Class

Windows and CUDA ： Shu Guo

Program from last time: Constant memory

Windows on CUDA

Reference: NVIDIA CUDA Getting Started Guide for Microsoft Windows

Whiting School has Visual Studio

Cuda 5.5 Installer from NVIDIA

(Make sure BIOS updated from manufacturer)

Update graphics card driver

Visual Studio

Manifest Tool: Embedded Manifest changed to No

Syntax highlighting:
  Text Editor File Extension .cu .cuh Visual C++

To make __global__ available: VAssistX trial version.

Include C/C++ Directories

Need to include <cuda_runtime.h> and <device.

Constant Memory

```
__constant__ float g;
__constant__ float m;

__global__ void multkernel(float *a, int N){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  if(x<N)
    { a[x]=m*a[x]*g; //energy replaces z
    }
}
////////////// in main


  cudaMemcpyToSymbol(g, &g_h, sizeof(float),0,cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(m, &m_h, sizeof(float));
```

Note: constant memory only accessible on the device

CUDA Memory Types

Global memory

Slow and uncached, all threads

Texture memory (read only)

Cache optimized for 2D access, all threads

Constant memory (read only)

Slow, cached, all threads

Shared memory

Fast, bank conflicts; limited; threads in block

Registers

Fast, only for one thread

Local memory

For what doesn’t fit in registers, slow but cached, one thread

Register memory

Variables defined in kernel; associated with each thread; fast

deviceQueryDrv: my laptop:
    64k registers per SM; max 1024 threads/block; 2048 max threads per SM;

If 1024 threads/block, then 2 blocks in SM at one time (based on max threads); 32 registers per thread.

If 256 threads/block, and one block in SM, 64k (=65536)/256= 256 registers per thread

If 256 threads/block, and eight blocks in SM, still 32 registers per thread, but if you had 64 registers per thread, then only four blocks could be in SM—too many registers; half of SM is empty

Shared Memory

Talked about before: Class 11

Statically or dynamically allocate

```
__global__ void MyFunc(float*) // __device__ or __global__ function
{
    extern __shared__ float shMemArray[];  //dynamically allocated
    // Size of shMemArray determined through the execution configuration
    // You can use shMemArray as you wish here…
}

// invoke like this
MyFunc<<< gridDim, blockDim, shMem >>>(parameter);
                                       ^
```

shMem is the amount of shared memory to use.

Shared Memory

Remember: subject to race conditions

Use __syncthreads(); to have all threads catch up

Shared Memory

For parallel machine, to allow many threads to access memory at the same time, there are separate memory bank to allow access

There are 32 banks for each streaming multiprocessor

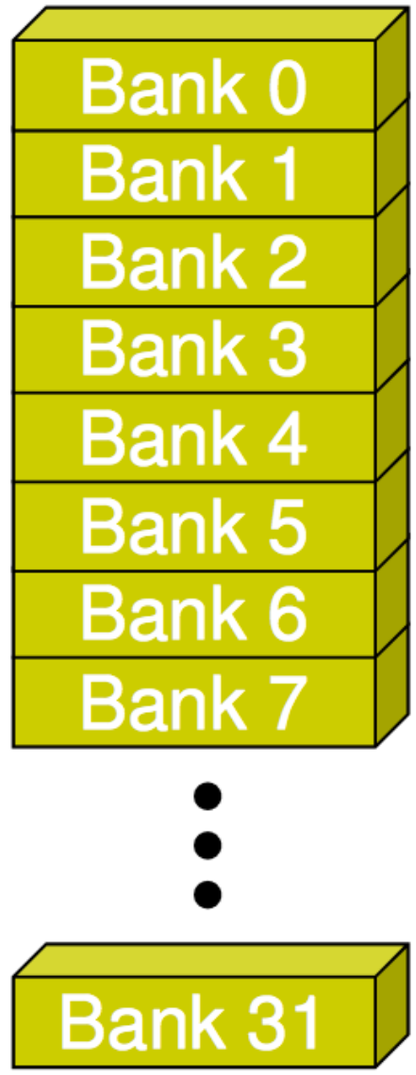
Maximum of 48 kB

When reading in 4 byte words*, 32 threads (1 warp) can read data simultaneously (128 bytes)

Bank conflict: when two threads try to read different words in the same bank. Serializes the process

No conflict if all threads read different banks or if all threads read same bank.

Worst case: all 32 threads read 32 words in same bank.



---

*4 bytes= 1 single precision float or one 32 bit integer

Successive 32-bit word addresses are assigned to successive banks

Bank you work with = (address of offset) % 32
This is because Fermi has 32 banks

Example: 1D shared mem array, myShMem, of 1024 floats (32 rows of 32)

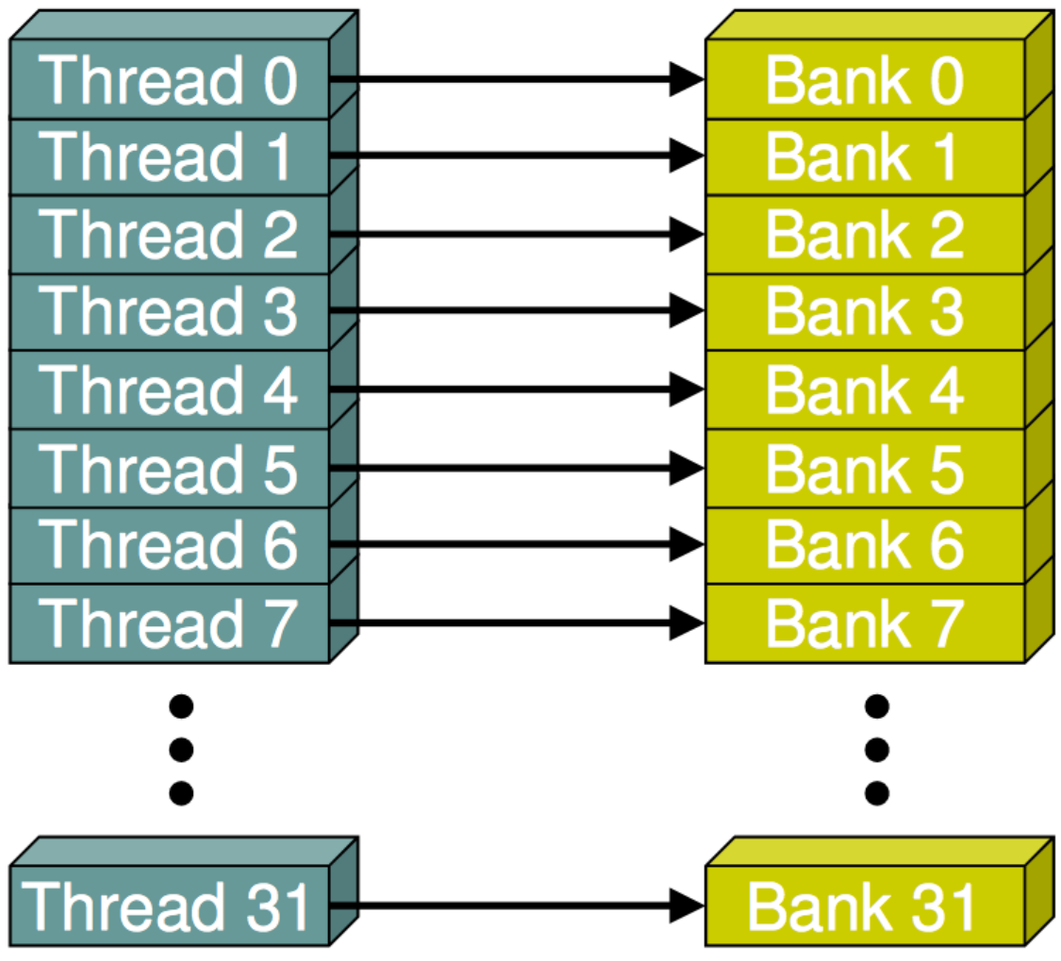
mySh**Mem**[4]: accesses bank #4 (physically, the fifth one – first row)
mySh**Mem**[31]: accesses bank #31 (physically, the last one – first row)
mySh**Mem**[50]: access bank #18 (physically, the 19th one – second row)
mySh**Mem**[128]: access bank #0 (physically, the first one – fifth row)
mySh**Mem**[178]: access bank #18 (physically, the 19th one – sixth row)

NOTE: If, for instance, the third thread in a warp accesses myShMem[50] and the eighth thread in the warp accesses myShMem[178], then you have a two-way bank conflict and the two transactions get serialized.

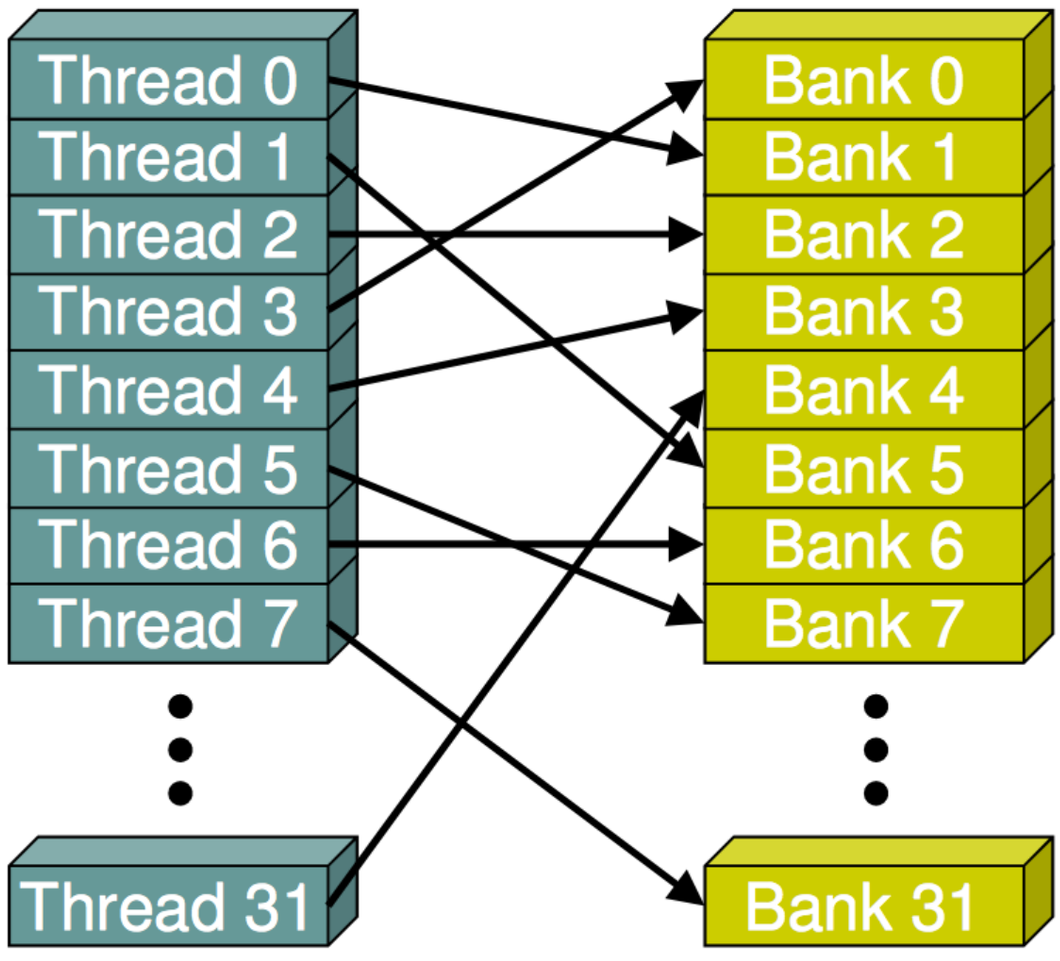
IMPORTANT: There is no such thing as “bank conflicts” between threads belonging to different warps

Bank Addressing

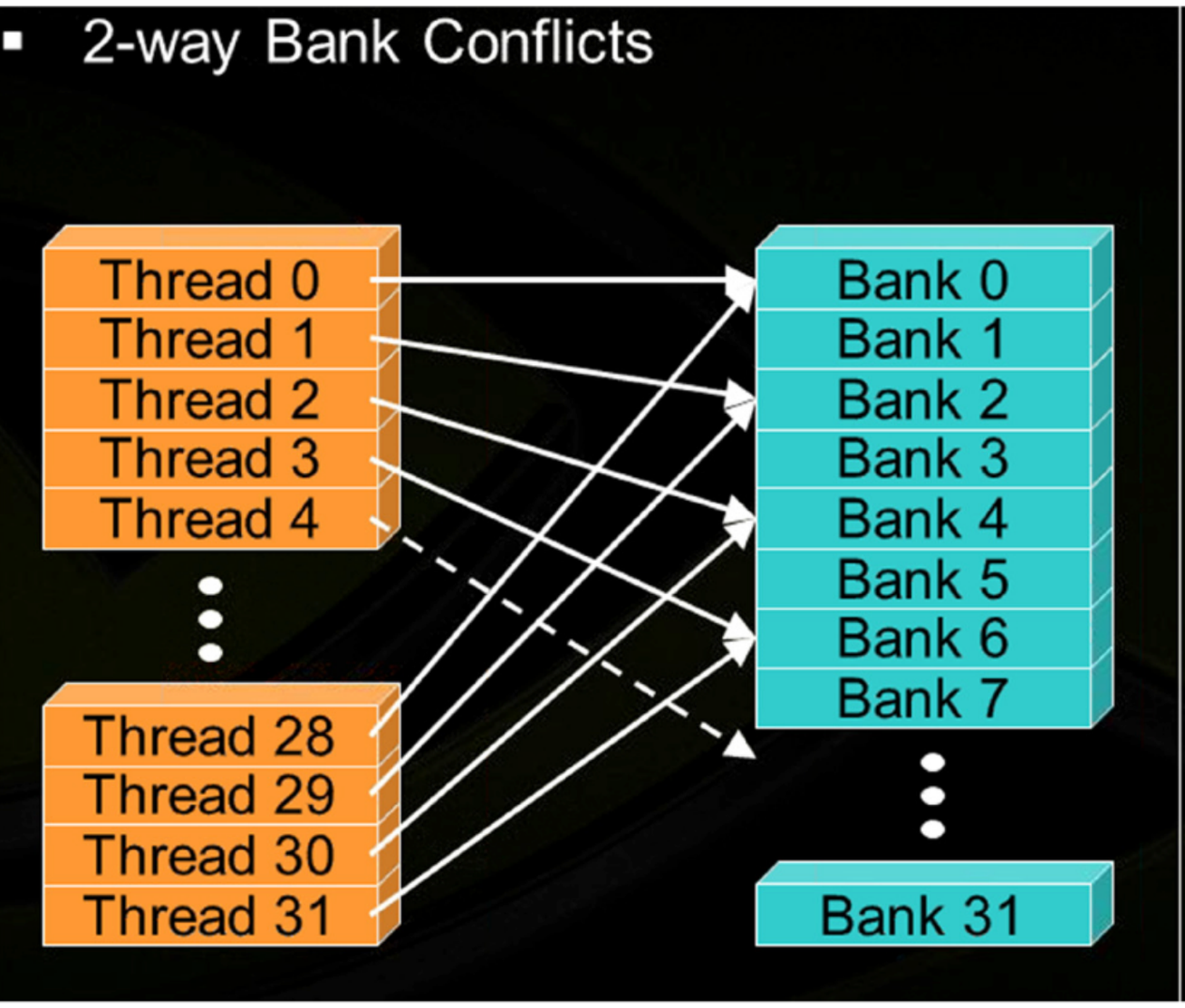
- No Bank Conflicts
  - Linear addressing stride == 1

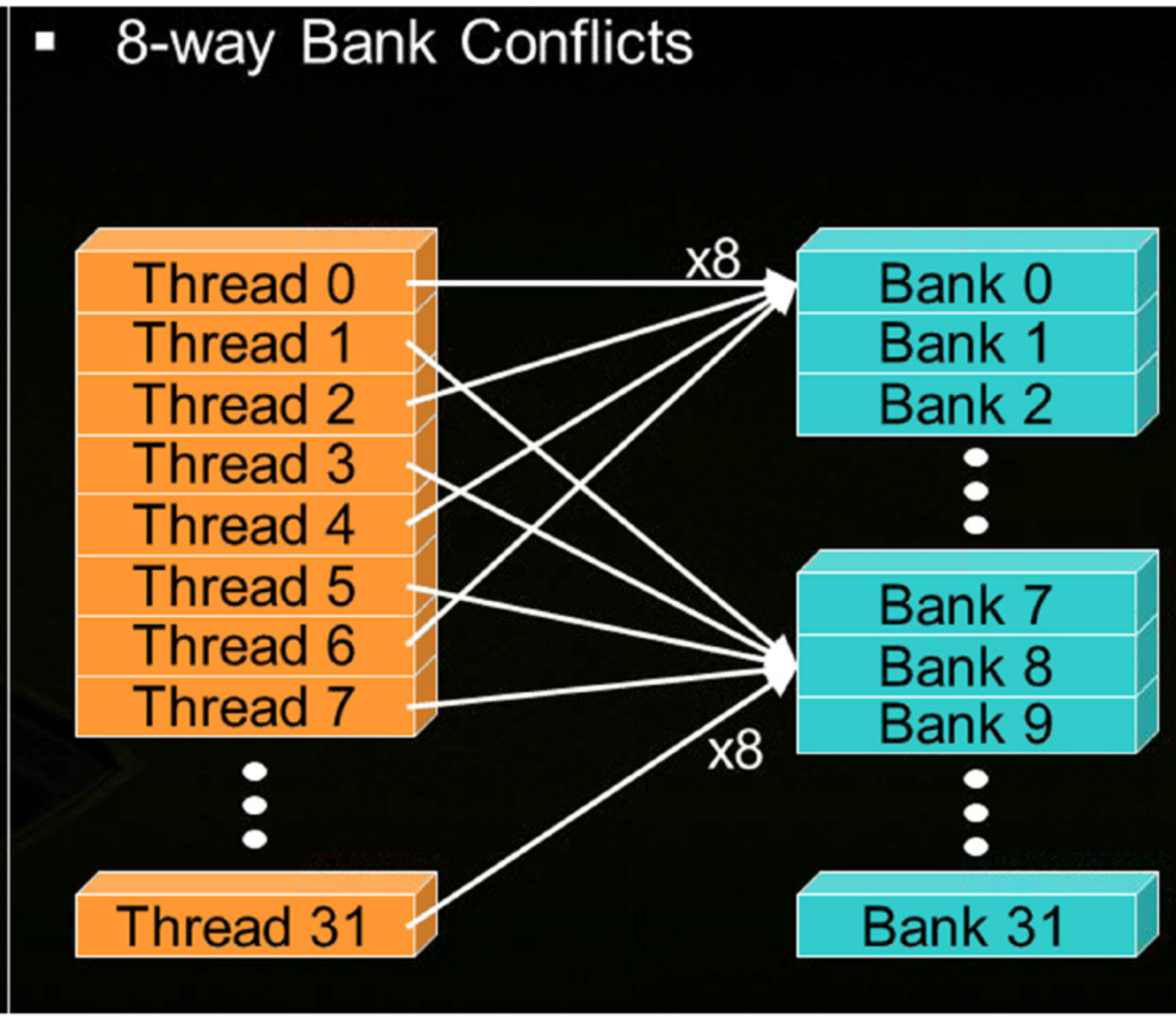


- No Bank Conflicts
  - Random 1:1 Permutation



Bank Conflicts



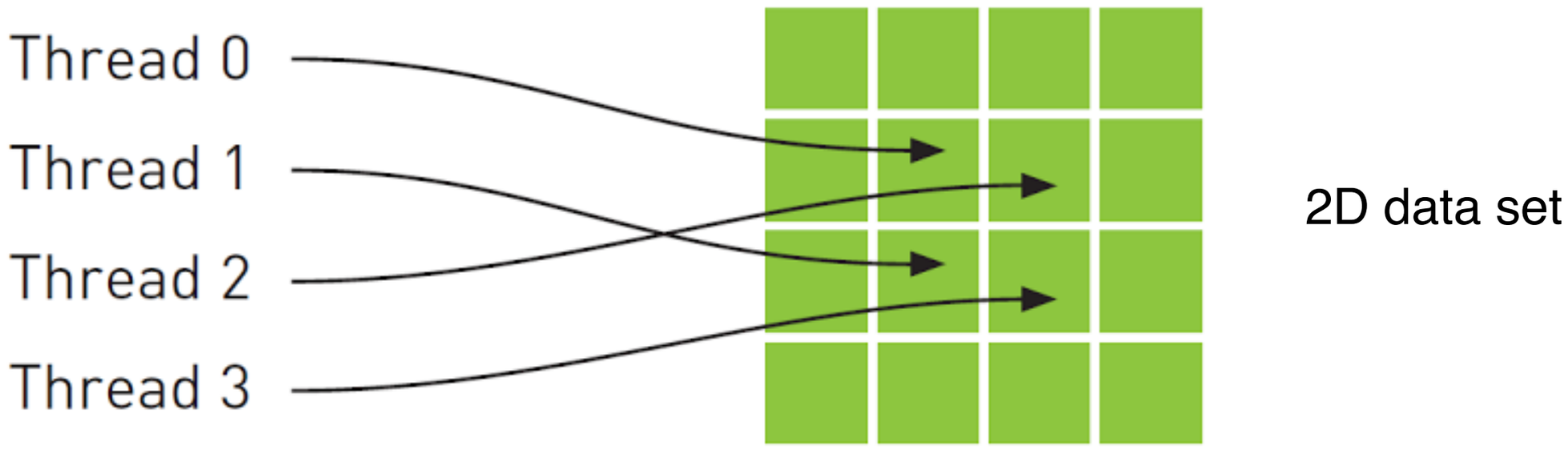


Texture Memory

Texture memory is **read-only** memory (like constant memory), which comes from the graphical roots of the graphics card. It is available to >all< threads, like global memory. Optimized for 2D, but small

Texture memory is cached on the GPU—faster access than global memory

Texture memory is ideal for applications where threads require nearby data:



Accessing Texture Memory

A variety of texture functions are available to save data to texture memory and to read the data for use in a computation.

Writing texture memory

Declare texture memory: *texture(type, dim, readmode) texture_ref*

texture_ref is the handle to the texture memory
readmode: cudaReadModeElementType—no conversions
cudaReadModeNormalizedFloat—[-1, 1]

Bind texture memory to the texture_reference

*cudaBindtexture (size *t offset, const struct texture<T, dim, readMode> & tex , const void * devptr, size_t size= UINT_MAX) ;*

Reading texture memory (texture fetch)

fetch texture memory for use

Unbind texture memory

Linear Memory Texture

```
Texture< float, 1, cudaReadModeNormalizedFloat> texRef;

in main()

unsigned short *d_A = 0;
cudaMalloc (&d_A, numBytes);

cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

cudaBindTexture(NULL, texRef, d_A, size /*in bytes*/);
```

Access:

```
unsigned short value = tex1Dfetch(texRef,10); //returns element 10
```

```
texture<float, 1, cudaReadModeElementType> texRef;

__global__ void readTexels(int n, float *d_out)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if(idx < n) {
    float x = tex1D(texRef, float(idx));
    d_out[idx] = x;}
}
#define NUM_THREADS 256

int main()
{
  int N = 10; // 10 is illustrative and should be larger in practice
  int nBlocks = N/NUM_THREADS + ((N % NUM_THREADS)?1:0);
  float *d_out;

  // allocate space on the device for the results
  cudaMalloc((void**)&d_out, sizeof(float) * N);

  // allocate space on the host for the results
  float *h_out = (float*)malloc(sizeof(float)*N);
```

```
// data fill array with increasing values
float data[N];
for (int i = 0; i <  N; i++) {
    data[i] = float(i);
    printf("%f\n",data[i]);
    }
printf("\n");// create a CUDA array on the device
cudaArray* cuArray;
cudaMallocArray (&cuArray, &texRef.channelDesc, N, 1);
cudaCheckError("cudaMalloc") ;
cudaMemcpyToArray(cuArray, 0, 0, data, sizeof(float)*N, cudaMemcpyHostToDevice);
 cudaCheckError("MemcpyToArray") ;

// bind a texture to the CUDA array
cudaBindTextureToArray (texRef, cuArray);
cudaCheckError("TextureToArray") ;

// host side settable texture attributes
texRef.normalized = false;
texRef.filterMode = cudaFilterModeLinear; //linear interpolation
                                          //Point is nearest neighbor
```

```
// read texels from texture
  readTexels<<<nBlocks, NUM_THREADS>>>(N, d_out);
   cudaCheckError("kernel") ;

  // copy texels to host
  cudaMemcpy(h_out, d_out, sizeof(float)*N, cudaMemcpyDeviceToHost);
   cudaCheckError("DeviceToHost") ;
 // look at them
  for (int i = 0; i < N; i++) {
    printf("%f\n",h_out[i]);
  }

  free(h_out);

  cudaFree(d_out);
  cudaFreeArray(cuArray);
  cudaUnbindTexture(texRef);
  cudaCheckError("cuda free operations");
}
```

2D Texture Memory

CudaArray

Texture<float, 2, cudaReadModeElementType> texRef;
^ dimensions

// set up the CUDA array
cudaChannelFormatDesc cf = cudaCreateChannelDesc<float>();
cudaArray *texArray = 0;
cudaMallocArray(&texArray, &cf, dimX, dimY);
cudaMempcyToArray(texArray, 0,0, hA, numBytes, cudaMemcpyHostToDevice);

Type tex2D(texRef, float x, float y);
cudaUnbindTexture (texRef)

Texture Memory

Address modes:

Border	Invalid texture coordinates return zero
Clamp	Clamp texture coordinates to closest valid coordinate
Wrap	Texture is repeated infinitely
Mirror	Texture is repeated and mirrored at each boundary

Filter modes:

Point	Return value of pixel nearest to point location
Linear	Linear interpolation between nearest elements

In **Elements** not bytes

- In CUDA Arrays:
  - 1D: 8K
  - 2D: 64K x 32K
  - 3D: 2K x 2K x 2K
- If in linear memory: 2^27
  - That's 128M elements

Textures are read-only

- Within a kernel

A kernel can produce an array

- Cannot write CUDA Arrays

Linear Memory can be copied from CUDA Arrays

- cudaMemcpyFromArray()
  - Copies linear memory array to a CudaArray
- cudaMemcpyToArray()
  - Copies CudaArray to linear memory array

Texture Objects

Like C++ objects; new with CUDA 5.0

Also called bindless texture. There is no overhead associated with binding

Texture memory is part of global memory

Linear memory: cudaMalloc()

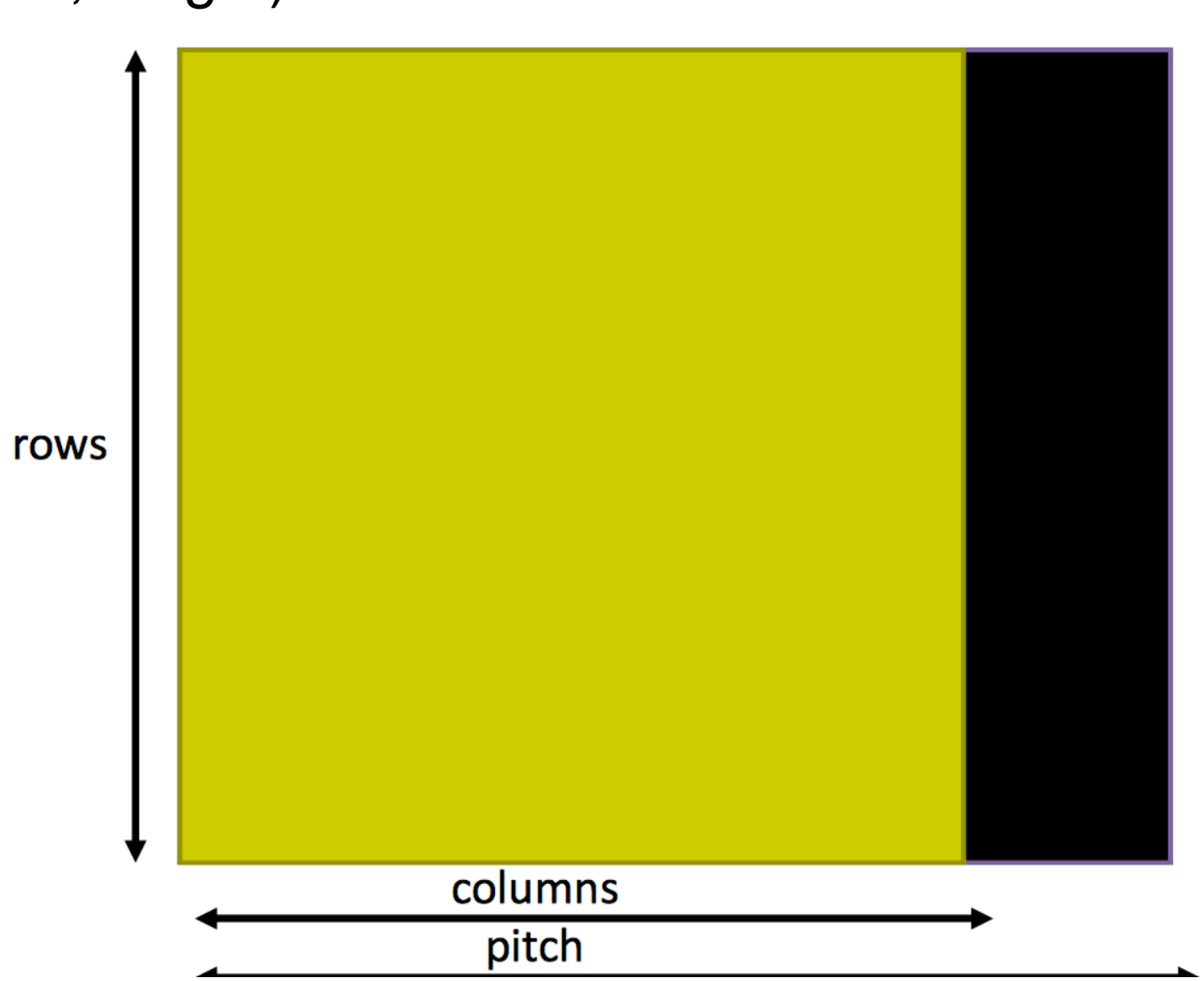
CUDA arrays: cudaMallocArray(); cudaMalloc3D()
Just for texture arrays

2D pitch linear memory: cudaMallocPitch() (better than linear memory)
cudaMallocPitch(&ptr, &pitch, width, height)
for 2D array of width * height
To align memory better (2^n)
CUDA tells you the pitch

Addressing:

row*pitch + column

instead of columns



Radix Sort Example (base 10)

5, 213, 55, 21, 430, 31, 20

Separate by units:

- zeros: 20, 430
- ones: 21, 31
- twos:
- threes: 213
- fours:
- fives: 5, 55

Back together: 20, 430, 21, 31, 213, 5, 55

Separate by tens:

- zeros: 05
- ones: 213
- twos: 20, 21
- threes: 430,
- fours:
- fives: 55

Back together: 5, 213, 20, 21, 430, 55

Example (2)

Separate by hundreds:

- zeros: 005, 020, 021, 055
- ones:
- twos: 213
- threes:
- fours: 430
- fives:

Back together: 5, 20, 21, 55, 213, 430

Use binary for integers.

Advantage: regardless of the length of the array to be sorted, only 32 loops are necessary to cover all bits in a 32 bit integer.

C++

```
void cpu_sort(int * data, int N)
{
    int temp_0[N];
    int temp_1[N];
    for (unsigned int bit = 0; bit<32; bit++)
    {
      int count_0 = 0;
      int count_1 = 0;
      for (int i = 0; i<N; i++) {
         unsigned int d= data[i];
         unsigned int bit_mask= (1 << bit);
         if ((d & bit_mask ) >1)   //bitwise & means 0 if 1 and 0, 0 and 0, and 1 if 1 and 1.
              {temp_1[count_1] = d;
               count_1++;
               } else
              {temp_0[count_0] = d;
               count_0++;} }
       // copy back to data; first the zero list
       for (int i=0; i< count_0; i++){
          data[i]=temp_0[i];}
        for (int i=0; i<count_1; i ++){
           data[count_0+i] = temp_1[i]; }
    }
}
```