GPU Programming

(in Cuda)

Julian Gutierrez

NUCAR

Session 4

Cuda Memory Model

Why do we need to understand the memory model?

Why do we need to understand the memory model?

- Memory access will affect performance drastically.

Why do we need to understand the memory model?

- Memory access will affect performance drastically.
- Memory patterns allow you to control these effects.

Why do we need to understand the memory model?

- Memory access will affect performance drastically.
- Memory patterns allow you to control these effects.
- Maximize resource usage by taking care of the LD/ST units.

Locality

- Usually applications don’t have random accesses.
- Random accesses are very hard to improve.
- Having a locality helps improve performance.

Locality

- What do we mean by Locality?

Locality

- What do we mean by Locality?
- Memory architectures make use of 2 types of locality

Locality

- Temporal locality
  - Useful data tends to continue being useful

Locality

- Temporal locality
  - Useful data tends to continue being useful
- Spatial locality
  - Useful data tends to be followed by more useful data

Locality

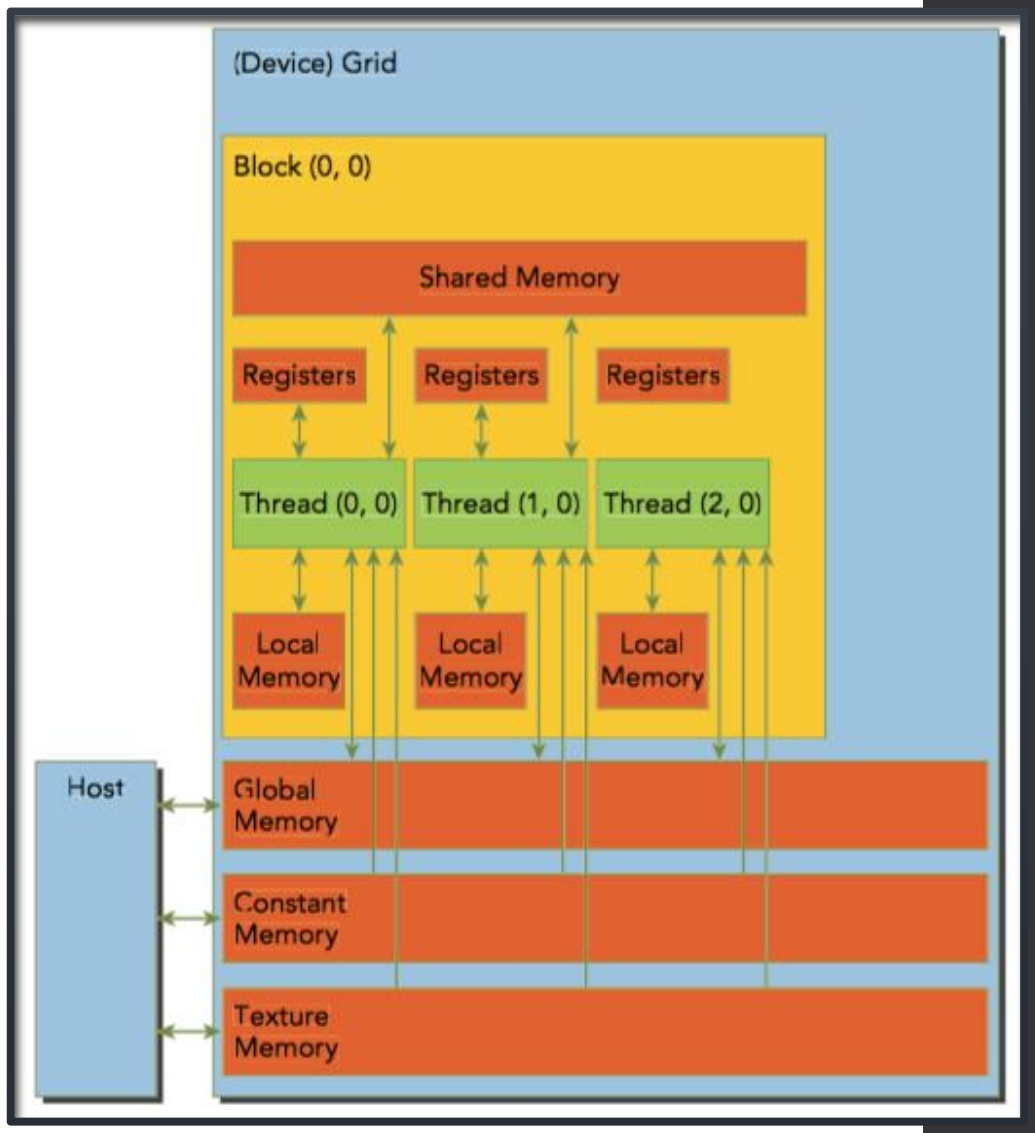
- Example:
  - CPU Caches
    - They bring more than just the address being requested (block of memory)
    - They remove the least recently used block from the cache when a new block comes in (eviction)

Locality

- 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.

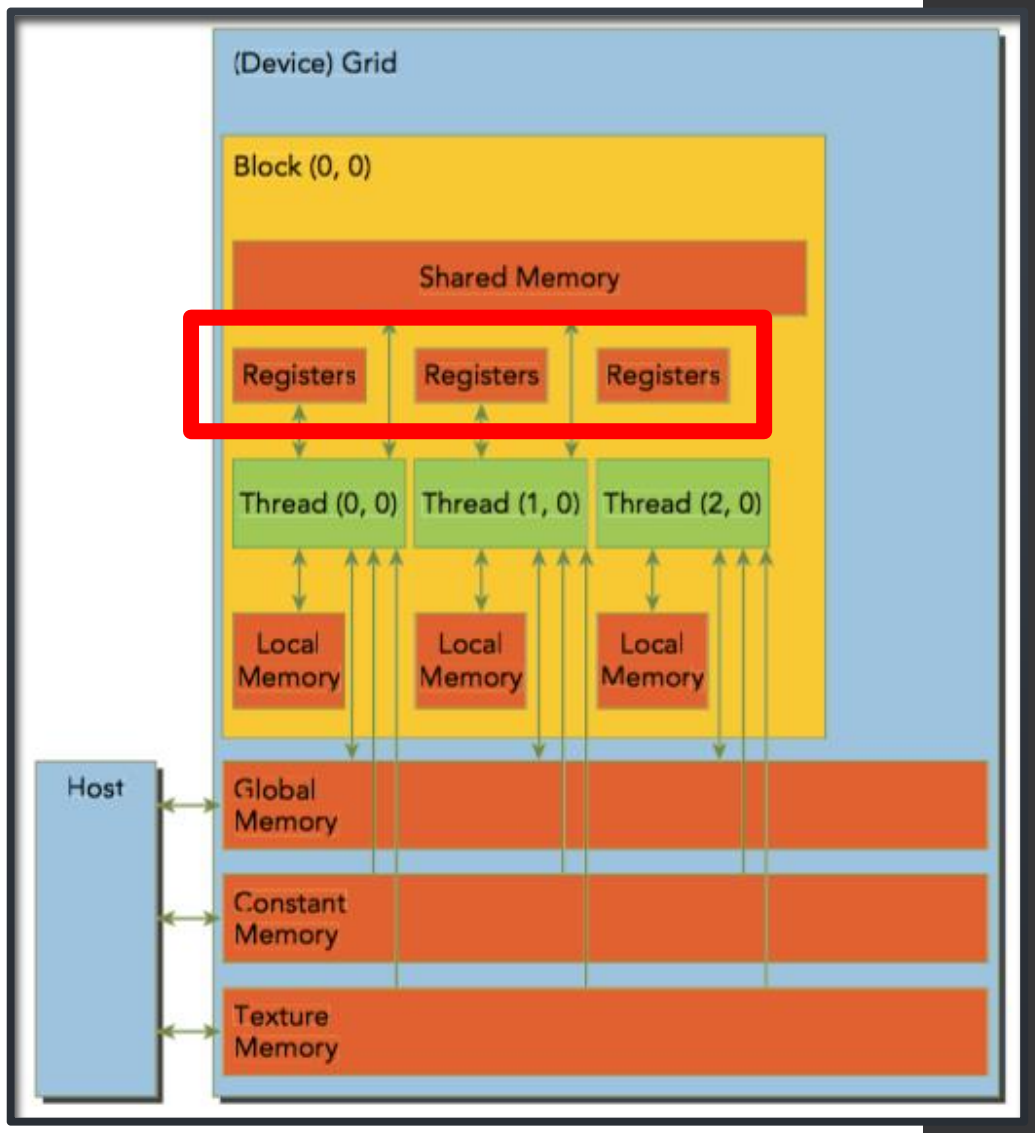
CUDA Memory Model

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



Registers

- The fastest memory
- Variables are automatically assigned to registers
- Some arrays might be declare as registers if the size is determined in compilation time and its constant



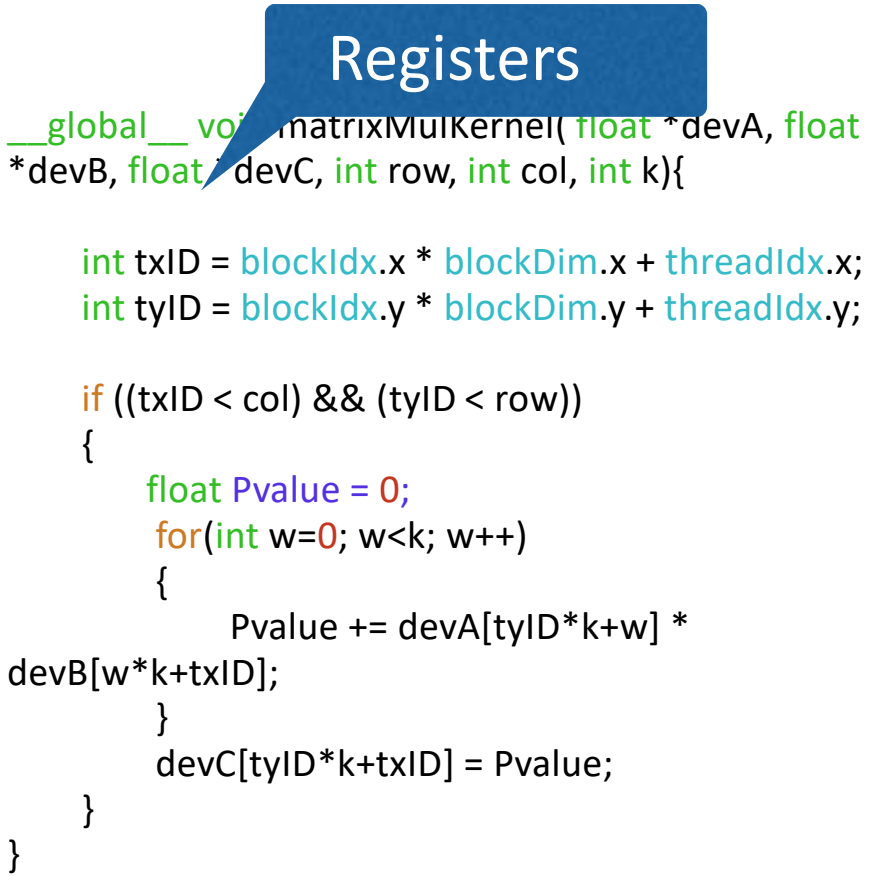
Registers

- The fastest memory
- Variables are automatically assigned to registers
- Some arrays might be declare as registers if the size is determined in compilation time and its constant

```
__global__ void matrixMulKernel( float *devA, float
*devB, float *devC, int row, int col, int 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;
      }
}
```

Registers

- The fastest memory
- Variables are automatically assigned to registers
- Some arrays might be declare as registers if the size is determined in compilation time and its constant



Registers

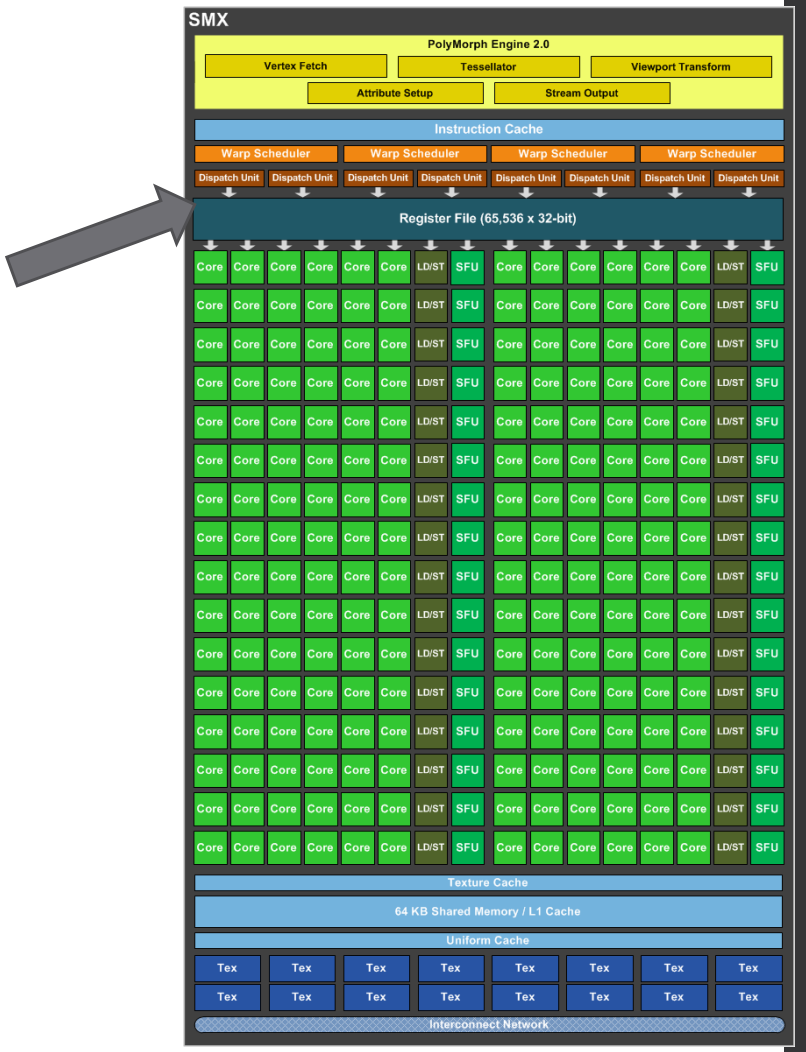
- Remember: Register File size was 64 K entries
- It is on a thread-level. Registers are private for each thread.
- Kepler supports 255 registers per thread max.
- Example:

```
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

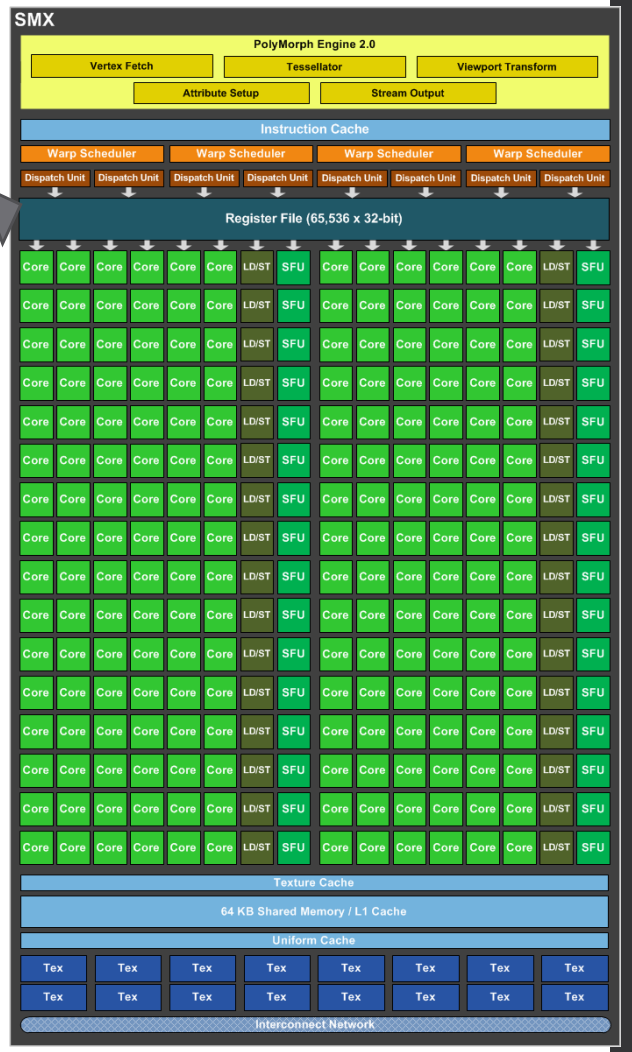
What happens if we exceed register hardware limit?

- Spills!
- It will significantly impact performance.
- It will spill to **local memory.**



What happens if we exceed register hardware limit?

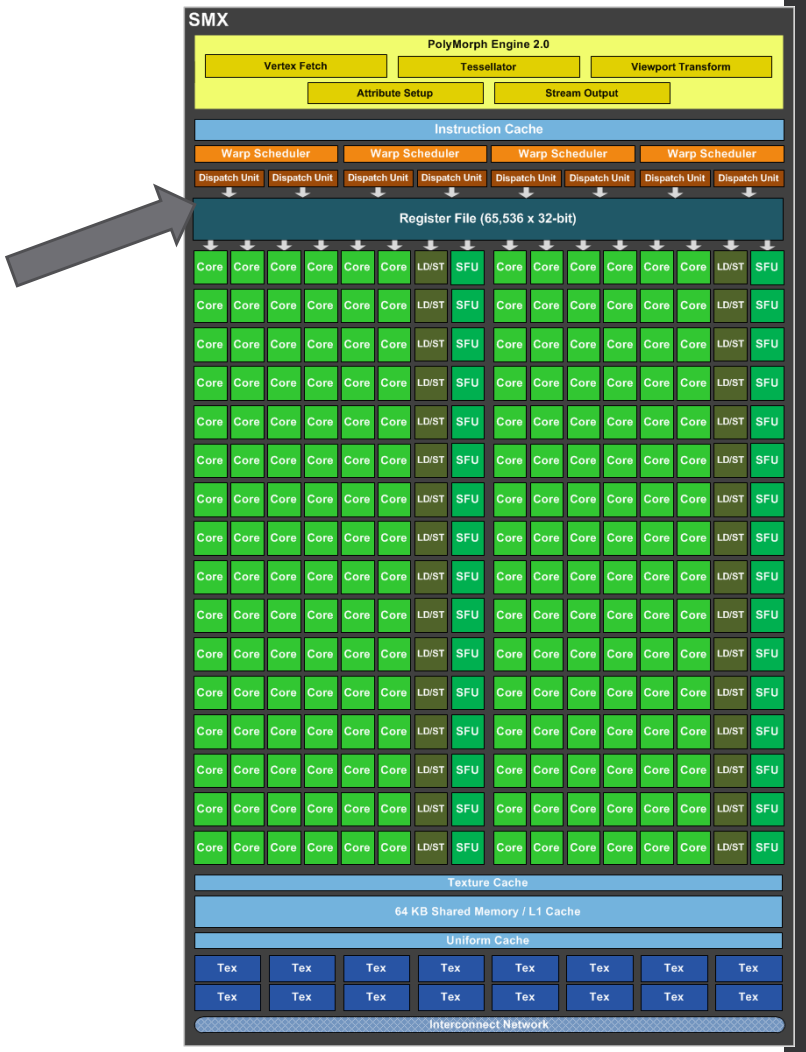
- Spills!
- It will significantly impact performance.
- It will spill to **local memory.**



Where is local memory???

What happens if we exceed register hardware limit?

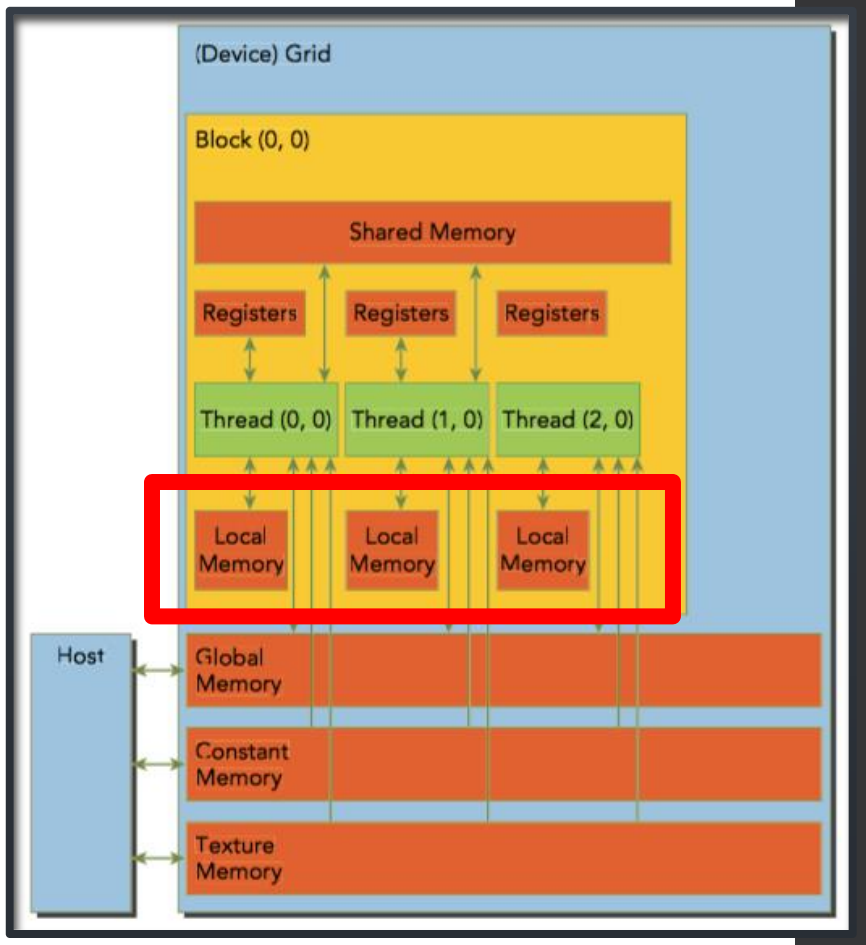
- Spills!
- It will significantly impact performance.
- It will spill to **local memory.**



It's in device memory!

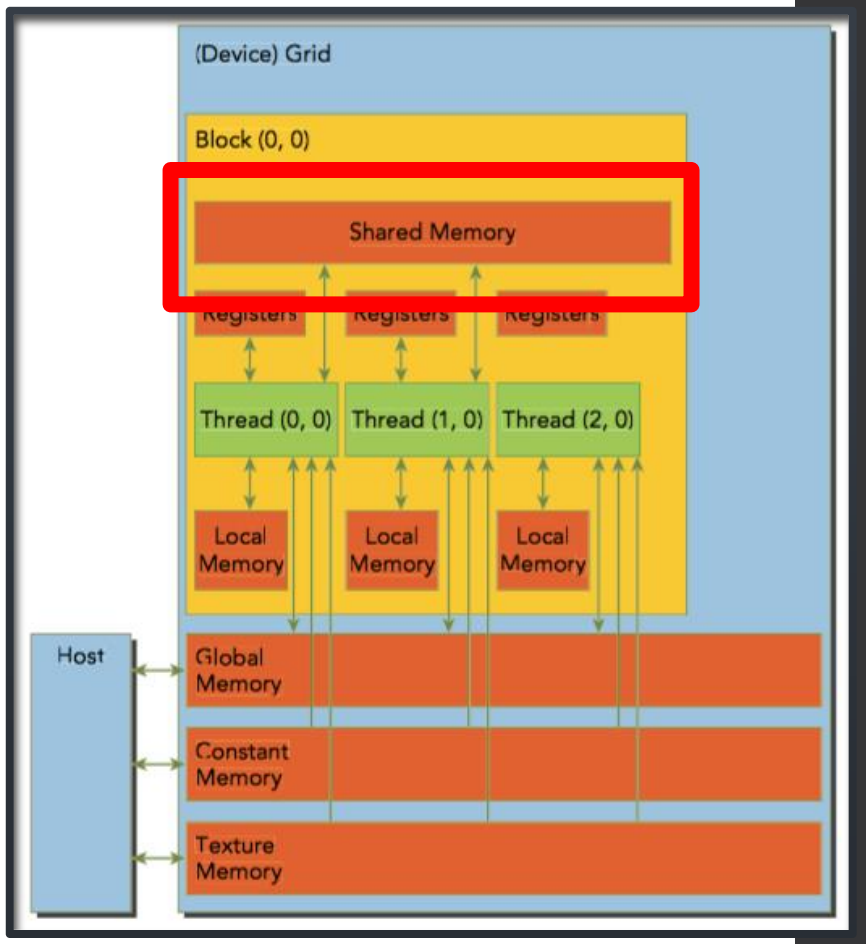
Local Memory

- 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
- Local memory is actually stored in DEVICE MEMORY!



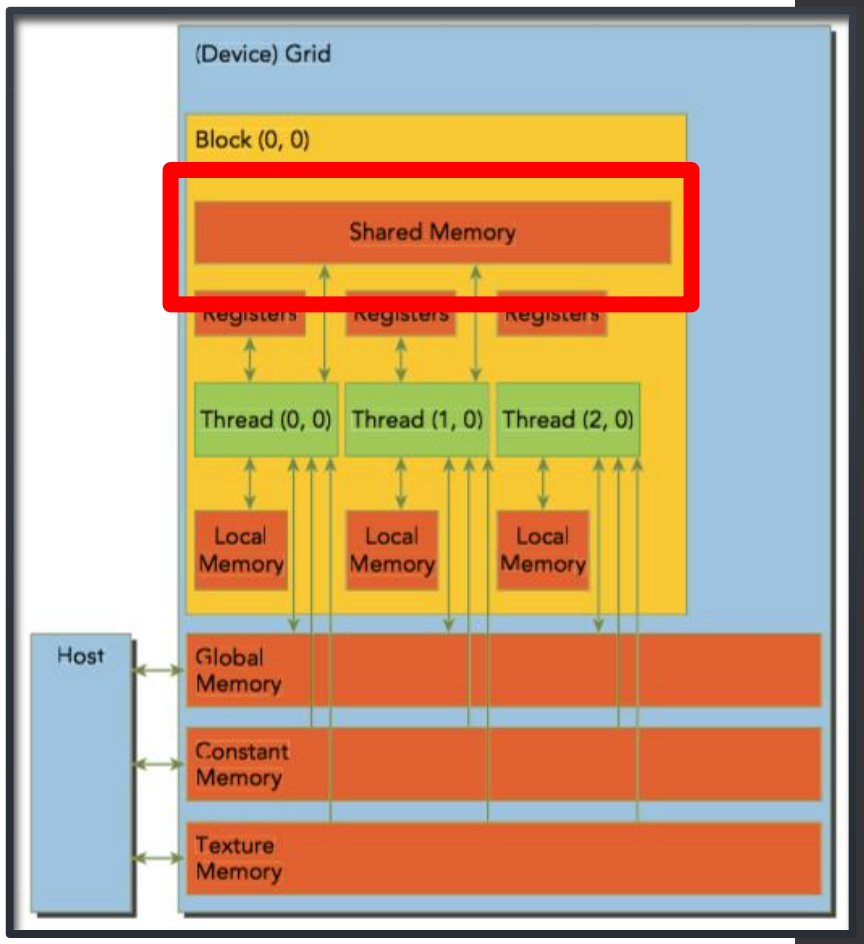
Shared Memory

- Use attribute: __shared__
- It is on-chip, therefore:
  - It has a much higher bandwidth
  - Lower latency
  - Much smaller size
- The BEST is that it is programmable. 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



Shared Memory

- It is declared on the kernel
- 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



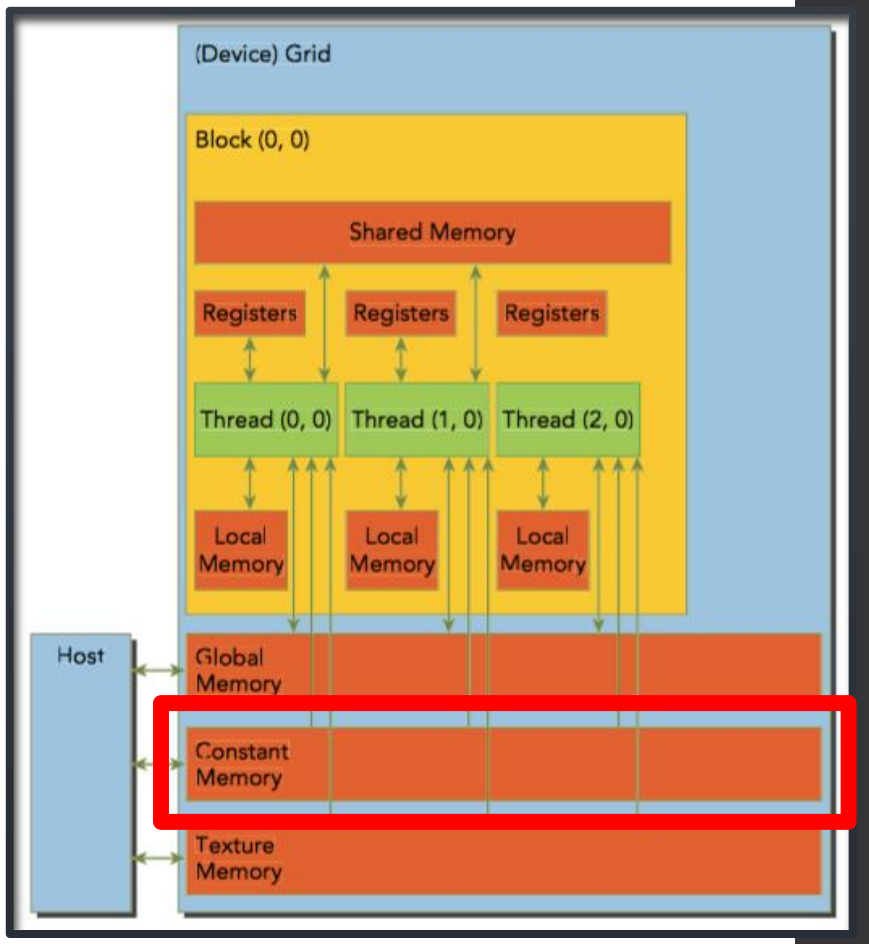
Shared Memory

- This is usually the best way to optimize your algorithms memory behavior.
- 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 unitialized 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 previousr 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];
}
```

Constant Memory

- Used attribute: __constant__
- 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.

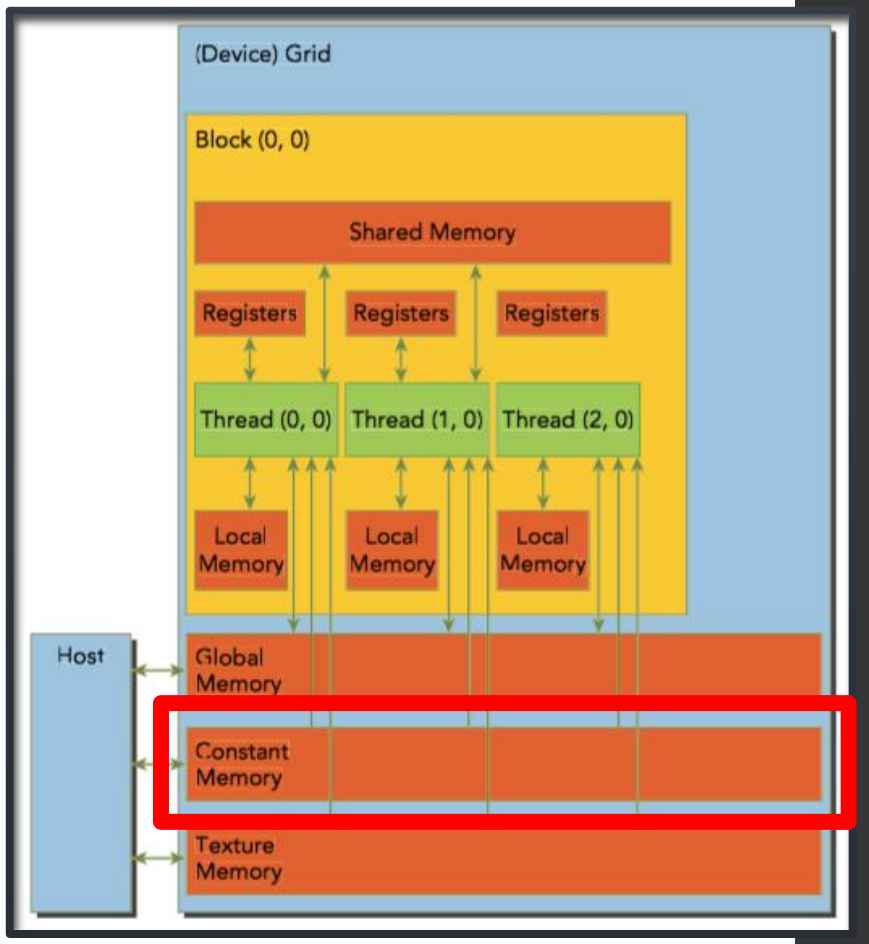


Constant Memory

- Host has to initizialize it:

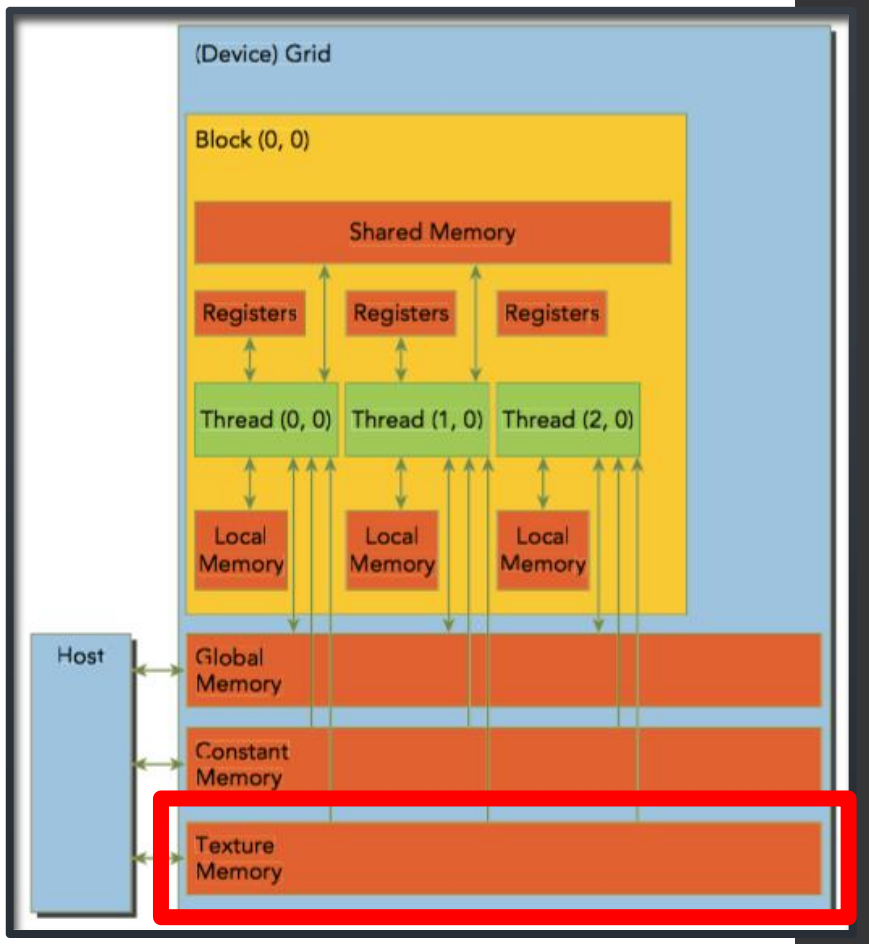
```
cudaError_t cudaMemcpyToSymbol(const void* symbol,
const void* src, size_t count);
```

- 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.



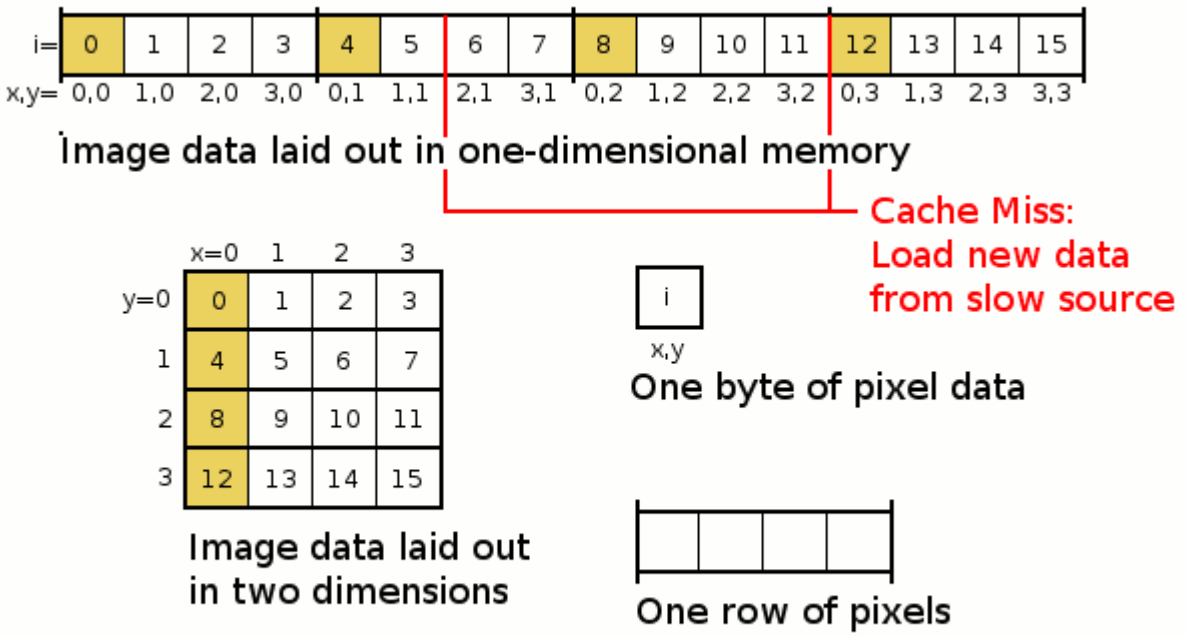
Texture Memory

- It resides in device memory as well
- It is cached in a per-SM fashion
- Read-only
- 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 spatial locality (i.e. imaging)



Texture Memory

- Example of how 2D spatial locality could work



Texture Memory

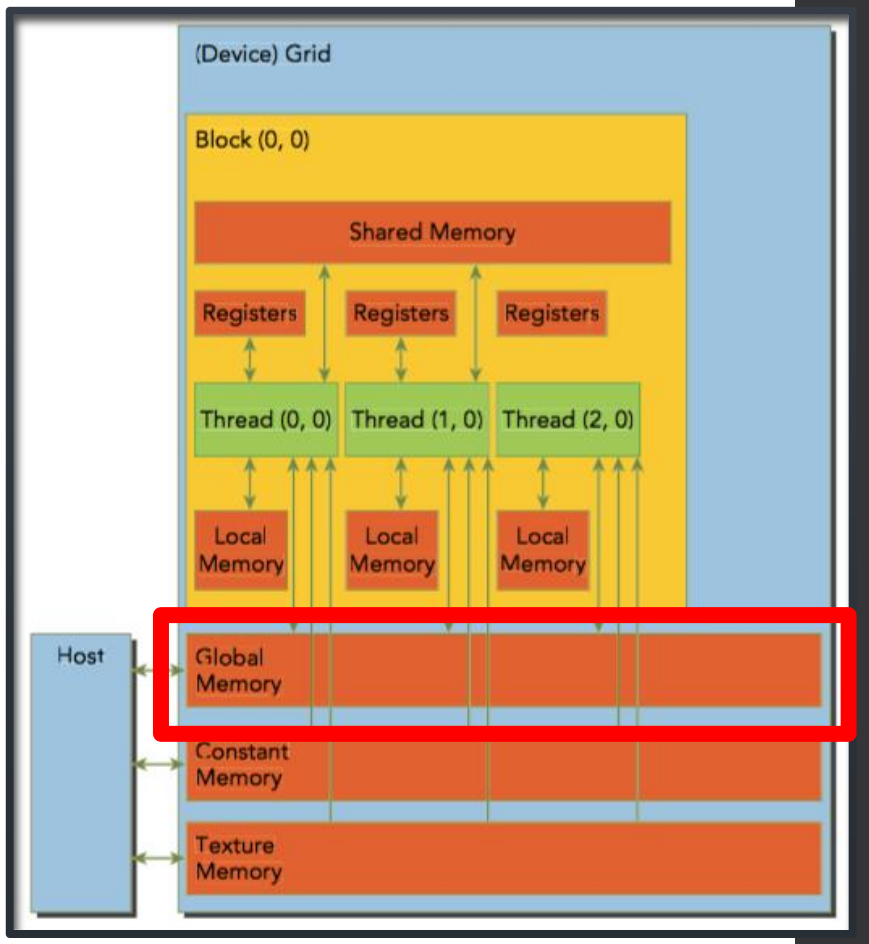
- Example of how 2D spatial locality could work (**Z-ORDER**)

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	2		
288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	3		
312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	3		
336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	3		
360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	3		

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

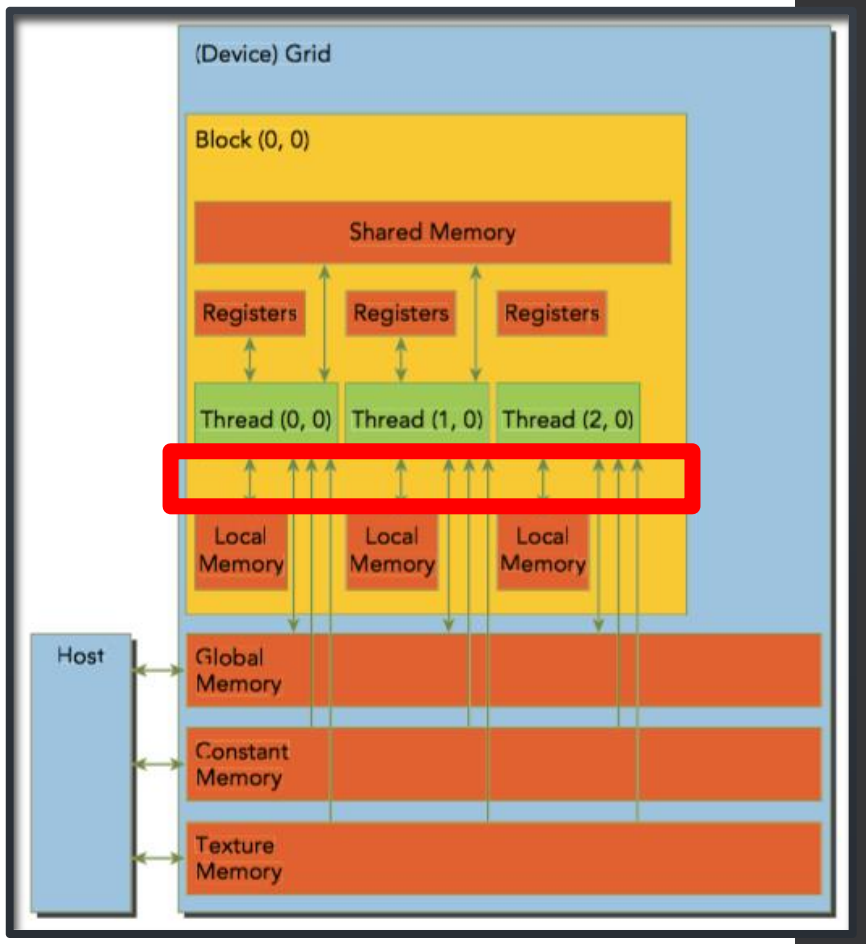
Global Memory

- Resides in device memory.
- Global scope: device level
- Largest memory on the GPU
- Highest latency
- How can we maximize the usage?



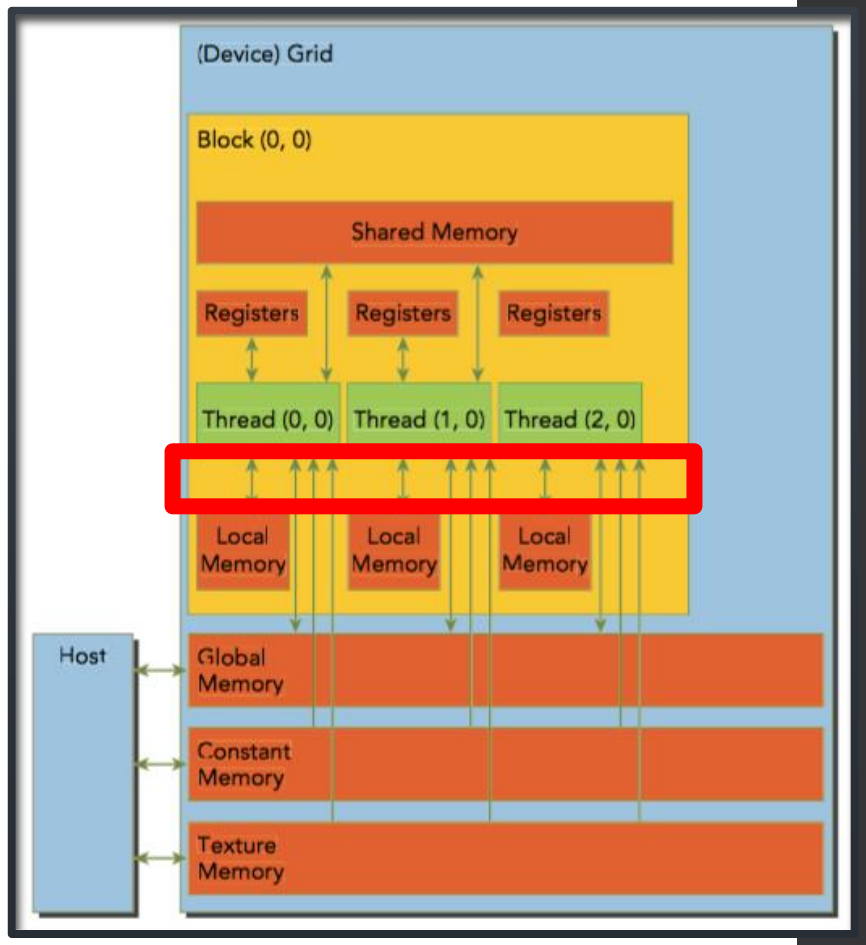
GPU Caches

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



GPU Caches

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



Summary of GPU Memories

QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	`float var`	Register	Thread	Thread
	`float var[100]`	Local	Thread	Thread
`__shared__`	`float var` †	Shared	Block	Block
`__device__`	`float var` †	Global	Global	Application
`__constant__`	`float var` †	Constant	Global	Application