GPU Programming
(in CUDA)

Summer 2019

Designed by Julian Gutierrez, Presented by Nicolas Agostini

Session 5

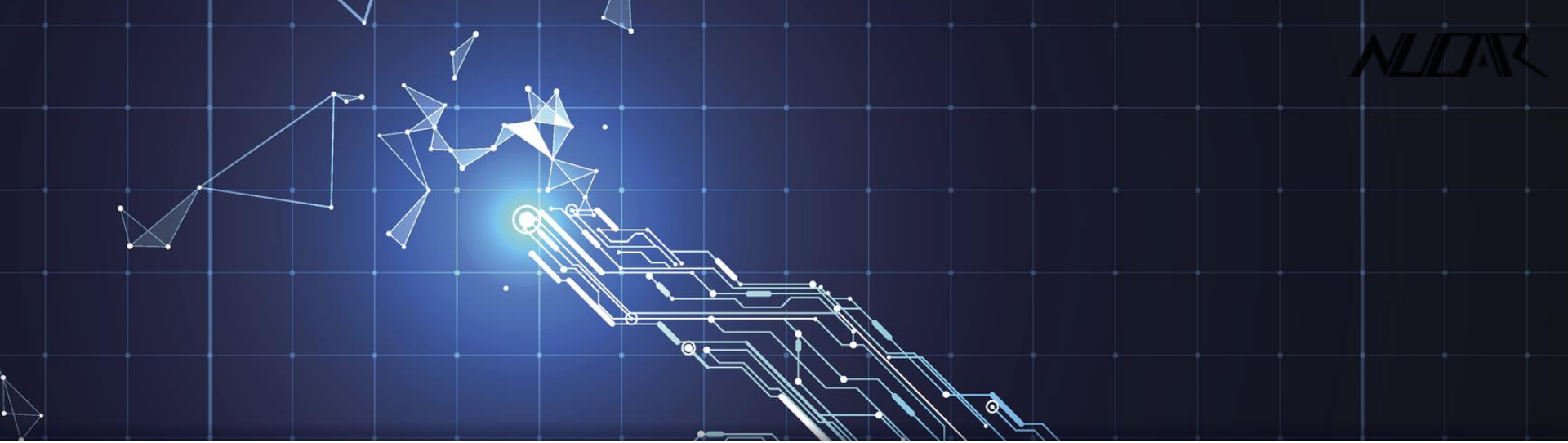
NUDAR

Outline

- Lab 2
- CUDA Memory Model
  - Overview and Concepts
  - Global Memory
  - Registers
  - Local Memory
  - Shared Memory
  - Caches
  - Constant Memory and Texture Memory
- Synchronization

Lab 2

- Any questions regarding the lab?
- Please note: I will provide a prewritten code for all of the labs. Before modifying it, go over it and understand what it does.



CUDA Memory Model

CUDA Memory Model

- Why do we need to understand the memory model?
  - Memory accesses will affect performance drastically
  - Memory patterns allow you to control these effects
  - Maximize resource usage by taking care of the LD/ST units
  - To do this, we use data locality

CUDA Memory Model

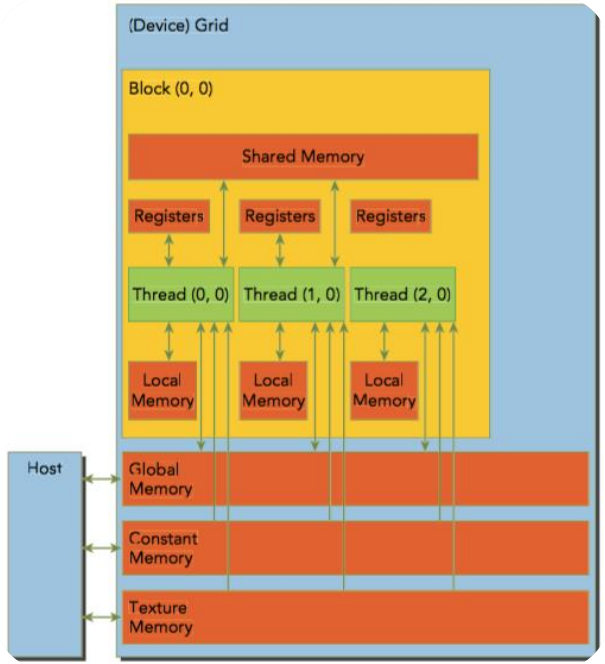
- Generally, applications don't have random accesses
  - Random accesses are very hard to improve
- Having locality in our data helps improve the performance
- There are two types of data locality that memory architectures make use of
- Temporal locality
  - Useful data tends to continue being useful
- Spatial locality
  - Useful data tends to be followed by more useful data

CUDA Memory Model

- Example
  - CPU Caches
    - They bring more than just the address being requested (block of memory)
    - They remove the least recently used block from the cache when a new block comes in (eviction)
  - GPUs exploit locality at different levels of the memory model

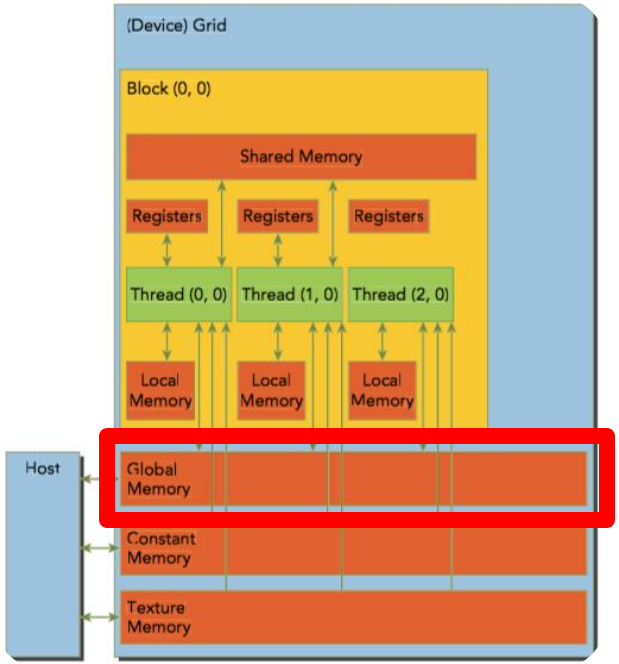
CUDA Memory Model

- Memories available on the GPU
  - Registers
  - Shared Memory
  - Local Memory
  - Constant Memory
  - Texture Memory
  - Global Memory



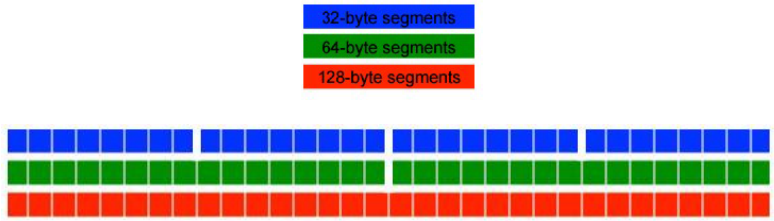
CUDA Memory Model

- Global Memory
  - Resides in device memory
  - Global scope: device level
  - Largest memory on the GPU
  - Highest latency (~100s of cycles)
  - If it's so slow, how can we use it efficiently?



CUDA Memory Model

- Global Memory
  - Accesses are organized by **half warps** (16 threads)
  - Accesses are done in 32, 64 and 128 byte segments
  - Consecutive addresses are read quickly
  - Certain non-sequential access patterns to global memory degrade performance
  - Allocations are aligned to multiples of 256 bytes (or more) for performance purposes
  - Accesses can be done in as little as one transactions for all the half warp, under certain conditions



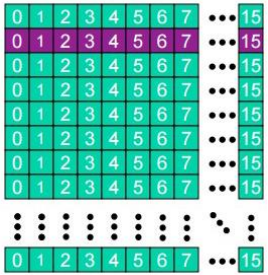
CUDA Memory Model

- Global Memory
  - This concept is called **Memory Coalescing**
  1. The memory controller looks at the segment containing the address the first active thread is requesting
  2. Looks for all other active threads requesting in the same segment
  3. Coalesces these transactions into fewer transactions (if possible)
  4. Mark the serviced threads as inactive
  5. Repeat until all threads in ½ warp are complete

CUDA Memory Model

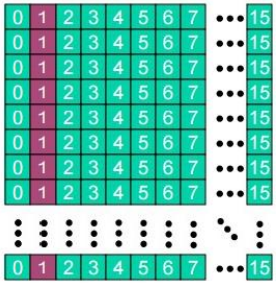
- Example: 2D data
- All warps in a block access consecutive elements within a row as they step through neighboring columns.

```
I =  blockIdx.y*by + ty;
J =  blockIdx.x*bx + tx;
int tx = threadIdx.x
a[ty][tx] = A[I*N+k*by+tx]
b[ty][tx] = B[J+N*(k*bx+ty)]
```



- Accesses by threads in a block along a column don't coalesce.

```
I =  blockIdx.x*bx + tx;
J =  blockIdx.y*by + ty;
a[tx][ty] = A[I*N+k*by+ty]
b[ty][tx] = B[J+N*(k*bx+tx)]
```

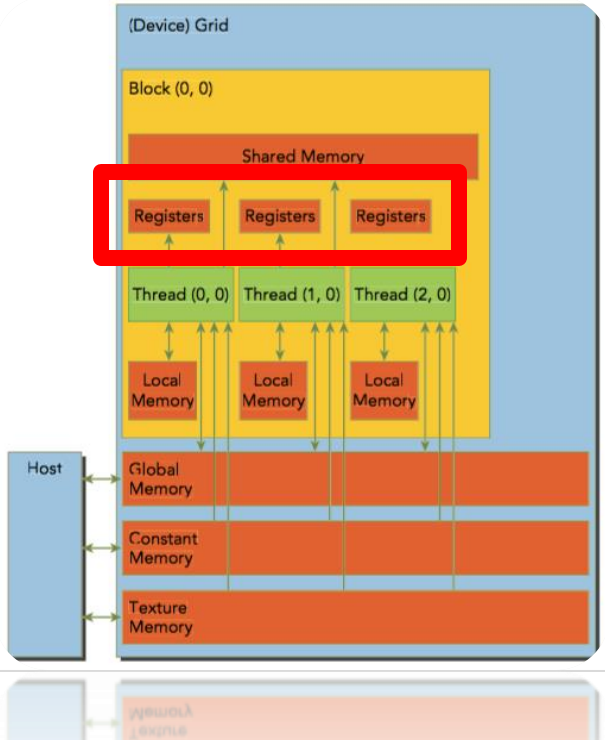


CUDA Memory Model

- In Summary
  - GM has very high latency (~100 cycles per read)
  - Resides in the RAM memory of the GPU
  - We want half a warp to access addresses that fall into same 32, 64 or 128 byte blocks to coalesce the reads (convert multiple memory requests into a single one) `allows memory coalescing`
  - Basically… We want very predictable and ordered accesses in memory. We will discuss this further in the hands-on lab.

CUDA Memory Model

- Registers
  - The fastest memory
  - Variables are automatically assigned to registers
  - Some arrays might be declare as registers if the size is determined at compilation time and are fairly small



CUDA Memory Model

- Registers
  - The fastest memory
  - Variables are automatically assigned to registers
  - Some arrays might be declare as registers if the size is determined at compilation time and are fairly small

Registers

```
__global__ voi                    at *devA, float
*devB, float *                    k){
    int txID = blockIdx.x * blockDim.x + threadIdx.x;
    int tyID = blockIdx.y * blockDim.y + threadIdx.y;

    if ((txID < col) && (tyID < row))
    {
        float Pvalue = 0;
        for(int w=0; w<k; w++)
        {
            Pvalue += devA[tyID*k+w] *
devB[w*k+txID];
        }
        devC[tyID*k+txID] = Pvalue;
    }
}
```

CUDA Memory Model

- Registers
  - Remember: Register File size is 64 K entries

This is for a single SM.
Matches with L3 Slide 43

  - Registers are private for each thread.
  - Kepler supports a maximum of 255 registers per thread.
  - Example:

```
nvcc -Xptxas -v,-abi=no matrixMul.cu -o matrixMul
ptxas warning : 'option -abi=no' might get deprecated in future
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function
'_Z15matrixMulKernelPfS_S_iii' for 'sm_20'
ptxas info    : Used 14 registers, 68 bytes cmem[0]
```

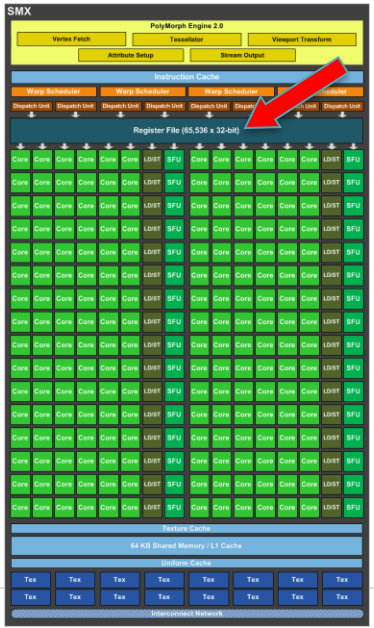
Note: Parallel Thread Execution (**PTX**) is a pseudo-assembly language used in Nvidia's **CUDA** programming environment

CUDA Memory Model

- Registers
  - What happens if we exceed register hardware limit?
    - Spills!
    - It will significantly impact performance.
    - It will spill to local memory.

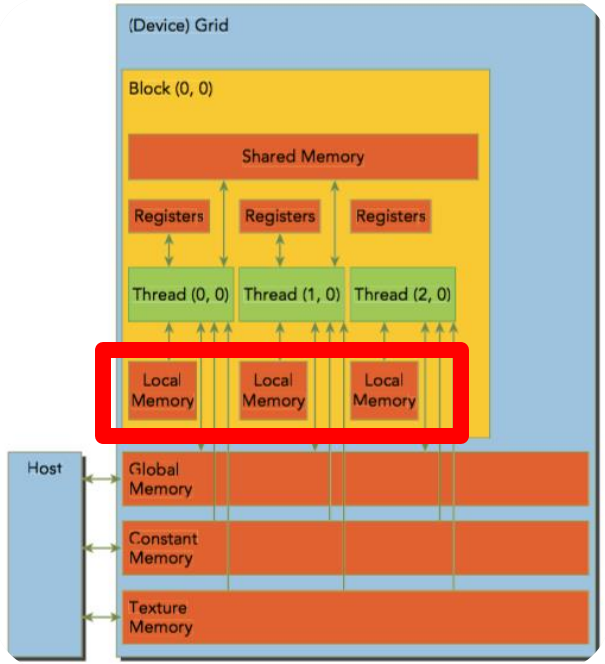
Where is Local Memory?

In DEVICE MEMORY!



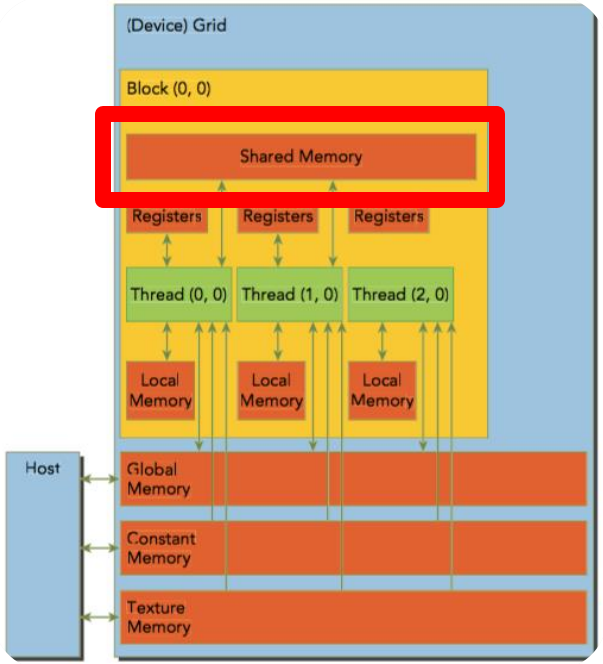
CUDA Memory Model

- Local Memory
  - Variables that cannot fit on the registers space go to Local Memory
    - Local array that indexes cannot be defined in compilation time
    - Large local array that consumes too much register space
    - Any variable that doesn't fit in the register limit
  - Remember, Local memory is actually stored in DEVICE MEMORY!
  - The reads to Local Memory are as slow as Global Memory… kind of…
  - In the Kepler Architecture, LM reads are cached in L1 cache, GM reads aren't (by default)



CUDA Memory Model

- Shared Memory
  - It is on-chip, therefore:
    - It has a much higher bandwidth
    - Lower latency
    - Much smaller size
  - It works at a block-level granularity
    - Threads in a block can access the same shared memory space
    - When a block is done, its shared memory will be released
    - Use __syncthreads() to synchronize threads in the block to avoid hazards or race conditions



CUDA Memory Model

- Shared Memory
  - It can be declared in the kernel (statically) or in the kernel call (dynamically).
  - Use attribute: __shared__

```
__global__ void staticReverse(int *d, int n) {
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n) {
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

CUDA Memory Model

- Shared Memory
  - It can be declared in the kernel (statically) or in the kernel call (dynamically).
  - Use attribute: __shared__

```
…
// run version with static shared memory
cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
staticReverse<<<1,n>>>(d_d, n);
cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
for (int i = 0; i < n; i++) if (d[i] != r[i])
    printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);

// run dynamic shared memory version
cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
for (int i = 0; i < n; i++) if (d[i] != r[i])
    printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
…
```

Third argument inside <<<?,?,?>>>

What if you need multiple dynamically sized arrays in a single kernel?

You must declare a single extern unsized array as before, and use pointers into it to divide it into multiple arrays

Basically, don't do it...

Just don't…

CUDA Memory Model

- Shared Memory
  - Another positive feature is that it is programmable on Kepler Architecture.
  - You can assign the amount of memory you need.
    - Shared memory and the L1 cache use the same physical space of 64K on-chip memory

```
cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);
```

Options are:

```
cudaFuncCachePreferL1
cudaFuncCachePreferShared
cudaFuncCachePreferEqual
```

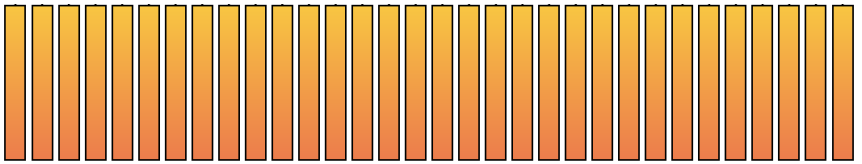
CUDA Memory Model

- Shared Memory
  - This is usually the best way to optimize your algorithms memory behavior if there are multiple accesses to those memory addresses.
  - Example:

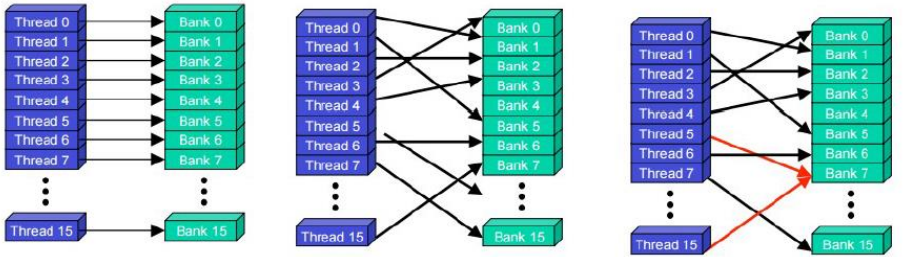
```
__global__ void compute_it(float *data) {
int tid = threadIdx.x;
__shared__ float myblock[1024];
float tmp;

// load the thread's data element into shared memory
myblock[tid] = data[tid];

// ensure that all threads have loaded their values into
// shared memory; otherwise, one thread might be computing
// on uninitialized data.
__syncthreads();

// compute the average of this thread's left and right neighbors
tmp = (myblock[tid>0?tid-1:1023] + myblock[tid<1023?tid+1:0]) * 0.5f);
// square the previous result and add my value, squared
myblock[tid] = tmp*tmp + myblock[tid]*myblock[tid];

// write the result back to global memory
data[tid] = myblock[tid];
}
```

CUDA Memory Model

- Shared Memory
  - So, why is Shared Memory so fast?
  - It is closer to the cores.
  - Shared memory is divided into banks.

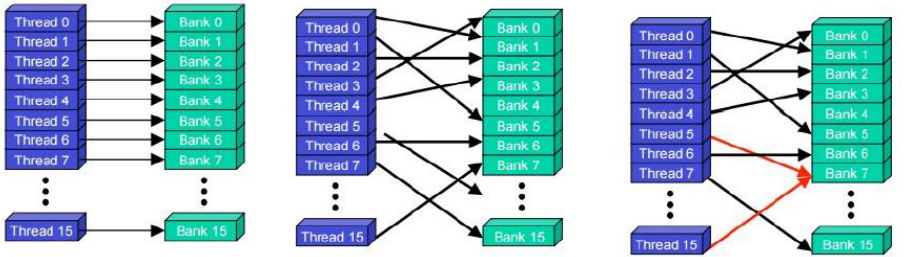


Shared Memory Banks



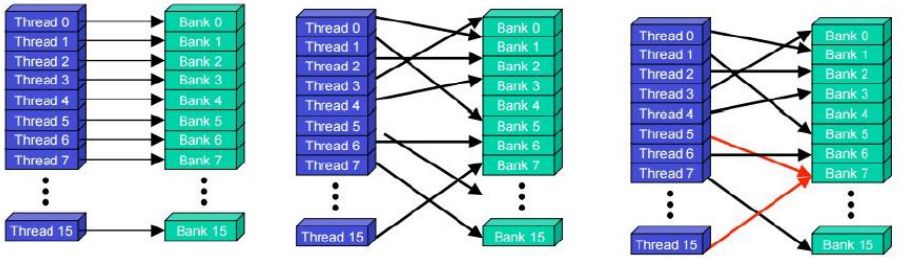
CUDA Memory Model

- Shared Memory
  - Successive 32 bit words are assigned to successive banks
  - For devices of compute capability 2.x or higher (fermi)
    - Number of banks = 32
    - Bandwidth is 32 bits per bank per 2 clock cycles
    - Shared memory requests for a warp are not split
    - Increased susceptibility to conflicts
    - But no conflicts if access to bytes in same 32 bit word



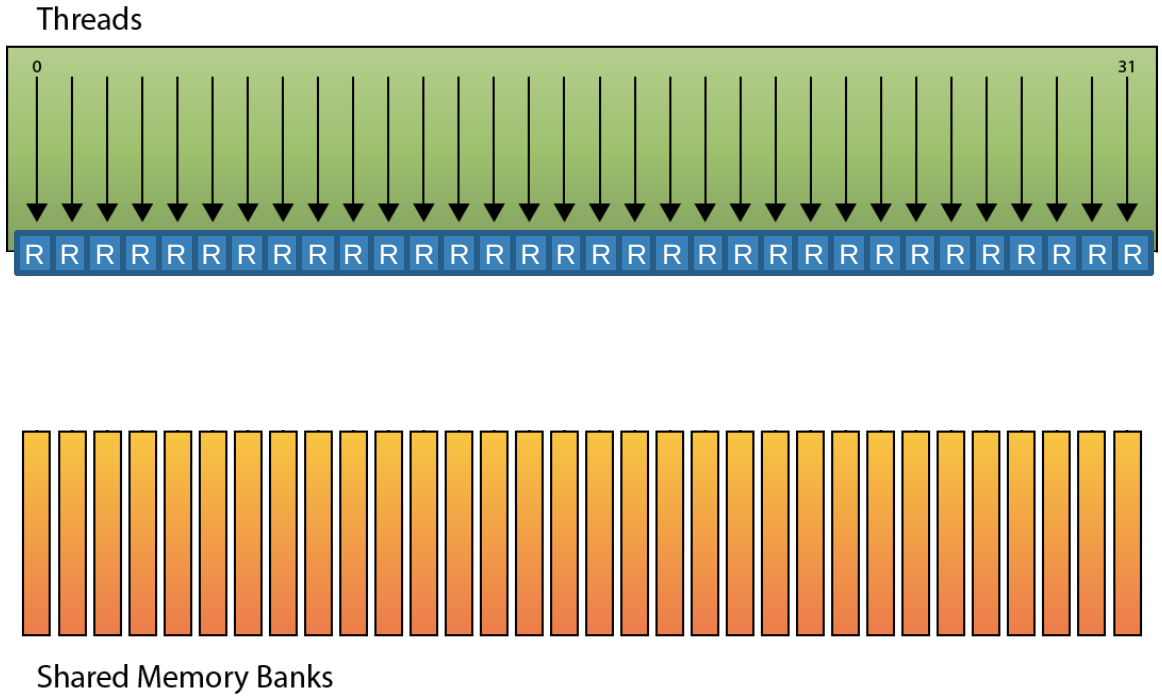
CUDA Memory Model

- Shared Memory
  - Access to shared memory is fast unless:
  - 2 or more threads in a warp access the same bank: we have a conflict
  - If two or more addresses of a memory request are in the same bank, the accesses are serialized (Bank conflicts exist only within a warp)
  - Exception: if all access the same address: broadcast



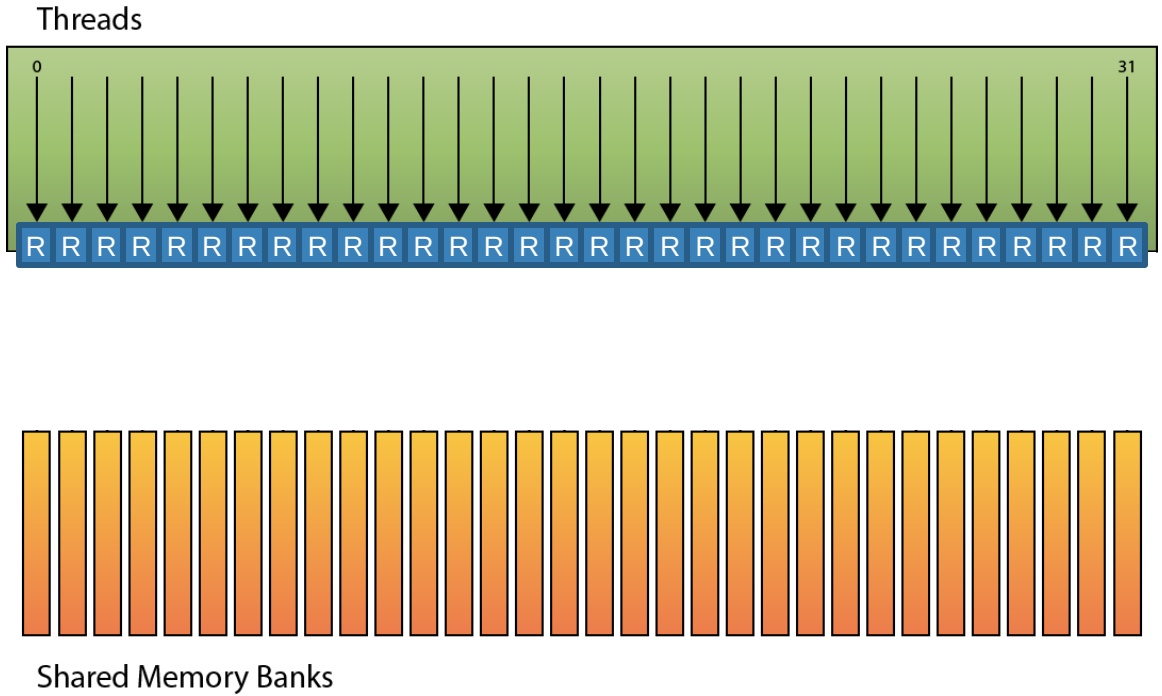
CUDA Memory Model

- Shared Memory
- Accessing from different banks
  - No contention



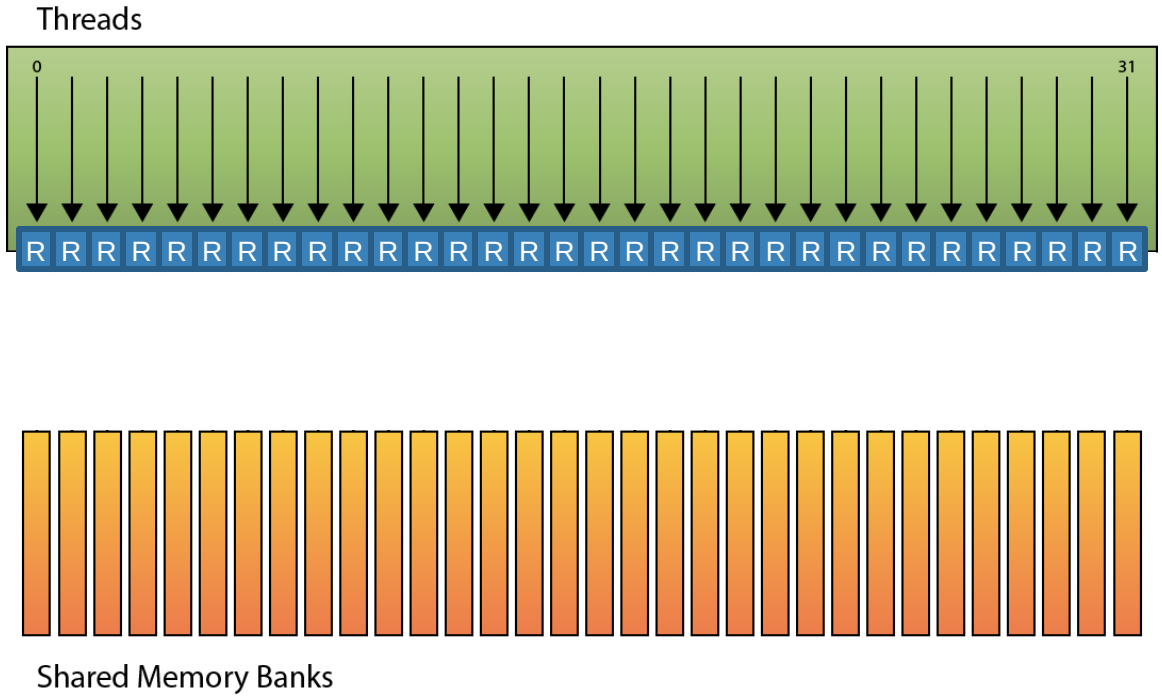
CUDA Memory Model

- Shared Memory
- Half warps accessing different memory from the same banks
  - Contention!



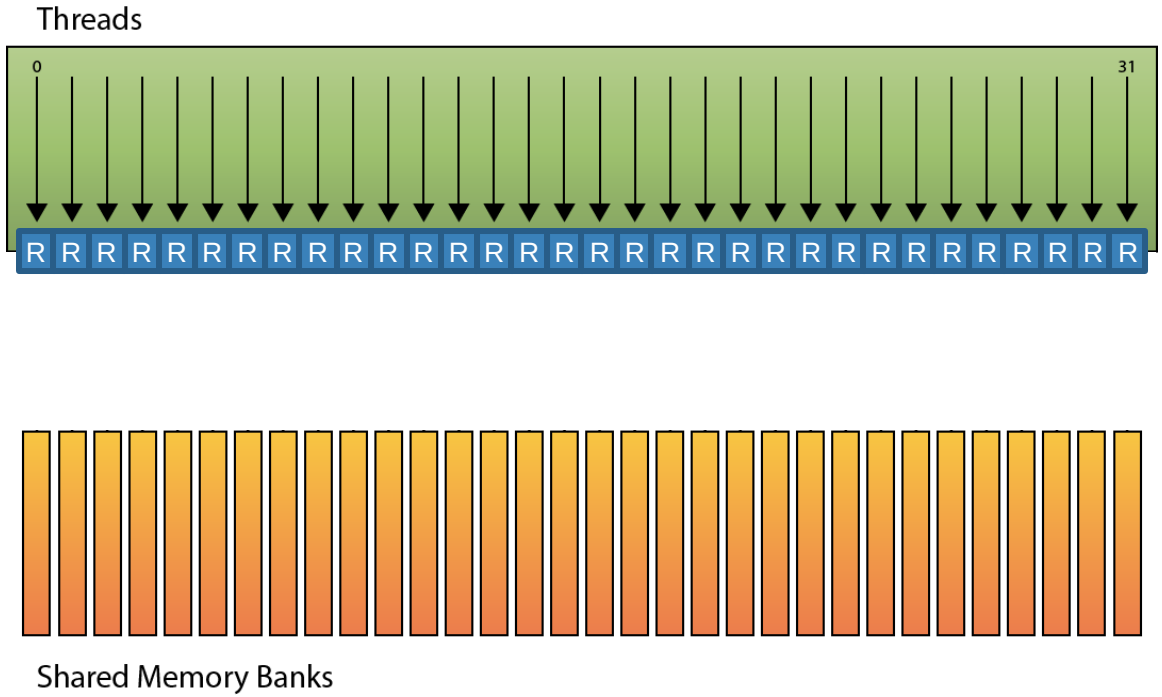
CUDA Memory Model

- Shared Memory
- Quarter warps accessing different memory from the same banks
  - More Contention



CUDA Memory Model

- Shared Memory
- Different threads accessing the SAME memory in the same Bank
  - Broadcast!
  - No contention

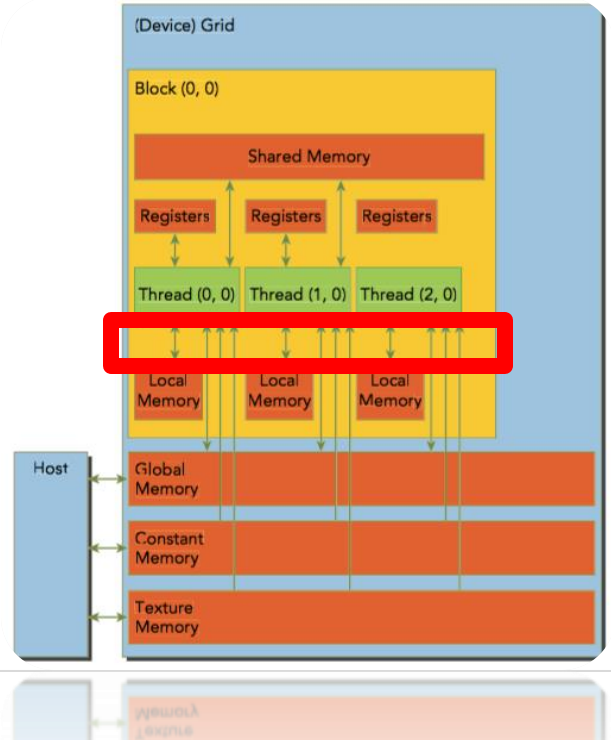


CUDA Memory Model

- In summary
  - Very Fast (2 cycles per read)
  - Divided into banks (32)
  - We don't want to access the same bank with threads in the same warp (bank conflicts, serialize the reads) unless they all access the same 32 bit address (we can broadcast the value to all the threads)
    - Very useful if addresses are requested multiple times from global memory (reduce the reads from global memory).
    - Requires extra manual work, and synchronization between threads in a block

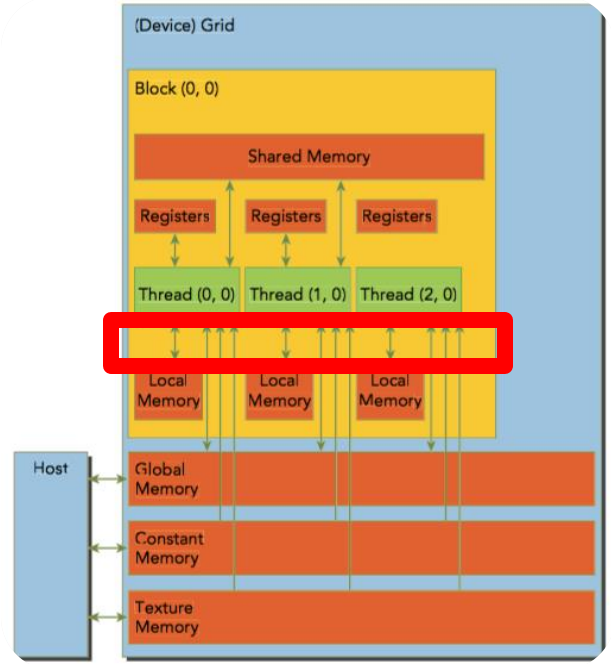
CUDA Memory Model

- Caches
  - Non-programmable memory
  - Four types of caches:
    - L1 cache
    - L2 cache
    - Read-only constant memory
    - Read-only texture memory



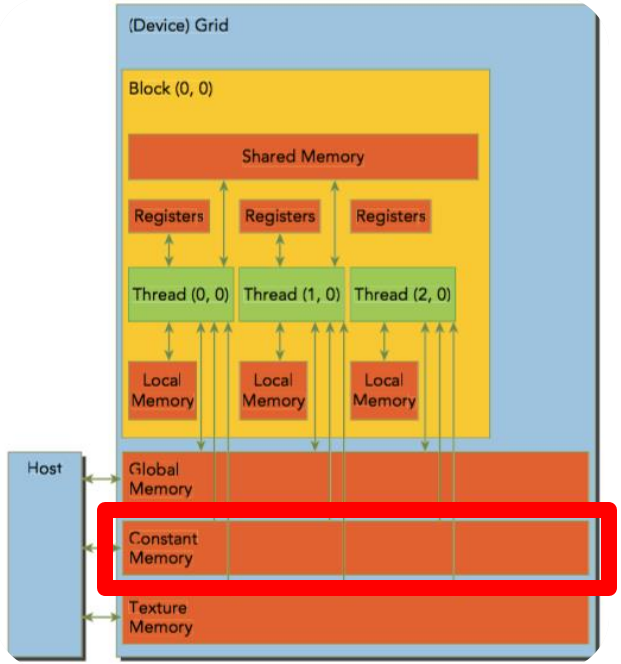
CUDA Memory Model

- Caches
  - One L1 per-SM
  - One L2 for all SMs
  - L1 and L2 store data for local and global memory (that includes register spills)
  - GPUs like Kepler can configure that reads are cached in L1 and L2, or just L2 (default)
  - Memory store operations cannot be cached



CUDA Memory Model

- Constant Memory
  - Read-only and resides on device memory
  - It has its own separate cache
  - It has to be declared on a global scope outside of the kernel
  - Limited resource: 64K for all compute capabilities
  - Use attribute: __constant__



CUDA Memory Model

- Constant Memory
  - Host has to initialize it:

```
cudaError_t cudaMemcpyToSymbol(const void* symbol,
const void* src, size_t count);
```

  - It performs better when all threads in a warp access the same memory address
  - **DON'T USE IT** if all threads in a warp access different memory addresses and is only accessed once
  - The variables in constant memory don't have to be declared in the kernel invocation

```
// CUDA global constants
  __constant__ int M;

int main(void)
{
...
  cudaMemcpyToSymbol("M", &M, sizeof(M));
...
}
```

```
__global__ void kernel(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N)
  {
    a[idx] =  a[idx] *M;
  }
}
```

CUDA Memory Model

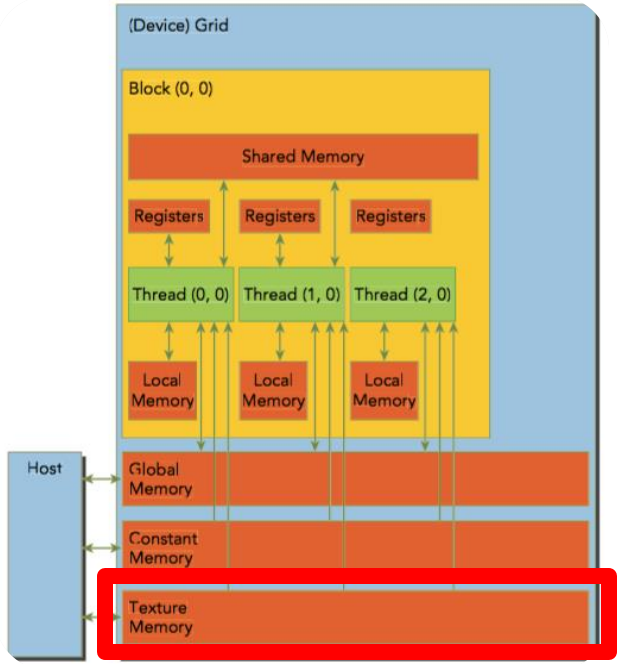
- Constant Memory
  - Declaring memory as constant constrains the usage to read-only
  - Reading from constant memory can conserve memory bandwidth when compared to reading the same data from global memory.
  - There are 2 reasons why reading from 64 KB constant memory saves bandwidth:
    - A single read from constant memory can be broadcasted to other threads, effectively saving up to 15 reads.
    - Constant memory is cached, so consecutive read of the same address will not incur any additional memory traffic.

CUDA Memory Model

- Constant Memory
  - There can be a downgrade of the performance when using constant memory
  - The half-warp broadcast feature can degrade the performance when all 16 threads read different addresses
  - If all 16 threads in a half-warp need different data from constant memory, the 16 different reads get serialized

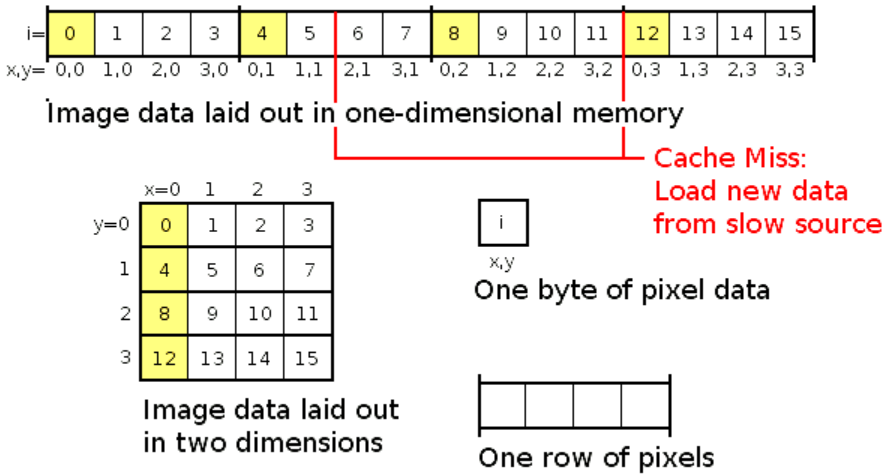
CUDA Memory Model

- Texture Memory
  - It resides in device memory as well
  - It is cached in a per-SM fashion
  - It belongs to the scope of global memory but it is accessed through a dedicated read-only cache
  - Recommended for use when data has 2D (3D) spatial locality (i.e. imaging)



CUDA Memory Model

- Texture Memory
  - Example of how 2D spatial locality could work



CUDA Memory Model

- Texture Memory
  - Example of how 2D spatial locality could work (Z-ORDER)
  - Regular Memory Addresses

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167
168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215
216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263
264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287
288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311
312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335
336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359
360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383

  - Reordered Memory Addresses

0	1	24	25	2	3	26	27	4	5	28	29	6	7	30	31	48	49	72	73	50	51	74	75
52	53	76	77	54	55	78	79	96	97	120	121	98	99	122	123	100	101	124	125	102	103	126	127
144	145	168	169	146	147	170	171	148	149	172	173	150	151	174	175	8	9	32	33	10	11	34	35
12	13	36	37	14	15	38	39	56	57	80	81	58	59	82	83	60	61	84	85	62	63	86	87
104	105	128	129	106	107	130	131	108	109	132	133	110	111	134	135	152	153	176	177	154	155	178	179
156	157	180	181	158	159	182	183	16	17	40	41	18	19	42	43	20	21	44	45	22	23	46	47
64	65	88	89	66	67	90	91	68	69	92	93	70	71	94	95	112	113	136	137	114	115	138	139
116	117	140	141	118	119	142	143	160	161	184	185	162	163	186	187	164	165	188	189	166	167	190	191
192	193	216	217	194	195	218	219	196	197	220	221	198	199	222	223	240	241	264	265	242	243	266	267
244	245	268	269	246	247	270	271	288	289	312	313	290	291	314	315	292	293	316	317	294	295	318	319
336	337	360	361	338	339	362	363	340	341	364	365	342	343	366	367	200	201	224	225	202	203	226	227
204	205	228	229	206	207	230	231	248	249	272	273	250	251	274	275	252	253	276	277	254	255	278	279
296	297	320	321	298	299	322	323	300	301	324	325	302	303	326	327	344	345	368	369	346	347	370	371
348	349	372	373	350	351	374	375	208	209	232	233	210	211	234	235	212	213	236	237	214	215	238	239
256	257	280	281	258	259	282	283	260	261	284	285	262	263	286	287	304	305	328	329	306	307	330	331
308	309	332	333	310	311	334	335	352	353	376	377	354	355	378	379	356	357	380	381	358	359	382	383

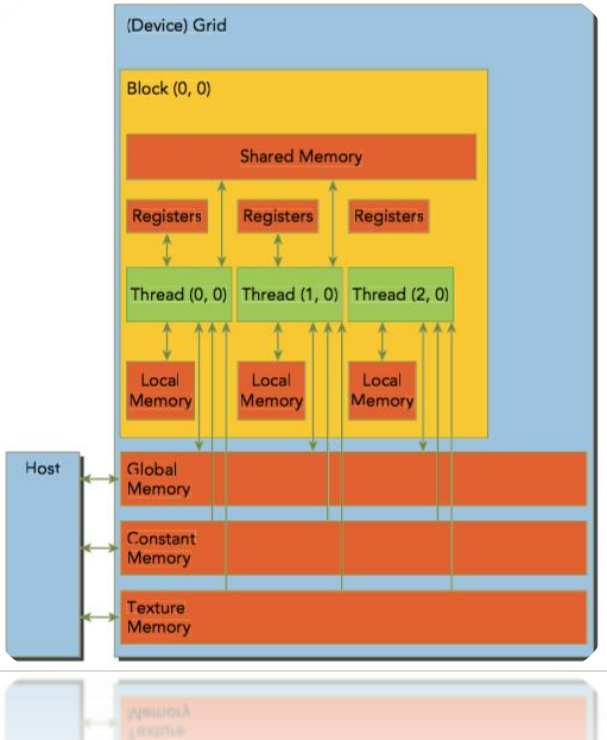
CUDA Memory Model

- Texture Memory
  - The easiest way to use texture memory, is by reference.

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

// texture reference name must be known at compile time
__global__ void kernel() {
  int i = blockIdx.x *blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
  // do some work using x...
}

void call_kernel(float *buffer) {
  // bind texture to buffer
  cudaBindTexture(0, tex, buffer, N*sizeof(float));

  dim3 block(128,1,1);
  dim3 grid(N/block.x,1,1);
  kernel <<<grid, block>>>();

  // unbind texture from buffer
  cudaUnbindTexture(tex);
}

int main() {
  // declare and allocate memory
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
```

CUDA Memory Model

- Summary
  - Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory



CUDA Memory Model

- Summary
  - Type Qualifier table
  - Notes:
    - __device__ not required for __local__, __shared__, or __constant__
    - Automatic variables without any qualifier reside in a register
      - Except arrays that reside in local memory
      - Or not enough registers available for automatic variables

Variable declaration	Memory	Scope	Lifetime
Int LocalVar;	Register	Thread	Thread
Int LocalArray[10];	Local	Thread	Thread
[__device__] __shared__ int SharedVar;	Shared	Block	Block
__device__ int GlobalVar;	Global	Grid	Application
[__device__] __constant__ int ConstantVar;	Constant	Grid	Application

CUDA Memory Model

- Summary
  - Type Qualifier table
  - Notes:
    - Scalar variables reside in on-chip registers (fast)
    - Shared variables reside in on-chip memory (fast)
    - Local arrays and global variables reside in off-chip memory (slow)
    - Constants reside in cached off-chip memory

Variable declaration	Memory	Scope	Lifetime
Int LocalVar;	Register	Thread	Thread
Int LocalArray[10];	Local	Thread	Thread
[__device__] __shared__ int SharedVar;	Shared	Block	Block
__device__ int GlobalVar;	Global	Grid	Application
[__device__] __constant__ int ConstantVar;	Constant	Grid	Application

Synchronization

- Race Condition
  - An undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

How can we avoid this on a GPU?

Synchronization

- Atomic Instructions
  - Atomic instructions are special hardware instructions that perform an operation on one or more memory locations atomically (indivisible or uninterruptible).
  - An atomic operation either succeeds or fails in its entirety, regardless of what instructions are being executed by other threads.

Synchronization

- Use of atomic operations
  - Collaboration
    - Atomics on an array that will be the output of the kernel
      - Histogram
  - Synchronization
    - Atomics on memory locations that are used for synchronization or coordination
      - Counters, locks, flags…
- CUDA provides atomic functions on shared memory and global memory

Synchronization

- Arithmetic functions
  - Add, sub, max, min, exch, inc, dec, CAS
- Bitwise functions
  - And, or, xor

```
CUDA: int atomicAdd(int*, int);
PTX: atom.shared.add.u32 &r25, [%rd14],1;
SASS:
```

GT200, Fermi, Kepler

```
/*00a0*/  LDSLK P0, R9, [R8];
/*00a8*/  @P0 IADD R10, R9, R7;
/*00b0*/  @P0 STSCUL P1, [R8], R10;
/*00b8*/  @!P1 BRA 0xa0;
```

Maxwell

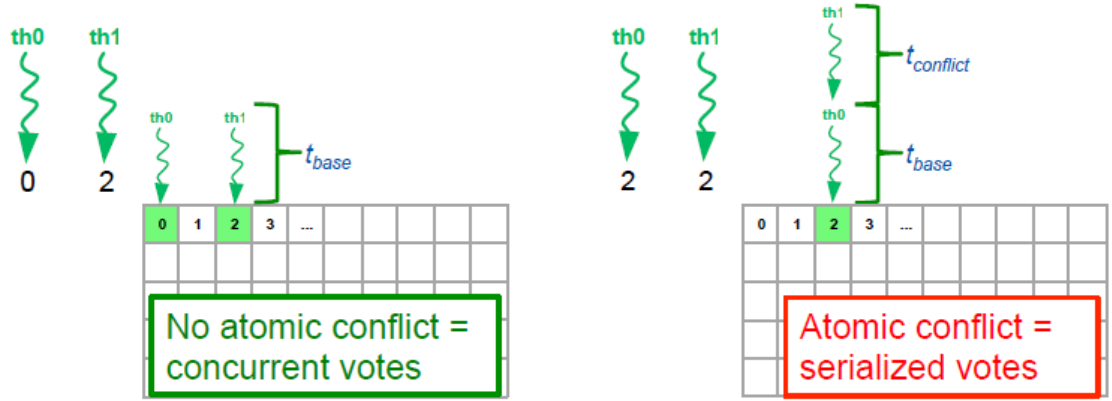
```
/*01f8*/  ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS

Lock/update/unlock vs native atomic

Synchronization

- Atomic conflict degree
  - Intra-warp conflict degree from 1 to 32



Synchronization

- Many applications require different means of synchronization.
  - Intra Block
  - Inter Block
  - CPU/GPU concurrent execution

How can we do these?

Synchronization

- Many applications require different means of synchronization.
  - Intra Block
    - __syncthreads()
      - Must be reached by all threads from the block.
      - Ensures that for all threads in the block, the code preceding the instruction is executed before the instructions following it.
    - Atomic operations (in shared memory)

Synchronization

- Many applications require different means of synchronization.
  - Inter Block
    - No explicit way of doing this
      - Terminate kernel at synchronization point, and then launch new kernel to continue after.
    - Atomic operations (in global memory, should be avoided)

Synchronization

- Many applications require different means of synchronization.
  - CPU/GPU concurrent execution
    - Copy data back and forth