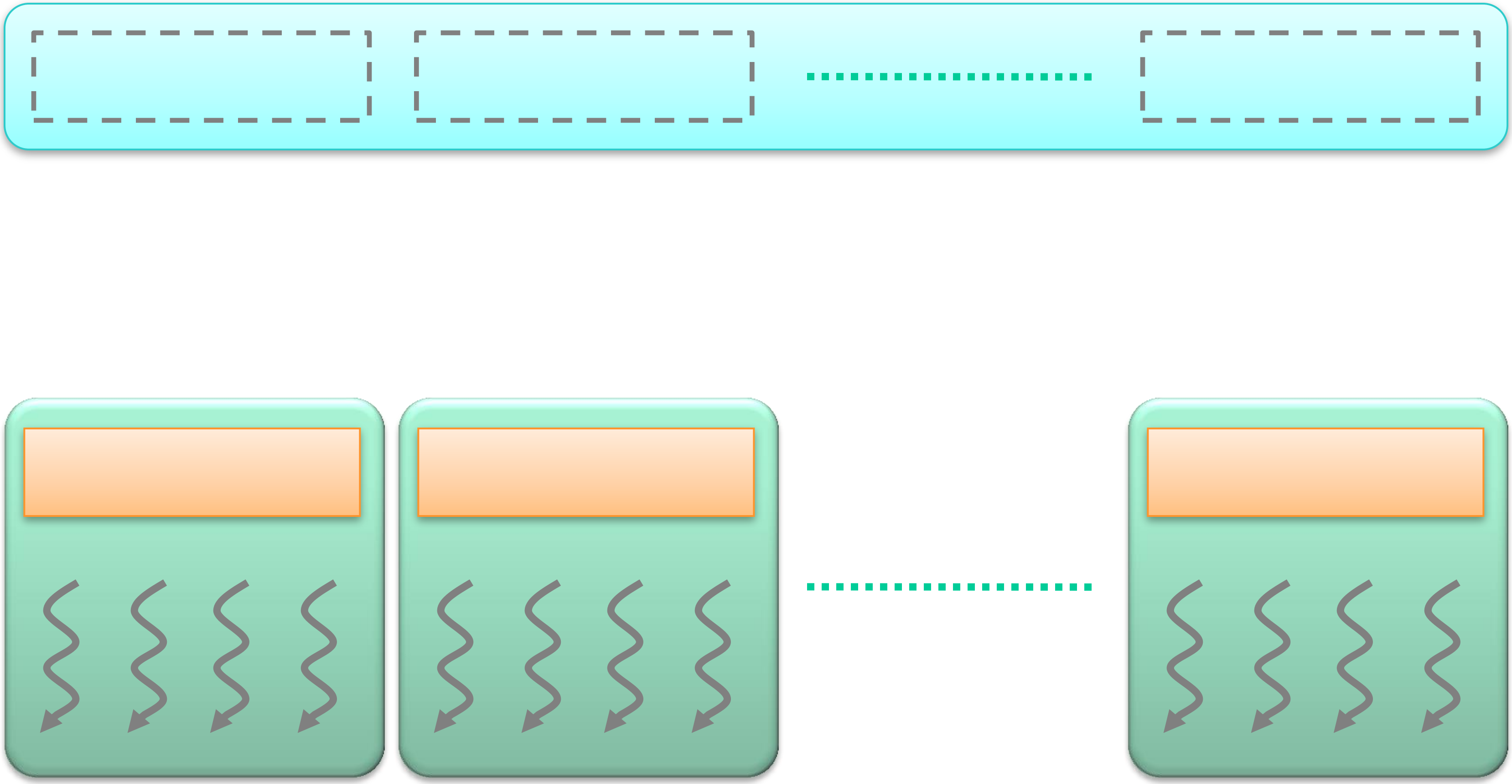
- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory:
  - Divide and conquer

A Common Programming Strategy



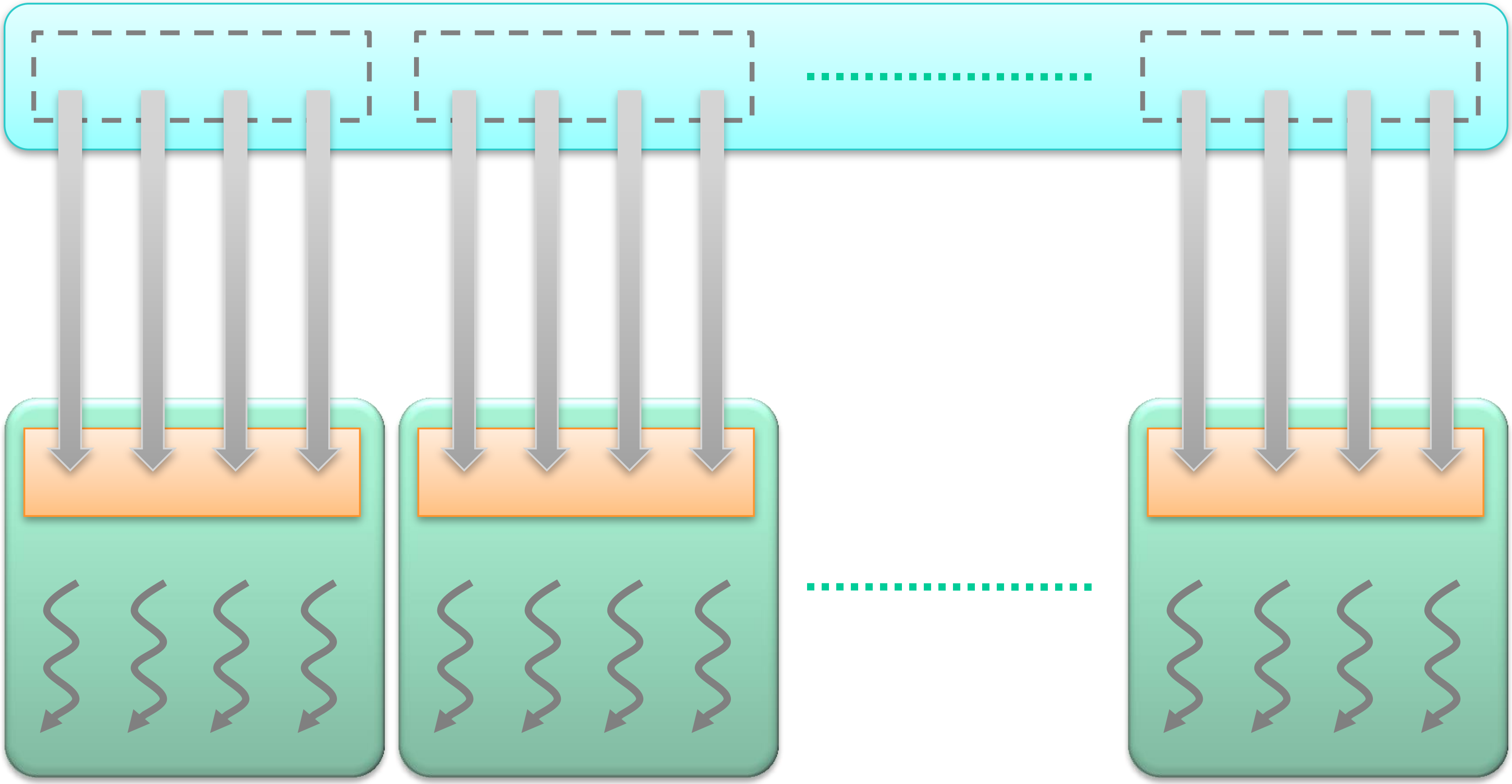
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



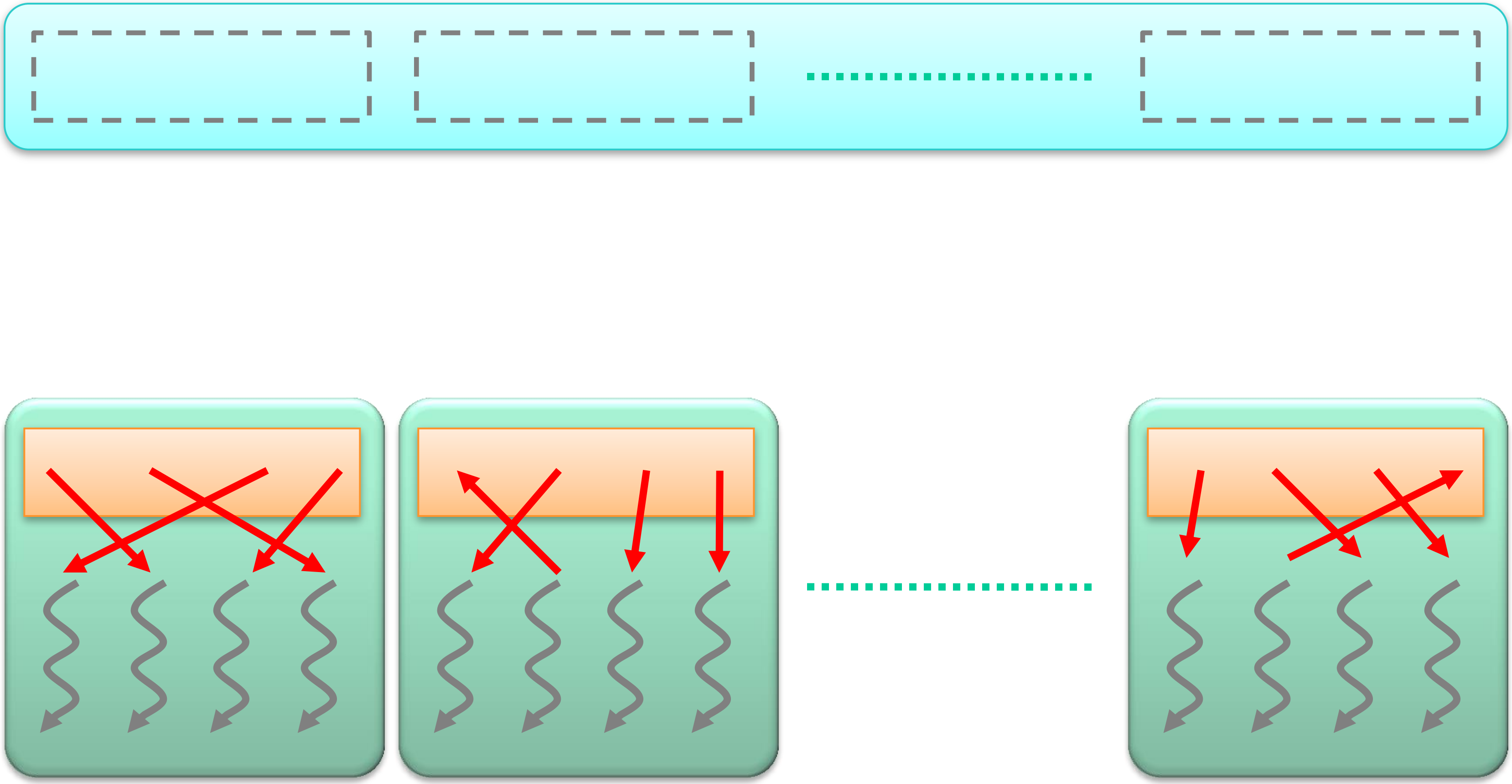
- Handle each data subset with one thread block

A Common Programming Strategy



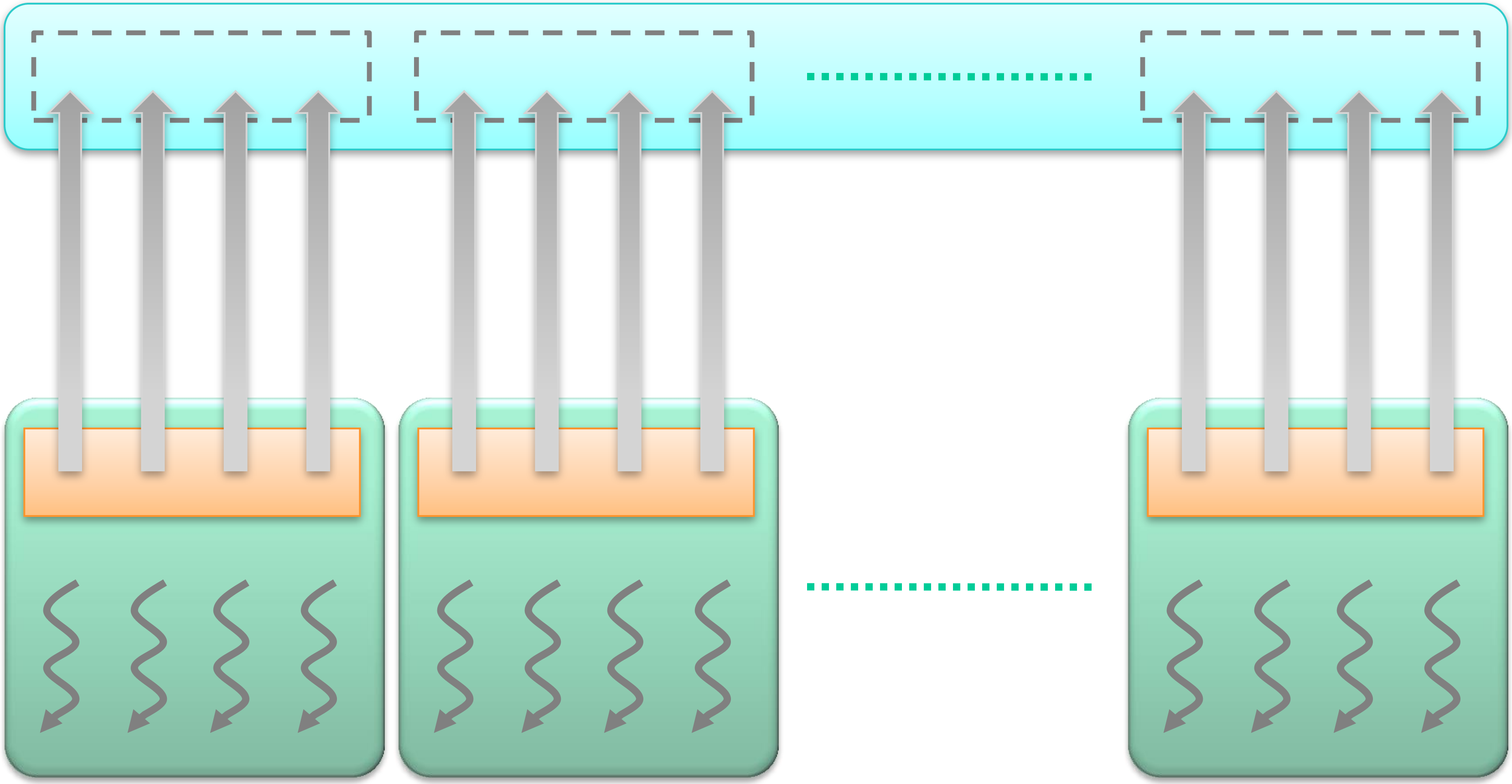
- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

A Common Programming Strategy



- Perform the computation on the subset from shared memory

A Common Programming Strategy



- Copy the result from shared memory back to global memory

A Common Programming Strategy

- Carefully partition data according to access patterns
- Read-only ➔ `__constant__` memory (fast)
- R/W & shared within block ➔ `__shared__` memory (fast)
- R/W within each thread ➔ registers (fast)
- Indexed R/W within each thread ➔ local memory (slow)
- R/W inputs/results ➔ `cudaMalloc`‘ed global memory (slow)

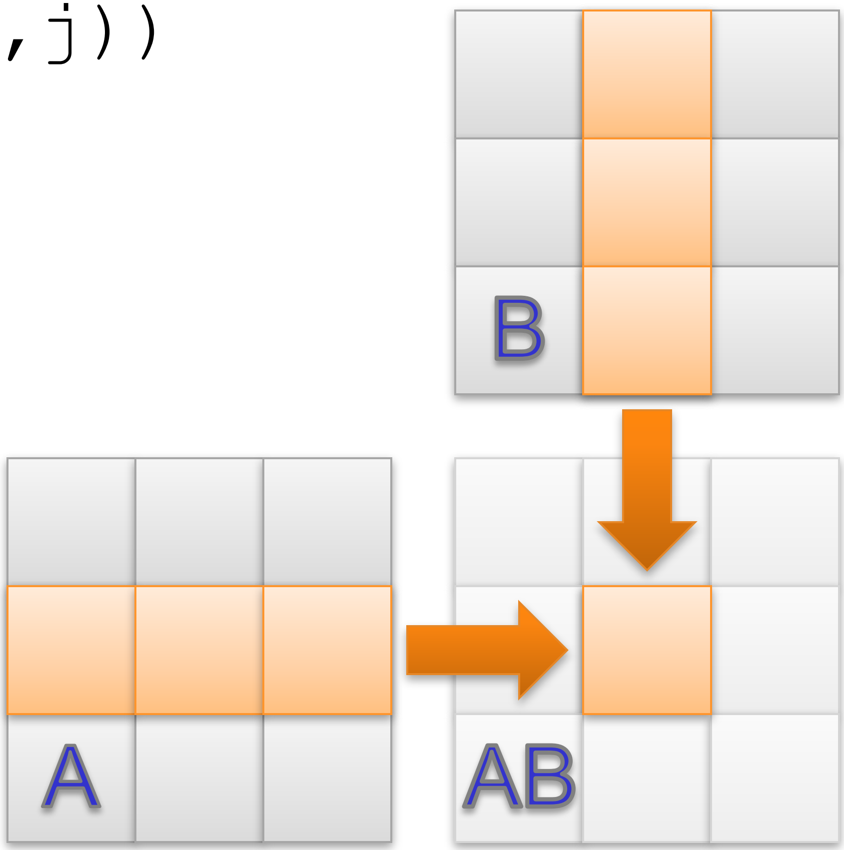
Matrix Multiplication Example

- AB = A * B

  Each element AB_{ij}

  `=dot(row(A,i),col(B,j))`

- Parallelization strategy
  - Thread → AB_{ij}
  - 2D kernel



Matrix Multiplication CPU

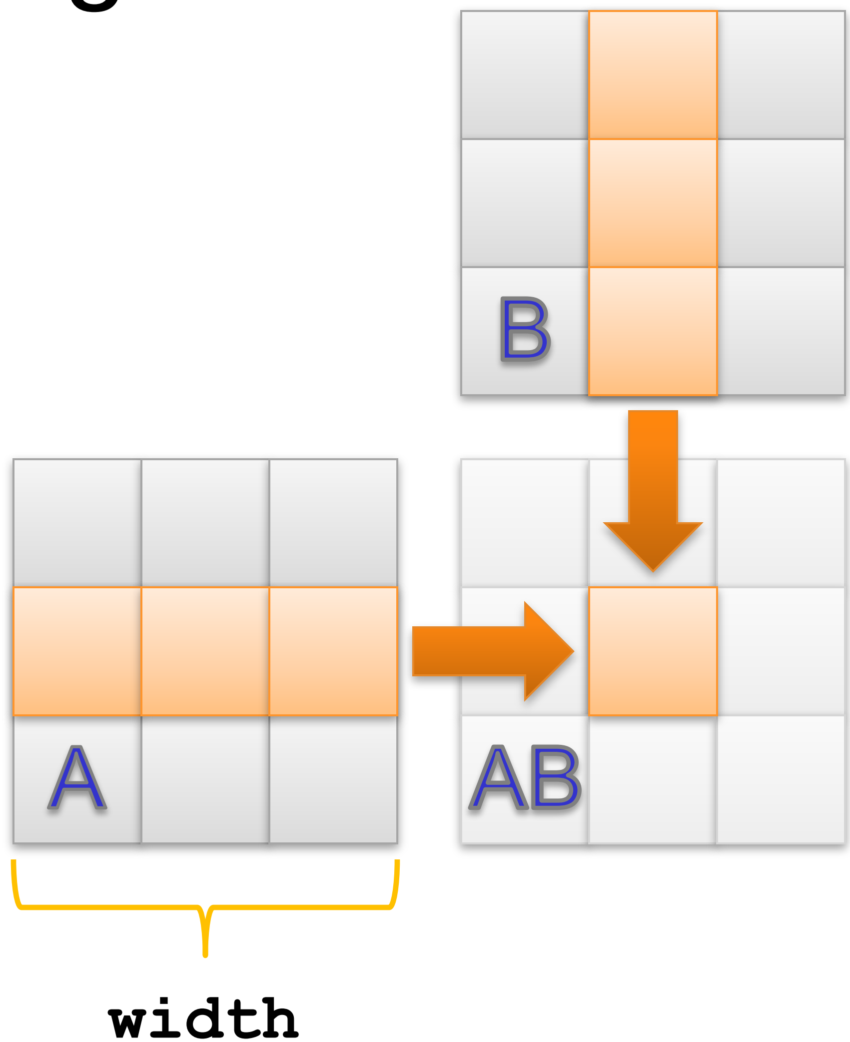
```
void mult(float a[][N], float b[][N], float c[][N]){
  for (i=0; i<N;i++)
     for(j=0; j<N; j++){
        c[i][j]=0;
        for(k=0;k<N;k++)
           c[i][j]+=a[i][k]*b[k][j];
     }
}
```

First GPU Implementation

```
__global__ void mat_mul(float *a, float *b,
                        float *ab, int width)
{
  // calculate the row & col index of the element
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;

  float result = 0;

  // do dot product between row of a and col of b
  for(int k = 0; k < width; ++k)
    result += a[row*width+k] * b[k*width+col];

  ab[row*width+col] = result;
}
//main
mat_mul<<<(width/16, width/16),(16,16)>>>(a_dev,
  b_dev, c_dev, width)
```

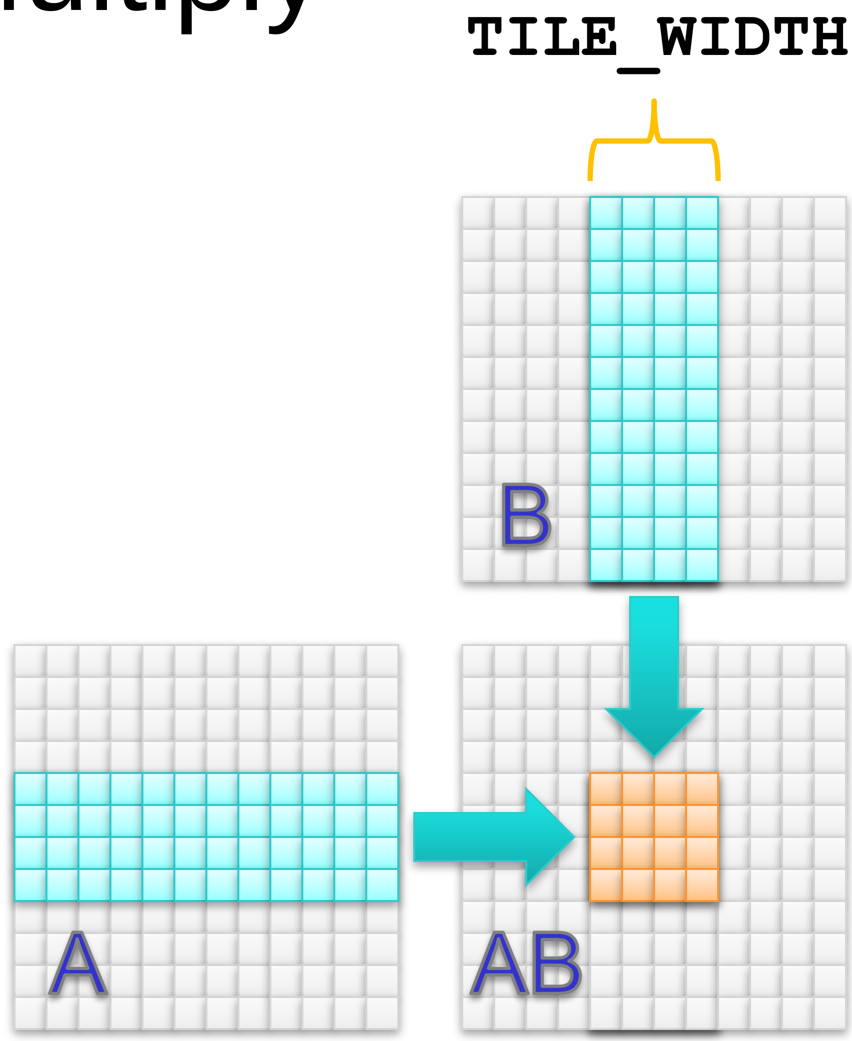
Idea: Use `__shared__` memory to reuse global data

- Each input element is read by `width` threads
- Load each element into `__shared__` memory and have several threads use the local version to reduce the memory bandwidth



Tiled Multiply

- Partition kernel loop into phases
- Load a tile of both matrices into `__shared__` each phase
- Each phase, each thread computes a partial result



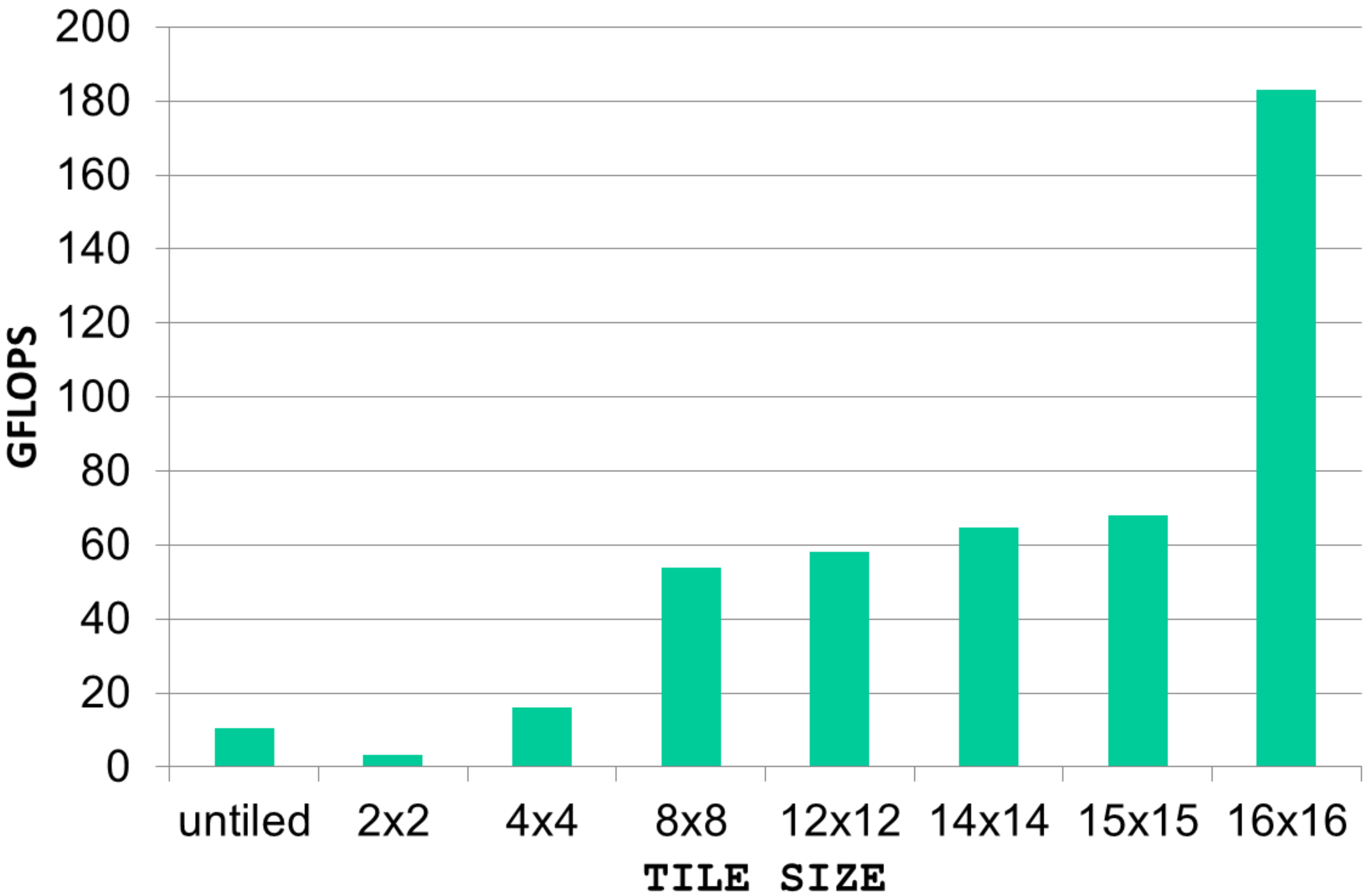
Better Implementation

```
__global__ void mat_mul(float *a, float *b,
                        float *ab, int width)
{
  int tx = threadIdx.x, ty = threadIdx.y;
  int bx = blockIdx.x,  by = blockIdx.y;
  // allocate tiles in __shared__ memory
  __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
  __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];
  // calculate the row & col index to identify
  //element to work on
  int row = by*blockDim.y + ty;
  int col = bx*blockDim.x + tx;

  float result = 0;
```

Better Implementation

```
  // loop over the tiles of the input in phases
  for(int p = 0; p < width/TILE_WIDTH; ++p)
  {
    // collaboratively load tiles into __shared__
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; ++k)
      result += s_a[ty][k] * s_b[k][tx];
    __syncthreads();
  }

  ab[row*width+col] = result;
}
```

TILE_SIZE Effects



Thoughts

- Effective use of CUDA memory hierarchy decreases bandwidth consumption to increase throughput
- Use `__shared__` memory to eliminate redundant loads from global memory
  - Use `__syncthreads` barriers to protect `__shared__` data
  - Use atomics if access patterns are sparse or unpredictable (Next)
- Optimization comes with a development cost
- Memory resources ultimately limit parallelism

For Not Squared Matrices PADDING

```
__global__ void matrixMultiplyShared(float * A, float * B, float * C,
                              int numARows, int numAColumns,
                              int numBRows, int numBColumns,
                              int numCRows, int numCColumns) {
   __shared__ float S_A[TILE_WIDTH][TILE_WIDTH];
   __shared__ float S_B[TILE_WIDTH][TILE_WIDTH];

    int bx= blockIdx.x; int tx= threadIdx.x;
    int by= blockIdx.y; int ty = threadIdx.y;
    //identify row and column to work on
    int Row=by*TILE_WIDTH+ty;
    int Col=bx*TILE_WIDTH+tx;
    float Pvalue=0;
   //loop over all tiles
    int ceiling=max(((numAColumns-1)/TILE_WIDTH)+1,((numBRows-
1)/TILE_WIDTH)+1);
```

For Not Squared Matrices
PADDING

```
for(int m=0;m<ceiling;++m){
      //copy to shared memory matrix A and B
       if((Row<numCRows)&&(m*TILE_WIDTH+tx)<numAColumns)
          S_A[ty][tx]=A[Row*numAColumns+(m*TILE_WIDTH+tx)];
      else
        S_A[ty][tx]=0; //pad with zeros

      if(Col<numCColumns && (m*TILE_WIDTH+ty)<numBRows)
         S_B[ty][tx]=B[(m*TILE_WIDTH+ty)*numBColumns+Col];
      else
        S_B[ty][tx]=0;
      __syncthreads();
```

For Not Squared Matrices
PADDING

```
    if(Row<numCRows && Col<numCColumns){
      for(int k=0; k<TILE_WIDTH;++k){
          Pvalue+=S_A[ty][k]*S_B[k][tx];
        }
     }
     __syncthreads();

     if(Row<numCRows && Col<numCColumns){
         C[Row*numCColumns+Col]=Pvalue;
     }

  }
}
```

Constant Memory

Constant Memory

With hundreds of ALU on the GPU, often the bottleneck is not the arithmetic throughput of the chip but rather the memory bandwidth of the chip.

So is worthy to investigate ways to decrease this memory bandwidth

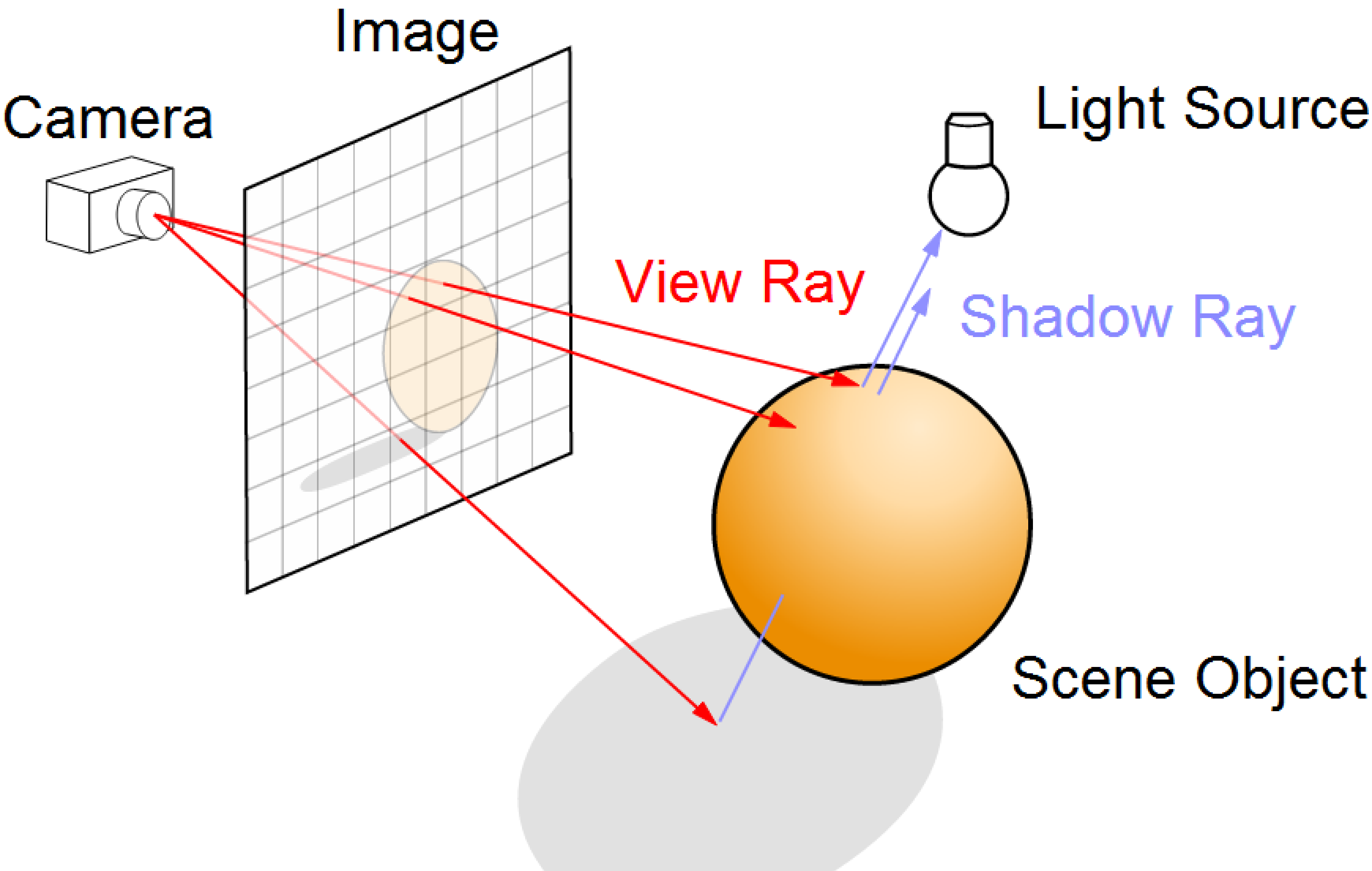
Constant Data: Data that is not going to be modified during the kernel execution .
Nvidia hardware provides ~64KB of constant memory that is treated differently than global memory

Ray Tracing Introduction

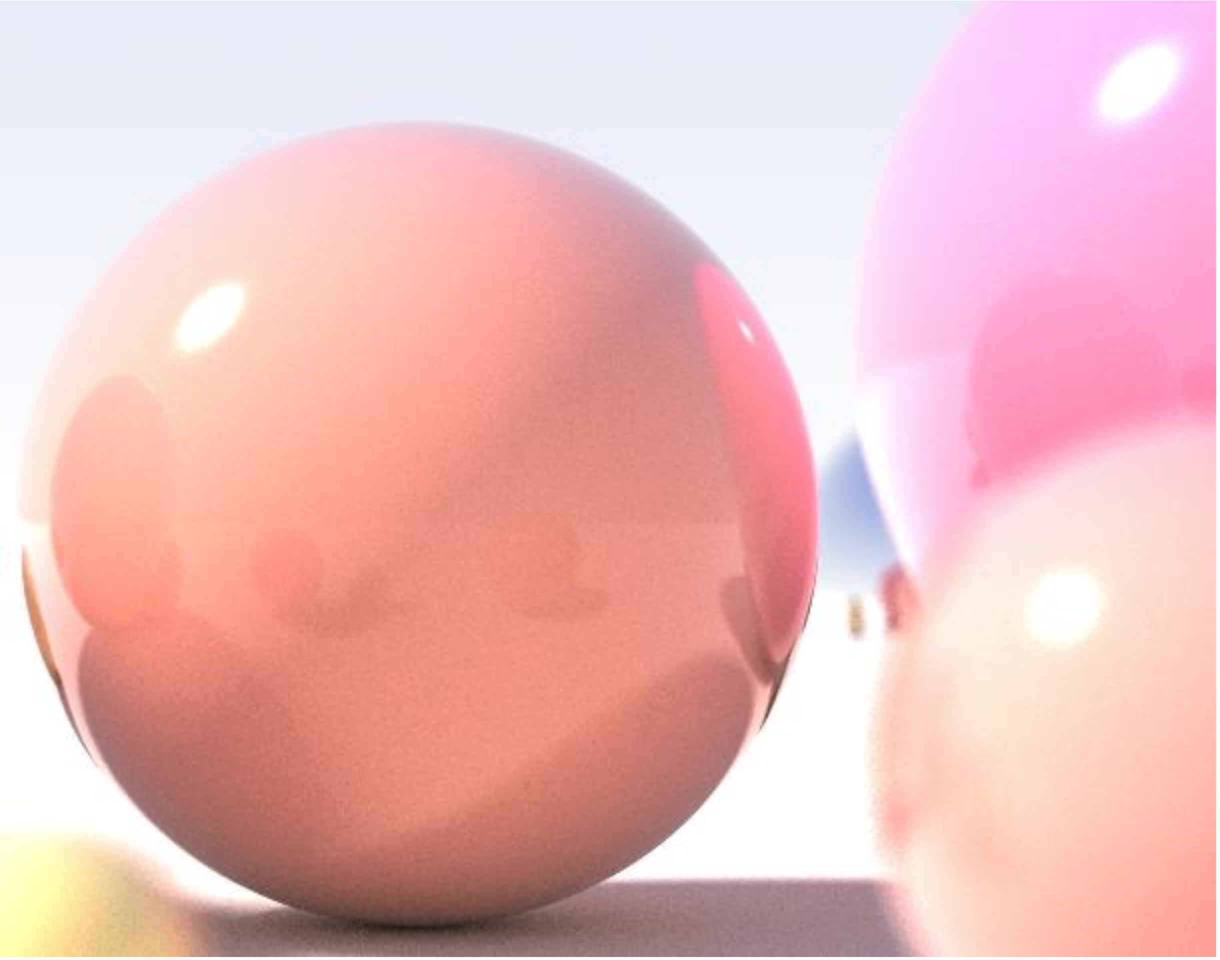
ray tracing

Technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. Ray tracing is capable of simulating a wide variety of optical effects, such
as reflection and refraction, scattering,
and dispersion phenomena

Ray Tracing Introduction



Ray Tracing Introduction



Ray Tracing on the GPU

Build a very simple Ray Tracing :

- Will only support scenes on spheres
- Camera is restricted to the z-axis facing the origin
- no lightning on the scene

We will do:

Fire a ray from each pixel and keep track of which rays hit which spheres.

It will track the depth of each of these hits

in case the ray passes several spheres we will only see the one closest to the camera

Ray Tracing

```
struct Sphere {
    float   r,b,g;
    float   radius;
    float   x,y,z; //location of the center of the spheres
    //given a ray shot from the pixel at (ox,oy), this method
    //computes whether the ray interset the sphere
    __device__ float hit( float ox, float oy, float *n ) {
        float dx = ox - x;
        float dy = oy - y;
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf( radius*radius - dx*dx - dy*dy );
            *n = dz / sqrtf( radius * radius );
            return dz + z;
        }
        return -INF;
    }
};
#define SPHERES 20
__constant__ Sphere s[SPHERES];
```

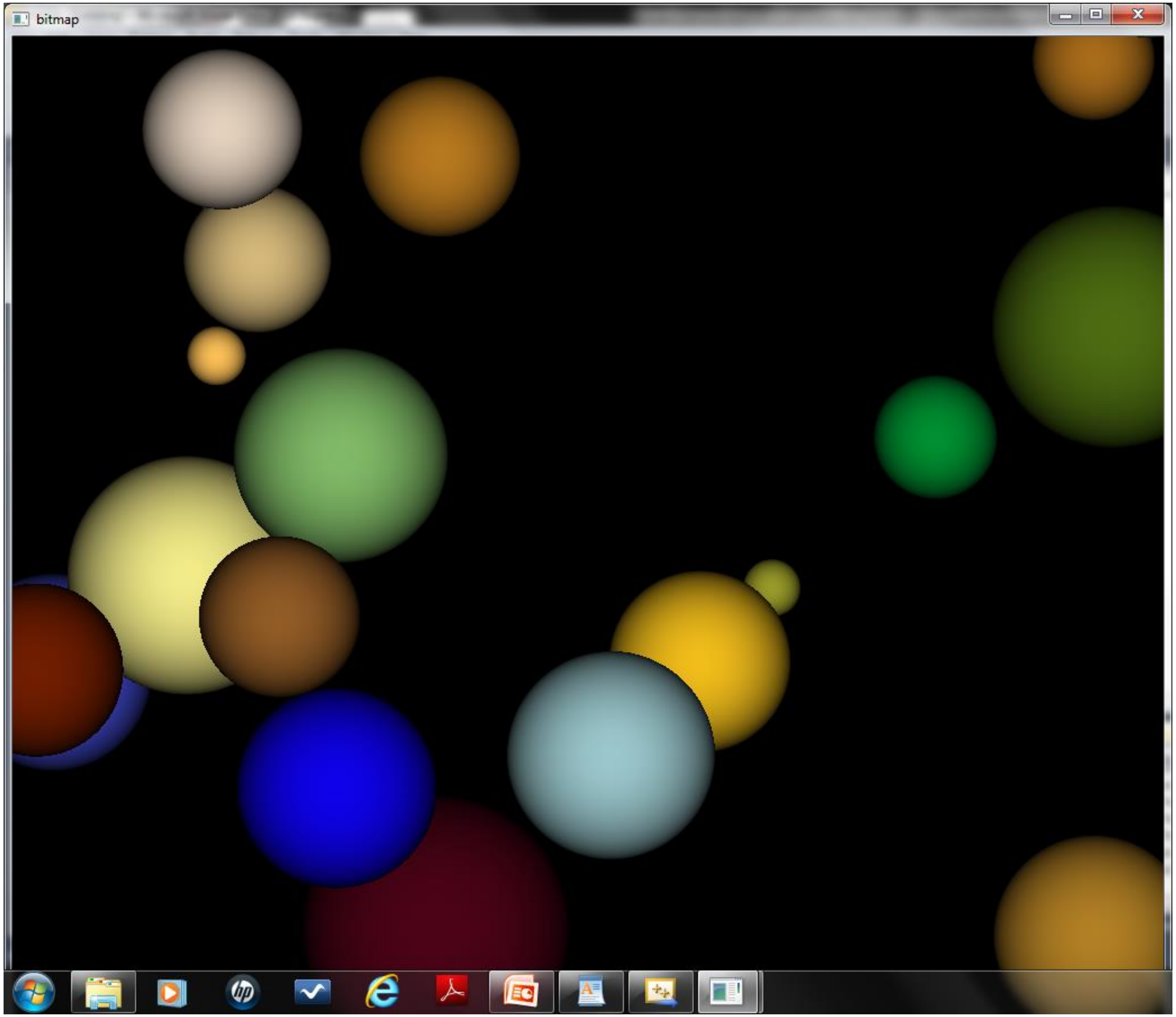
Ray Tracing

```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float   ox = (x - DIM/2); float oy = (y - DIM/2);//z-axis in center
    float   r=0, g=0, b=0; float   maxz = -INF;
    for(int i=0; i<SPHERES; i++) {
        float   n; float   t = s[i].hit( ox, oy, &n );
        if (t > maxz) {
            float fscale = n;
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t; //if the ray hits the sphere show just the closer
        }
    }
    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
}
```

```
struct DataBlock {
    unsigned char   *dev_bitmap;
};
int main( void ) {
    DataBlock   data;
    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char   *dev_bitmap;

    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR(cudaMalloc((void**)&dev_bitmap,bitmap.image_size()));
    // allocate temp memory, initialize it, copy to constant
    // memory on the GPU, then free our temp memory
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s, sizeof(Sphere) * SPHERES) );
    free( temp_s );
```

```
    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( dev_bitmap );

    // copy our bitmap back from the GPU for display
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    // get stop time, and display the timing results
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
   //we can not read the value of the stop until GPU completed
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );

    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );

    // display
    bitmap.display_and_exit();
}
```

Shared Memory Bitmap



Performance with constant memory

There are two reasons why reading from the 64 kb constant memory can save bandwidth over using standard global memory:

1. A single read from constant memory can be broadcast to other nearby threads, effectively saving up to 15 reads that are woven together and are executed in lockstep
   - In CUDA a warp is a collection of 32 threads
   - Every thread in a wrap executes the same instructions on different data
   - Nvidia hardware can broadcast a single memory read to each half-warp
2. Constant memory is cached, so consecutive reads of same address will not incur any additional memory traffic

There can be a downside in performance when using constant memory when all threads read different addresses

Measuring Performance with Events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
…..

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
// cuda events are asynchronous, so to safely read the value from stop we wait
//until GPU has completed the work and completed the task. This function will
    //Instruct the CPU to synchronize on an event


cudaEventElapsedTime(&elapsedTime, start, stop);
//Print eleapsedTime

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Texture Memory

Texture Memory

Another variety of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns (spatial locality)

Traditionally designed for graphics applications but can also be used for general purpose computing

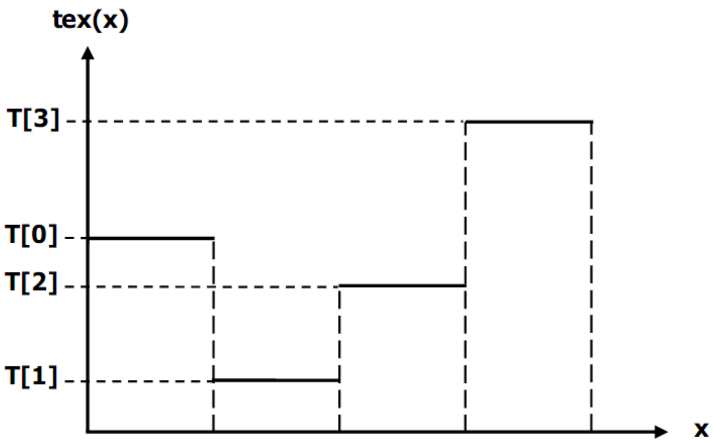
Texture memory as constant memory is cached on chip

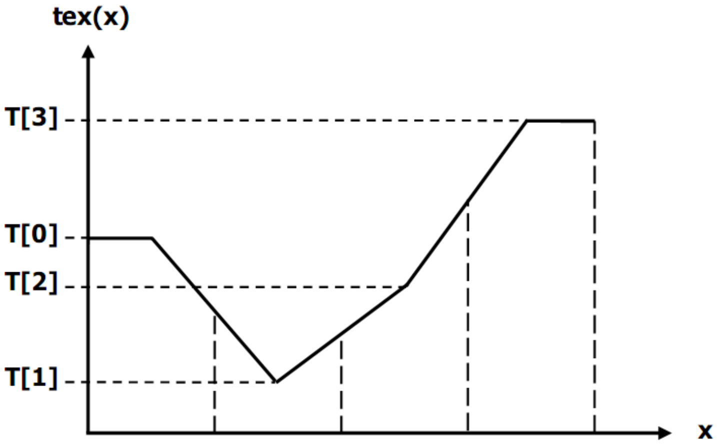
Texture Memory Cache

- Optimized for 2D spatial locality via z-order curve
- Inherits some nice features from the graphics pipeline.
- Get some things for free:

More Nice Texture Memory Features

– Linear interpolation of adjacent data values





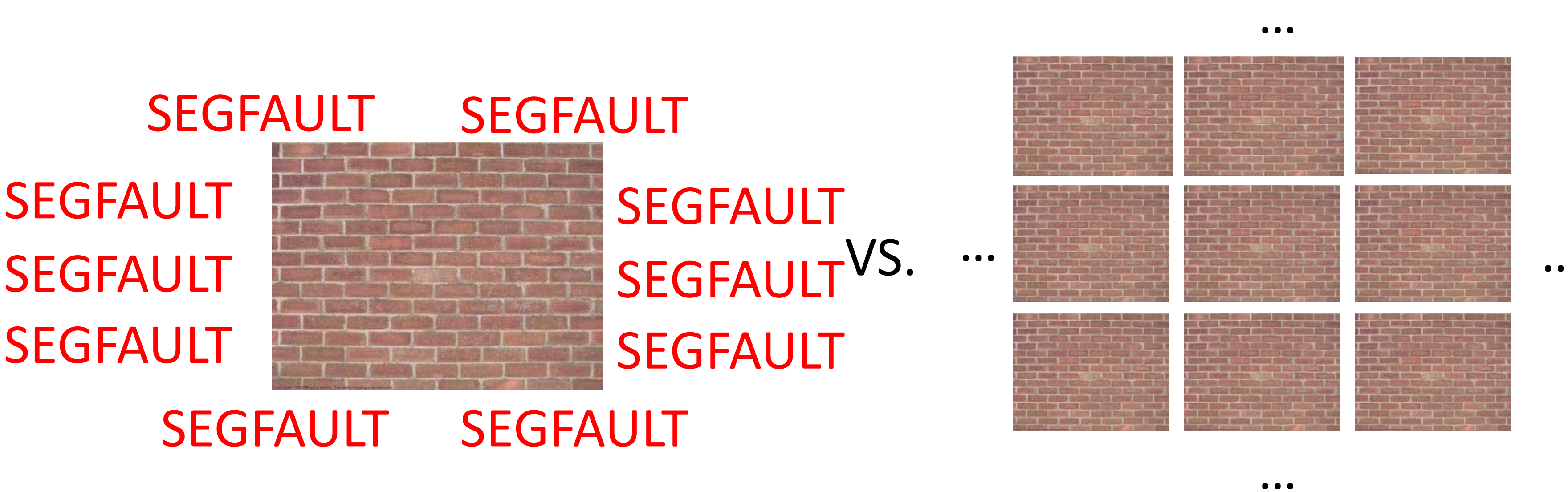
[NVIDIA]

More Nice Texture Memory Features

- Automatic normalization of data when you fetch it
  - [0,1] for unsigned values
  - [-1,1] for signed values
- Automatic normalization of array indices to [0,1]
  - e.g. same code for all problem sizes

More Nice Texture Memory Features

– Automatic boundary handling



When to Use Texture Memory

If you update your data rarely but read it often especially if there tends to be some kind of spatial locality to the read access pattern:

> i.e. nearby threads access nearby locations in the texture

Also, you need to use texture memory in order to visualize your data using the graphics pipeline

When Not to Use Texture Memory

- If you read your data exactly once after you update it.
  - e.g. It would not be appropriate to store the vector $\mathbf{y}$ ($\mathbf{y}=a\mathbf{x}+\mathbf{y}$) in texture memory, since there is a predicable one-to-one relationship between reads and writes.

Image Processing with CUDA

$$\mathbf{I}_{out}(x,y) = f(\text{neighborhood around } \mathbf{I}_{in}(x,y))$$

- Note the memory access pattern:

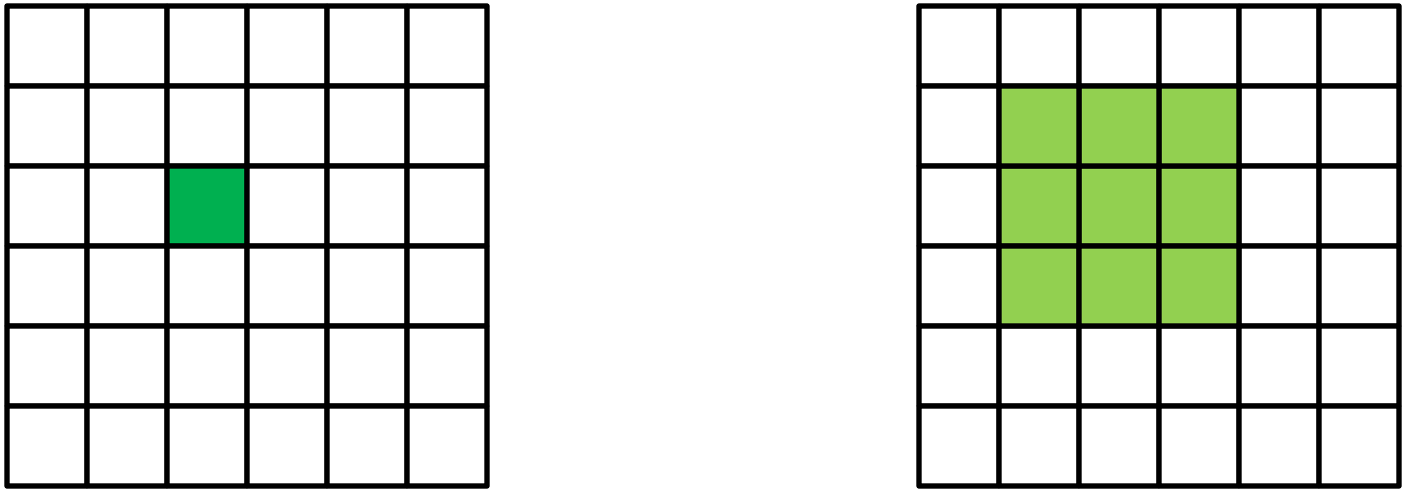


Image Processing with CUDA

$$\mathbf{I}_{\mathbf{out}}(x,y) = f(\text{neighborhood around } \mathbf{I}_{\mathbf{in}}(x,y))$$

- Note the memory access pattern:

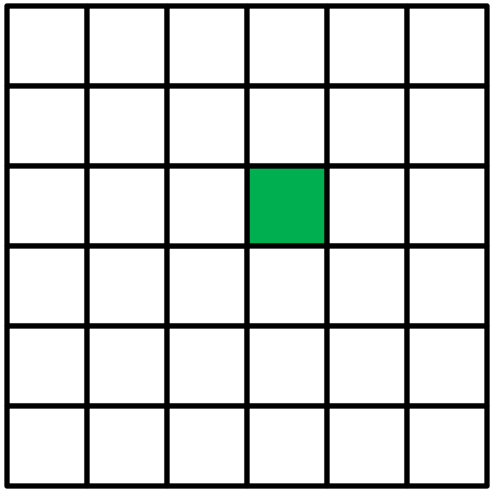
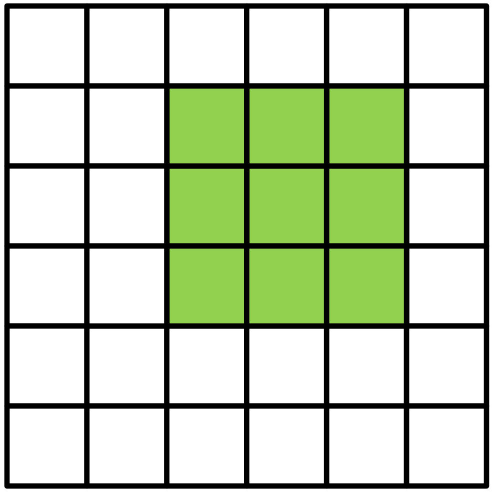
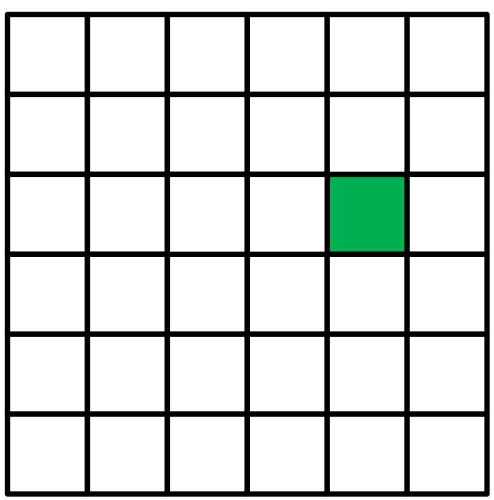



Image Processing with CUDA

$$\mathbf{I}_{\mathbf{out}}(x,y) = f(\text{neighborhood around } \mathbf{I}_{\mathbf{in}}(x,y))$$

- Note the memory access pattern:



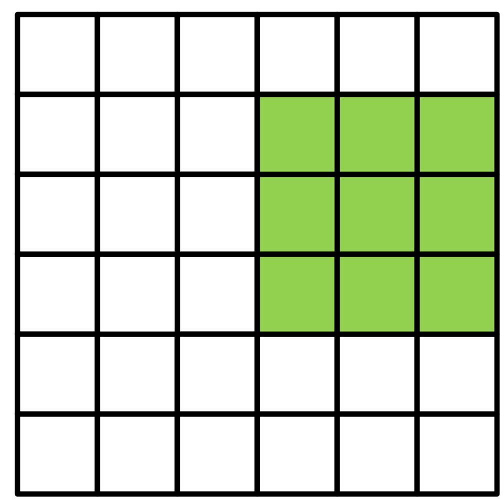
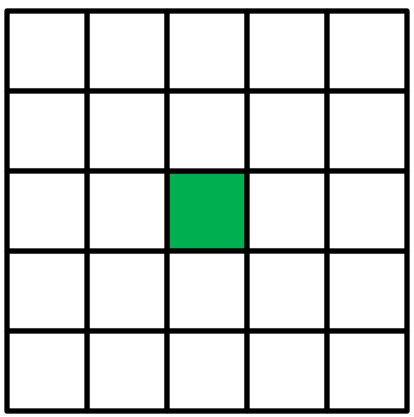


Image Processing with CUDA

- We don't want to have to read *k* elements from global memory for each element we write to in global memory



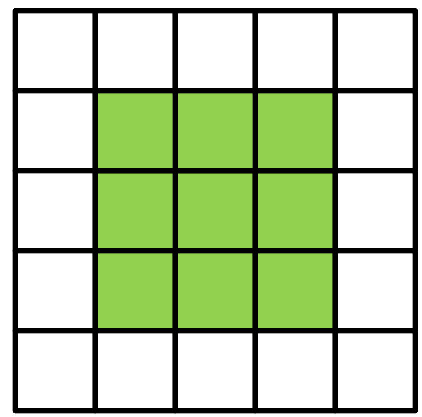


Image Processing with CUDA

- We can use the hardware managed texture cache to help us

Heat transfer example

- Simple two dimensional heat transfer simulation
  - Assume we have a rectangular room divided into a grid
  - In the grid we have some heaters at some fixed temperatures
  - We want to simulate what happened to temperature on each grid cell as time progresses

Heat transfer example

- Simple two dimensional heat transfer simulation

Tnew=Told+$\sum_{neighbors} k(Tneighbor \; - Told)$

Tnew=Told+k(Ttop+Tbottom+Tleft+Tright-4Told)

Heat transfer example

- Simple two dimensional heat transfer simulation

1. Given some grid of input temperatures, copy the temperature of cells with heaters to this grid. To make sure the temperatures on the heater cells remains constant. copy_const_kernel()
2. Given the input temp grid, compute the output temperatures based on prev equation
3. Swap input and output buffers in preparation for next step

Heat transfer example

```
//copy the temperature of cells with heaters into
this
//output grid from prev. time step, overwriting
//calculated values. So cells with heaters remain
// at constant temperatures
__global__ void copy_const_kernel( float *iptr ,
                                    const float
*cptr) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if (cptr[offset] != 0)
        iptr[offset] = cptr[offset];
}
```

Heat transfer example

```
__global__ void blend_kernel( float *outSrc,const float *inSrc
) {
      int x = threadIdx.x + blockIdx.x * blockDim.x;
     int y = threadIdx.y + blockIdx.y * blockDim.y;
     int offset = x + y * blockDim.x * gridDim.x;

     int left = offset - 1;
     int right = offset + 1;
     if (x == 0)   left++;
     if (x == DIM-1) right--;
     int top = offset - DIM;
     int bottom = offset + DIM;
     if (y == 0)   top += DIM;
     if (y == DIM-1) bottom -= DIM;

     outScr[offset] = inScr[offset] + SPEED *(inScr[top] +
inScr[bottom] + inScr[left] + inScr[right] - 4 *
inScr[offset]);
}
```

Heat transfer example. Animation

```
#define DIM 1024
#define PI 3.1415926535897932f
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED   0.25f

// globals needed by the update routine
struct DataBlock {
    unsigned char   *output_bitmap;
    float           *dev_inSrc;
    float           *dev_outSrc;
    float           *dev_constSrc;
    CPUAnimBitmap  *bitmap;

    cudaEvent_t     start, stop;
    float           totalTime;
    float           frames;
};
```

Heat transfer example. Animation

```
void anim_gpu( DataBlock *d, int ticks ) {
    HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
    dim3    blocks(DIM/16,DIM/16);
    dim3    threads(16,16);
    CPUAnimBitmap  *bitmap = d->bitmap;

    for (int i=0; i<40; i++) {
   copy_const_kernel<<<blocks,threads>>>(d->dev_inStr, d->dev_constScr);
   blend_kernel<<<blocks,threads>>>(d->dev_outSrc, d->dev_constSrc);
   swap(d->dev_inSrc, d->dev_outSrc);
    }
    float_to_color<<<blocks,threads>>>( d->output_bitmap,
                                    d->dev_inSrc );
    HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(), d->output_bitmap,
                              bitmap->image_size(), cudaMemcpyDeviceToHost )
);
    HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    d->start, d->stop ) );
    d->totalTime += elapsedTime;
    ++d->frames;
    printf( "Average Time per frame:  %3.1f ms\n",
            d->totalTime/d->frames  );
}
```

Heat transfer example. Animation

```
// clean up memory allocated on the GPU
void anim_exit( DataBlock *d ) {
    HANDLE_ERROR( cudaFree( d->dev_inSrc ) );
    HANDLE_ERROR( cudaFree( d->dev_outSrc ) );
    HANDLE_ERROR( cudaFree( d->dev_constSrc ) );

    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}
```

Heat transfer example. Main

```
int main( void ) {
    DataBlock   data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );

    int imageSize = bitmap.image_size();

    HANDLE_ERROR( cudaMalloc( (void**)&data.output_bitmap,
   imageSize ) );

    // assume float == 4 chars in size (ie rgba)
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc, imageSize
   ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,imageSize
   ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
   imageSize ) );
```

Heat transfer example. Main

```
  // intialize the constant data
  float *temp = (float*)malloc( imageSize );
  for (int i=0; i<DIM*DIM; i++) {
      temp[i] = 0;
      int x = i % DIM;
      int y = i / DIM;
      if ((x>300) && (x<600) && (y>310) && (y<601))
          temp[i] = MAX_TEMP;
  }
  temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
  temp[DIM*700+100] = MIN_TEMP;
  temp[DIM*300+300] = MIN_TEMP;
  temp[DIM*200+700] = MIN_TEMP;
  for (int y=800; y<900; y++) {
      for (int x=400; x<500; x++) {
          temp[x+y*DIM] = MIN_TEMP;
      }
  }
  HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,
imageSize, cudaMemcpyHostToDevice ) );
```

Heat transfer example. Main

```
  // initialize the input data
    for (int y=800; y<DIM; y++) {
        for (int x=0; x<200; x++) {
            temp[x+y*DIM] = MAX_TEMP;
        }
    }
    HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                              imageSize,
                              cudaMemcpyHostToDevice ) );
    free( temp );

    bitmap.anim_and_exit( (void (*)(void*,int))anim_gpu,
                           (void (*)(void*))anim_exit );
}
```

Heat transfer example. Using Texture Memory

//Declare inputs in the texture references

// these exist on the GPU side
texture<float> texConstSrc;
texture<float> texIn;
texture<float> texOut;
We need to bind now the references to the memory buffer using CudaBindTexture():

We will use the buffer as texture

we will name that texture

```
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc, imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc, imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc, imageSize ) );

HANDLE_ERROR(cudaBindTexture(NULL,texConstSrc,data.dev_constSrc,imageSize))
HANDLE_ERROR(cudaBindTexture( NULL, texIn,data.dev_inSrc,imageSize ) );
HANDLE_ERROR(cudaBindTexture( NULL, texOut, data.dev_outSrc,imageSize ) );
```

Heat transfer example. Using Texture Memory

//Declare inputs in the texture references

Need to use text1fetch() when reading from memory

Since textures reference must be declared globally we can no longer pass the input and output buffers as parameters to the function so we will pass a flag dstOut which indicates witch buffer to use

Heat transfer example

```
__global__ void copy_const_kernel( float *iptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex1Dfetch(texConstSrc,offset);
    if (c != 0)
        iptr[offset] = c;
}
```

Heat transfer example

```
__global__ void blend_kernel( float *dst,
                              bool dstOut ) {
  // map from threadIdx/BlockIdx to pixel position
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int offset = x + y * blockDim.x * gridDim.x;

  int left = offset - 1;
  int right = offset + 1;
  if (x == 0)   left++;
  if (x == DIM-1) right--;

  int top = offset - DIM;
  int bottom = offset + DIM;
  if (y == 0)   top += DIM;
  if (y == DIM-1) bottom -= DIM;

  float   t, l, c, r, b;
```

Heat transfer example

```
  if (dstOut) {
      t = tex1Dfetch(texIn,top);
      l = tex1Dfetch(texIn,left);
      c = tex1Dfetch(texIn,offset);
      r = tex1Dfetch(texIn,right);
      b = tex1Dfetch(texIn,bottom);

  } else {
      t = tex1Dfetch(texOut,top);
      l = tex1Dfetch(texOut,left);
      c = tex1Dfetch(texOut,offset);
      r = tex1Dfetch(texOut,right);
      b = tex1Dfetch(texOut,bottom);
  }
  dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

Heat transfer example. Animation

```
#define DIM 1024
#define PI 3.1415926535897932f
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED   0.25f

// these exist on the GPU side
texture<float>  texConstSrc;
texture<float>  texIn;
texture<float>  texOut;

// globals needed by the update routine
struct DataBlock {
    unsigned char   *output_bitmap;
    float           *dev_inSrc;
    float           *dev_outSrc;
    float           *dev_constSrc;
    CPUAnimBitmap  *bitmap;

    cudaEvent_t     start, stop;
    float           totalTime;
    float           frames;
};
```

Heat transfer example. Animation

```
void anim_gpu( DataBlock *d, int ticks ) {
    HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
    dim3    blocks(DIM/16,DIM/16);
    dim3    threads(16,16);
    CPUAnimBitmap  *bitmap = d->bitmap;

    // since tex is global and bound, we have to use a flag to
    // select which is in/out per iteration
    volatile bool dstOut = true;
    for (int i=0; i<90; i++) {
        float   *in, *out;
        if (dstOut) {
            in  = d->dev_inSrc;
            out = d->dev_outSrc;
        } else {
            out = d->dev_inSrc;
            in  = d->dev_outSrc;
        }
        copy_const_kernel<<<blocks,threads>>>( in );
        blend_kernel<<<blocks,threads>>>( out, dstOut );
        dstOut = !dstOut;
    }
    float_to_color<<<blocks,threads>>>( d->output_bitmap,d->dev_inSrc );
```

Heat transfer example. Animation

```
    HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(),
                              d->output_bitmap,
                              bitmap->image_size(),
                              cudaMemcpyDeviceToHost ) );

    HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        d->start, d->stop ) );
    d->totalTime += elapsedTime;
    ++d->frames;
    printf( "Average Time per frame:  %3.1f ms\n",
            d->totalTime/d->frames  );
}
```

Heat transfer example. Animation

```
void anim_exit( DataBlock *d ) {
    cudaUnbindTexture( texIn );
    cudaUnbindTexture( texOut );
    cudaUnbindTexture( texConstSrc );
    HANDLE_ERROR( cudaFree( d->dev_inSrc ) );
    HANDLE_ERROR( cudaFree( d->dev_outSrc ) );
    HANDLE_ERROR( cudaFree( d->dev_constSrc ) );

    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}
```

Heat transfer example. Main

```
int main( void ) {
    DataBlock   data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );

    int imageSize = bitmap.image_size();

    HANDLE_ERROR( cudaMalloc( (void**)&data.output_bitmap,    imageSize ) );

    // assume float == 4 chars in size (ie rgba)
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc, imageSize ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,imageSize ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc, imageSize ) );

    HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc, data.dev_constSrc,
                                   imageSize ) );

    HANDLE_ERROR( cudaBindTexture( NULL, texIn,data.dev_inSrc, imageSize ) );
```

Heat transfer example. Main

```
HANDLE_ERROR( cudaBindTexture( NULL, texOut,data.dev_outSrc,  imageSize ) );

// intialize the constant data
float *temp = (float*)malloc( imageSize );
for (int i=0; i<DIM*DIM; i++) {
    temp[i] = 0;
    int x = i % DIM;
    int y = i / DIM;
    if ((x>300) && (x<600) && (y>310) && (y<601))
        temp[i] = MAX_TEMP;
}
temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
temp[DIM*700+100] = MIN_TEMP;
temp[DIM*300+300] = MIN_TEMP;
temp[DIM*200+700] = MIN_TEMP;
for (int y=800; y<900; y++) {
    for (int x=400; x<500; x++) {
        temp[x+y*DIM] = MIN_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp, imageSize,
                          cudaMemcpyHostToDevice ) );
```

Heat transfer example. Main

```
    // initialize the input data
    for (int y=800; y<DIM; y++) {
        for (int x=0; x<200; x++) {
            temp[x+y*DIM] = MAX_TEMP;
        }
    }
    HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                              imageSize,
                              cudaMemcpyHostToDevice ) );
    free( temp );

    bitmap.anim_and_exit( (void (*)(void*,int))anim_gpu,
                          (void (*)(void*))anim_exit );
}
```

Heat transfer example. Using Two-Dimensional Texture Memory

```
__global__ void copy_const_kernel( float *iptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex2D(texConstSrc,x,y);
    if (c != 0)
        iptr[offset] = c;
}
```

Heat transfer example

```
__global__ void blend_kernel( float *dst,     bool dstOut ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0)   left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0)   top += DIM;
    if (y == DIM-1) bottom -= DIM;
```

Heat transfer example

```
    float   t, l, c, r, b;
    if (dstOut) {
        t = tex2D(texIn,x,y-1);
        l = tex2D(texIn,x-1,y);
        c = tex2D(texIn,x,y);
        r = tex2D(texIn,x+1,y);
        b = tex2D(texIn,x,y+1);
    } else {
        t = tex2D(texOut,x,y-1);
        l = tex2D(texOut,x-1,y);
        c = tex2D(texOut,x,y);
        r = tex2D(texOut,x+1,y);
        b = tex2D(texOut,x,y+1);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

Heat transfer example

```
// assume float == 4 chars in size (ie rgba)
   HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc, imageSize ) );
   HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc, imageSize ) );
   HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc, imageSize ) );

   cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
   HANDLE_ERROR( cudaBindTexture2D( NULL, texConstSrc,
                              data.dev_constSrc,
                              desc, DIM, DIM,
                              sizeof(float) * DIM ) );
   HANDLE_ERROR( cudaBindTexture2D( NULL, texIn,
                              data.dev_inSrc,
                              desc, DIM, DIM,
                              sizeof(float) * DIM ) );
   HANDLE_ERROR( cudaBindTexture2D( NULL, texOut,
                              data.dev_outSrc,
                              desc, DIM, DIM,
                              sizeof(float) * DIM ) );
```