# MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

SPIE Medical Imaging

## CUDA Computing

Klaus Mueller, Wei Xu, Ziyi Zheng, Fang Xu

Computer Science
Center for Visual Computing
Stony Brook University, NY

Siemens USA Research
Princeton, NJ

## Outline

Goal: To develop medical imaging applications in CUDA environment

- CUDA Hardware
- CUDA Programming API
- CUDA Graphics API
- CUDA Performance



## Setup CUDA

Compute Unified Device Architecture

- Driver, Toolkit and SDK http://www.nvidia.com/object/cuda_get.html

Other resource

- Cuda-dgb beta 2.1
- CUDA occupancy calculator
- Visual studio syntax highlighting
- Template wizard



## CUDA Hardware

**CUDA Hardware**

- Host & Device
- CG Model & CUDA Model
- Thread Hierarchy
- Memory Hierarchy

--Know your weapon.

## Hardware Architecture

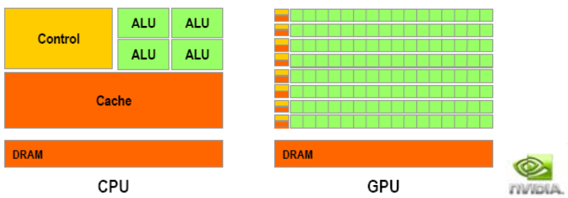
SPIE Medical Imaging

GPU

- Compute-intensive
- Highly data parallel

CUDA

- Expose the parallel capabilities of GPUs.






## Host & Device

SPIE Medical Imaging

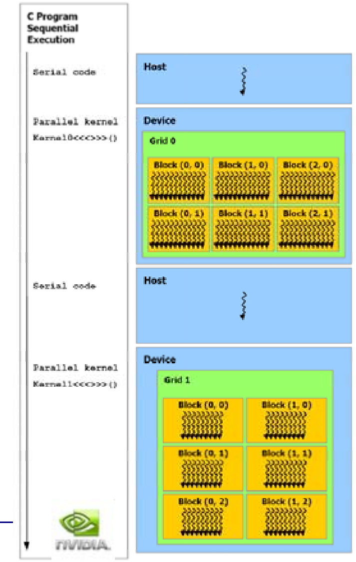
Host (CPU)

- Program flow
- Thread management
- Load GPU programs (kernels)

Device (GPU)

- Load data
- Perform computations

Heterogeneous Programming

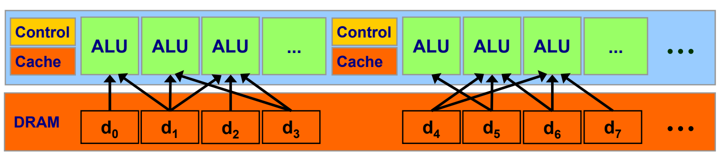




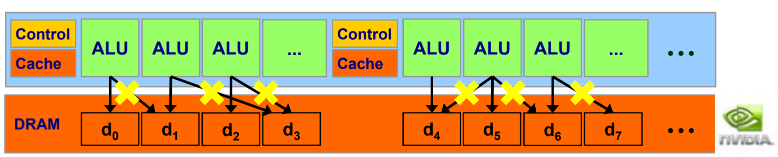

## CG Model

SPIE Medical Imaging

Collect




Can not scatter!



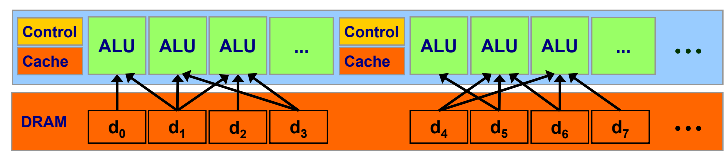

Threads can not cooperate! →Multi-pass render



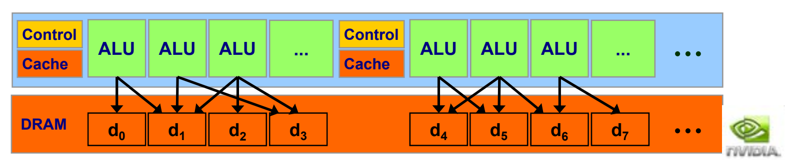
## CUDA Model

SPIE Medical Imaging

Collect




Scatter




Threads can cooperate!

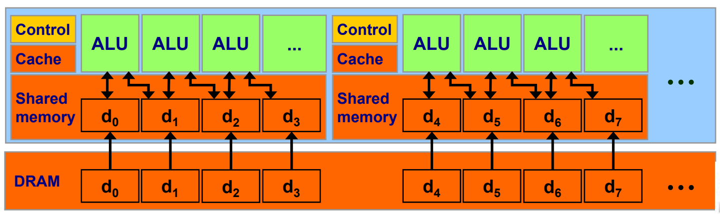
# Decomposition

Can all threads cooperate?

- NO

Cooperate with a smaller batch of threads (block)

- Same multiprocessor
- Shared memory
- Highlight of CUDA computing







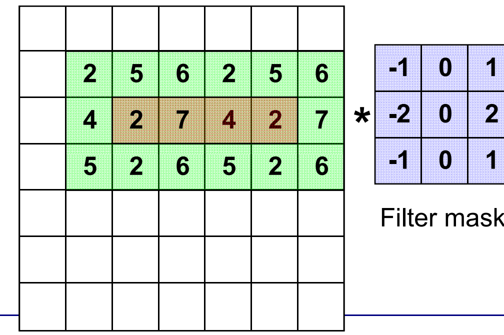

# Decomposition

CUDA Tips From NVidia

1. Decompose program into a sequence of steps ( Grids)
2. Decompose grid into independent parallel blocks ( Thread blocks)
3. Decompose block into cooperating parallel elements (Threads)

Examples

- Adding vector
- Sobel filter




Filter mask

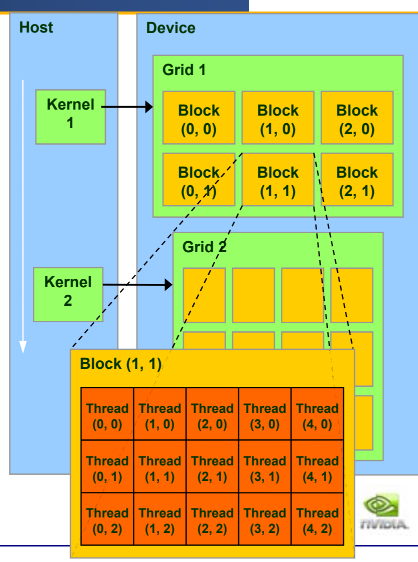




# Thread Hierarchy

Thread, Block & Grid

- Each kernel executes as a batch of threads
- This batch is organized as grid of blocks
- Threads in a block is an array of threads that can cooperate.
- Threads with a same block are executed by one multiprocessor. They share memory and can be synchronized





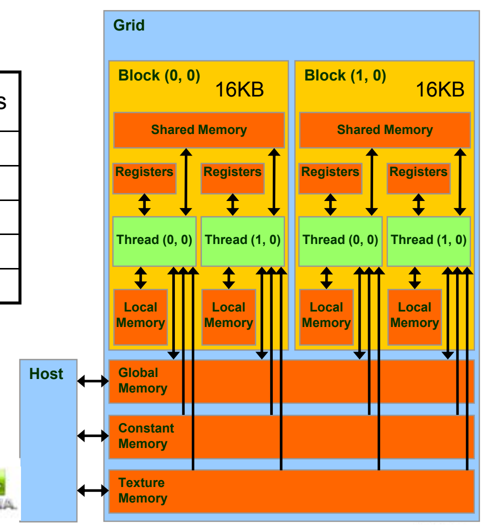



# Memory Model

Memory Hierarchy

| Memory | On-chip | Cached | Access |
|---|---|---|---|
| Local | N | N | RW |
| Shared | Y | NA | RW |
| Global | N | N | RW |
| Constant | N | Y | R |
| Texture | N | Y | R |

- Simply start by using just global memory
- Then optimize
- More about this later



# CUDA vs. CG

| | CUDA | CG |
|---|---|---|
| Application | General purpose | Graphics |
| Program | Kernel | Shader |
| Collect | Yes | Yes |
| Scatter | Yes | No |
| Thread Synchronization | Yes | No |
| Memory | Local<br>Shared<br>Constant<br>Global<br>Texture | Texture |



# CUDA Programming API

## CUDA Programming API

- Functions
- Variables
- Execution
- Memory Management
- Thread Synchronization

--Control the ALU army



# Function Qualifiers

Device Global, & Host

- To specify whether a function executes on the host or device
- __global__ must return void

| Function | Exe on | Call from |
|---|---|---|
| __device__ | GPU | GPU |
| __global__ | GPU | CPU |
| __host__ | CPU | CPU |

```
__global__  void Func (float* parameter);
```



# Variable Qualifiers

Shared, Device & Constant

- To specify the memory location on the device of a variable
- __shared__ and __constant__ are optionally used together with __device__

| Variable | Memory | Scope | Lifetime |
|---|---|---|---|
| __shared__ | Shared | Block | Block |
| __device__ | Global | Grid | Application |
| __constant__ | Constant | Grid | Application |

```
__constant__        float ConstantArrary[16];
__shared__          float SharedArray[16];
__device__           ……
```

# Execution Configuration

<<< Grids, Blocks>>>

- Kernel function must specify the number of threads for each call (dim3)

<<< Grids, Blocks, Shared>>>

- It can specify the number of bytes in shared memory that is dynamically allocated per block (size_t)

```
dim3 dimBlock(8, 8, 2);
dim3 dimGrid(10, 10, 1);
KernelFunc<<<dimGrid, dimBlock>>>(…);


KernelFunc<<< 100, 128 >>>(...);
```



# Device Side Parameters

Threads get parameters from execution configuration

- Dimensions of the grid in blocks

```
dim3 gridDim;
```

- Dimensions of the block in threads

```
dim3 blockDim;
```

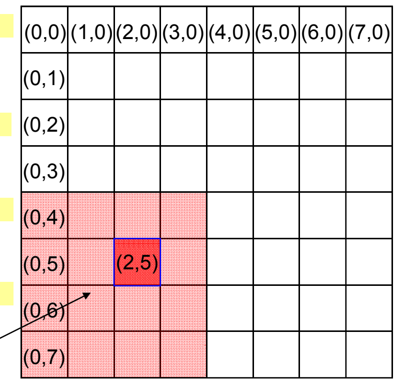
- Block index within the grid

```
dim3 blockIdx;
```

- Thread index within the block

```
dim3 threadIdx;
```

```
blockIdx.x *blockDim.x + threadIdx.x;
blockIdx.y *blockDim.y + threadIdx.y;
```



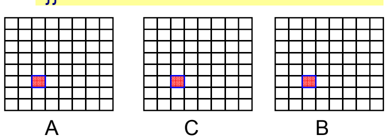

<<<dim3(2,2), dim3(4,4)>>>



# Example 1: Adding Matrix

C++

```
void AddMatrix (int *A, int *B, int *C,
                int w,  int d)
{   for ( int j = 0; j < d; j++)
      for( int i = 0; i < w; i++)
        { int index=j*w+i;
          C[index] = A[index] + B[index];
}}
```

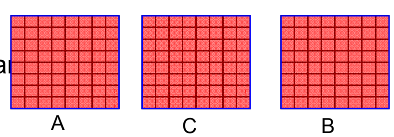



CUDA

```
AddMatrix<<<1,dim3(w,d,1)>>>(A,B,C);
```

```
__global__ void AddMatrix (int *A,int *B,int *C)
{
      int i = threadIdx.x;
      int j = threadIdx.y;
      int id = j*blockDim.x+i;
      C[id]=A[id]+B[id];
}
```

- A simple parallel computing kernel
- It rewrites "for" loops as execution para
- Block ←→ multiprocessor (w*d<=512)






# Memory Management

Device memory allocation

```
cudaMalloc(), cudaFree()
```

Memory copy

```
cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()
```

Memory addressing

```
cudaGetSymbolAddress()
```

# Example 1: Adding Matrix



```
void * a,*b,*c;
cudaMalloc((void**)&a, w*d*sizeof(int));
cudaMalloc((void**)&b, w*d*sizeof(int));
cudaMalloc((void**)&c, w*d*sizeof(int));
…//load data into a, b;
cudaMemcpy(a, w*d*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(b, w*d*sizeof(int),cudaMemcpyHostToDevice);
dim3 dimBlock(BLOCKSIZE , BLOCKSIZE );
dim3 dimGrid( w/dimBlock.x, d/dimBlock.y );
AddMatrix <<<dimGrid, dimBlock>>> (a,b,c,w );
cudaMemcpy(c, w*d*sizeof(int),cudaMemcpyDeviceToHost);
cudaFree(a); cudaFree(b); cudaFree(c);
```

Allocate global memory (read and write)

Launch kernel

Assume multiple of BLOCKSIZE

Copy result

Free memory

```
__global__ void AddMatrix (int *A, int *B, int *C, int w)
{
        int i = blockIdx.x *blockDim.x + threadIdx.x;
        int j = blockIdx.y *blockDim.y + threadIdx.y;
        int id = j* w +i;
        C[id]=A[id]+B[id];
}
```

Define kernel

Get threadIdx, blockIdx here



# Parallel and Synchronize



Threads execute in asynchronous manner in general

- Threads with one block share memory and can synchronize

```
void __syncthreads();
```

- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory
- No such function in CG. CG can do this by multi-pass render



# CUDA Graphics API



## CUDA Graphics API

- Texture (1D 2D 3D)
- PBO (Pixel Buffer Object)
- FBO (Frame Buffer Object)

--Go back to our goal



# Texture Memory Advantage



Texture fetch versus global or constant memory read

- Cached, better performance if fetch with locality
- Not subject to the memory coalescing constraint for global and constant memory
- 2D address
- Filtering
- Normalized coordinates
- Handling boundary address

# Texture

1. Declaring texture reference, format and cudaArray

```
texture<Type, Dim, ReadMode> texRef; cudaArray* cu_array;
cudaChannelFormatDesc cudaCreateChannelDesc<T>();
```

| Feature | Use | Caveat |
|---|---|---|
| Filtering | Fast, low-precision interpolation | Only valid if the texture reference returns float-point data |
| Normalized coordinates | Resolution-independent | |
| Addressing modes | Handling boundary | Normalized texture only |

2. Memory management

```
cudaMallocArray(), cudaFreeArray(), cudaMemcpyToArray()
```

3. Bind/Unbind texture before/after texture fetching

```
cudaBindTextureToArray(), cudaUnbindTexture()
```



# Example 2: Texture Example

```
texture<unsigned char, 2, cudaReadModeElementType> texr;
…

cudaChannelFormatDesc chDesc = cudaCreateChannelDesc<unsigned char>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &chDesc, w, h);
cudaMemcpyToArray( cuArray, 0, 0, input, data_size, cudaMemcpyHostToDevice );

cudaBindTextureToArray(texr, cuArray);
 …
//launch kernel. Inside kernel,  use tex2d(texr, idxX, idxY );
…//read result from device
cudaUnbindTexture(texr );
cudaFreeArray( cuArray );
cudaFree( data );
```

Bind texture before reference

Unbind texture after reference

Free cudaArray and memory



# OPENGL PBO/FBO

Mapping PBO/FBO from OPENGL into CUDA

1. Register

```
cudaError_t cudaGLRegisterBufferObject(GLuint bufferObj);
```

2. Map

```
cudaError_t cudaMapBufferObject(void** devPtr,unsigned int* size,
            GLuint bufferObj);
```

3. UnMap

```
cudaError_t cudaGLUnmapBufferObject(GLuint bufferObj);
```

4. Unregister

```
cudaError_t cudaGLUnregisterBufferObject(GLuint bufferObj);
```



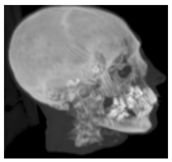
# Example 3: PBO in OPENGL

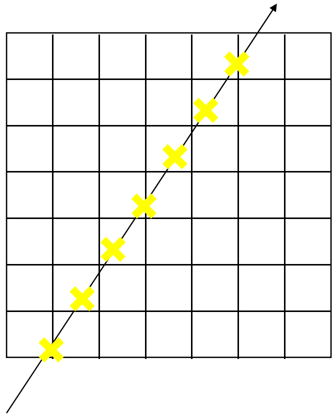
```
GLuint pbo;
glGenBuffersARB(1, &pbo);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB, width*height*sizeof(GLubyte)*4, 0,
    GL_STREAM_DRAW_ARB);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
cudaGLRegisterBufferObject(pbo);
....
cudaGLMapBufferObject((void**)&d_output, pbo));
..//lanch kernel
..
cudaGLUnmapBufferObject(pbo) ;
cuGLUnregisterBufferObject(pbo);
```

# Example 4: X Ray Rendering

SPIE Medical Imaging

- Direct volume rendering
- 3D texture only
- Pixel Buffer Object (PBO)
- Ray-casting
- Bounding volume
- No shared memory. Similar to CG.







# Example 4: X Ray Rendering

SPIE Medical Imaging

```
// calculate eye ray in world space
// find intersection with box
……….
float4 sum = make_float4(0.0f);
float t = far;
 for(int i=0; i<maxSteps; i++) {
   float3 pos = eyeRay.o + eyeRay.d*t;
   pos = pos*0.5f+0.5f;
   float sample = tex3D(tex, pos.x, pos.y, pos.z);
  sample *= transferScale;
 float4 col = make_float4(sample,sample,sample,1.0);
 sum+=col;
   t -= tstep;
   if (t < near) break;}
```

March along ray from back to front,

Map position to [0, 1] coordinates

Read from 3D texture

Accumulating color



# CUDA Performance

SPIE Medical Imaging

## CUDA Performance

- Instruction Optimization
- Global Memory Coalescing
- Shared Memory Bank Conflicts

--Save time, save lives



# Instruction Optimization

SPIE Medical Imaging

Compiling with "-usefastmath"

Single or double precision

Unrolling loops

- Overhead by loop/branching is relatively high

```
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
    sum += data[sharMemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

- Results in 2-fold performance increase

```
sum = data[sharMemPos - 1] * d_Kernel[2]
    + data[sharMemPos + 0] * d_Kernel[1]
    + data[sharMemPos + 1] * d_Kernel[0];
```

# Optimizing Memory Usage

Minimizing data transfers with low bandwidth

- Minimizing host & device transfer
- Maximizing usage of shared memory
- Re-computing can sometimes be cheaper than transfer
- Low-parallelism computation can sometimes be faster

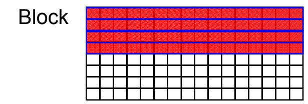
Organizing memory accesses based on the optimal memory access patterns

- Important for global memory access (low bandwidth)
- Shared memory accesses are usually worth optimizing only in case they have a high degree of bank conflicts



# Global Memory Coalescing

Warps & global memory



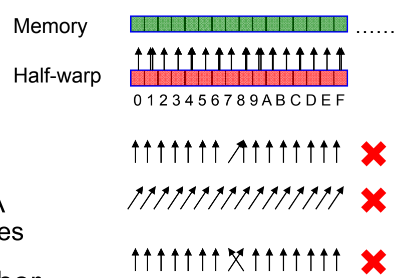

- Threads execute by warp (32)
- Memory read/write by half warp (16)
- Global memory is considered to be partitioned into segments of size equal to 32, 64, or 128 bytes and aligned to this sizes.
- Block width must be divisible by 16 for coalescing
- Check your hardware (Compute Capability 1.x)
- Great improve throughput (Can yield speedups of >10)



# Global Memory Coalescing

Compute Capability 1.0 or 1.1

- Aligned 64 or 128 bytes segment
- Sequential warp
- Divergent warp
- See some good patterns in CUDA document and CUDA SDK samples



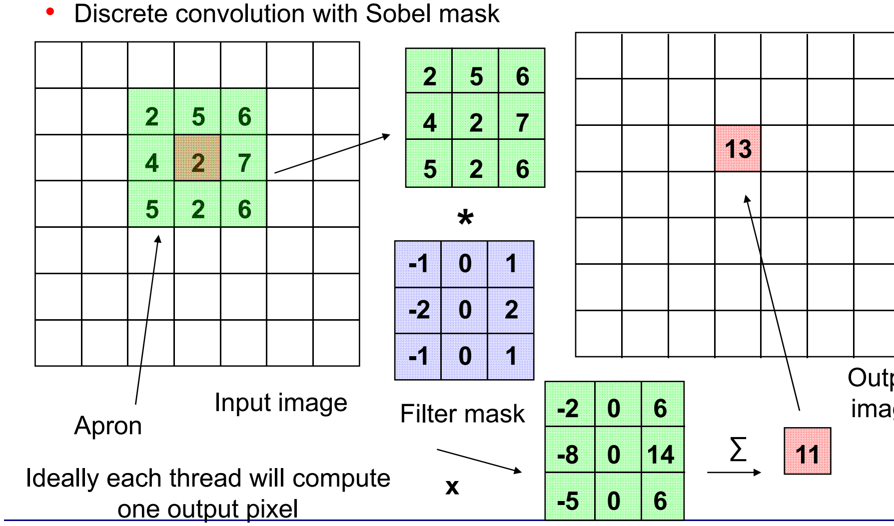

Compute Capability 1.2 or higher

- 32, 64 or 128 bytes segment
- Any pattern as long as inside segment



# Example 5: Sobel Filter

- Discrete convolution with Sobel mask



# R/W Global Memory

Bad access pattern

- Global memory only. No texture memory or shared memory. Hundreds of clock cycles, compared to 1 or 2 for reading from shared memory
- Unstructured read (non sequential) mostly
- No cache, up to 12 global memory reads per thread

Input from global memory

Output to another global memory

```
__global__ void
SobelBadKernel(unsigned char* Input, unsigned char* output,unsigned int width, unsigned int
    height)
{
….//calculate the index for ur, ul, um, ml, mr, ll, lm, lr.
 float Horz=Input[ur] +Input[lr] +2.0*Input[mr] -2.0*Input[ml] -Input[ul] -Input[ll] ;
 float Vert=Input[ur] +Input[ul] +2.0*Input[um] -2.0*Input[lm] -Input[ll] -Input[lr] ;
 output[resultindex]   = abs(Horz)+abs(Vert);
}
```



# Reduce Global Memory Read

```
__device__ unsigned char ComputeSobel(
        unsigned char ul,
        unsigned char um,
        unsigned char ur,
        unsigned char ml,
        unsigned char mm, //not used
        unsigned char mr,
        unsigned char ll,
        unsigned char lm,
        unsigned char lr,
        float fScale ){
  short Horz = ur + 2*mr + lr - ul - 2*ml - ll;
  short Vert = ul + 2*um + ur - ll - 2*lm - lr;
  short Sum = (short) (fScale*(abs(Horz)+abs(Vert)));
  if ( Sum < 0 ) return 0; else if ( Sum > 255 ) return 255;
  return (unsigned char) Sum;}
```

| | | |
|---|---|---|
| 2 | 5 | 6 |
| 4 | 2 | 7 |
| 5 | 2 | 6 |

Reduce 12 reads into 8 or 9 reads

SPIE Medical Imaging 2009

# Reading Texture Memory

CG approach

- Each kernel computes one pixel
- Reading 9 pixels in apron
- Read/output → 9/1

Take advantage of CUDA (texture memory)

- Using cache ( texture memory ) to enhance performance
- Each kernel can compute more than one pixels. This can help to exploit locality for cache
- Texture memory itself is optimized for coalescing

SPIE Medical Imaging 2009

# Reading Texture Memory

- Texture memory only. No shared memory
- Almost the same as collecting in CG (A little different

Global memory as output. Need consider coalescing when write back

One thread computes (width/ blockDim.x) pixels

Read from texture memory

```
unsigned char *pSobel = (unsigned char *) (((char *) pSobelOrig
for ( int i = threadIdx.x; i < w; i += blockDim.x ) {
   unsigned char pix00 = tex2D( tex, (float) i-1, (float) blockIdx.x-1 );
   unsigned char pix01 = tex2D( tex, (float) i+0, (float) blockIdx.x
   unsigned char pix02 = tex2D( tex, (float) i+1, (float) blockIdx.x-1 );
   unsigned char pix10 = tex2D( tex, (float) i-1, (float) blockIdx.x+0 );
   unsigned char pix11 = tex2D( tex, (float) i+0, (float) blockIdx.x+0 );
   unsigned char pix12 = tex2D( tex, (float) i+1, (float) blockIdx.x+0 );
   unsigned char pix20 = tex2D( tex, (float) i-1, (float) blockIdx.x+1
   unsigned char pix21 = tex2D( tex, (float) i+0, (float) blockIdx.x+1 );
   unsigned char pix22 = tex2D( tex, (float) i+1, (float) blockIdx.x+1 );
   pSobel[i] = ComputeSobel(pix00, pix01, pix02, pix10, pix11, pix12,
                      pix20, pix21, pix22, fScale );}
```

SPIE Medical Imaging 2009

# Improve Caching?

SPIE Medical Imaging

Advantage

- Texture memory read is better than global or constant memory

Disadvantage

- Only using hardware cache to handle spatial locality
- A pixel may be still loaded 9 times in total due to cache miss

Take advantage of CUDA Shared Memory

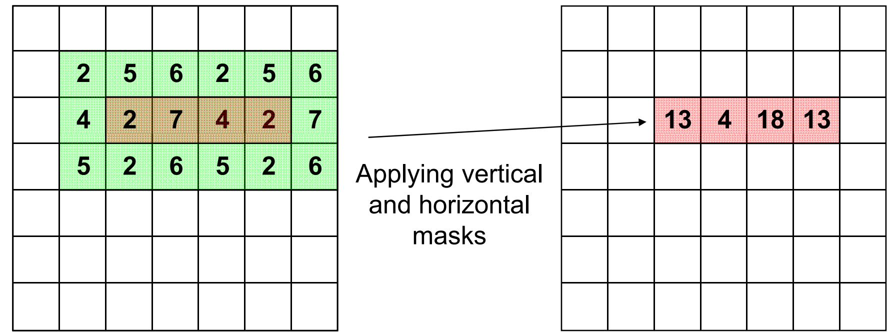
- Shared memory can be as fast as register! As a user-controlled cache.

1. Together with texture memory, load a block of the image into shared memory
2. Each thread compute a consecutive rows of pixels (sliding window)
3. Writing result to global memory



# Example 5: Sobel Filter

SPIE Medical Imaging




Each thread will compute a number of consecutive rows of pixel

Computing all pixels inside one block (without apron)



# Reading Shared Memory

SPIE Medical Imaging

- Shared memory + texture memory.

```
__shared__ unsigned char shared[];
kernel<<<blocks, threads, sharedMem>>>(…);
```

| | | | | | |
|---|---|---|---|---|---|
| 2 | 5 | 6 | 2 | 5 | 6 |
| 4 | 2 | 7 | 4 | 2 | 7 |
| 5 | 2 | 6 | 5 | 2 | 6 |

```
……// copy a large tile of pixels into shared memory
  __syncthreads();
……// read 9 pixels from shared memory
 out.x = ComputeSobel(pix00, pix01, pix02,  pix10, pix11, pix12, pix20, pix21, pix22, fScale );
……//read p00, p10, p20
out.y = ComputeSobel(pix01, pix02, pix00,  pix11, pix12, pix10, pix21, pix22, pix20, fScale );
……// read p01, p11, p21
out.z = ComputeSobel( pix02, pix00, pix01,  pix12, pix10, pix11, pix22, pix20, pix21, fScale );
……// read p02, p12, p22
out.w = ComputeSobel( pix00, pix01, pix02,  pix10, pix11, pix12, pix20, pix21, pix22, fScale );
  __syncthreads();
```

Loading data under current window, 9 reads

Sliding window right, reuse 6, update 3

Sliding window right, reuse 6, update 3

Sliding window right, reuse 6, update 3



# Shared Memory Bank Conflicts

SPIE Medical Imaging

Shared memory banks

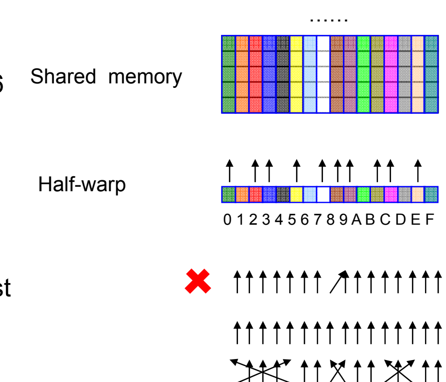
- Shared memory are divided into 16 banks to reduce the conflicts
- In a half-warp, each thread can access 32-bit from different banks simultaneously to achieve high memory bandwidth
- Conflict-free shared memory as fast as registers
- Linear

```
shared__ float shared[32];
float data = shared[BaseIndex + 1* tid];
```

Thread ID

- Random

1, 3, 5 ,7 …… (Any odd number)



# Shared Memory Bank Conflicts

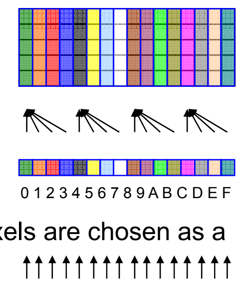
SPIE Medical Imaging

4-way bank conflicts

```
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

No bank conflicts

```
char data = shared[BaseIndex + 4 * tid];
```




- In shared memory edge detection example, 4 pixels are chosen as a group (4 unsigned char = 32 bit)
- If data is larger than 32 bits, one way to avoid bank conflicts in this case is to split data. It might not always improve and will perform worse in future architectures
- Structure assignment can be used



# Reading Shared Memory

SPIE Medical Imaging

Shared memory 9 reads

```
unsigned char pix00 = shared[BaseIndex+4*tid+0*Pitch+0];
unsigned char pix01 = shared [BaseIndex+4*tid+0*Pitch+1];
unsigned char pix02 = shared[BaseIndex+4*tid+0*Pitch+2];
unsigned char pix10 = shared[BaseIndex+4*tid+1*Pitch+0];
unsigned char pix11 = shared[BaseIndex+4*tid+1*Pitch+1];
unsigned char pix12 = shared[BaseIndex+4*tid+1*Pitch+2];
unsigned char pix20 = shared[BaseIndex+4*tid+2*Pitch+0];
unsigned char pix21 = shared[BaseIndex+4*tid+2*Pitch+1];
unsigned char pix22 = shared[BaseIndex+4*tid+2*Pitch+2];
```

Pad arrays so that every row starts at a 64-byte (16x32bit)boundary address will improve performance.

| | | | |
|---|---|---|---|
| 2 | 5 | 6 | 2 |
| 4 | 2 | 7 | 4 |
| 5 | 2 | 6 | 5 |

Shared memory update 3 reads

```
pix00 = shared[BaseIndex+4*tid+0*Pitch+3];
pix10 = shared [BaseIndex+4*tid+1*Pitch+3];
pix20 = shared[BaseIndex+4*tid+2*Pitch+3];
```

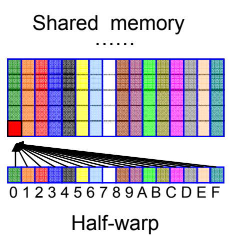


# Shared Memory Broadcasting

SPIE Medical Imaging

Shared memory read a 32-bit word and broadcast to several threads simultaneously

- Read
- Reduce or resolve bank conflicts if set to broadcasting
- Which word is selected as the broadcast word and which address is picked up for each bank at each cycle are unspecified






# Sample From Nvidia CUDA SDK

SPIE Medical Imaging

Image processing samples

- Histogram
- Bicubic filter
- Sobel filter (FBO/PBO)
- Boxfilter
- Volume-render (3D PBO)
- ……

Code examples in this lecture reference above Nvidia SDK Sample

## To Probe Further

SPIE Medical Imaging

NVIDIA CUDA Zone:

- http://www.nvidia.com/object/cuda_home.html
- Lots of information and code examples
- NVIDIA CUDA Programming Guide

GPGPU community:

- http://www.gpgpu.org
- User forums, tutorials, papers
- Good source: conference tutorials http://www.gpgpu.org/developer/index.shtml#conference-tutorial



## Conclusion

SPIE Medical Imaging

- CUDA Hardware

Threads cooperate using shared memory

- CUDA Programming API

Launch parallel kernels

- CUDA Graphics API

Visualize the result

- CUDA Performance

Memory is complex but important



## Course Schedule

SPIE Medical Imaging

| | |
|---|---|
| 1:30 – 2:00: | Introduction (Klaus Mueller) |
| 2:00 – 2:45: | Graphics-style GPU programming with CG (Wei Xu) |
| 2:45 – 3:00: | GPGPU-style GPU programming with CUDA (Ziyi Zheng) |
| | Coffee Break |
| 3:30 – 4:00: | GPGPU-style GPU programming with CUDA (Ziyi Zheng) |
| 4:00 – 4:20: | CT reconstruction pipeline components (Klaus Mueller) |
| 4:20 – 5:20: | GPU-accelerated CT reconstruction (Fang Xu) |
| 5:20 – 5:30: | Extensions and final remarks (all) |