

Designed by Julian Gutierrez, Presented by Nicolas Agostini

Session 5



Outline



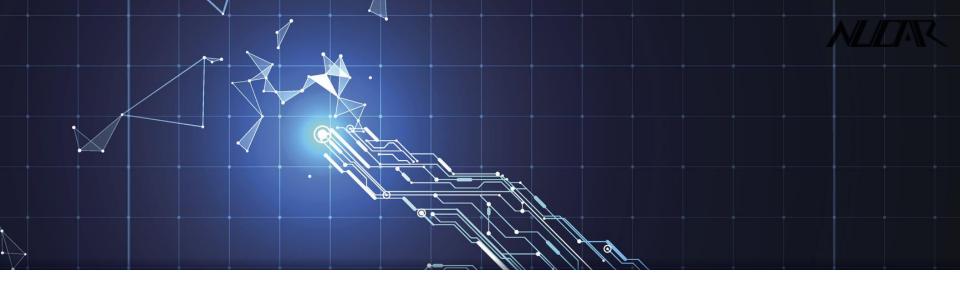


- Lab 2
- CUDA Memory Model
 - Overview and Concepts
 - Global Memory
 - Registers
 - Local Memory
 - Shared Memory
 - Caches
 - Constant Memory and Texture Memory
- Synchronization

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.





- 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



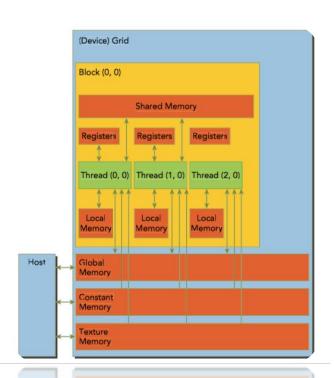
- Generally, applications don't have random accesses
 - Random accesses are very hard to improve
- Having (ocality) in our data helps improve the performance
- There are two types of data locality that memory architectures make use of
- Temporal locality
 - Useful data tends to continue being useful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



- 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

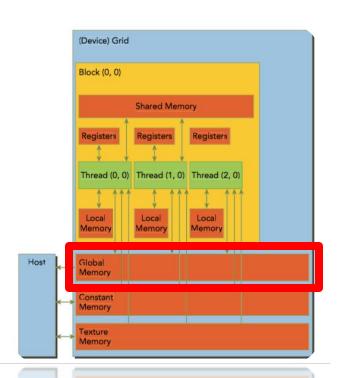


- Memories available on the GPU
 - Registers
 - Shared Memory
 - Local Memory
 - Constant Memory
 - Texture Memory
 - Global Memory





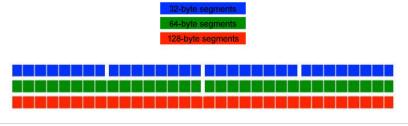
- Global Memory
 - Resides in device memory
 - Global scope: device level
 - Largest memory on the GPU
 - Highest latency (~100s of cycles)
 - If it's so slow, how can we use it efficiently?





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

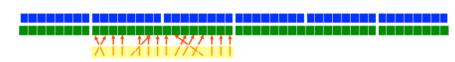




- Global Memory
 - This concept is called **Memory Coalescing**
 - 1. The memory controller looks at the segment containing the address the first active thread is requesting
 - 2. Looks for all other active threads requesting in the same segment
 - Coalesces these transactions into fewer transactions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

1 transaction - 64B segment

32-byte segments
64-byte segments





Example: 2D data

All warps in a block access consecutive elements within a row as they step through

neighboring columns.

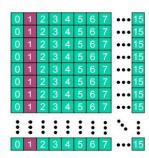
Accesses by threads in a block along a column don't coalesce.

```
I = blockldx.x*bx + tx;

J = blockldx.y*by + ty;

a[tx][ty] = A[I*N+k*by+ty]

b[ty][tx] = B[J+N*(k*bx+tx)]
```



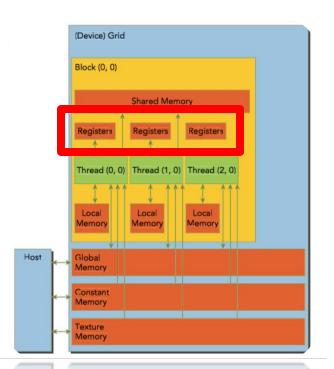


- In Summary
 - GM has very high latency (~100 cycles per read)
 - Resides in the RAM memory of the GPU
 - We want half a warp to access addresses that fall into same 32, 64 or 128
 byte blocks to coalesce the reads (convert multiple memory requests into a single one) allows memory coalescing
 - Basically...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



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

- 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

```
at *devA, float
               Registers
    int txID = blockIdx.x * blockDim.x + threadIdx.x;
    inttyID = 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;
```



Registers

Remember: Register File size is 64 K entries

Match

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]

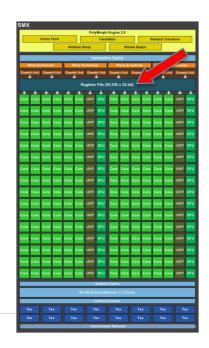
Note: Parallel Thread Execution (PTX) is a pseudo-assembly language used in Nvidia's CUDA programming environment



- 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?

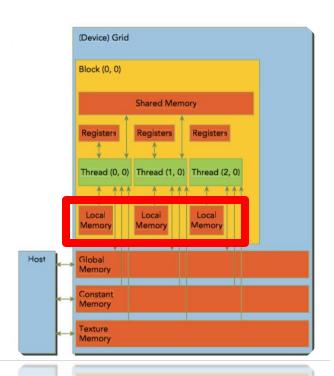
In DEVICE MEMORY!





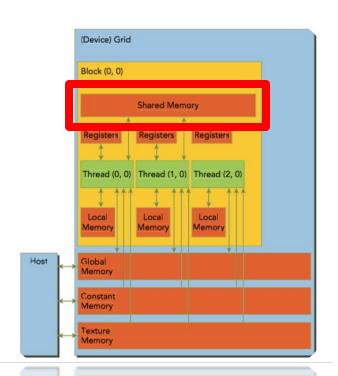
Local Memory

- 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)





- 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





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



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

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



- 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

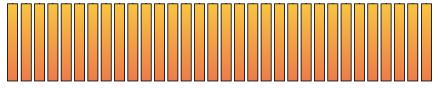


- Shared Memory
 - This is usually the best way to optimize your algorithms memory behavior if there are multiple accesses to those memory addresses.
 - Example:

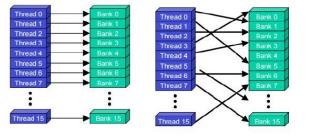
```
global void compute it(float *data) {
int tid = threadIdx.x;
  shared float myblock[1024];
float tmp;
// load the thread's data element into shared memory
myblock[tid] = data[tid];
// ensure that all threads have loaded their values into
// shared memory; otherwise, one thread might be computing
// 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];
```

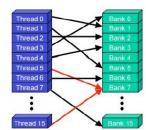


- Shared Memory
 - So, why is Shared Memory so fast?
 - It is closer to the cores.
 - Shared memory is divided into banks.



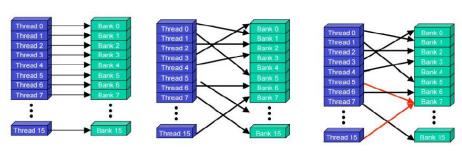
Shared Memory Banks





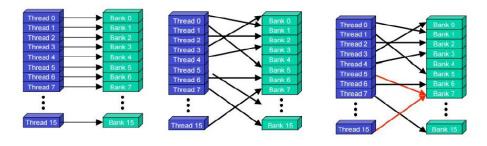


- Shared Memory
 - Successive 32 bit words are assigned to successive banks
 - For devices of compute capability 2.x or higher (fermi)
 - Number of banks = 32
 - Bandwidth is 32 bits per bank per 2 clock cycles
 - Shared memory requests for a warp are not split
 - Increased susceptibility to conflicts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



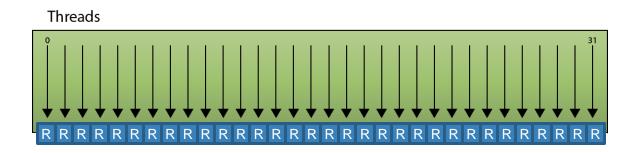


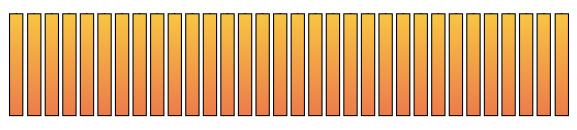
- Shared Memory
 - Access to shared memory is fast unless:
 - 2 or more threads in a warp access the same bank: we have a conflict
 - If two or more addresses of a memory request are in the same bank, the accesses are serialized (Bank conflicts exist only within a warp)
 - Constitution of all as a second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





- Shared Memory
- Accessing from different banks
 - No contention

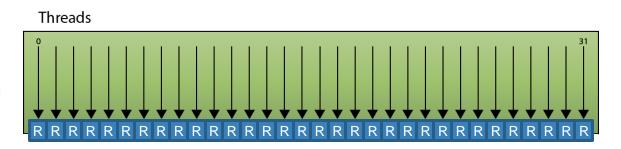


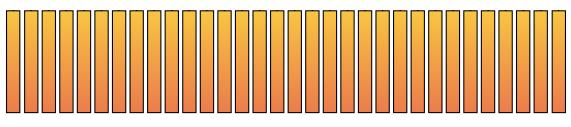


Shared Memory Banks



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

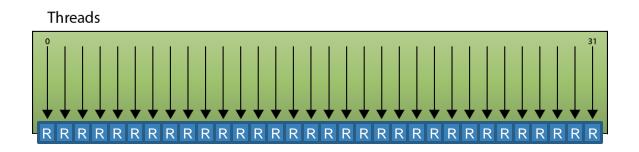


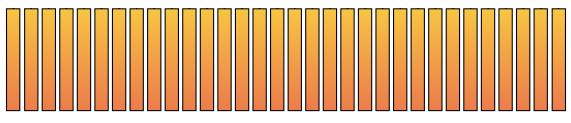


Shared Memory Banks



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

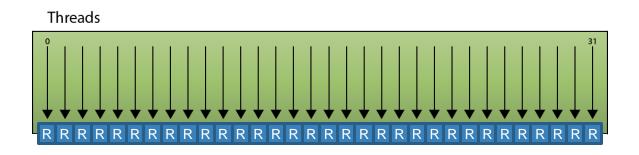


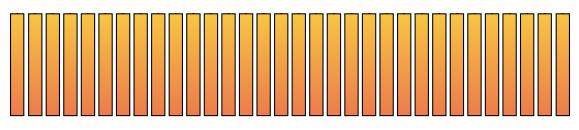


Shared Memory Banks



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





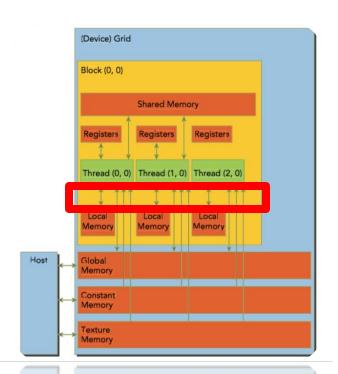
Shared Memory Banks



- 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



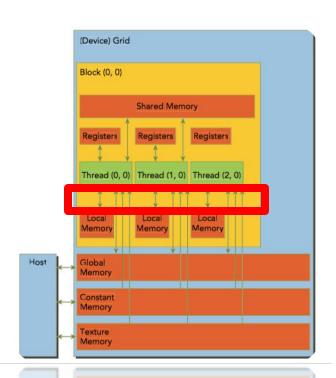
- Caches
 - Non-programmable memory
 - Four types of caches:
 - L1 cache
 - L2 cache
 - Read-only constant memory
 - Read-only texture memory





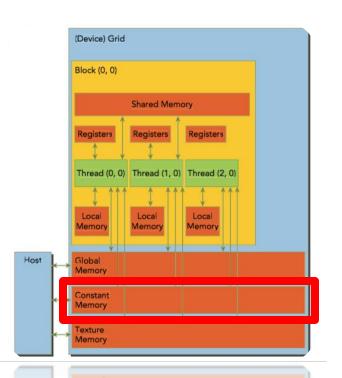
Caches

- 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





- 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__





- 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;
    }
}</pre>
```



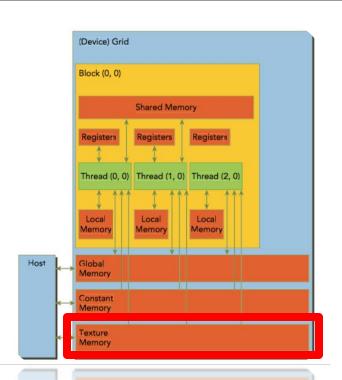
- Constant Memory
 - Declaring memory as constant constrains the usage to read-only
 - Reading from constant memory can conserve memory bandwidth when compared to reading the same data from global memory.
 - There are 2 reasons why reading from 64 KB constant memory saves bandwidth:
 - A single read 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.



- 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

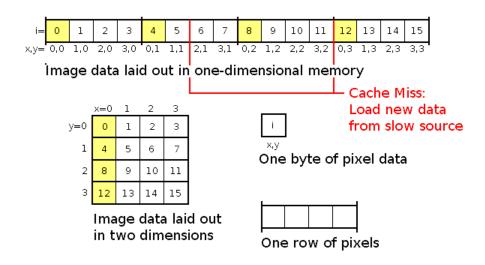


- Texture Memory
 - It resides in device memory as well
 - It is cached in a per-SM fashion
 - It belongs to the scope of global memory but it is accessed through a dedicated read-only cache
 - Recommended for use when data has 2D (3D) spatial locality (i.e. imaging)



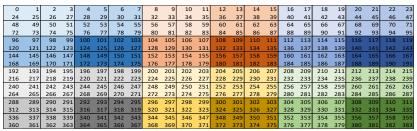


- Texture Memory
 - Example of how 2D spatial locality could work





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



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



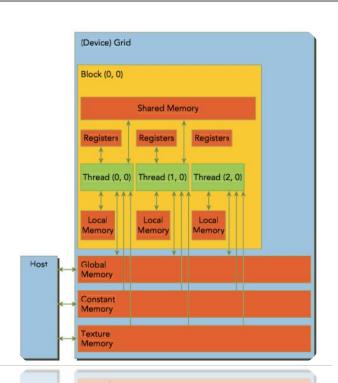
- Texture Memory
 - The easiest way to use texture memory, is by reference.

```
#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));
```



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





- 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



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



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?



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.



- 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



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

Maxwell

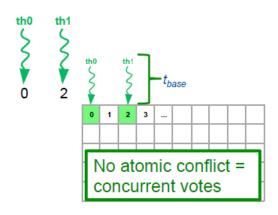
```
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

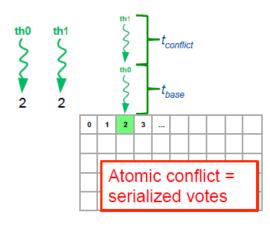
Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS

Lock/update/unlock vs native atomic



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







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

How can we do these?



- 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)



- 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)



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