Effective Use of Memory and Optimization - I

Dr. Alptekin Temizel
atemizel@metu.edu.tr

2nd Lab

- Will be on 12th March Tuesday at 9:40 in A-212
- Will be on memory (slides 58th onwards)

1st Quiz

- Will also be on 12th March, before we start the lab
- Please be on time, it will start exactly at 9:40. If you are late, you will not be compensated for late arrival
- The quiz questions will be from the contents of the slides up to the 58th slide, so please study these slides carefully

Compute Capability

- The compute capability of a device is defined by a major revision number and a minor revision number.

- Devices with the same major revision number are of the same core architecture.
  - Pascal architecture is 6.x, Volta 7.x.

- The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.
  - Pascal has versions 6.0, 6.1, 6.2
  - Turing architecture 7.5 (An increment on Volta core architecture)

Compute Capability- Feature Support

Feature support (unlisted features are supported for all compute abilities)	*Compute capability (version)*											
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.0/2 (Volta)	7.5 (Turing)
Integer atomic functions operating on 32-bit words in global memory	No	Yes										
atomicExch() operating on 32-bit floating point values in global memory												
Integer atomic functions operating on 32-bit words in shared memory	No	Yes										
atomicExch() operating on 32-bit floating point values in shared memory												
Integer atomic functions operating on 64-bit words in global memory												
Warp vote functions												
Double-precision floating-point operations	No				Yes							
Atomic functions operating on 64-bit integer values in shared memory	No				Yes							
Floating-point atomic addition operating on 32-bit words in global and shared memory												
_ballot()												
_threadfence_system()												
_syncthreads_count(), _syncthreads_and(), _syncthreads_or()												
Surface functions												
3D grid of thread block												
Warp shuffle functions	No					Yes						
Funnel shift	No						Yes					
Dynamic parallelism	No							Yes				
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No								Yes			
Atomic addition operating on 64-bit floating point values in global memory and shared memory	No									Yes		
Tensor core	No										Yes	

Compute Capability – Tech Specs

Technical specifications	Compute capability (version)															
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128
Maximum dimensionality of grid of thread blocks	2				3											
Maximum x-dimension of a grid of thread blocks	65535				$2^{31} - 1$											
Maximum y-, or z-dimension of a grid of thread blocks	65535															
Maximum dimensionality of thread block	3															
Maximum x- or y-dimension of a block	512				1024											
Maximum z-dimension of a block	64															
Maximum number of threads per block	512				1024											
Warp size	32															
Maximum number of resident blocks per multiprocessor	8					16				32						
Maximum number of resident warps per multiprocessor	24		32		48	64										
Maximum number of resident threads per multiprocessor	768		1024		1536	2048										
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K						
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K	64 K				32 K	64 K		32 K	64 K
Maximum number of 32-bit registers per thread	124				63	255										
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB
Maximum amount of shared memory per thread block	48 KB															48/96 KB
Number of shared memory banks	16				32											
Amount of local memory per thread	16 KB				512 KB											
Constant memory size	64 KB															

Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity
  - Aim for higher calculation/bandwidth ratio
- Sometimes it's better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host

Optimize Memory Access

- Coalesced vs. Non-coalesced
  - order of magnitude difference
- Global/Local device memory
- Optimize for spatial locality in cached texture memory
- In shared memory, avoid high-degree bank conflicts

Take Advantage of Shared Memory

Hundreds of times faster than global memory

- Threads can cooperate via shared memory
- Use one / a few threads to load, then compute data shared by all threads
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

Use Parallelism Efficiently

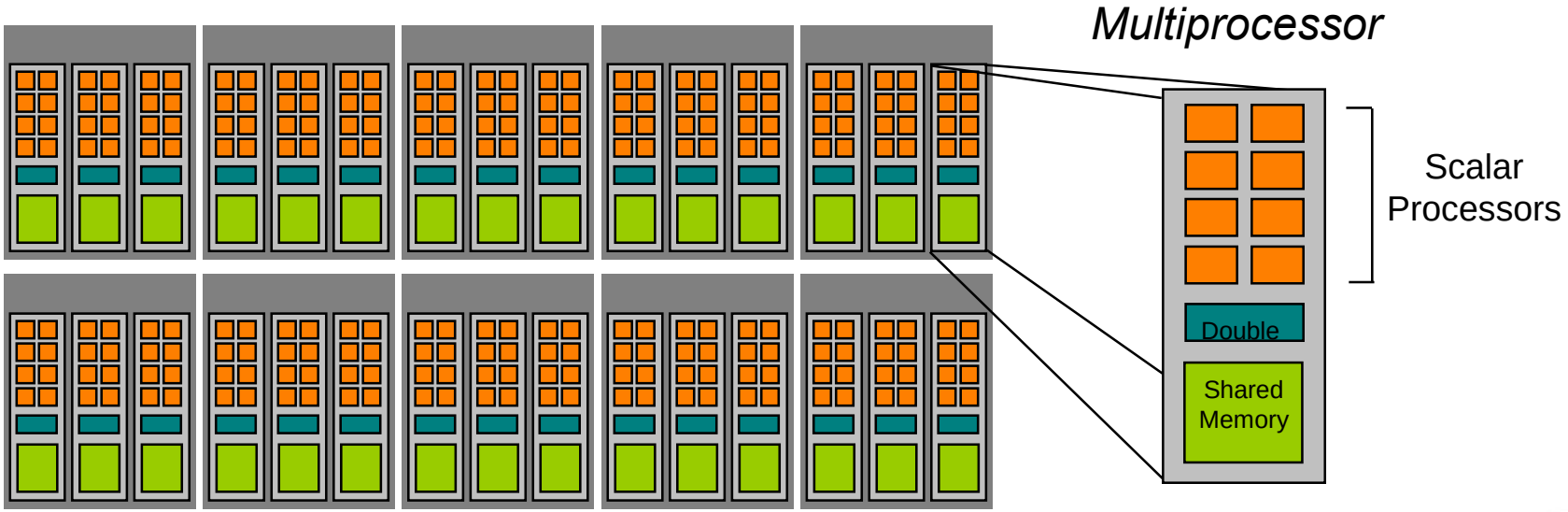
- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads
  - Many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Use registers and shared memory wisely

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance – effective bandwidth
    - Coalescing
    - Shared Memory
    - Textures
- Summary

10-Series Architecture

- 240 *Scalar Processor (SP) cores* execute kernel threads
- 30 Streaming *Multiprocessors (SMs)* each contain
  - 8 scalar processors
  - 2 Special Function Units (SFUs)
  - 1 double precision unit
  - *Shared memory* enables thread cooperation



Execution Model

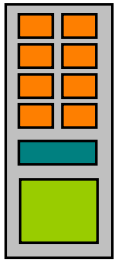
Software **Hardware**

Thread

Scalar Processor

Threads are executed by scalar processors

Thread Block

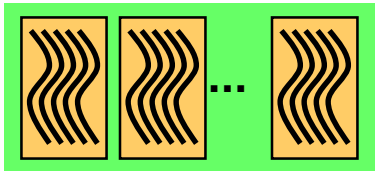


Multiprocessor

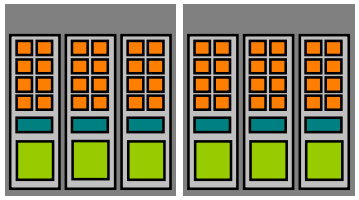
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



Grid



Device

A kernel is launched as a grid of thread blocks

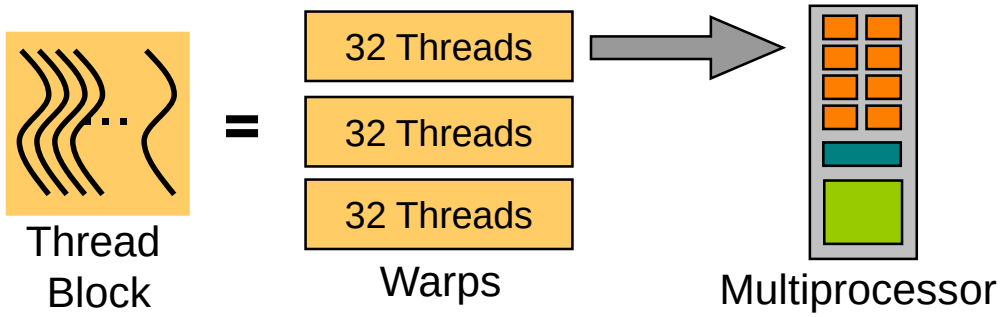
Concurrent kernel execution is possible (with newer Fermi devices)

Execution Model - Warps

- **Warp**: a group of threads executed *physically* in parallel
  - Warp is the unit of thread scheduling in SMs
  - Each block is further divided into warps for execution
  - Each warp consists of **32** threads of consecutive *threadIdx* values
  - Threads 0-31 form the first warp, 32-63 second warp, and so on
  - If block has a size not multiple of 32, the last block will be padded, i.e. if the block has 48 threads, warp0 will have 32 threads and warp 1 will be padded with 16 threads to have 32 threads

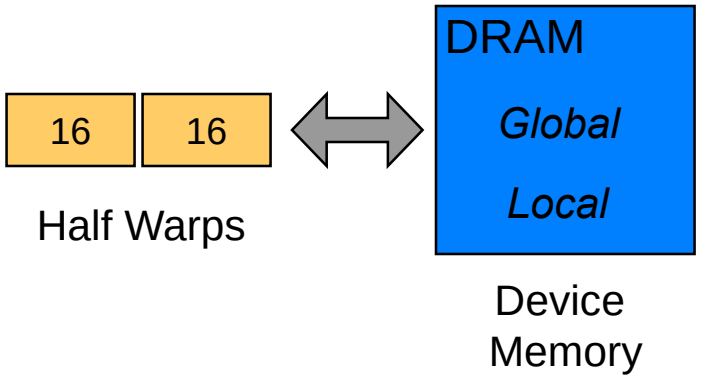
- A SM executes all threads in a warp following SIMD model
  - All threads in a warp will apply the same instruction to different parts of the data and they will have the same execution timing.

Warps and Half Warps



A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor



A half-warp of 16 threads can coordinate global memory accesses into a single transaction

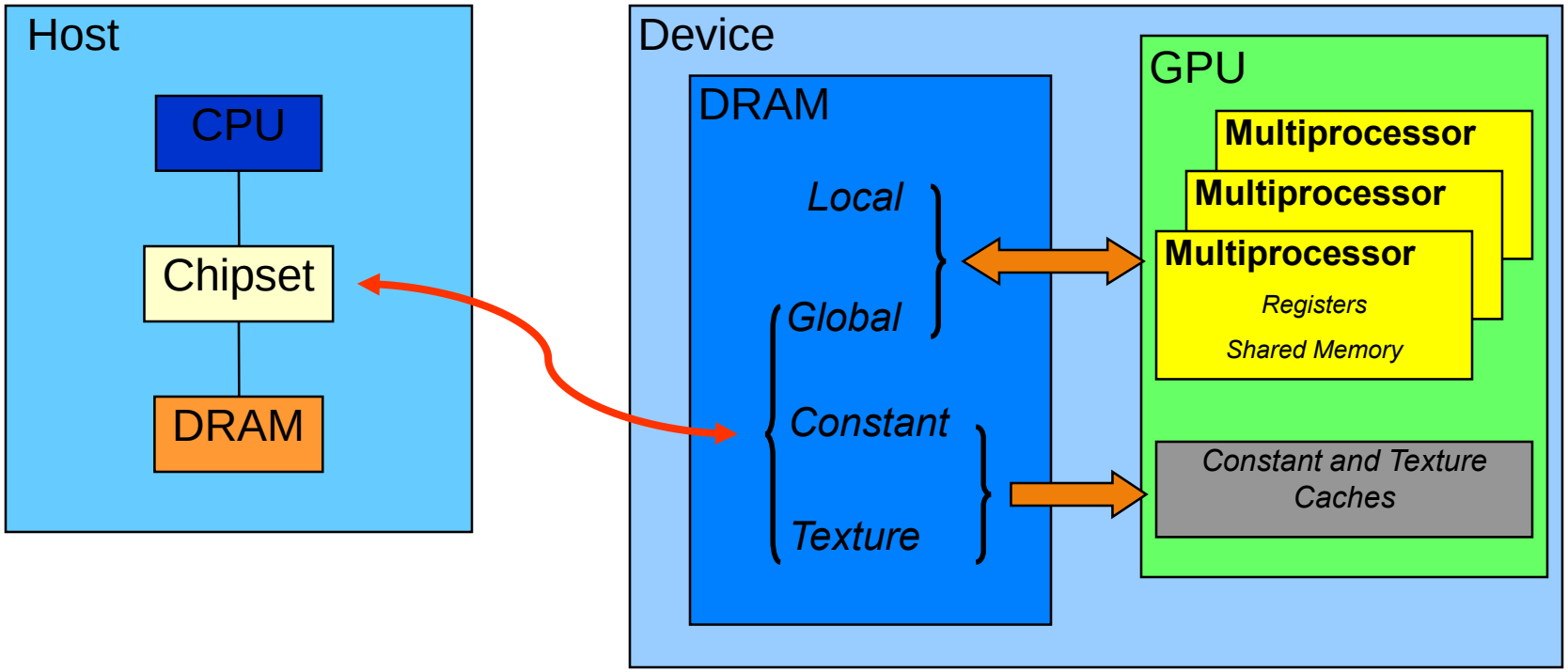
Latency Hiding

Why do we need so many warps in an SM for better occupancy?

Latency hiding

- If an instruction executed by the threads in a warp needs to wait for the result of previously initiated long latency operation, the warp is not selected for execution.
- Another resident warp that is no longer waiting for the results will be selected for execution.

Memory Architecture



Memory Architecture

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Fermi Architecture

The same on-chip memory is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call. Experimentation is recommended to find out the best combination for a given kernel

- Program configurable:
  - 16KB shared / 48 KB L1
  - 48KB shared / 16KB L1
- **`cudaDeviceSetCacheConfig()`**

  `cudaFuncCachePreferNone`

  no preference for shared memory or L1 (default)

  `cudaFuncCachePreferShared`

  prefer larger shared memory and smaller L1 cache

  `cudaFuncCachePreferL1`

  prefer larger L1 cache and smaller shared memory

Maxwell Architecture

- With Fermi and Kepler, shared memory and the L1 cache shared the same on-chip storage.
- Maxwell, by contrast, provides dedicated space to the shared memory of each SMM, since the functionality of the L1 and texture caches have been merged in SMM.
- This increases the shared memory space available per SMM as compared to SMX
  - GM107 provides 64 KB shared memory per SMM
  - GM204 further increases this to 96 KB shared memory per SMM.
- Applications no longer need to select a preference of the L1/shared split.
  - the preference will be ignored on Maxwell and shared memory will always be 64 KB per SMM

Declaration of memory types

- Variable decleration by __device__
  - Variable reside in global memory
  - Slow access – while the latency and throughput have been improved with caches in newer devices
  - It will be a global variable – visible to all threads of all kernels.
  - Can be used to make threads collaborate across blocks: Often used to pass information from one kernel invocation to another kernel invocation

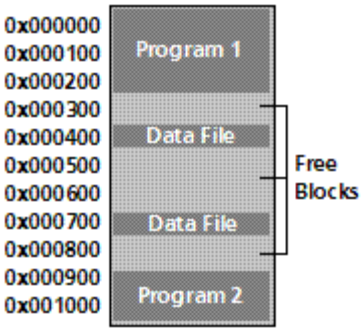
Declaration of memory types

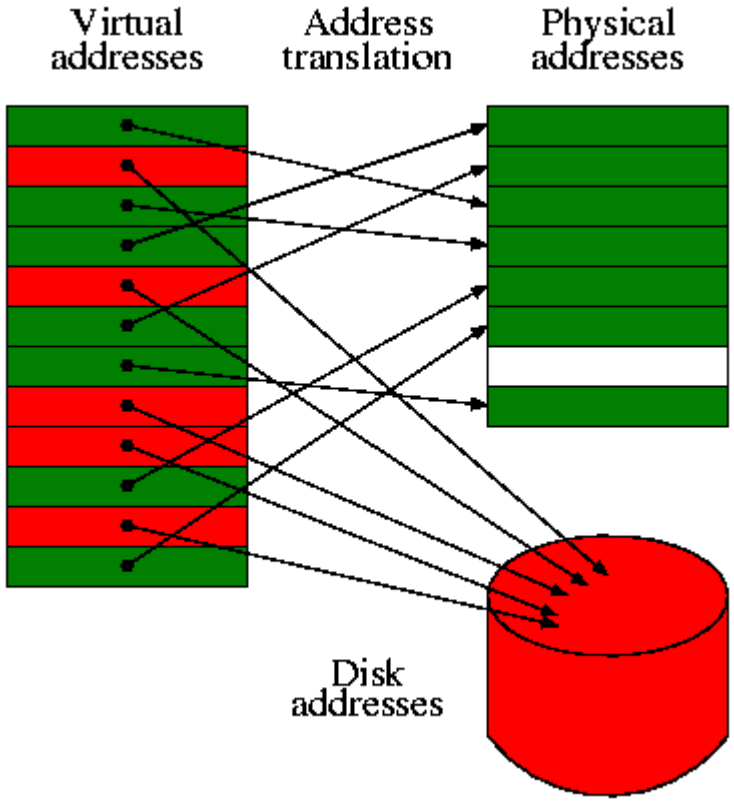
- Variable decleration preceeded by __shared__
  - Variable reside in shared memory
  - Scope is within a thread block – all threads in a block see the same version of the shared variable
  - When kernel exits, the contents of its shared memory cease to exist
- Variable decleration preceeded by __constant__
  - It must be declared outside any function body!
  - Scope is the whole grid – all threads in a grid see the same version of the constant variable
  - Stored in the global memory but cached for efficient access
  - Total size is limited to 65536 bytes
- Both could be preceeded with __device__, it will have the same effect.

Host-Device Data Transfers

- Device to host memory bandwidth much lower than device to device bandwidth
  - 8 GB/s peak (PCI-e x16 Gen 2) (Device to Host)
  - 141 GB/s peak (GTX 280) (Device to Device)
- Minimize transfers
  - Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory
- Group transfers
  - One large transfer much better than many small ones

Virtual Memory and Paging

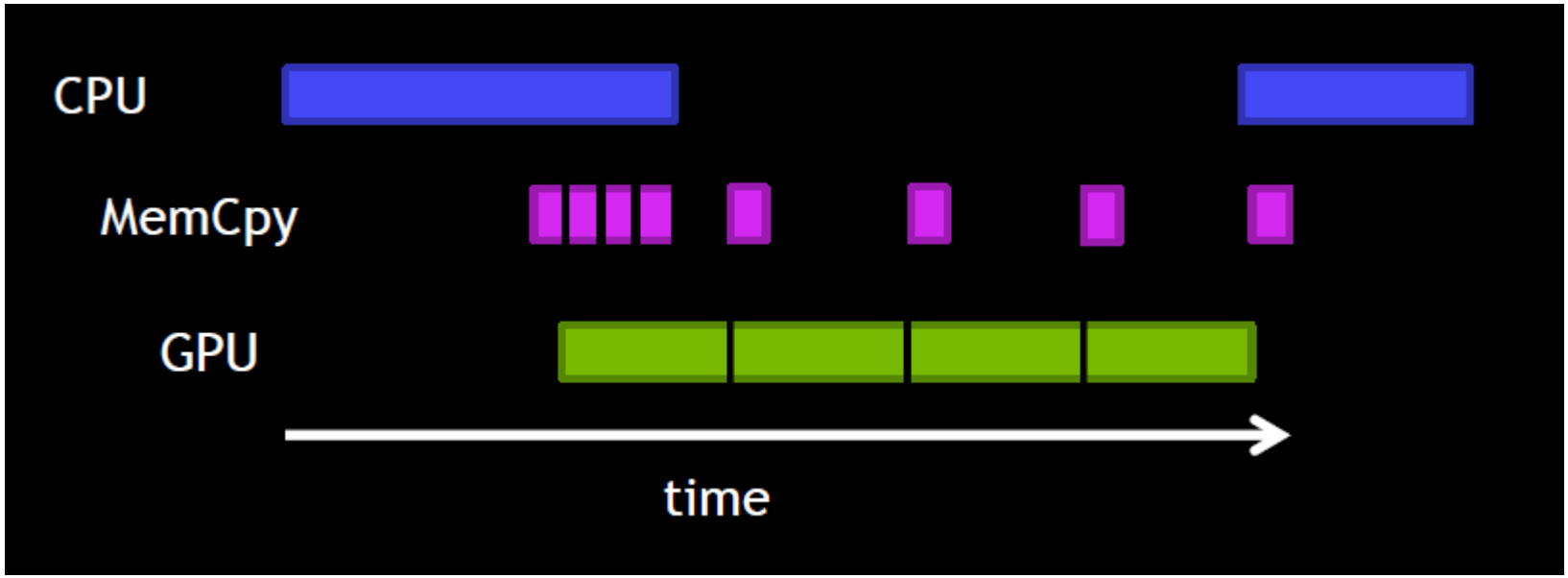




Page-Locked Data Transfers

- `cudaMallocHost()` allows allocation of page-locked ("pinned") host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2
- See the "bandwidthTest" CUDA SDK sample
- Use with caution!!
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits

Overlapping Data Transfers and Computation



Overlapping Data Transfers and Computation

- Async and Stream APIs allow overlap of H2D or D2H data transfers with computation
  - CPU computation can overlap data transfers on all CUDA capable devices
  - Kernel (GPU) computation can overlap data transfers on devices with “Concurrent copy and execution” (roughly compute capability >= 1.1)
- Stream = sequence of operations that execute in order on GPU
  - Operations from different streams can be interleaved
  - Stream ID used as argument to async calls and kernel launches

Asynchronous Data Transfers

- Asynchronous host-device memory copy returns control immediately to CPU
  - `cudaMemcpyAsync(dst, src, size, dir, stream);`
  - requires *pinned* host memory (allocated with "`cudaMallocHost`")
- Overlap CPU computation with data transfer
  - `0` = default stream

```
cudaMemcpyAsync(a_d, a_h, size,
    cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

overlapped

Overlapping kernel and data transfer

- "Concurrent copy and execute"
  - `deviceOverlap` field of a `cudaDeviceProp` variable
- Kernel and transfer use different, *non-zero* streams
  - A CUDA call to stream-0 blocks until all previous calls complete and cannot be overlapped

- **Example:**

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
```

Overlapping kernel and data transfer

- **cudaHostAlloc(void** pHost, size_t size, int flags)**
  (or use cudaMallocHost, it is effectively the same)
- **cudaFree(void * pHost)**
  - deallocates pinned (page-locked) CPU memory
- **cudaHostRegister(void* pHost, size_t size, int flags)**
- **cudaHostUnregister(void* pHost)**
  - pins/unpins previously allocated memory

GPU/CPU Synchronization

- Context based
  - `cudaDeviceSynchronize()`
    - Blocks until all previously issued CUDA calls from a CPU thread complete
- Stream based
  - `cudaStreamSynchronize(stream)`
    - Blocks until all CUDA calls issued to given stream complete
  - `cudaStreamQuery(stream)`
    - Indicates whether stream is idle
    - Returns `cudaSuccess`, `cudaErrorNotReady, ...`
    - Does not block CPU thread

GPU/CPU Synchronization

Stream based using events

- Events can be inserted into streams:

```
cudaEventRecord(event, stream)
```

- Event is recorded when GPU reaches it in a stream
  - Recorded = assigned a timestamp (GPU clocktick)
  - Useful for timing

- `cudaEventSynchronize(event)`
  - Blocks until given event is recorded

- `cudaEventQuery(event)`
  - Indicates whether event has recorded
  - Returns `cudaSuccess`, `cudaErrorNotReady, ...`
  - Does not block CPU thread

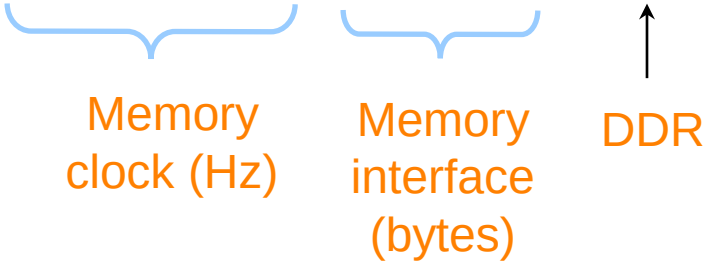
GTX 280 Specs

CUDA Cores	**240**
Graphics Clock (MHz)	**602 MHz**
Processor Clock (MHz)	**1296 MHz**
Texture Fill Rate (billion/sec)	**48.2**
Memory Type	**DDR2**
Memory Clock (MHz)	**1107 MHz**
Standard Memory Config	**1 GB**
Memory Interface Width	**512-bit**
Memory Bandwidth (GB/sec)	**141.7**

DDR*x*: Double Data Rate (x2 throughput), *x*: version (no direct effect on throughput)

Theoretical Bandwidth

- Device Bandwidth of GTX 280
  - $1107 * 10^6 * (512 / 8) * 2 / 1024^3 = 131.9$ GB/s

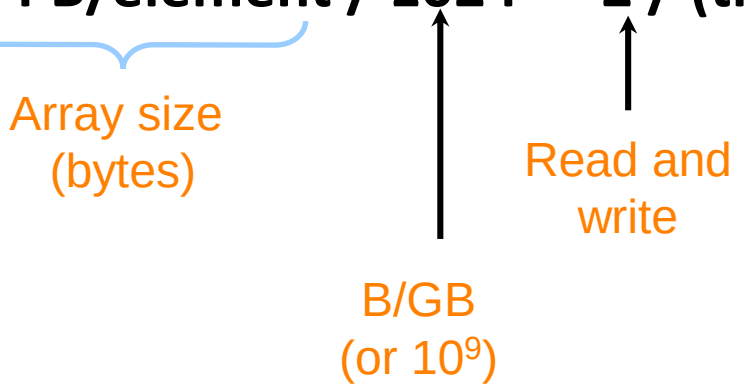


  - Specs report 141 GB/s
    - Use 10^9 B/GB conversion rather than 1024^3
    - Whichever you use, be consistent

Memory Type	DDR2
Memory Clock (MHz)	1107 MHz
Standard Memory Config	1 GB
Memory Interface Width	512-bit
Memory Bandwidth (GB/sec)	141.7

Effective Bandwidth

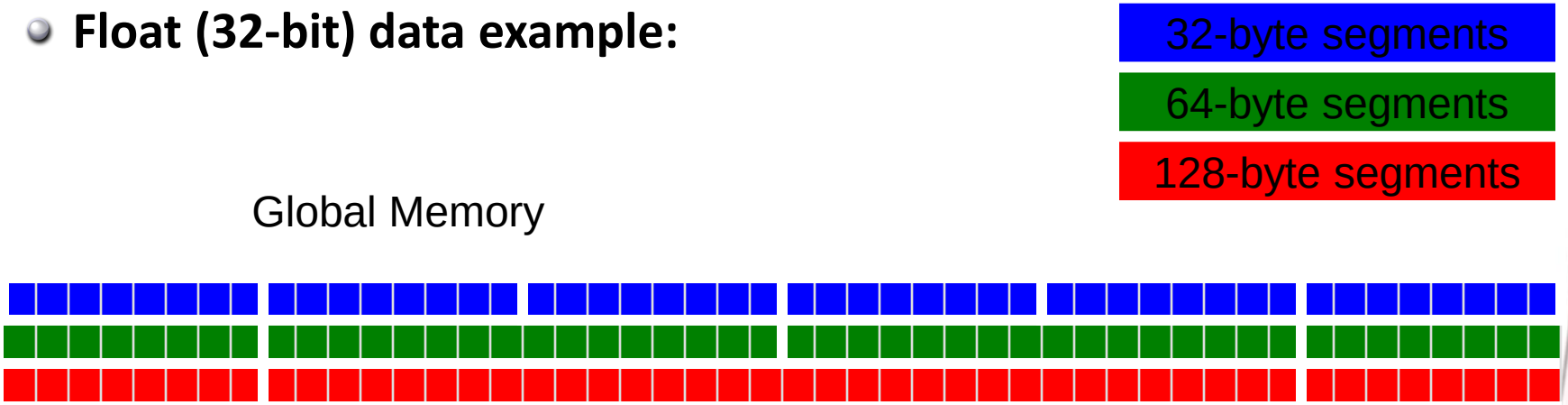
- **Effective Bandwidth (for copying array of N floats)**
  - **N * 4 B/element / 1024^3 * 2 / (time in secs) = GB/s**



Coalescing

Coalasce: to unite so as to form one mass (Dictionary.com)
-Bir araya gelmek, birleşmek-

- **Global memory access of 32, 64, or 128-bit (4B, 8B, or 16B) words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met**
- **Depends on compute capability**
  - **1.0 and 1.1 have stricter access requirements**
- **Float (32-bit) data example:**

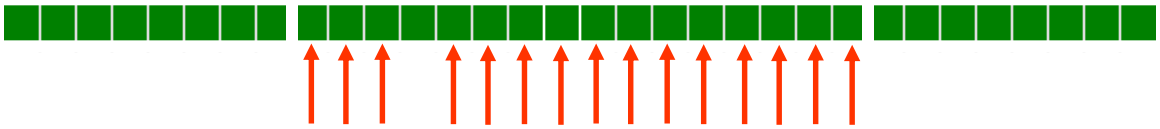


Coalescing

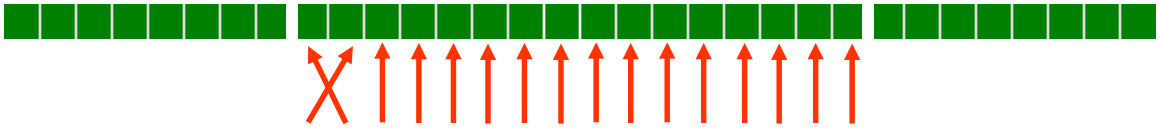
Compute capability 1.0 and 1.1

- K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate

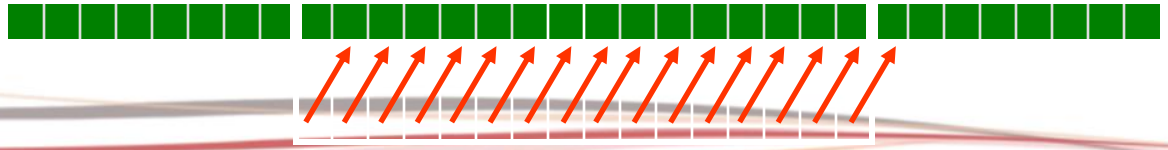
Coalesces – 1 transaction



Out of sequence – 16 transactions



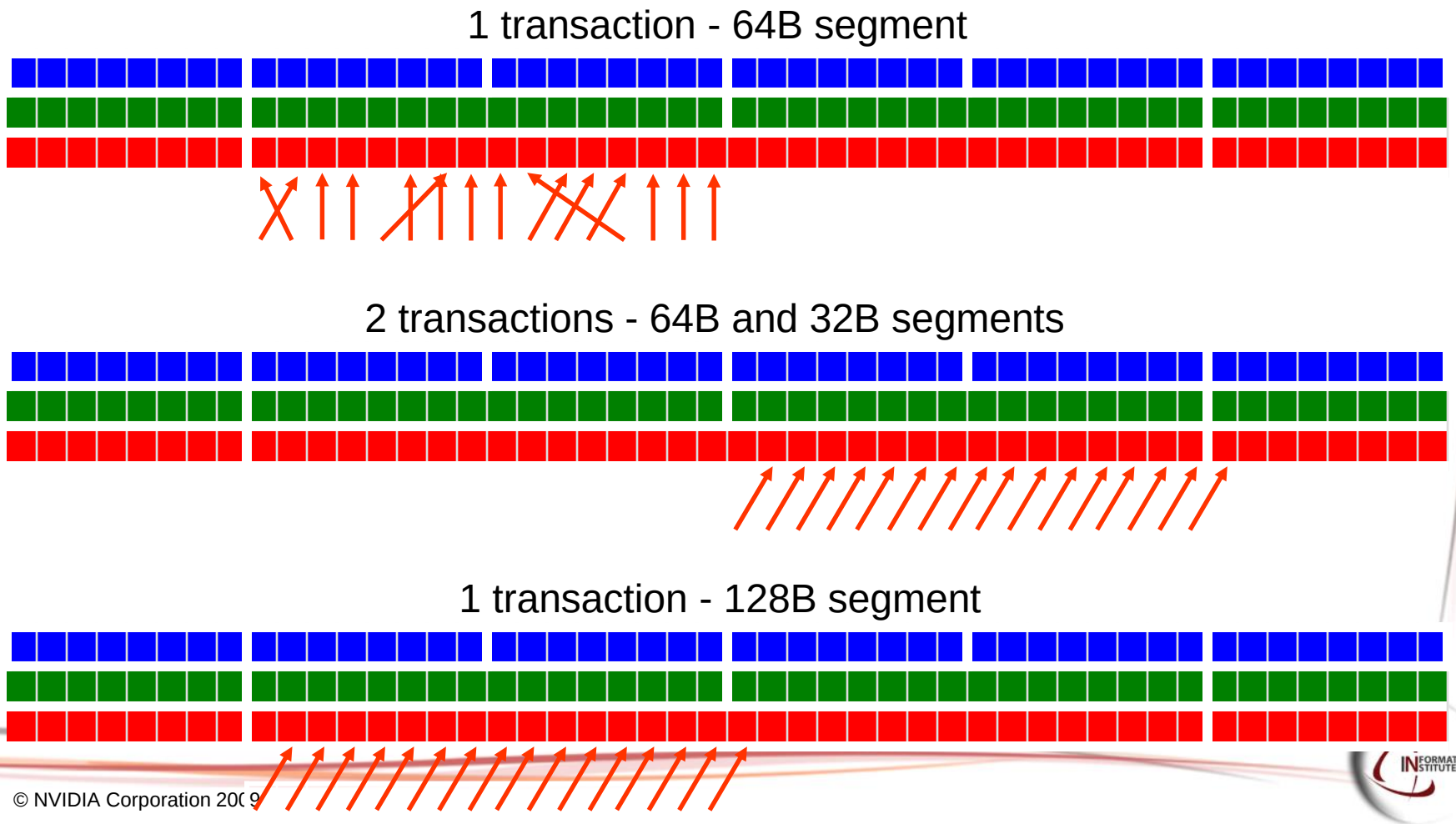
Misaligned – 16 transactions



Coalescing

Compute capability 1.2 and higher

- Issues transactions for segments of 32B, 64B, and 128B
- Smaller transactions used to avoid wasted bandwidth



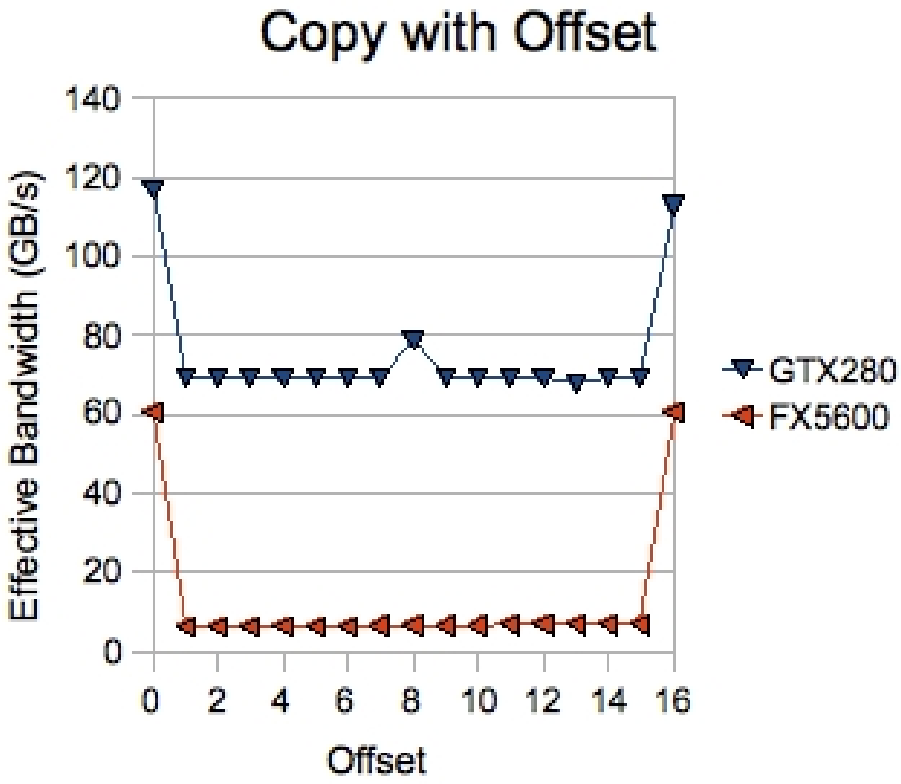
Coalescing Examples

- Effective bandwidth of small kernels that copy data
  - Effects of offset and stride on performance

- Two GPUs
  - GTX 280
    - Compute capability 1.3
    - Peak bandwidth of 141 GB/s
  - FX 5600
    - Compute capability 1.0
    - Peak bandwidth of 77 GB/s

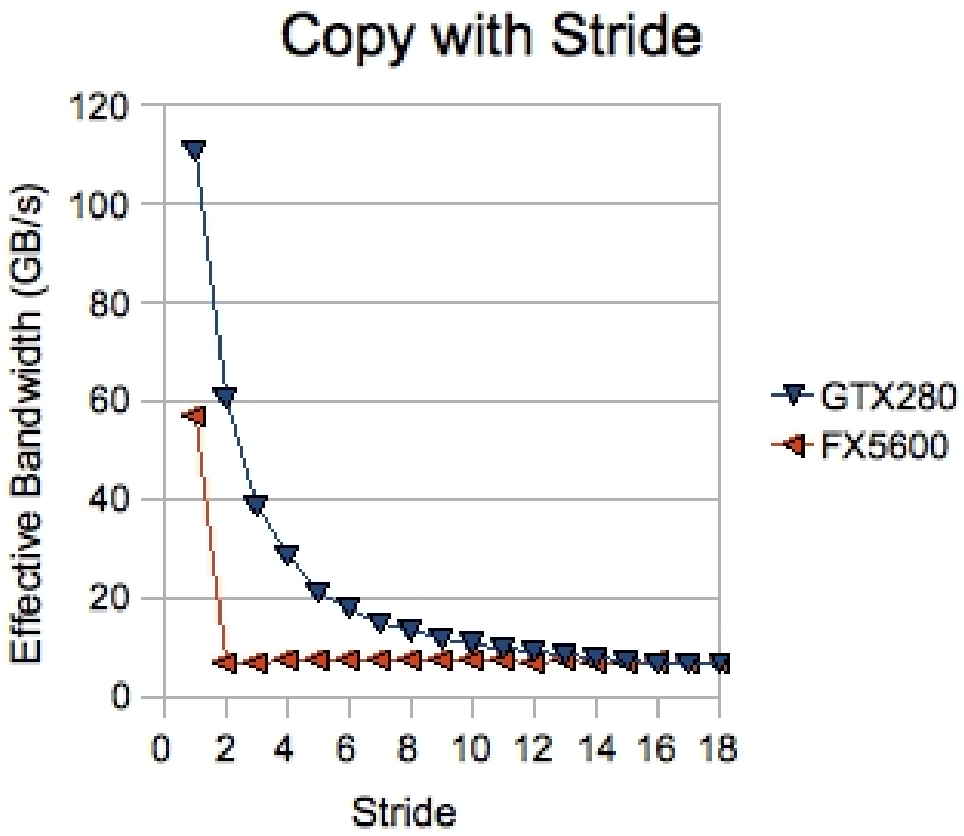
Coalescing Examples

```
__global__ void offsetCopy(float *odata, float *idata,
                           int offset)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
  odata[xid] = idata[xid];
}
```



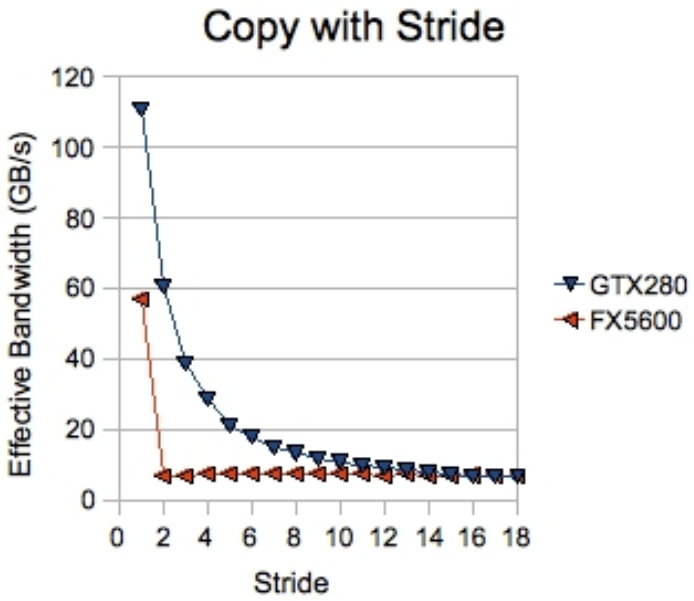
Coalescing Examples

```
__global__ void strideCopy(float *odata, float *idata,
                           int stride)
{
  int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
  odata[xid] = idata[xid];
}
```



Coalescing Examples

- Strided memory access is inherent in many multidimensional problems
  - Stride is generally large (>> 18)



However …

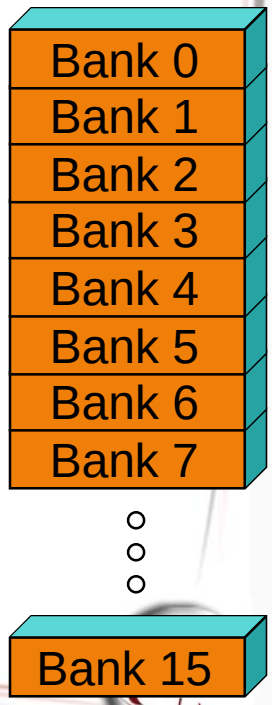
- Strided access to ***global memory*** can be avoided by using ***shared memory***

Shared Memory

- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

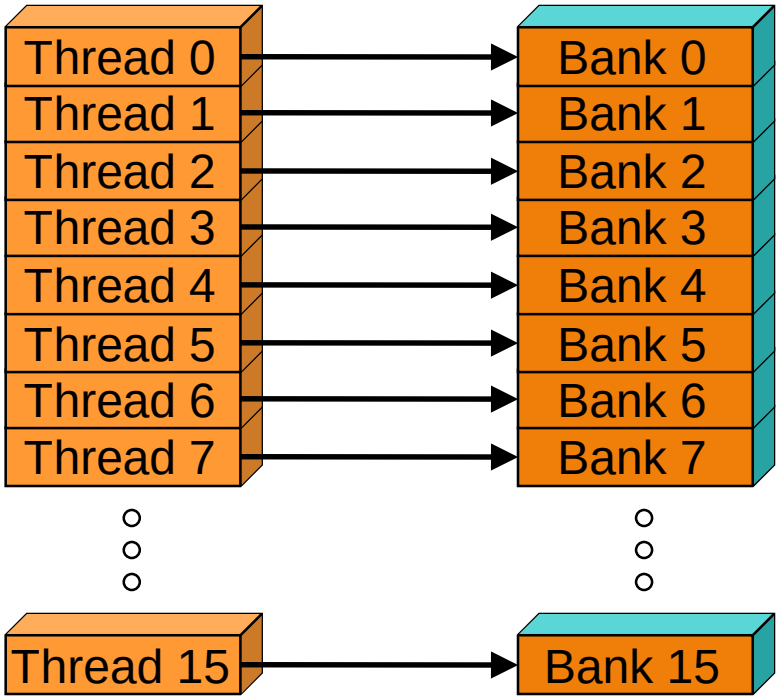
Shared Memory Architecture

- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

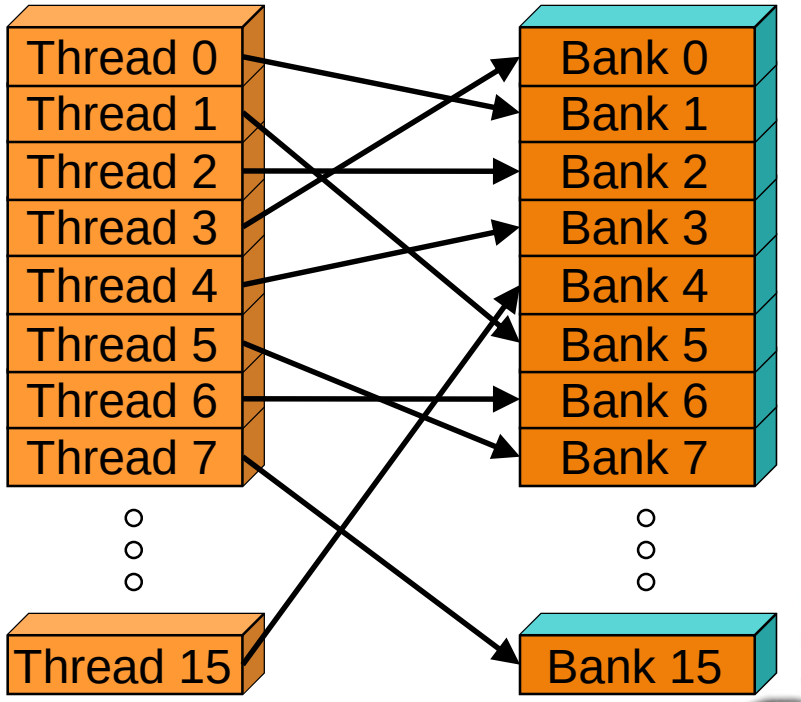


Bank Addressing Examples

- **No Bank Conflicts**
  - **Linear addressing stride == 1**

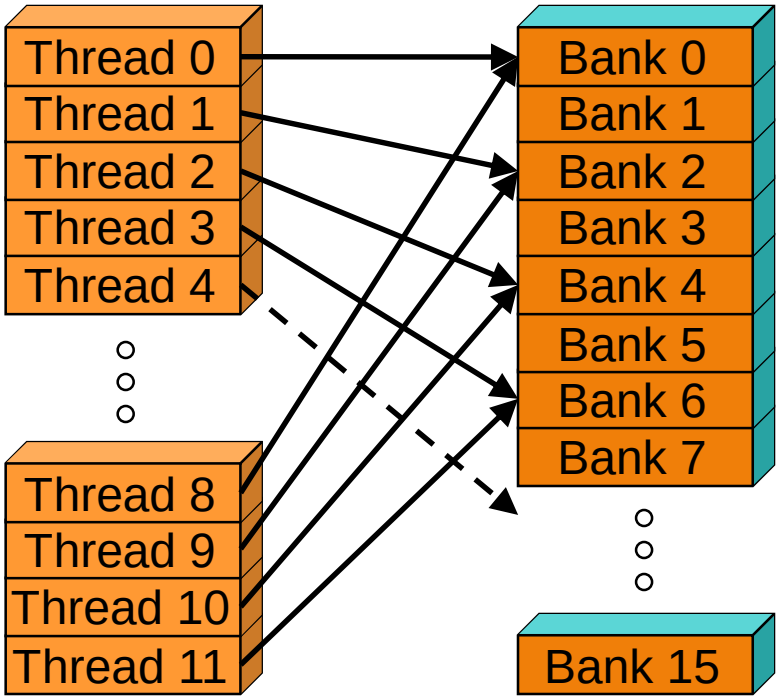


- **No Bank Conflicts**
  - **Random 1:1 Permutation**

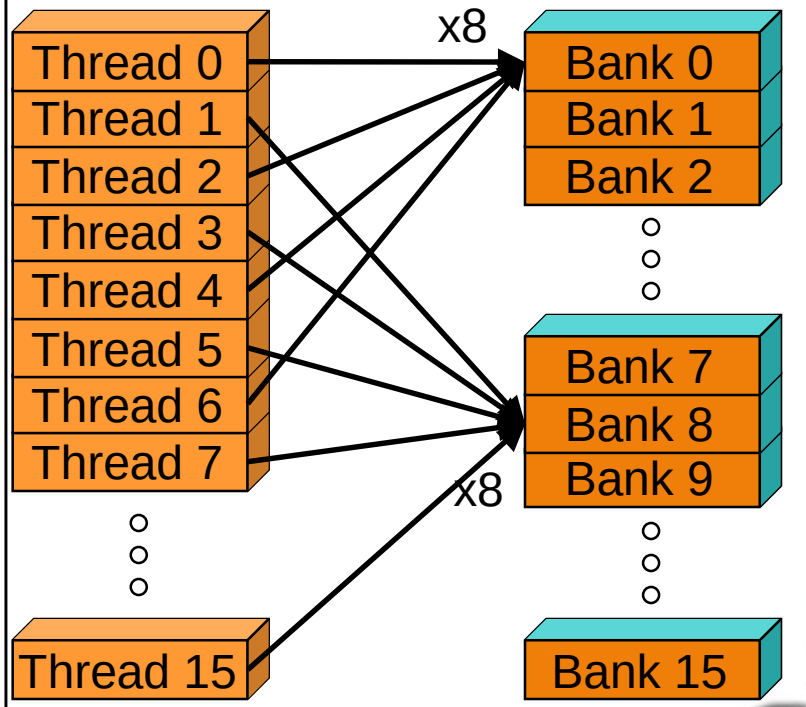


Bank Addressing Examples

- **2-way Bank Conflicts**
  - **Linear addressing stride == 2**



- **8-way Bank Conflicts**
  - **Linear addressing stride == 8**



Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts
- warp_serialize profiler signal reflects conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

Shared Memory Multi-cast
Fermi and Newer Architectures

- Devices of compute capability 2.0 and higher have the additional ability to multicast shared memory accesses
- Multiple accesses to the same **location** by any number of threads within a warp are served simultaneously.

Shared Memory Bank Conflicts
Fermi and Kepler Architectures

- Shared memory has 32 banks
  - Successive 32-bit words are assigned to successive banks
- Bandwidth:
  - Fermi: Each bank 32-bits per two clock cycles
  - Kepler: Each bank 64-bits per clock cycle

Shared Memory Configuration Kepler Architecture

- Has 32-bit and 64-bit modes
  - Changes the bank conflict behaviour accordingly
- Set shared memory configuration for a device function:

```
__host__ cudaError_t CUDARTAPI cudaFuncSetSharedMemConfig    (
        const char *  func,
        enum cudaSharedMemConfig   config
)
```

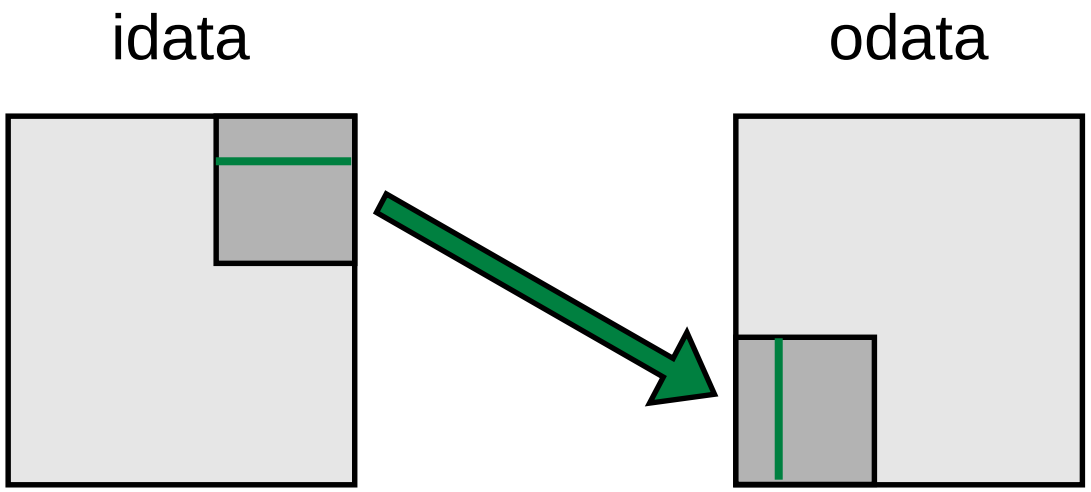
- cudaSharedMemBankSizeFourByte - shared memory bank width is four bytes (32-bit).
- cudaSharedMemBankSizeEightByte - shared memory bank width is eight bytes (64-bit).

Shared Memory Configuration

- Kepler SMX introduced an optional 8-byte shared memory banking mode, which had the potential to increase shared memory bandwidth per SM over Fermi for shared memory accesses of 8 or 16 bytes. However, applications could only benefit from this when storing these larger elements in shared memory (i.e., integers and fp32 values saw no benefit), and only when the developer explicitly opted into the 8-byte bank mode via the API.
- To simplify this, Maxwell returns to the Fermi style of shared memory banking, where banks are always four bytes wide. Aggregate shared memory bandwidth across the chip remains comparable to that of corresponding Kepler chips, given increased SM count.
- Read more at: http://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html#ixzz3GgCvZVnB

Shared Memory Example: Transpose

- Each thread block works on a tile of the matrix
- Naive implementation exhibits strided access to global memory

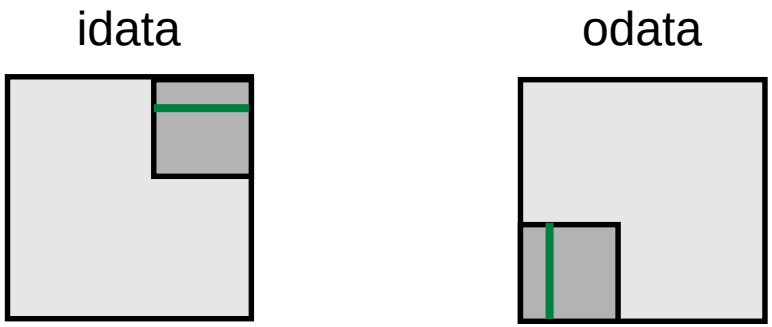


Elements transposed by a half-warp of threads

Naïve Transpose

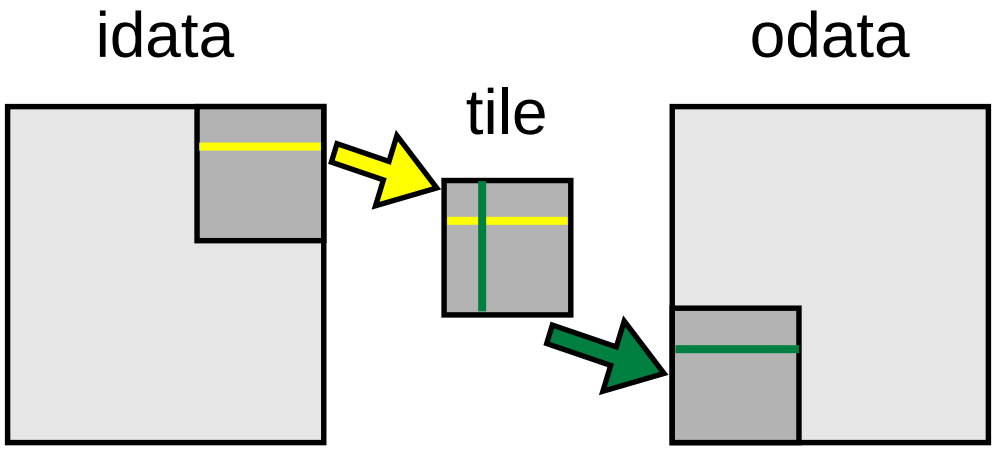
- Loads are coalesced, stores are not (strided by height)

```
__global__ void transposeNaive(float *odata, float *idata,
                               int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;

  odata[index_out] = idata[index_in];
}
```



Coalescing through shared memory

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored by other threads



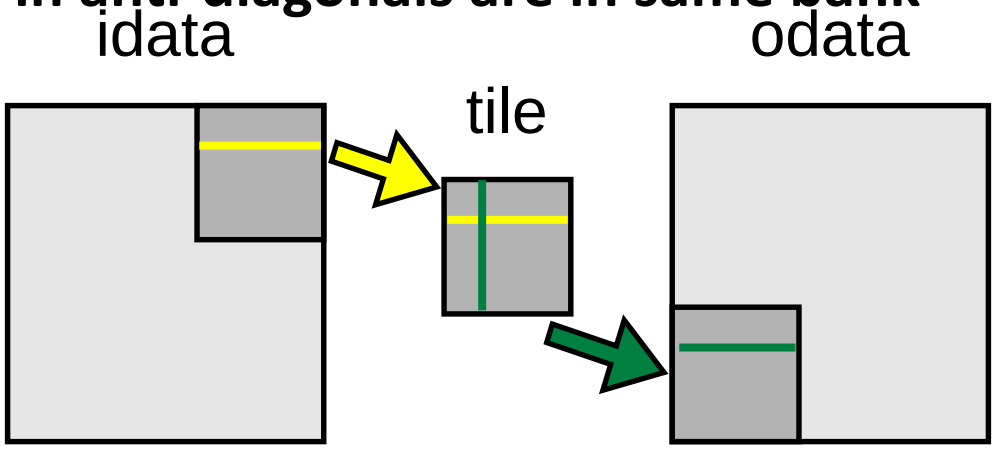
Elements transposed by a half-warp of threads

Coalescing through shared memory

```
__global__ void transposeCoalesced(float *odata, float *idata,
                                   int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  tile[threadIdx.y][threadIdx.x] = idata[index_in];

  __syncthreads();

  odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Bank Conflicts in Transpose

- 16x16 shared memory tile of floats
  - Data in columns are in the same bank
  - 16-way bank conflict reading columns in tile
- Solution - pad shared memory array
  - `__shared__ float tile[TILE_DIM][TILE_DIM+1];`
  - Data in anti-diagonals are in same bank



Effective Use of Memory and Optimization - II

Dr. Alptekin Temizel
atemizel@metu.edu.tr

Zero Copy

- Zero Copy is a way to map host memory and access it directly over PCIe without doing an explicit memory transfer. It allows CUDA kernels to directly access host memory.

- Instead of reading data from global memory (limit ~200GB/s), data would be read over PCIe and be limited by PCIe bandwidth (upto 16GB/s). Hence there was no real performance advantage for most applications.

Zero Copy

- However, this could be advantageous with integrated devices and devices like Jetson TK1/TX1/TX2.

- The Jetson boards have physical memory (8GB in TX2) that is shared by the ARM CPU and the CUDA GPU.

- If a cudaMemcpy Host->Device is done on the Jetson, the memory is simply being copied to a new location on the same physical memory and retrieving a CUDA pointer from it.

Zero Copy

- Zero copy will always be a win for integrated devices that utilize CPU memory, you can check this:

```
integrated cudaDeviceProp
```

- Potentially easier and faster alternative to using `cudaMemcpyAsync`
  - For example, can both read and write CPU memory from within one kernel

- Note that most devices use pointers that are 32-bit so there is a limit of *4GB per context*

- But we now have Unified Memory feature as well!

Zero Copy

- **Check `canMapHostMemory` field of `cudaDeviceProp` variable**
- **See the example simpleZeroCopy in SDK**

```
cudaSetDeviceFlags(cudaDeviceMapHost);
...
cudaHostAlloc((void **)&a_h, nBytes,cudaHostAllocMapped);
cudaHostGetDevicePointer((void **)&a_d, (void *)a_h, 0);
for (i=0; i<N; i++) a_h[i] = i;
increment<<<grid, block>>>(a_d, N)
```

Constant Memory

- Available since compute capability 1.0.
- Data in constant memory:
  - Resides in a 64KB partition of device memory
  - Is accessed through an 8KB cache on each SM(X)
  - Is intended to be broadcast to all threads in a warp

Constant Memory

- If all the threads in the warp request the same value, that value is delivered to all threads in a single cycle.
- If the threads in a warp request N different values, the request is serialized and the N values are delivered one at a time over N clock cycles.
- Make sure that your indexes into constant memory arrays are not functions of threadIdx.x.

Read Only Cache

- The read-only data cache was introduced with Compute Capability 3.5 architectures (e.g. Tesla K20c/K20X and GeForce GTX Titan/780 GPUs).

- Similar functionality has been available since Compute Capability 1.0 devices - although you needed to use the textures to take advantage of it.

Read Only Cache

- Each SMX has a 48KB read-only cache.
- The CUDA compiler automatically accesses data via the read-only cache when it can determine that data is read-only for the lifetime of your kernel.
- In practice, you need to qualify pointers as const and __restrict__ before the compiler can satisfy this condition.
- You can also specify a read-only data cache access with the __ldg() intrinsic function.

Usage Examples

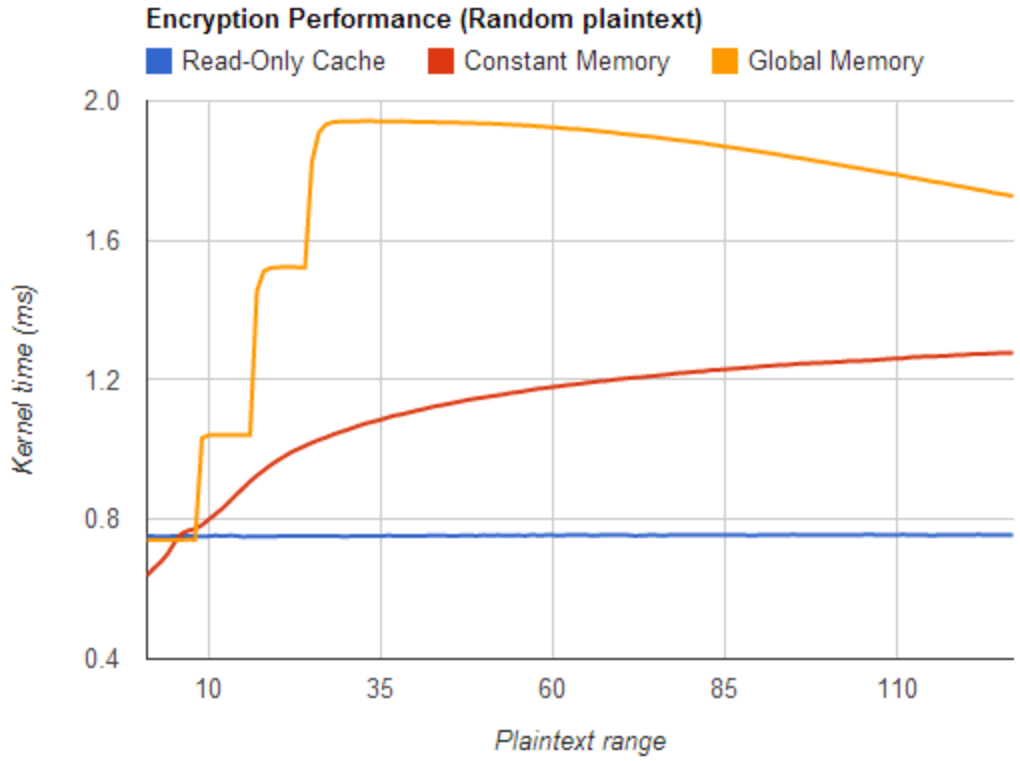
http://acceleware.com/blog/constant-cache-vs-read-only-cache-part-2?source=cuda-newsletter-1013

```
__global__ void SubstitutionEncrypt_ReadOnlyCache(const int * __restrict__
plaintext,  const int * __restrict__ substTable,
                int * __restrict__ ciphertext)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    ciphertext[index] = __ldg(&substTable[plaintext[index]]);
}

__global__ void SubstitutionEncrypt_ConstantMemory(const int * __restrict__
plaintext,   int * __restrict__ ciphertext)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    ciphertext[index] = substTable_c[plaintext[index]];
}

__global__ void SubstitutionEncrypt_GlobalMemory(int * plaintext, int *
substTable, int * ciphertext)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    ciphertext[index] = substTable[plaintext[index]];
}
```

Usage Examples

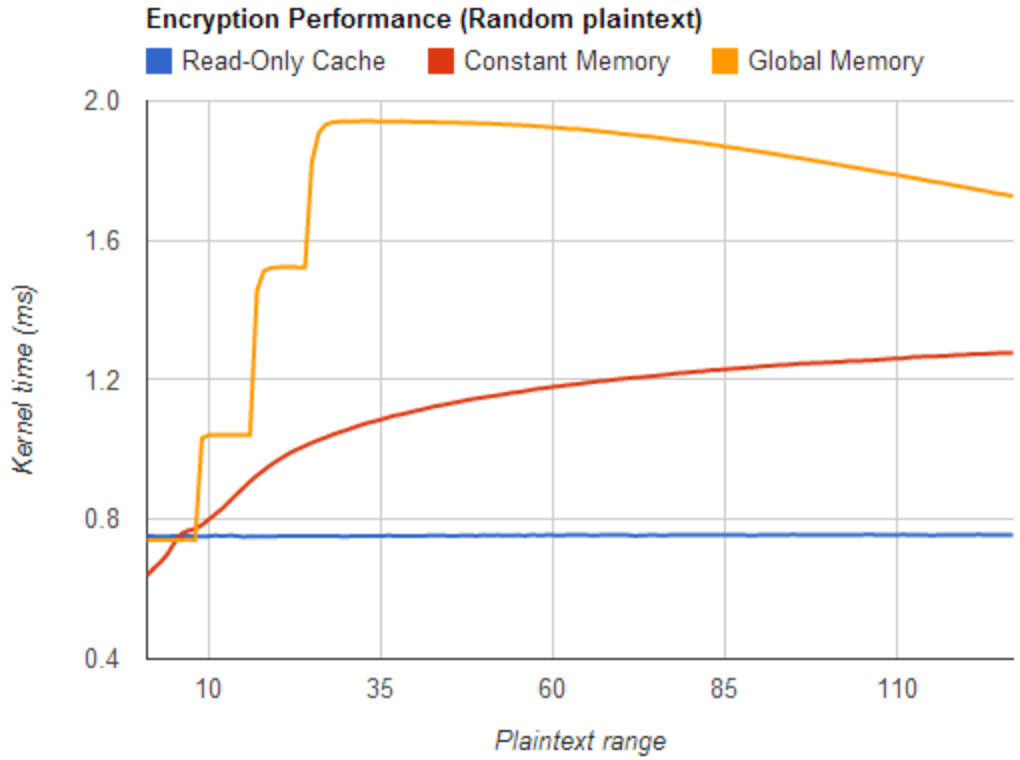
- It is important to note that the memory access pattern to the substitution table is data-dependent. By varying the range of ‘letters’ [M] in the plaintext, we can examine differences in performance between the three memory spaces.
- The kernels were compiled for Compute Capability 3.5, using the CUDA 5.5 compiler.
- Results were collected on a Tesla K20c GPU.
- The average of 5 iterations (ignoring initial CUDA module load time) for various plaintext ranges (M = 1 to 16384) on a plaintext of length 10 240 000 are recorded.

Usage Examples



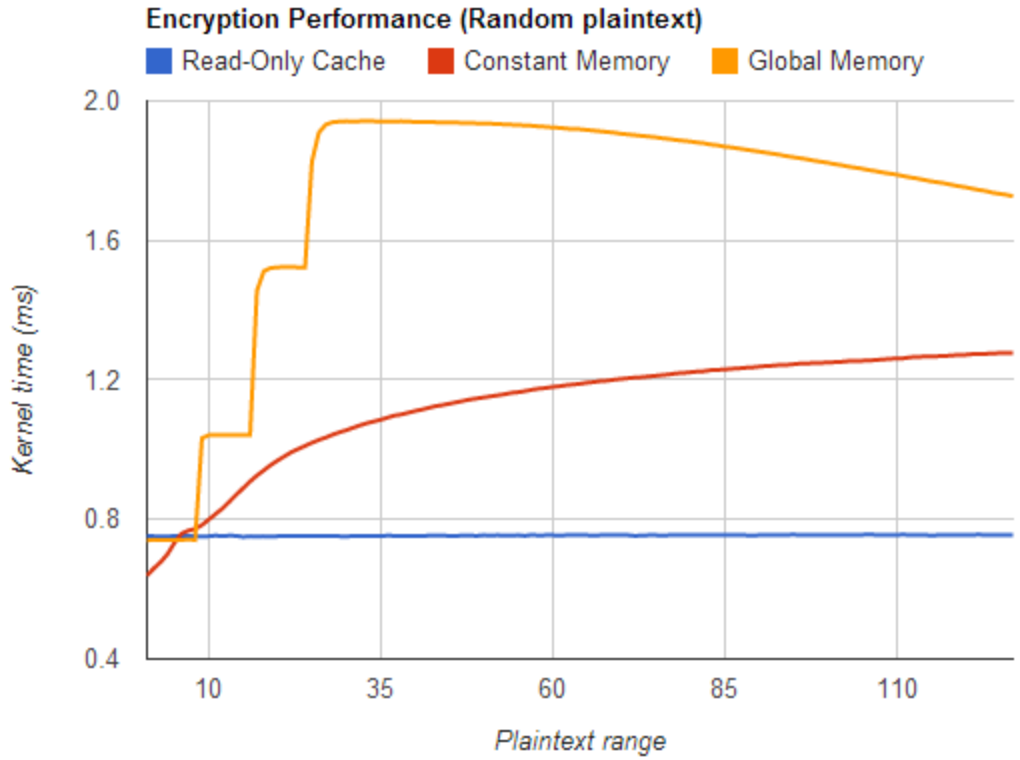
•For M = 1, all threads read the same value from memory, and the broadcast mechanism on the constant memory outperforms both the read-only cache and global memory.

Usage Examples



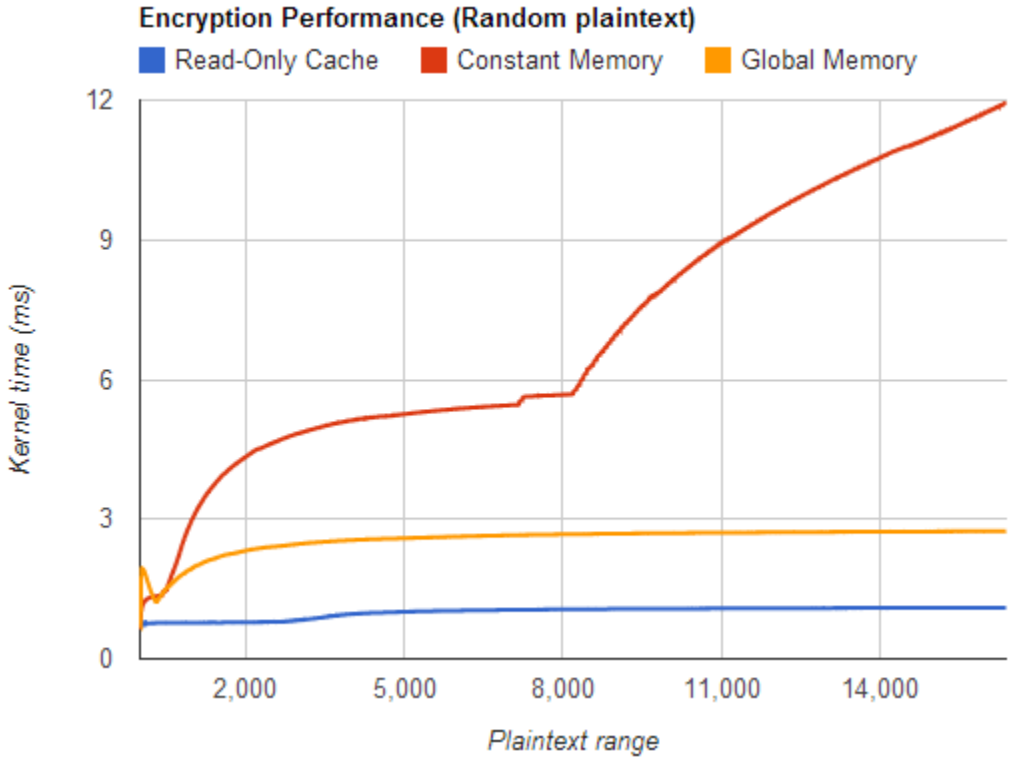
•Constant memory performs faster than both the read-only cache and global memory up to M = 5 inclusively, after which the serialized access to the constant memory elements hinders performance.

Usage Examples



•The important piece of information to take away from this is that constant memory should still considered an option even if the memory access pattern is not a pure broadcast -as long as the access is limited to a small range of addresses-.

Usage Examples



•For sequential plaintext, larger values of M exhibit a repetitive pattern that repeats every multiple of 128, however kernel times remain stable, averaging at about 3.5 ms.

•For randomized plaintext, larger values of M show a dramatic performance decrease compared to the sequential input, and kernel times continue to increase as M increases.

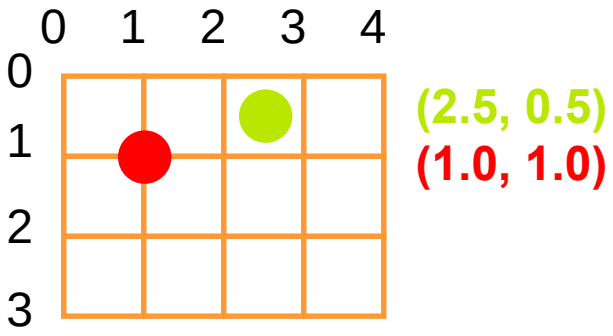
Textures in CUDA

- Texture is an object for *reading* data
- Benefits:
  - Data is cached
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear interpolation
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable in 1D, 2D, or 3D
    - Using integer or normalized coordinates

Textures in CUDA

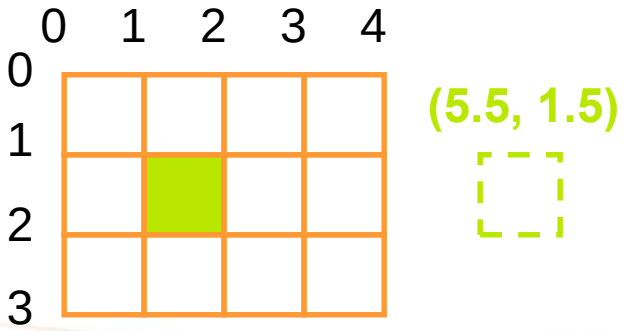
- On devices of compute capability 1.x, some kernels can achieve a speedup when using (cached) texture fetches rather than regular global memory loads (e.g., when the regular loads do not coalesce well). Unless texture fetches provide other benefits such as address calculations or texture filtering
- This optimization can be counterproductive on devices of compute capability 2.x, however, since global memory loads are cached in L1 and the L1 cache has higher bandwidth than the texture cache

Texture Addressing



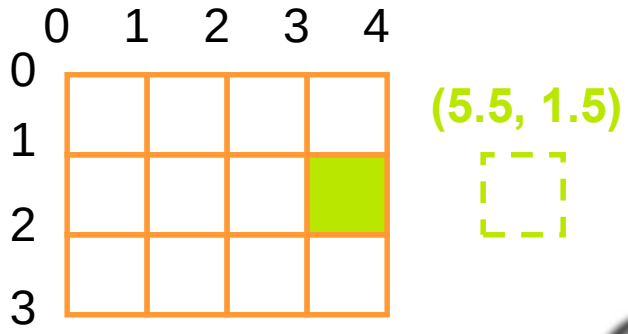
Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



Clamp

- Out-of-bounds coordinate is replaced with the closest boundary



CUDA Texture Types

- Bound to linear memory
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes
- Bound to CUDA arrays
  - Block linear CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- Bound to pitch linear (CUDA 2.2)
  - Global memory address is bound to a texture
  - 2D
  - Float/integer addressing, filtering, and clamp/repeat addressing modes similar to CUDA arrays

CUDA Texturing Steps

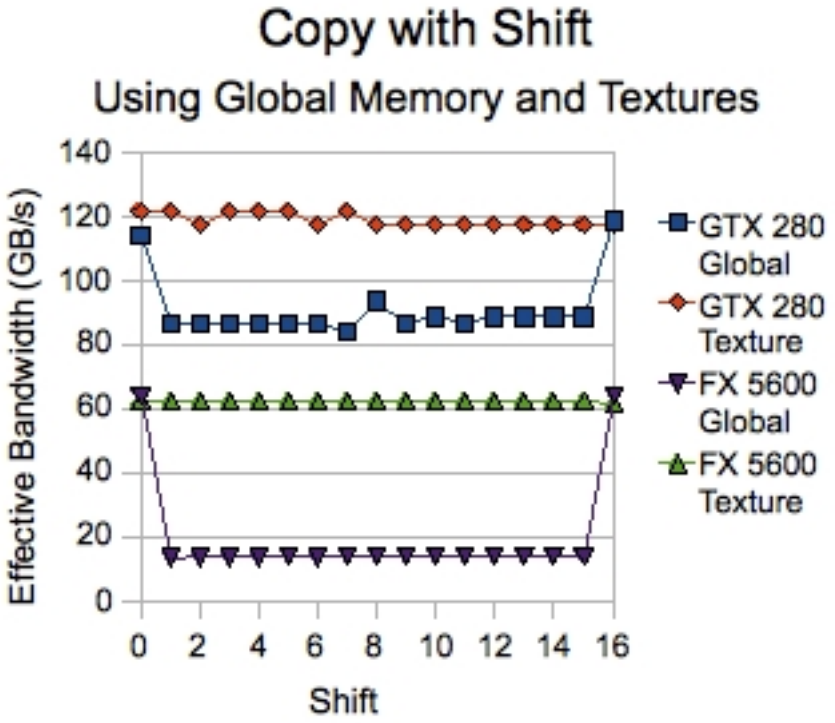
- Host (CPU) code:
  - Allocate/obtain memory
    - global linear/pitch linear, or CUDA array
  - Create a texture reference object
    - Currently must be at file-scope
  - Bind the texture reference to memory/array
  - When done:
    - Unbind the texture reference, free resources

- Device (kernel) code:
  - Fetch using texture reference
  - Linear memory textures: tex1Dfetch()
  - Array textures: tex1D() or tex2D() or tex3D()
  - Pitch linear textures: tex2D()

Texture Example

```
__global__ void
shiftCopy(float *odata,
          float *idata,
          int shift)
{
  int xid = blockIdx.x * blockDim.x
          + threadIdx.x;
  odata[xid] = idata[xid+shift];
}
```

```
texture <float> texRef;

__global__ void
textureShiftCopy(float *odata,
                 float *idata,
                 int shift)
{
  int xid = blockIdx.x * blockDim.x
          + threadIdx.x;
  odata[xid] = tex1Dfetch(texRef, xid+shift);
}
```



An Example from SDK (main)

```
// #includes etc omitted
// declare texture at file scope
texture<float, 2, cudaReadModeElementType> mytex;
#include <simpleTexture_kernel.cu>
int main( int argc, char *argv[])
{
// code to read image and set size,width, height & angle ommitted
// allocate CudaArray
cudaChannelFormatDesc channelDesc =
cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);
cudaArray* cu_array;
cudaMallocArray( &cu_array, &channelDesc, width, height );
```

An Example from SDK (main)

```
// copy image to device array cu_array – used as texture mytex on device
cudaMemcpyToArray( cu_array, 0, 0, h_data, size,
    cudaMemcpyHostToDevice);
// set texture parameters
mytex.addressMode[0] = cudaAddressModeWrap;
mytex.addressMode[1] = cudaAddressModeWrap;
mytex.filterMode = cudaFilterModeLinear;
mytex.normalized = true; //NB coordinates in [0,1]
// Bind the array to the texture
cudaBindTextureToArray( mytex, cu_array, channelDesc);
// allocate device memory for result
float* d_data = NULL;
cudaMalloc( (void**) &d_data, size);
```

An Example from SDK (main)

```
// Invoke kernel
dim3 dimBlock(8, 8, 1);
dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
transformKernel<<< dimGrid, dimBlock, 0 >>>( d_data, width, height,
    angle);
// copy result from device to host
cudaMemcpy( h_data, d_data, size, cudaMemcpyDeviceToHost);
// free memory & exit
cudaFree(d_data);
cudaFreeArray(cu_array);
return 0;
} // end main
```

An Example from SDK (kernel)

```
__global__ void
transformKernel( float* g_odata, int width, int height, float theta)
{
// calculate normalized texture coordinates
unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
float u = x / (float) width;
float v = y / (float) height;
// transform coordinates – rotate about centre of image
u -= 0.5f;
v -= 0.5f;
float tu = u*cosf(theta) - v*sinf(theta) + 0.5f;
float tv = v*cosf(theta) + u*sinf(theta) + 0.5f;
// read from texture and write to global memory
g_odata[ y*width + x] = tex2D( mytex, tu, tv);
}
```

Summary

- GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
  - Use parallelism efficiently
  - Coalesce memory accesses if possible
  - Take advantage of shared memory
  - Explore other memory spaces
    - Texture
    - Constant
  - Reduce bank conflicts