Practical Course: GPU Programming in Computer Vision

CUDA Memories

Björn Häfner, Benedikt Löwenhauser, Thomas Möllenhoff

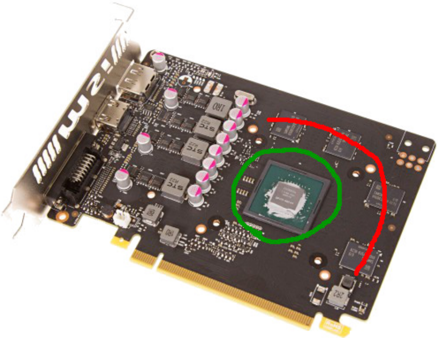
Technische Universität München
Department of Informatics
Computer Vision Group

Summer Semester 2017
September 11 - October 8

Outline

Outline

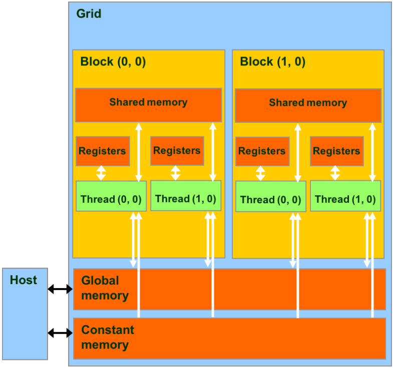
CUDA Memories



- red line is global memory (off-chip)
- green circle is the chip, contains SMs and on-chip memory

CUDA Memories

- Each thread can:
  - read / write per-thread registers
  - read / write per-block shared memory
  - read / write per-grid global memory
  - read per-grid constant memory

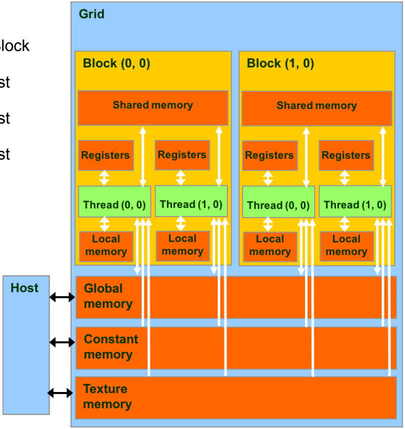


CUDA Memories

Memory	Location	Access	Scope
Register	On-Chip	Read/Write	1 Thread
Local	Off-Chip	Read/Write	1 Thread
Shared	On-Chip	Read/Write	All Threads in 1 Block
Global	Off-Chip	Read/Write	All Threads + Host
Constant	Off-Chip	Read	All Threads + Host
Texture	Off-Chip	Read/(Write)	All Threads + Host

Other memories:

- local memory
- texture memory
- both are part of global memory



CUDA Variable Type Qualifiers

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

Rules of thumb:

- scalar variables without qualifier reside in a register
- (compiler may spill to local memory)
- array variables without qualifier reside in local memory

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 memory resides in fast, on-chip memories
- thread local arrays & global variables reside in off-chip memory (though cached on modern architectures)
- constant variables reside in cached off-chip memory

CUDA Variable Type Scale

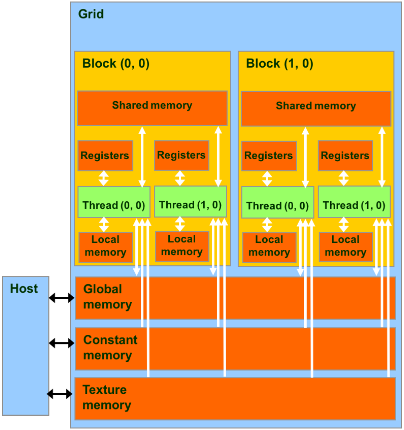
Variable declaration	Instances	Visibility
`int var;`	100,000s	1
`int array_var[10];`	100,000s	1
`__shared__ int shared_var;`	100s	100s
`__device__ int global_var;`	1	100,000s
`__constant__ int constant_var;`	1	100,000s

- 100,000s per-thread variables, read/write by 1 thread
- 100s shared variables, each read/write by 100s of threads
- 1 global variable, is read/write by 100,000s of threads
- 1 constant variable, is read by 100,000s of threads

Local Memory

Compiler might place variables in local memory:

- too many register variables
- a structure consumes too much register space
- an array is not indexed with constant quantities, i.e., when the addressing of the array is not known at compile time



Outline

Global and Shared Memory

- Global memory is located off-chip
  - high latency (often the bottleneck of computation)
  - important to minimize accesses
  - cached (L1 and L2) for reasonably modern GPUs (not cached for CC 1.x GPUs)
  - difficulty: try to coalesce accesses (more later)
- Shared memory is on-chip
  - low latency
  - like a user-managed per-SM cache
  - GPUs in lab: 48kb per multiprocessor
  - minor difficulty: try to minimize or avoid bank conflicts (more tomorrow)

Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Avoid multiple loads of same data by different threads of the block
- Use one/a few threads to load/compute data shared by all threads in the block

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_global(float *result, float *input, int n)
{
  int i = threadIdx.x + blockDim.x*blockIdx.x;

  float res = 0;
  if (i+1 < n)
  {
    // each thread loads two elements from global memory
    float xplus1 = input[i+1];
    float x0 = input[i];
    res = xplus1 - x0;
  }
  if(i<n) result[i] = res;
}
```

- `input[i]` is read by thread `i-1` and by thread `i`
- Idea: eliminate redundancy by sharing data

Shared Memory: Example

```
#define BLOCK_SIZE 32

// forward differences discretization of derivative
__global__ void diff_shared(float *result, float *input, int n)
{
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int iblock = threadIdx.x; // local "block" version of i

  // allocate shared array, of constant size BLOCK_SIZE
  __shared__ float sh_data[BLOCK_SIZE];

  // each thread reads one element and writes into sh_data
  if (i<n) sh_data[iblock] = input[i];

  // ensure all threads finish writing before continuing
  __syncthreads();
  ...
```

Shared Memory: Example

```
  ...

  float res = 0;
  if (i+1 < n)
  {
    // handle thread block boundary
    int xplus1 = (iblock+1<blockDim.x)? sh_data[iblock+1] : input[i+1];
    int x0 = sh_data[iblock];
    res = xplus1 -x0;
  }
  if(i<n) result[i] = res;
}
```

Shared Memory: Example

```
__global__ void diff_global(float *result,
  float *input, int n) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  float res = 0;
  if (i+1 < n) {
    // each thread loads
    // two elements from global memory
    float xplus1 = input[i+1];
    float x0 = input[i];
    res = xplus1 - x0;
  }
  if(i<n) result[i] = res;
}
```

```
__global__ void diff_shared(float *result,
  float *input, int n) {
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  // local "block" version of i
  int iblock = threadIdx.x;

  // allocate shared array of size BLOCK_SIZE
  __shared__ float sh_data[BLOCK_SIZE];

  // each thread reads one element
  // and writes into sh_data
  if (i<n) sh_data[iblock] = input[i];

  // ensure all threads finish
  // writing before continuing
  __syncthreads();

  float res = 0;
  if (i+1 < n) {
    // handle thread block boundary
    int xplus1 = (iblock+1<blockDim.x)?
      sh_data[iblock+1] : input[i+1];
    int x0 = sh_data[iblock];
    res = xplus1 - x0;
  }
  if (i<n) result[i] = res;
}
```

Shared Memory: Dynamic Allocation

Size known at compile time:

```
__global__ void kernel (...)
{
  ...
  __shared__ float s_data[BLOCK_SIZE];
  ...
}

int main(void)
{
  ...



  kernel <<<grid,block>>> (...);
  ...
}
```

Size known at kernel launch:

```
__global__ void kernel (...)
{
  ...
  extern __shared__ float s_data[];
  ...
}

int main(void)
{
  ...
  // allocate enough shared memory
  size_t smBytes = block.x * block.y * block.z
                   * sizeof(float);
  kernel <<<grid,block,smBytes>>> (...);
  ...
}
```

- Always use dynamic allocation
  - flexibility w.r.t. maximal block size: can specify at run time
  - no waste of resources: more blocks can run in parallel

Shared Memory: Synchronization

- `__syncthreads();`
- Synchronizes all threads in a block
  - generates a barrier synchronization instruction
  - no thread can pass this barrier until all threads in the block reach it
  - used to avoid Read-After-Write / Write-After-Read / Write-After-Write hazards for shared memory accesses
- Allowed in conditional code („if“, „while“, etc.) only if the conditional is uniform across the block
  - e.g. every thread follows the same „if“- or „else“-path

Shared Memory: Synchronization

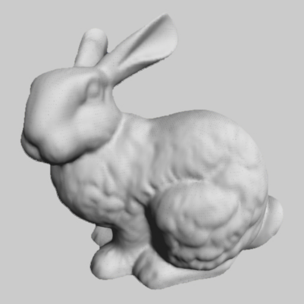
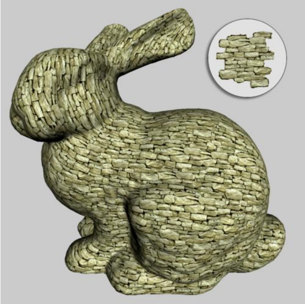
- Always use `__syncthreads();` after writing to the shared memory to ensure that data is ready for accessing
- Don't synchronize or serialize unnecessarily

```
__global__ void share_data(int *input)
{
  extern __shared__ int data[];
  data[threadIdx.x] = input[threadIdx.x];
  __syncthreads();
  // the state of the entire data array
  // is now well-defined for all threads in the block
}
```

Outline

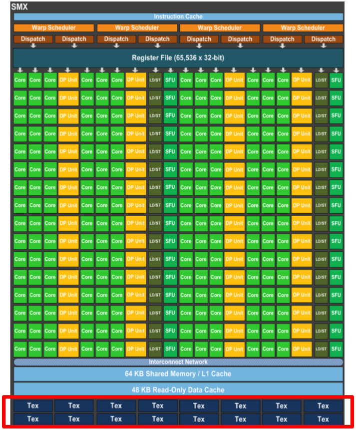
Texture Memory

- GPUs were originally intended to do computer graphics
- Still contain specialized hardware for frequent operations such as texture mapping

Textures

- Texture memory is part of global memory
- (Read-only), cached
- Global memory reads are performed through extra hardware for texture manipulation



Why use Textures?

- Data is cached, cache is optimized for 2D spatial locality
- Filtering (interpolation) with no additional costs
  - linear / bilinear / trilinear
- Wrap modes with no additional costs
  - for „out-of-bounds“ addresses
- Addressable in 1D, 2D, or 3D
  - using integer or normalized [0,1) coordinates

Texture Usage: Overview

- Host (CPU) code:
  - allocate global memory
  - create a texture reference object
  - bind the texture reference to the allocated memory
  - use texture reference in kernels
  - when done: unbind texture reference
- Device (GPU) code:
  - fetch (read) using texture reference
  - `tex1D(texRef,x)`, `tex2D(texRef,x,y)`, `tex3D(texRef,x,y,z)`
- Work best together with `cudaArray` (more later)

Texture Usage: Texture Reference

- Define a texture reference at file scope:

```
texture <Type, Dim, ReadMode> texRef;
```

- `Type`: `int, float, float2, float4, …`
- `Dim`: 1, 2, or 3, data dimension
- `ReadMode`:
  - `cudaReadModeElementType` for integer-valued textures: return value as is
  - `cudaReadModeNormalizedFloat` for integer-valued textures: normalize value to [0,1)

Texture Usage: Set Parameters

- Set boundary conditions for x and y
  - `texRef.addressMode[0] = cudaAddressModeClamp;`
  - `texRef.addressMode[1] = cudaAddressModeClamp;`
  - `cudaAddressModeClamp`, `cudaAddressModeWrap`
- Enable/disable filtering
  - `texRef.filterMode = cudaFilterModePoint;`
  - `cudaFilterModePoint`, `cudaFilterModeLinear`
- Set whether coordinates are normalized to $[0, 1)$
  - `texRef.normalized = false;`

Texture Usage: Bind and Unbind

- Bind texture to array:

```
cudaBindTexture2D(NULL, &texRef, ptr, &desc,
width, height, pitch)
```

  - `ptr`: pointer to allocated array in global memory
  - `width`: width of array
  - `height`: height of array
  - `pitch`: pitch of array in bytes, if ptr was allocated using `cudaMalloc`, this is `width*sizeof(ptr[0])`
  - `desc`: number of bits for each texture channel, e.g., `cudaCreateChannelDesc<float>()`

- Unbind texture:

```
cudaUnbindTexture(texRef);
```

Textures: Example

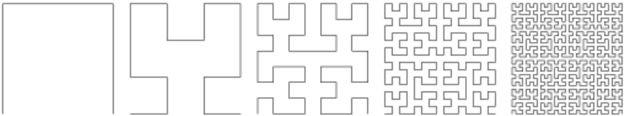
```
texture<float,2,cudaReadModeElementType> texRef; // at file scope

__global__ void kernel (...)
{
  int x = threadIdx.x + blockDim.x*blockIdx.x;
  int y = threadIdx.y + blockDim.y*blockIdx.y;
  float val = tex2D(texRef, x+0.5f, y+0.5f); // add 0.5f to get center of pixel
  ...
}

int main()
{
  ...
  texRef.addressMode[0] = cudaAddressModeClamp; // clamp x to border
  texRef.addressMode[1] = cudaAddressModeClamp; // clamp y to border
  texRef.filterMode = cudaFilterModeLinear; // linear interpolation
  texRef.normalized = false; // access as (x+0.5f,y+0.5f), not as ((x+0.5f)/w,(y+0.5f)/h)
  cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
  cudaBindTexture2D(NULL, &texRef, d_ptr, &desc, w, h, w*sizeof(d_ptr[0]));
  kernel <<<grid,block>>> (...);
  cudaUnbindTexture(texRef);
  ...
}
```

Textures: cudaArray

- Textures can use memory stored in a space filling curve



- Better texture cache hit rate due to improved 2D locality
- Copying data to a `cudaArray` will cause it to be formatted to such a curve
- { `cudaArray`, `cudaMallocArray`, `cudaMemcpyToArray`, `cudaBindTextureToArray`, `cudaFreeArray` }

```
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);
cudaMemcpyToArray(cuArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);
cudaBindTextureToArray(texRef, cuArray, channelDesc);
...
cuFreeArray(cuArray); // free device memory
```

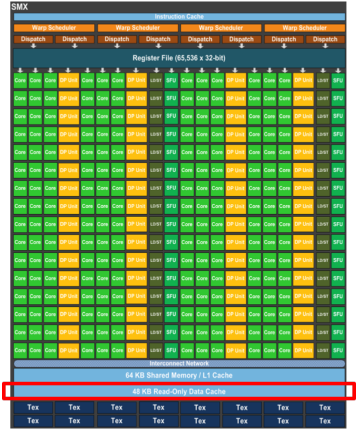
Surface Memory

- Device code (CC $\geq$ 2.0) can read/write to `cudaArray` by using `surfaces`
- See CUDA SDK example `simpleSurfaceWrite`
- Surface operations have
  - no interpolation or data conversion
  - but some boundary handling
- Some caveats:
  - texture cache is <u>not</u> notified of `cudaArray` modifications
  - similarly, texture cache is also <u>not</u> notified of global memory modifications
  - start new kernel to pick up modifications
  - surface write/reads take x coordinates in <u>byte</u> size

Outline

Constant Memory

- Part of global memory
- Read-only, cached
    - cache is dedicated
    - will not be overwritten by other global memory reads
- Fast!
- Limited size, use it to store a few cruical parameters (convolution kernel, 4×4 camera matrix, ...)



Constant Memory

- Defined at file scope
- Qualifier: `__constant__`
- Examples:
  - `__constant__ float myparam;`
  - `__constant__ float constKernel[KERNEL_SIZE];`
  - array size must be known, no dynamic allocation possible
- Reading only on device:

  ```
  float val = myparam; val = constKernel[0];
  ```
- Writing only on host:

  ```
  cudaMemcpyToSymbol(constKernel, h_ptr, szBytes);
  ```

Outline

Global Memory: Coalescing

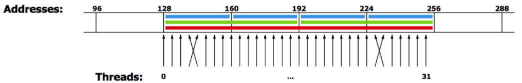
- Global memory access is slow (400-800 clock cycles)
- Hardware coalesces (combines) memory accesses
  - chunks of size 32 B, 64 B, 128 B
  - aligned to multiples of 32 B, 64 B, 128 B, respectively
- Coalescing is per warp
  - each thread reads a char: 1B*32 = 32 B chunk
  - each thread reads a float: 4B*32 = 128 B chunk
  - each thread reads a int2: 8B*32 = 2*128 B chunks

Global Memory: Coalescing

- Make sure threads within a warp access
  - a contiguous memory region
  - as few 128 B segments as possible
- Huge performance hit for non-coalesced accesses
  - memory accesses per warp will be serialized
  - worst case: reading chars from random locations

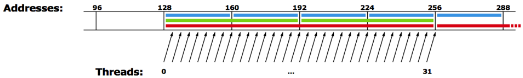
Global Memory: Coalescing

Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis-aligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	7x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224 1x 32B at 256	1x 128B at 128 1x 64B at 192 1x 32B at 256	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

The Most Important CUDA Optimization

Minimize the number of global memory accesses

- they are the slowest operations
- essentially the only reason for slow kernel run time
- if you access global memory, do it coalesced

Rules of thumb:

- neighboring threads must access neighboring elements
  - `array[threadId.x + blockDim.x * blockIdx.x]`
- (two `float` arrays are better than one `float2` array)
  - use layered memory layout for multi-channel images
- value is used a lot in same thread: load in local variable
  - even if used just more than once
- if one value is used by lots of threads: shared memory
  - but if used only by 2 or so threads, global mem is still OK

Recommended Further Reading

CUDA Programming Guide (linked on course page)

- Appendix B.1 – B.4
- Chapter 3, sections 3.2.1 – 3.2.3

Best Practices Guide (linked on course page)

- Chapter 9, section 9.2