CS 179: GPU Programming

Lecture 10

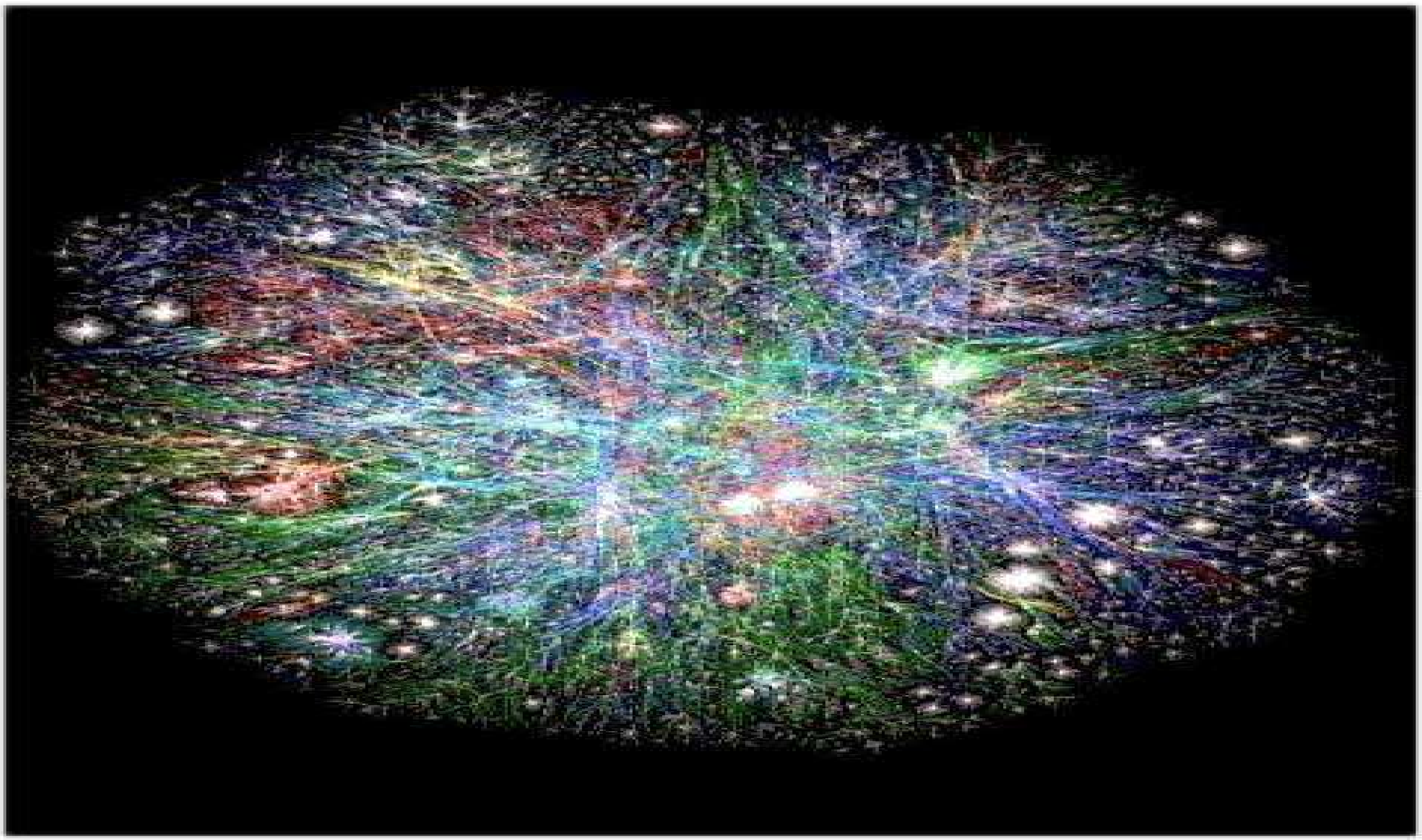
Topics

- Non-numerical algorithms
  - Parallel breadth-first search (BFS)
- Texture memory

- GPUs – good for many numerical calculations…

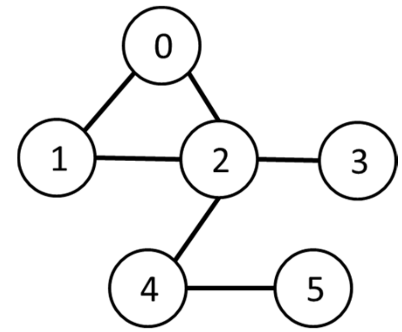
- What about “non-numerical” problems?

Graph Algorithms



Graph Algorithms

- Graph G(V, E) consists of:
  - Vertices
  - Edges (defined by pairs of vertices)

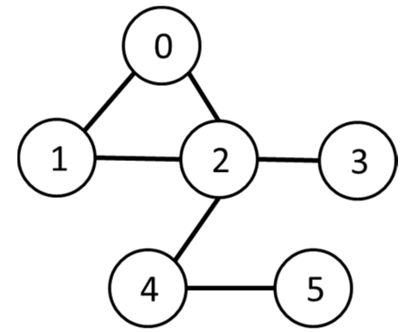


- Complex data structures!
  - How to store?
  - How to work around?

- Are graph algorithms parallelizable?

Breadth-First Search*

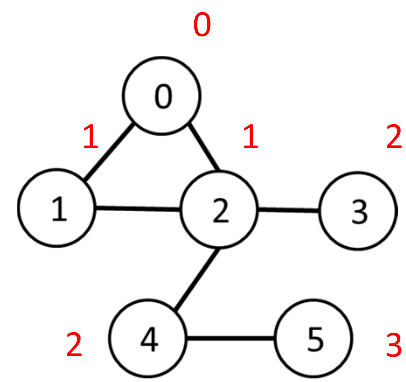
- Given source vertex S:
  - Find min. #edges to reach every vertex from S



*variation

Breadth-First Search*

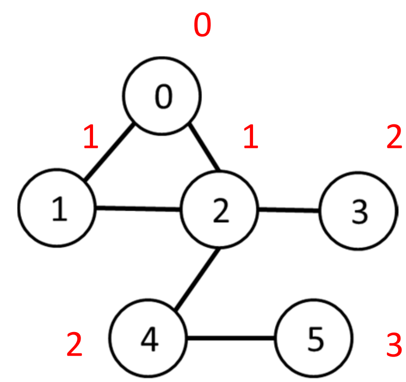
- Given source vertex S:
  - Find min. #edges to reach every vertex from S
  - (Assume source is vertex 0)



*variation

Breadth-First Search*

- Given source vertex S:
  - Find min. #edges to reach every vertex from S
  - (Assume source is vertex 0)



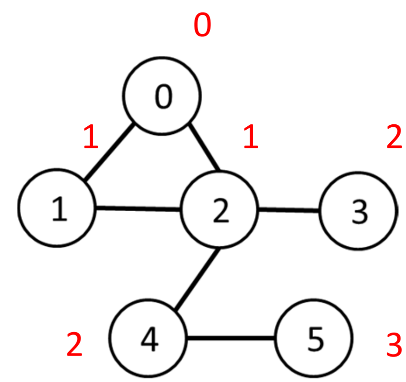
- Sequential pseudocode:

```
let Q be a queue
Q.enqueue(source)
label source as discovered
source.value <- 0

while Q is not empty
   v ← Q.dequeue()
   for all edges from v to w in G.adjacentEdges(v):
       if w is not labeled as discovered
           Q.enqueue(w)
           label w as discovered, w.value <- v.value + 1
```

Breadth-First Search*

- Given source vertex S:
  - Find min. #edges to reach every vertex from S
  - (Assume source is vertex 0)



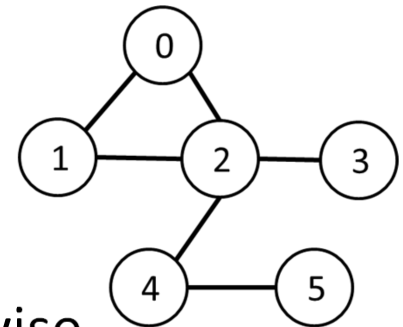
- Sequential pseudocode:

```
let Q be a queue
Q.enqueue(source)
label source as discovered
source.value <- 0

while Q is not empty
   v ← Q.dequeue()
   for all edges from v to w in G.adjacentEdges(v):
       if w is not labeled as discovered
           Q.enqueue(w)
           label w as discovered, w.value <- v.value + 1
```

Runtime:
O(|V| + |E|)

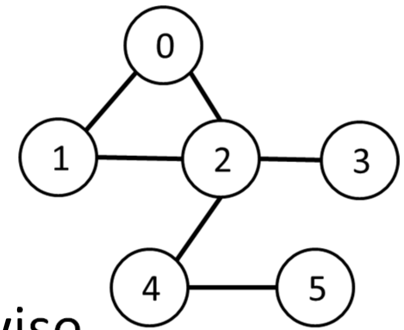
Representing Graphs



- "Adjacency matrix"
  - A: |V| x |V| matrix:
    - A_{ij} = 1 if vertices i,j are adjacent, 0 otherwise
  - $O(V^2)$ space

- "Adjacency list"
  - Adjacent vertices noted for each vertex
  - O(V + E) space

Representing Graphs

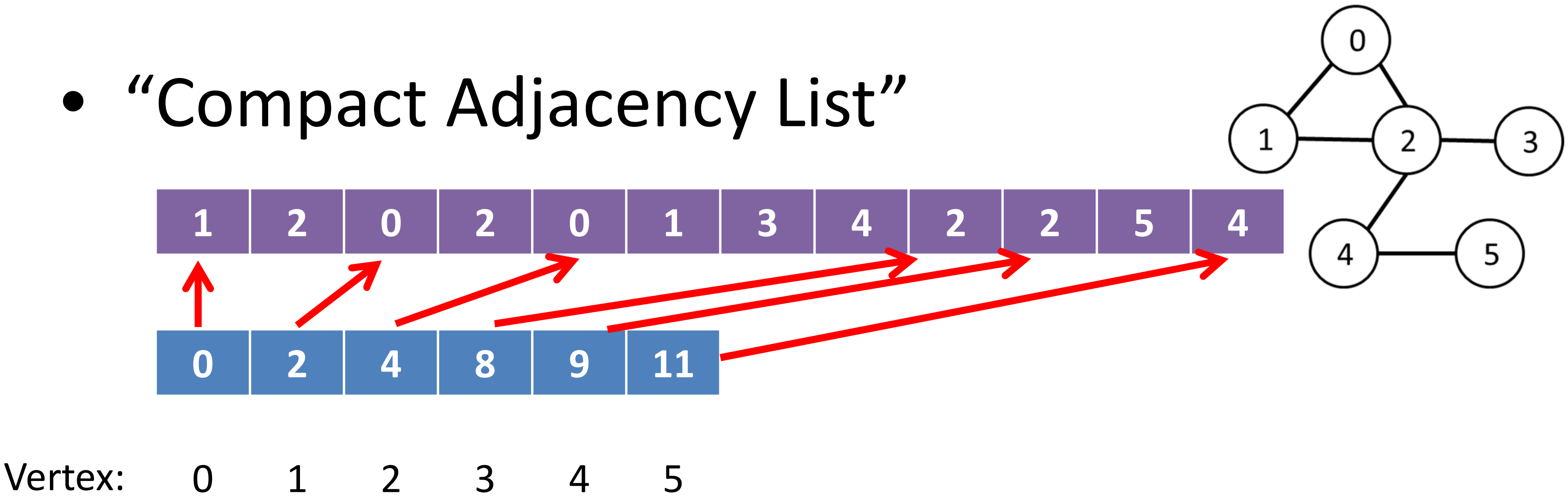


- "Adjacency matrix"
  - A: |V| x |V| matrix:
    - A_{ij} = 1 if vertices i,j are adjacent, 0 otherwise
  - $O(V^2)$ space <- hard to fit, more copy overhead

- "Adjacency list"
  - Adjacent vertices noted for each vertex
  - O(V + E) space <- easy to fit, less copy overhead

Representing Graphs

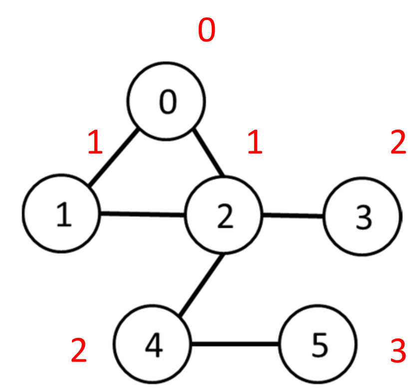
- "Compact Adjacency List"



  - Array E_a: Adjacent vertices to vertex 0, then vertex 1, then … size: O(E)
  - Array V_a: Delimiters for E_a size: O(V)

Breadth-First Search*

- Given source vertex S:
  - Find min. #edges to reach every vertex from S
  - (Assume source is vertex 0)



- Sequential pseudocode:

```
let Q be a queue
Q.enqueue(source)
label source as discovered
source.value <- 0

while Q is not empty
   v ← Q.dequeue()
   for all edges from v to w in G.adjacentEdges(v):
       if w is not labeled as discovered
           Q.enqueue(w)
           label w as discovered, w.value <- v.value + 1
```

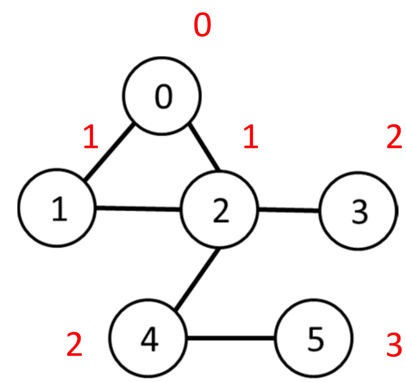
How to “parallelize” when there’s a queue?

Breadth-First Search*

- Sequential pseudocode:

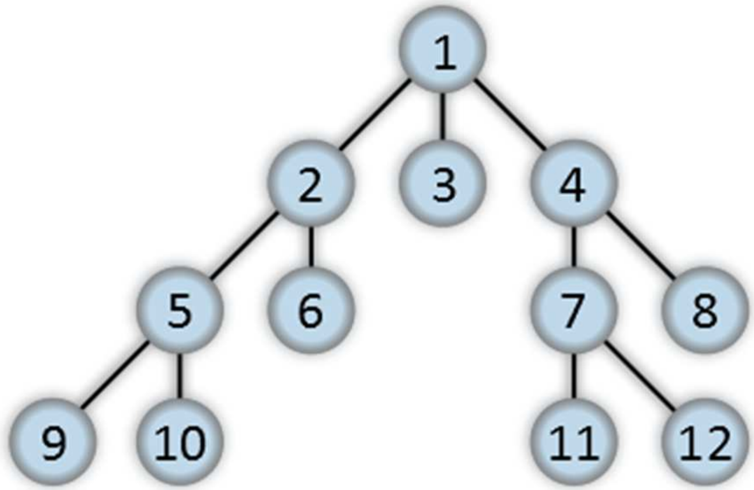
```
let Q be a queue
Q.enqueue(source)
label source as discovered
source.value <- 0

while Q is not empty
   v ← Q.dequeue()
   for all edges from v to w in G.adjacentEdges(v):
       if w is not labeled as discovered
           Q.enqueue(w)
           label w as discovered, w.value <- v.value + 1
```



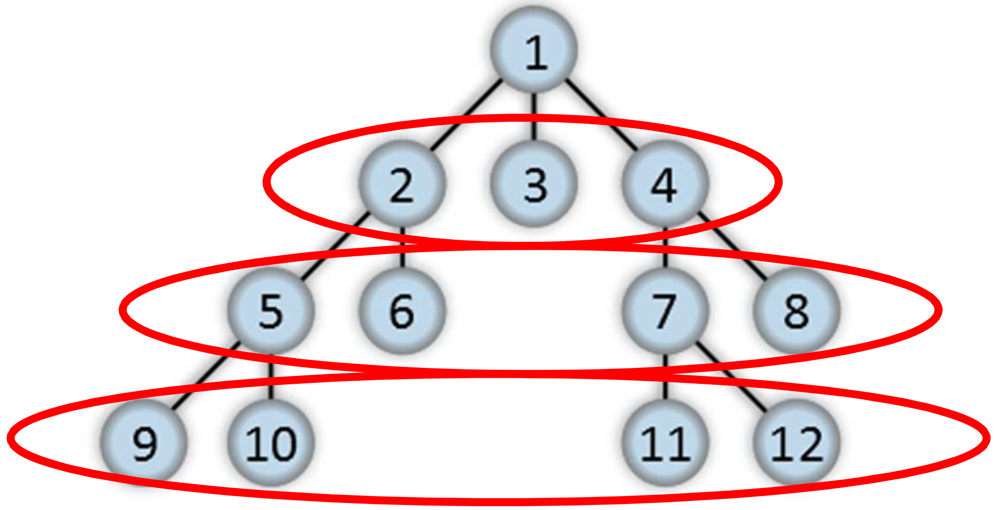
- Why do we use a queue?

BFS Order



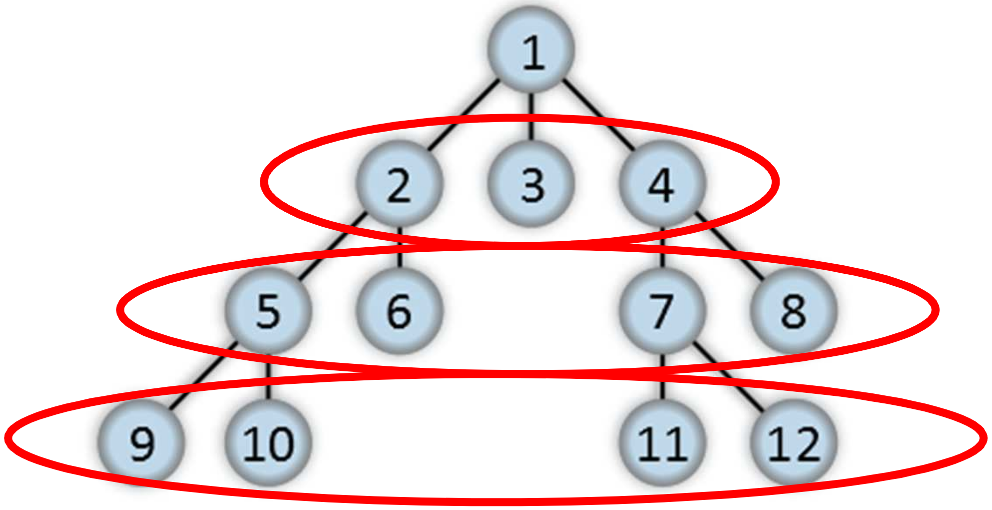
Here, vertex #s are possible BFS order

BFS Order



Permutation within ovals preserves BFS!

BFS Order



Permutation within ovals preserves BFS!

- Queue replaceable if layers preserved!

Alternate BFS algorithm

- Construct arrays of size |V|:
  - “Frontier” (denote F):
    - Boolean array - indicating vertices “to be visited” (at beginning of iteration)
  - “Visited” (denote X):
    - Boolean array - indicating already-visited vertices
  - “Cost” (denote C):
    - Integer array - indicating #edges to reach each vertex

- Goal: Populate C

Alternate BFS algorithm

- New sequential pseudocode:

```
Input: Va, Ea, source        (graph in “compact adjacency list” format)
Create frontier (F), visited array (X), cost array (C)
F <- (all false)
X <- (all false)
C <- (all infinity)

F[source] <- true
C[source] <- 0
while F is not all false:

   for 0 ≤ i < |Va|:
      if F[i] is true:

         F[i] <- false
         X[i] <- true

         for all neighbors j of i:
            if X[j] is false:

               C[j] <- C[i] + 1
               F[j] <- true
```

Alternate BFS algorithm

- New sequential pseudocode:

```
Input: Va, Ea, source       (graph in "compact adjacency list" format)
Create frontier (F), visited array (X), cost array (C)
F <- (all false)
X <- (all false)
C <- (all infinity)

F[source] <- true
C[source] <- 0
while F is not all false:

   for 0 ≤ i < |Va|:
      if F[i] is true:

         F[i] <- false
         X[i] <- true

         for Ea[Va[i]] ≤ j < Ea[Va[i+1]]:
            if X[j] is false:

               C[j] <- C[i] + 1
               F[j] <- true
```

Alternate BFS algorithm

- New sequential pseudocode:

```
Input: Va, Ea, source        (graph in “compact adjacency list” format)
Create frontier (F), visited array (X), cost array (C)
F <- (all false)
X <- (all false)
C <- (all infinity)

F[source] <- true
C[source] <- 0
while F is not all false:

   for 0 ≤ i < |Va|:
      if F[i] is true:

         F[i] <- false
         X[i] <- true

         for Ea[Va[i]] ≤ j < Ea[Va[i+1]]:
            if X[j] is false:

               C[j] <- C[i] + 1
               F[j] <- true
```

Parallelizable!

GPU-accelerated BFS

- CPU-side pseudocode:

```
Input: Va, Ea, source        (graph in "compact adjacency list" format)
Create frontier (F), visited array (X), cost array (C)
F <- (all false)
X <- (all false)
C <- (all infinity)

F[source] <- true
C[source] <- 0
while F is not all false:
   call GPU kernel( F, X, C, Va, Ea )
```

Can represent boolean values as integers

- GPU-side kernel pseudocode:

```
if F[threadId] is true:

   F[threadId] <- false
   X[threadId] <- true

   for Ea[Va[threadId]] ≤ j < Ea[Va[threadId + 1]]:
      if X[j] is false:
         C[j] <- C[threadId] + 1
         F[j] <- true
```

GPU-accelerated BFS

- CPU-side pseudocode:

```
Input: Va, Ea, source        (graph in "compact adjacency list" format)
Create frontier (F), visited array (X), cost array (C)
F <- (all false)
X <- (all false)
C <- (all infinity)

F[source] <- true
C[source] <- 0
while F is not all false:
   call GPU kernel( F, X, C, Va, Ea )
```

Can represent boolean values as integers

- GPU-side kernel pseudocode:

```
if F[threadId] is true:

   F[threadId] <- false
   X[threadId] <- true

   for Ea[Va[threadId]] ≤ j < Ea[Va[threadId + 1]]:
      if X[j] is false:
         C[j] <- C[threadId] + 1
         F[j] <- true
```

Unsafe operation?

GPU-accelerated BFS

- CPU-side pseudocode:

```
Input: Va, Ea, source       (graph in “compact adjacency list” format)
Create frontier (F), visited array (X), cost array (C)
F <- (all false)
X <- (all false)
C <- (all infinity)

F[source] <- true
C[source] <- 0
while F is not all false:
   call GPU kernel( F, X, C, Va, Ea )
```

Can represent boolean values as integers

- GPU-side kernel pseudocode:

```
if F[threadId] is true:

   F[threadId] <- false
   X[threadId] <- true

   for Ea[Va[threadId]] ≤ j < Ea[Va[threadId + 1]]:
      if X[j] is false:
         C[j] <- C[threadId] + 1
         F[j] <- true
```

Safe! No ambiguity!

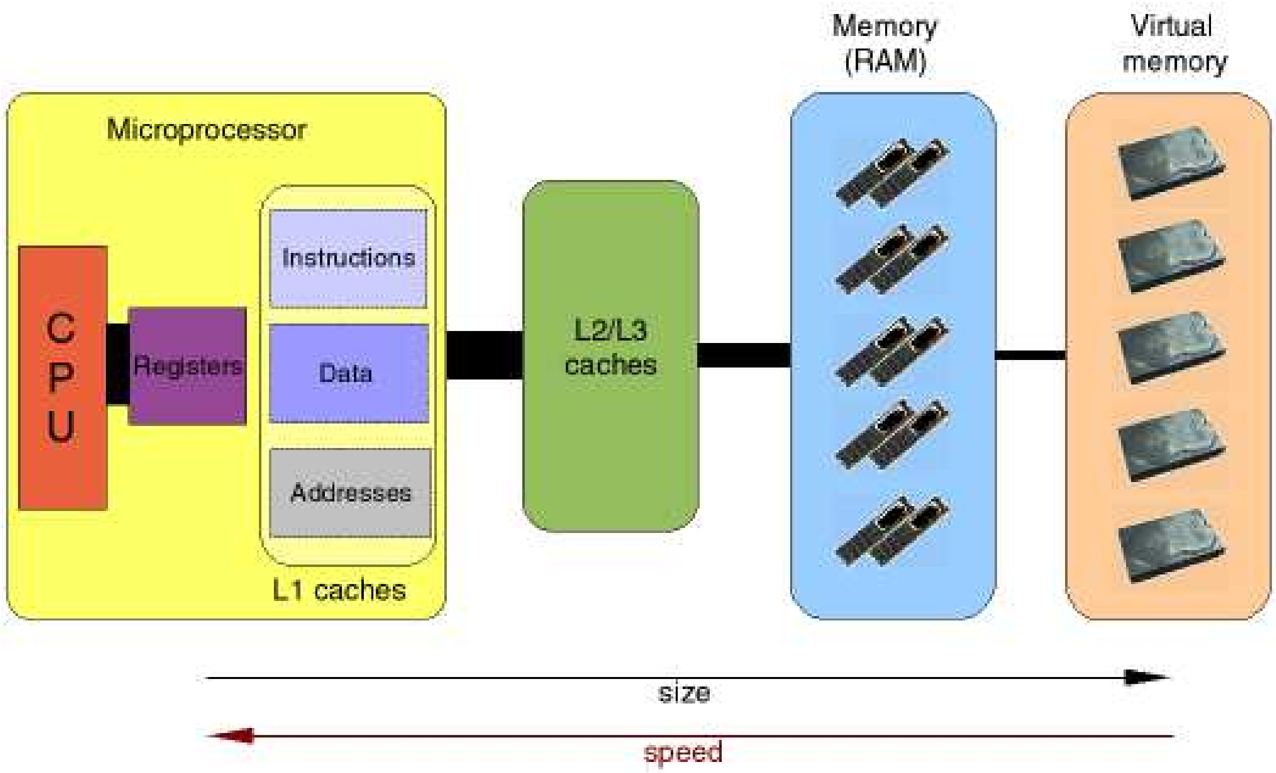
Summary

- Tricky algorithms need drastic measures!

- Resources
  - “Accelerating Large Graph Algorithms on the GPU Using CUDA” (Harish, Narayanan)

Texture Memory

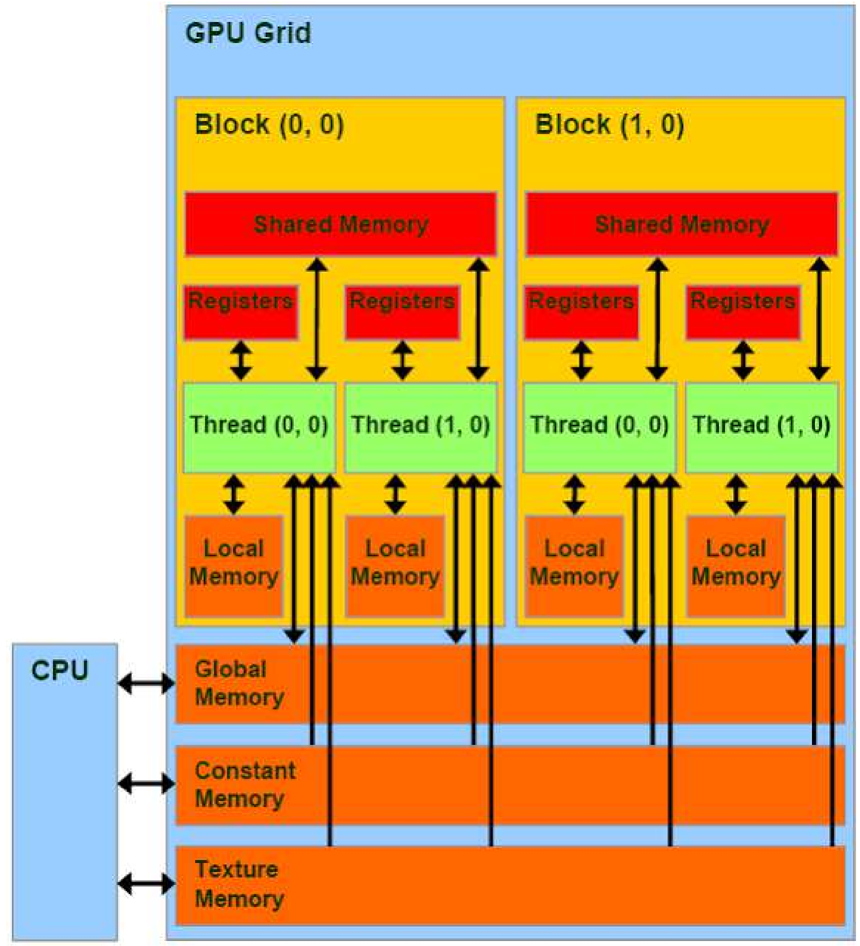
“Ordinary” Memory Hierarchy



http://www.imm.dtu.dk/~beda/SciComp/caches.png

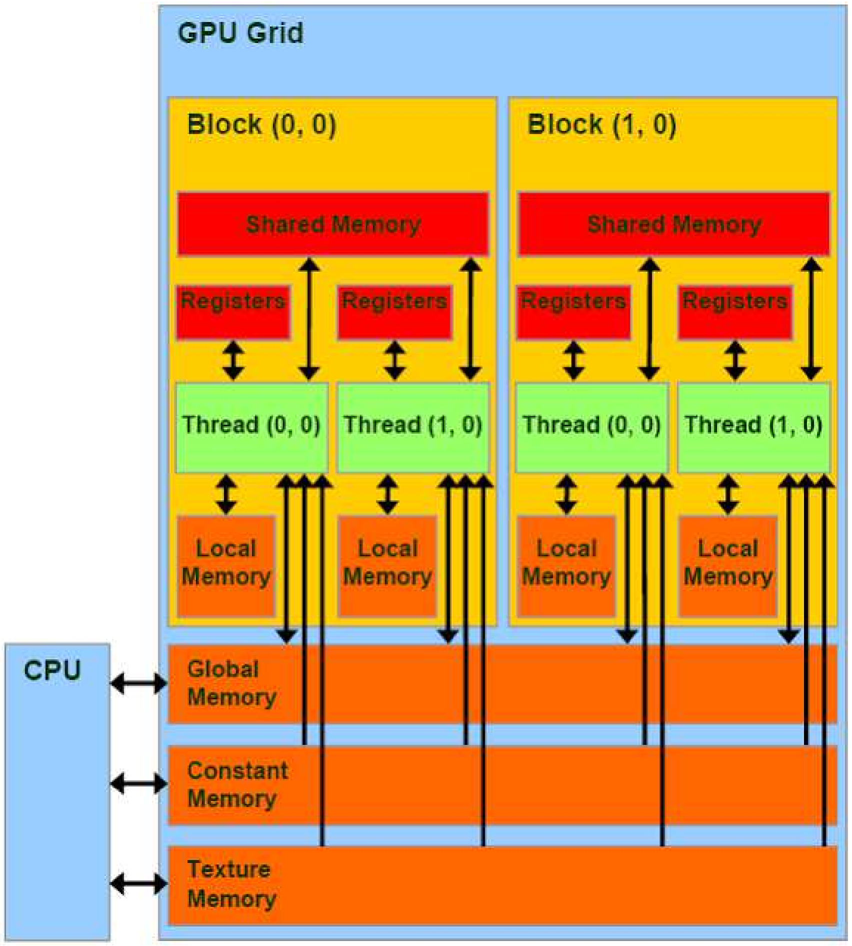
GPU Memory

- Lots of types!
  - Global memory
  - Shared memory
  - Constant memory

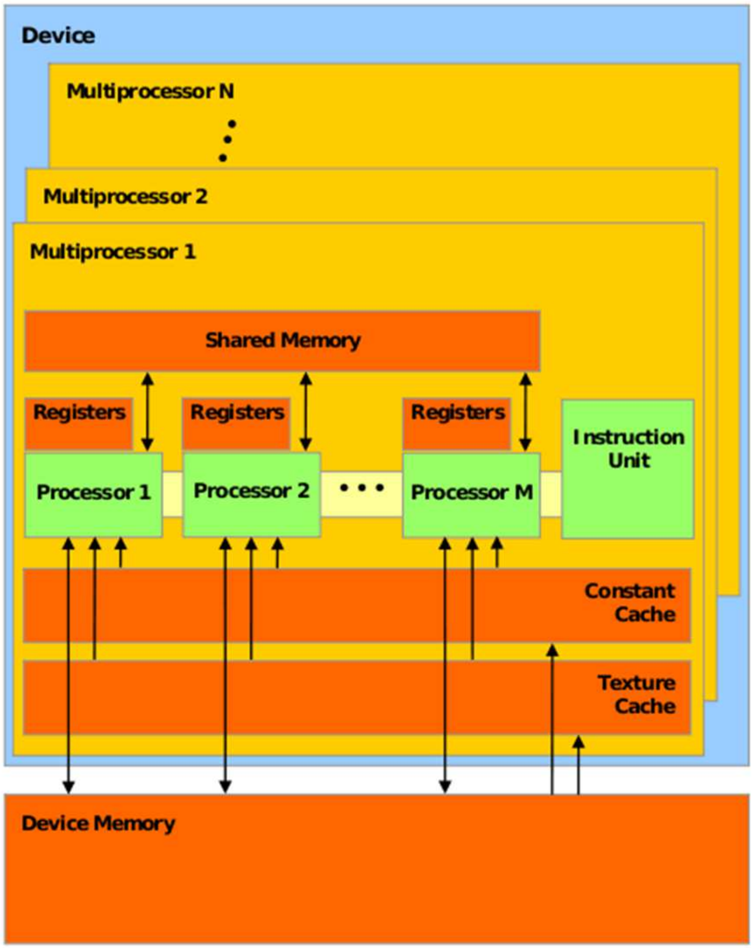


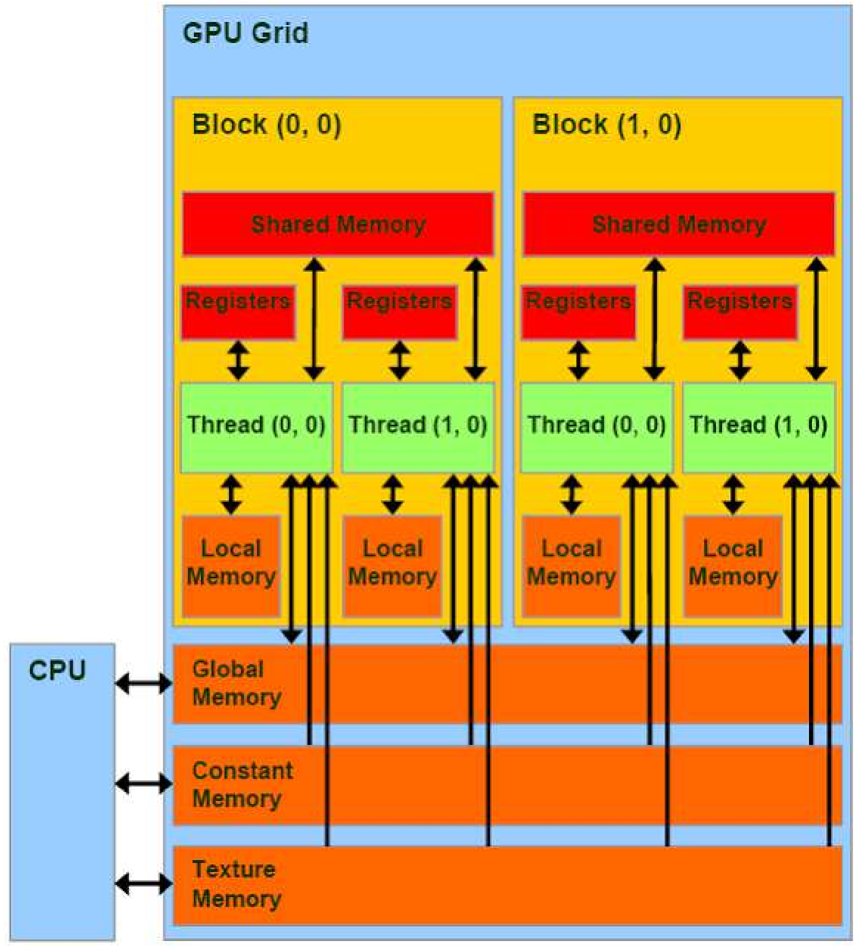
GPU Memory

- Lots of types!
  - Global memory
  - Shared memory
  - Constant memory
- Must keep in mind:
  - Coalesced access
  - Divergence
  - Bank conflicts
  - Random serialized access
  - …
  - Size!



Hardware vs. Abstraction





Hardware vs. Abstraction

- Names refer to *manner of access* on device memory:
  - "Global memory"
  - "Constant memory"
  - "Texture memory"

Review: Constant Memory

- Read-only access
- 64 KB available, 8 KB cache – small!
- *Not* "const"!
  - Write to region with cudaMemcpyToSymbol()

Review: Constant Memory

- Broadcast reads to half-warps!
  - When all threads need same data: Save reads!
- Downside:
  - When all threads need different data: Extremely slow!

Review: Constant Memory

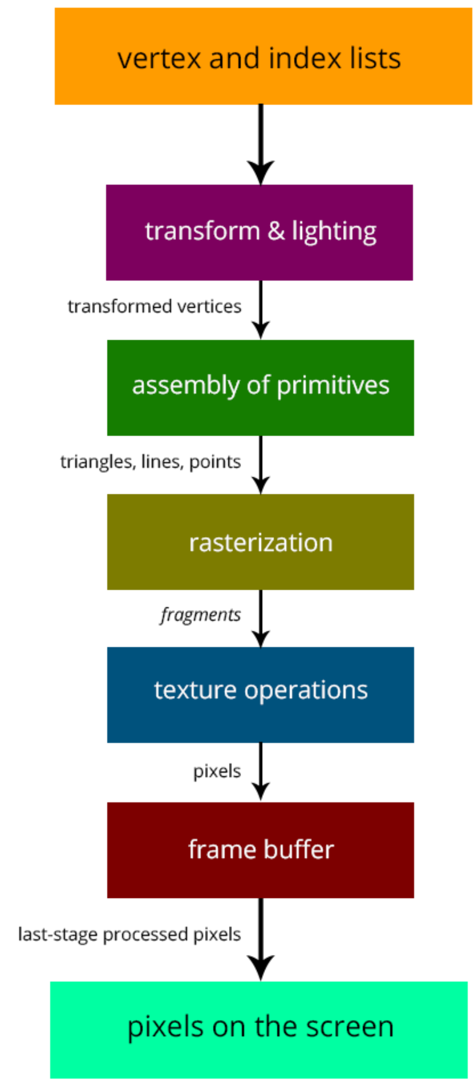
- Example application: Gaussian impulse response (from HW 1):
  - Not changed
  - Accessed simultaneously by threads in warp

Texture Memory (and co-stars)

- Another type of memory system, featuring:
  - Spatially-cached read-only access
  - Avoid coalescing worries
  - Interpolation
  - (Other) fixed-function capabilities
  - Graphics interoperability

Fixed Functions

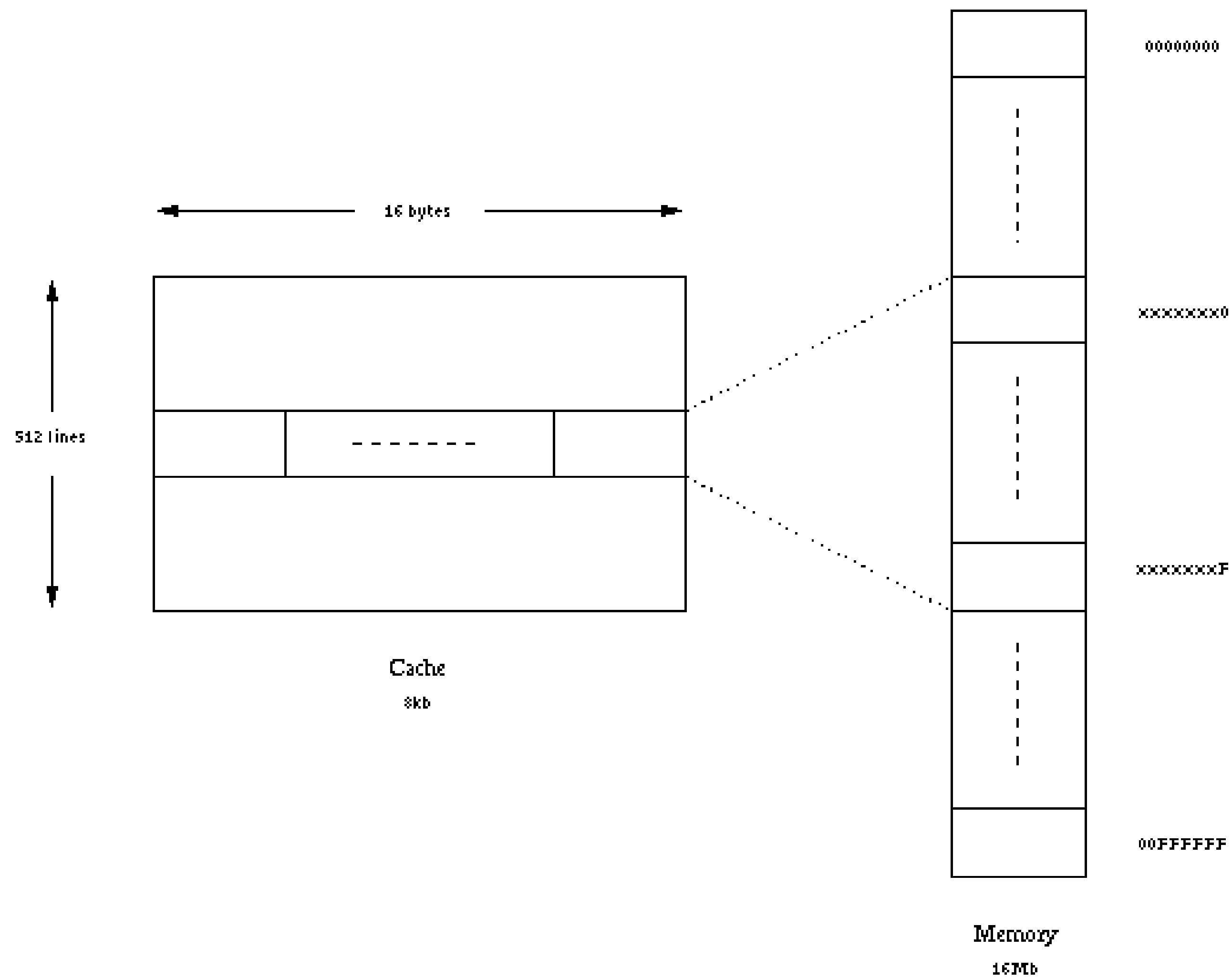
- Like GPUs in the old days!
- Still important/useful for certain things



Traditional Caching

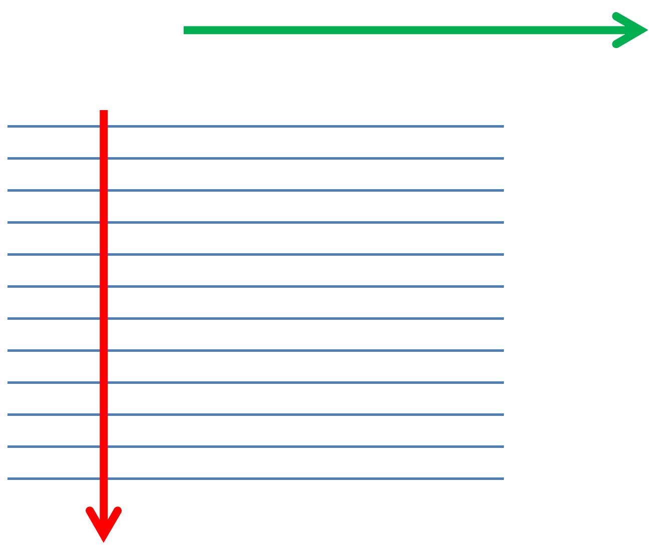
- When reading, cache “nearby elements”
  - (i.e. cache line)
  - Memory is linear!
  - Applies to CPU, GPU L1/L2 cache, etc

Traditional Caching



Traditional Caching

- 2D array manipulations:
  - One direction goes “against the grain” of caching
    - E.g. if array is stored row-major, traveling along “y-direction” is sub-optimal!



Texture-Memory Caching

- Can cache “spatially!” (2D, 3D)
  - Specify dimensions (1D, 2D, 3D) on creation

- 1D applications:
  - Interpolation, clipping (later)
  - Caching when e.g. coalesced access is infeasible

Texture Memory

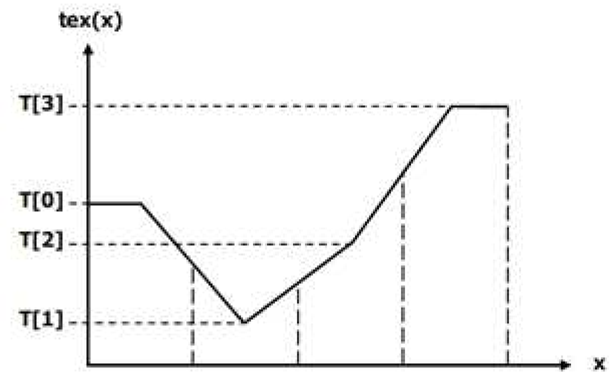
- “Memory is just an unshaped bucket of bits” (CUDA Handbook)
- Need *texture reference* in order to:
  - Interpret data
  - Deliver to registers

Texture References

- “Bound” to regions of memory
- Specify (depending on situation):
  - Access dimensions (1D, 2D, 3D)
  - Interpolation behavior
  - “Clamping” behavior
  - Normalization
  - …

Interpolation

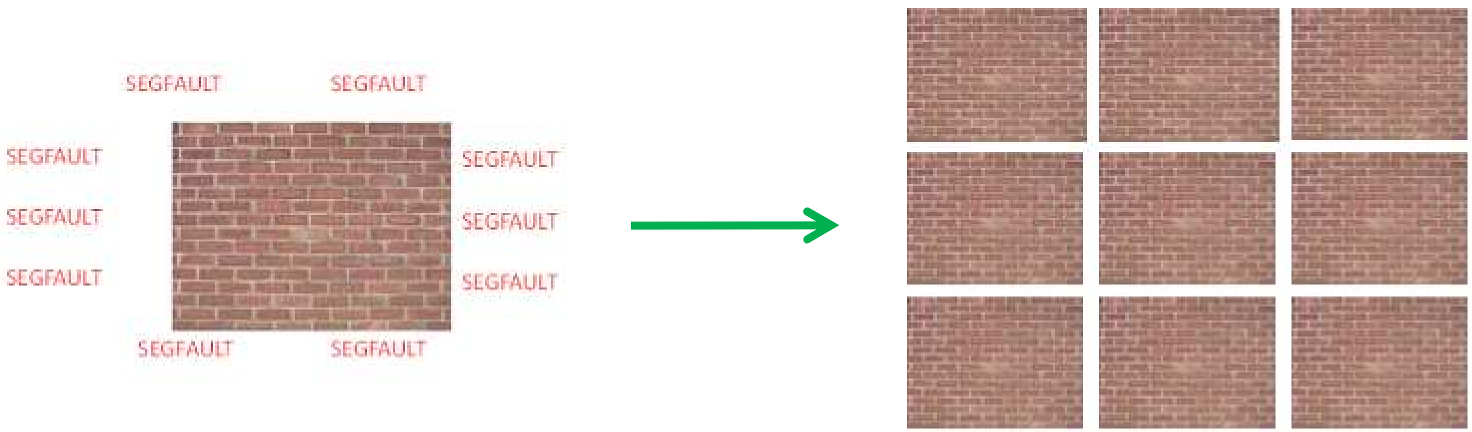
- Can “read between the lines!”



http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

Clamping

- Seamlessly handle reads beyond region!



http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

“CUDA Arrays”

- So far, we’ve used standard linear arrays

- “CUDA arrays”:
  - Different addressing calculation
    - Contiguous addresses have 2D/3D locality!

  - Not pointer-addressable
  - (Designed specifically for texturing)

Texture Memory

- Texture reference can be attached to:
  - Ordinary device-memory array
  - “CUDA array”
    - Many more capabilities

Texturing Example (2D)

Texture Memory

```
#define size 3200

//declare texture reference
texture<float,2,cudaReadModeElementType> texreference;

int main(int argc,char** argv)
{
   dim3 blocknum;
   dim3 blocksize;

   float* hmatrix;
   float* dmatrix;

   cudaArray* carray;
   cudaChannelFormatDesc channel;

   //allocate host and device memory
   hmatrix=(float*)malloc(sizeof(float)*size*size);
   cudaMalloc((void**)&dmatrix,sizeof(float)*size*size);

   //initialize host matrix before usage
   for(int loop=0;loop<size*size;loop++)
   hmatrix[loop]=float)rand()/(float)(RAND_MAX-1);
```


http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf
http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf

Texturing Example (2D)

Texture Memory

```
//create channel to describe data type
channel=cudaCreateChannelDesc<float>();

//allocate device memory for cuda array
cudaMallocArray(&carray,&channel,size,size);

//copy matrix from host to device memory
bytes=sizeof(float)*size*size;
cudaMemcpyToArray(carray,0,0,hmatrix,bytes,cudaMemcpyHostToDevice);

//set texture filter mode property
//use cudaFilterModePoint or cudaFilterModeLinear
texreference.filterMode=cudaFilterModePoint;

//set texture address mode property
//use cudaAddressModeClamp or cudaAddressModeWrap
texreference.addressMode[0]=cudaAddressModeWrap;
texreference.addressMode[1]=cudaaddressModeClamp;
```


http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf

Texturing Example (2D)

Texture Memory

```
    //bind texture reference with cuda array
    cudaBindTextureToArray(texreference,carray);

    blocksize.x=16;
    blocksize.y=16;

    blocknum.x=(int)ceil((float)size/16);
    blocknum.y=(int)ceil((float)size/16);

    //execute device kernel
    kernel<<<blocknum,blocksize>>>(dmatrix,size);

    //unbind texture reference to free resource
    cudaUnbindTexture(texreference);

    //copy result matrix from device to host memory
    cudaMemcpy(hmatrix,dmatrix,bytes,cudaMemcpyDeviceToHost);

    //free host and device memory
    free(hmatrix);
    cudaFree(dmatrix);
    cudaFreeArray(carray);

    return 0;
}
```


http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf

Texturing Example (2D)

Texture Memory

```
__global__ void kernel(float* dmatrix,int size)
{
    int xindex;
    int yindex;

    //calculate each thread global index
    xindex=blockIdx.x*blockDim.x+threadIdx.x;
    yindex=blockIdx.y*blockDim.y+threadIdx.y;

    //fetch cuda array through texture reference
    dmatrix[yindex*size+xindex]=tex2D(texreference,xindex,yindex);

    return;
}
```