CUDA Programming Model (Part 3)

ECE 277

Cheolhong An

Function type qualifier in CUDA

- __global__ qualifier is an entry point of device functions
- __device__ qualifier is used for the device functions

QUALIFIERS	EXECUTION	CALLABLE	NOTES
`__global__`	Executed on the device	Callable from the host Callable from the device for devices of compute capability 3	Must have a `void` return type (no return)
`__device__`	Executed on the device	Callable from the device only	
`__host__`	Executed on the host	Callable from the host only	Can be omitted

__global__ kernel function arguments

- CPU variables and pointers can be passed through __global__ kernel function arguments
- However, the pointers should be allocated to Device memory, otherwise, memory access violations (**GPU functions cannot access CPU memory directly**)
- CPU cannot access Device memory directly through pointer

  fprintf(1, “%f\n” d_MatA[0]); ← Access violation

```
__global__ void sumMatrixOnGPU1D(float *MatA, int nx, int ny)
{}

int main(int argc, char **argv)
{
    int nx = 1 << 14;          <-- CPU variable
    int ny = 1 << 14;
    float *d_MatA;             <-- CPU variable
    cudaMalloc((void **)&d_MatA, nBytes);
    sumMatrixOnGPU1D<<<grid, block>>>(d_MatA, nx, ny);
}
```

__global__ kernel function arguments

- Even though CPU variables can be passed through arguments, kernel cannot return variables directly
- cudaMemCpy should be used to return variables to CPU

```
__global__ void sumMatrixOnGPU1D(float *MatA, int &nx, int ny){
    if (ny == 4)
        nx = 3;
}

int main(int argc, char **argv)
{
    int nx = 1 << 14;
    int ny = 1 << 14;
    float *d_MatA;
    cudaMalloc((void **)&d_MatA, nBytes);
    sumMatrixOnGPU1D<<<grid, block>>>(d_MatA, nx, ny);
}
```

you can read but not allowed to write (cannot update, no return)

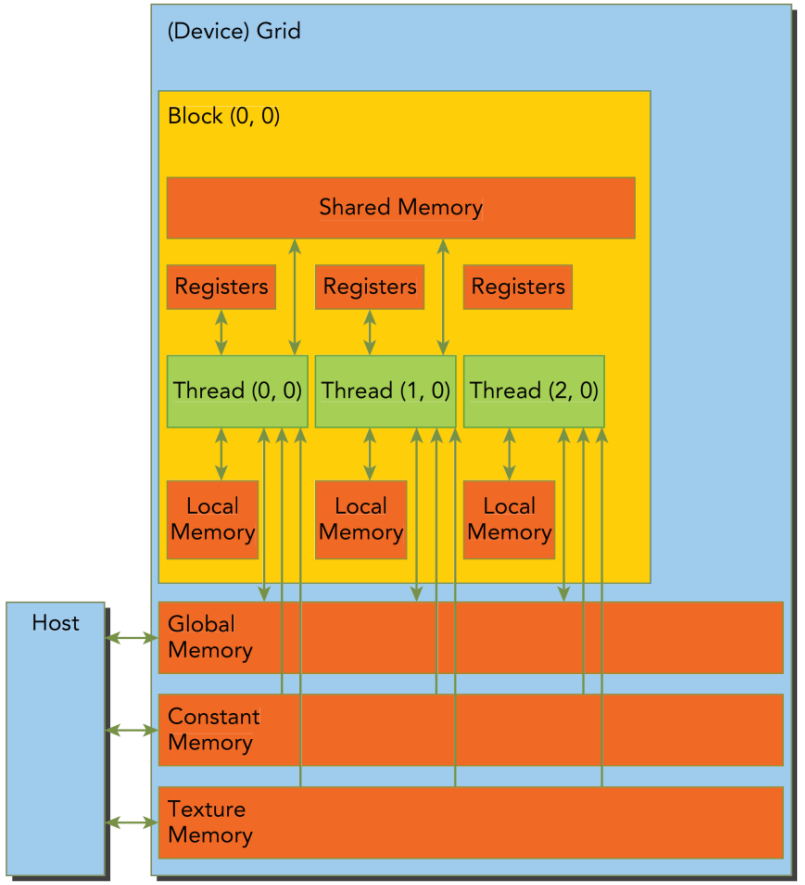
check the class lab
(c1_block1d_grid1d)

CUDA Memory Hierarchy and Models (1/2)

- Registers
- Shared memory (SMEM)
- Local memory (LMEM)
- Global memory (GMEM)
- Constant memory (CMEM)
- Texture memory

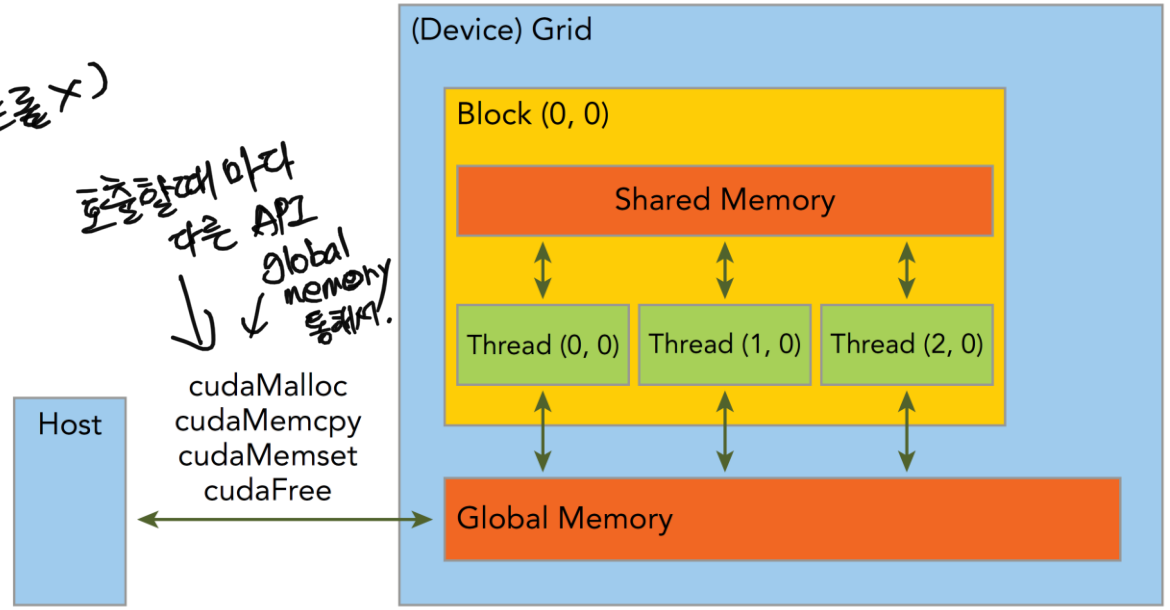
physically existed inside GPU core → on chip

→ external DRAM. (logically creation) → on board (GDDR)



CUDA Memory Hierarchy and Models (2/2)

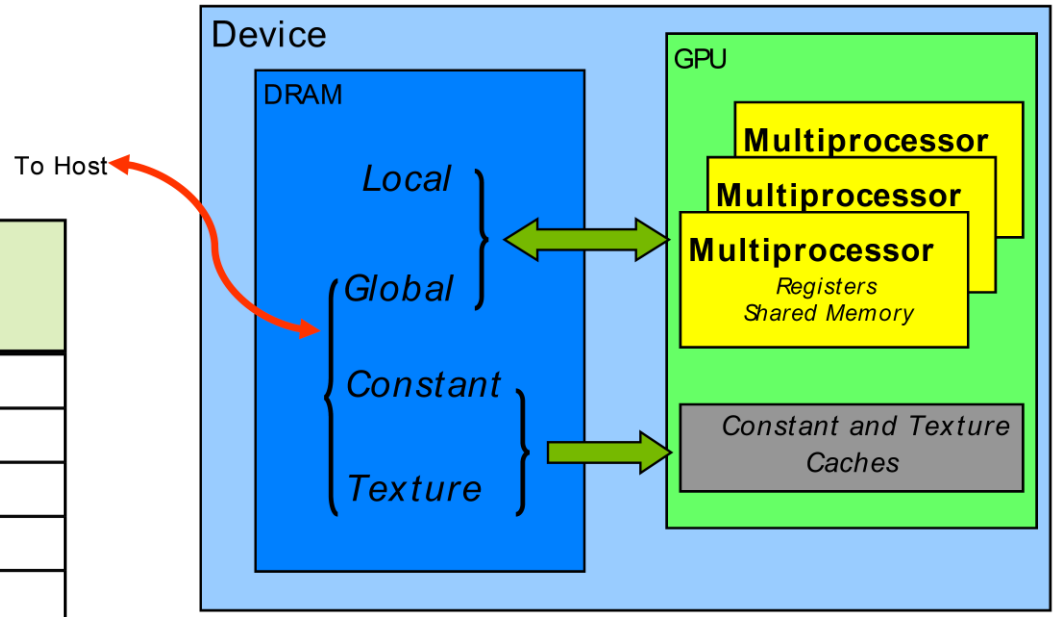
- Registers — on chip
- Shared memory (SMEM) — on chip
- Local memory (LMEM) — 컴파일러가 자동으로 (우리가 컨트롤 X)
- Global memory (GMEM) — on board (GDDR)
- Constant memory (CMEM) — on board (GDDR)
- Texture memory (TMEM) — on board (GDDR)



Different approaches are required to create GMEM, CMEM, SMEM, TMEM except the local memory, which is created automatically

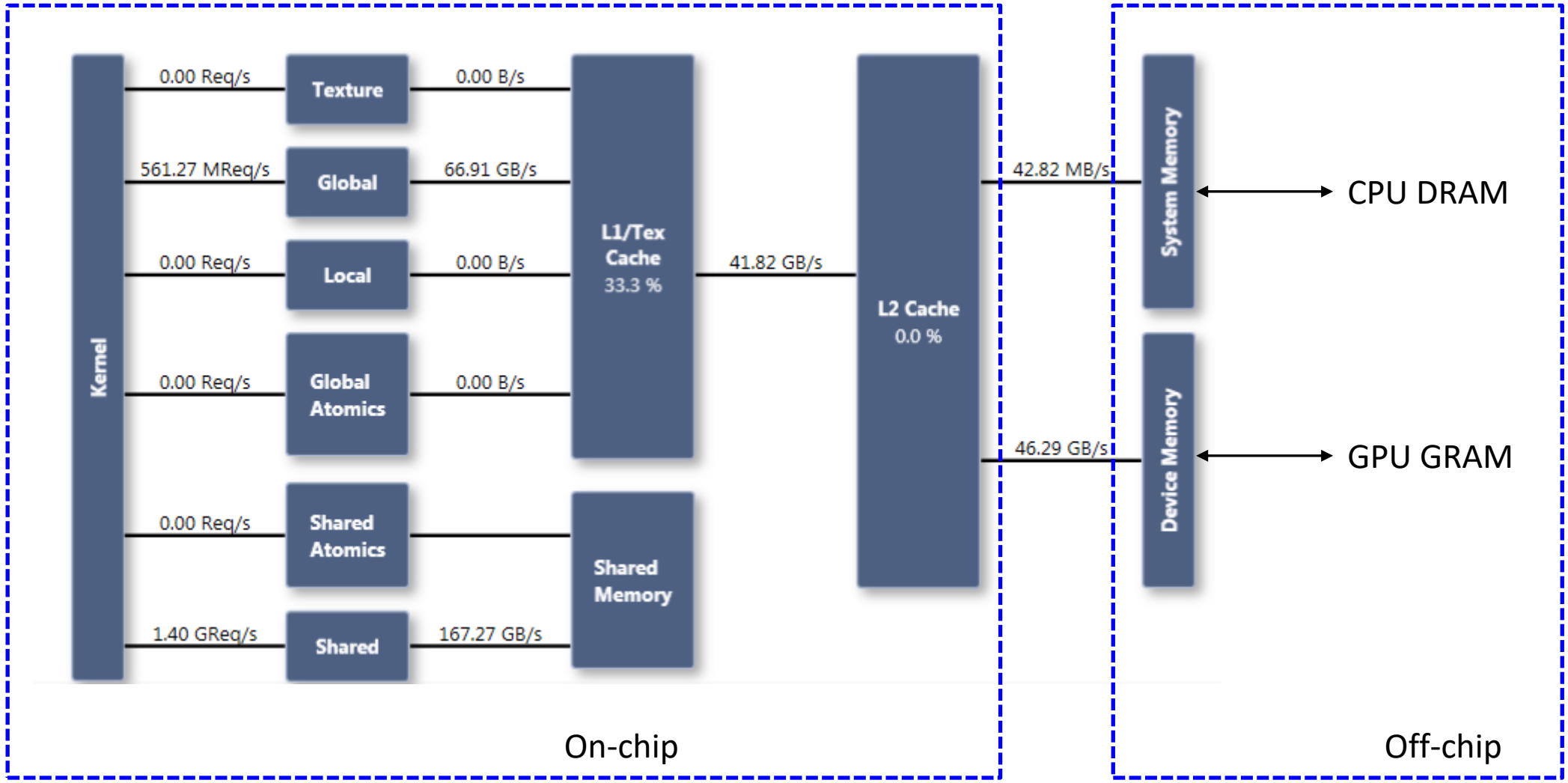
Memory Hierarchy Scope and Lifetime

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† Cached only on devices of compute capability 2.x.					



Nvidia, CUDA C BEST PRACTICES GUIDE, Feb, 2014

Memory Statistics Example



Summary of CUDA Memory Model

QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	float var	Register	Thread	Thread
	float var[100]	Local	Thread	Thread
__shared__	float var	SMEM	Block	Block
__device__	float var	GMEM	Global	Application
__constant__	float var	GMEM (Read only)	Global	Application

Global Memory (GMEM) (1/3)

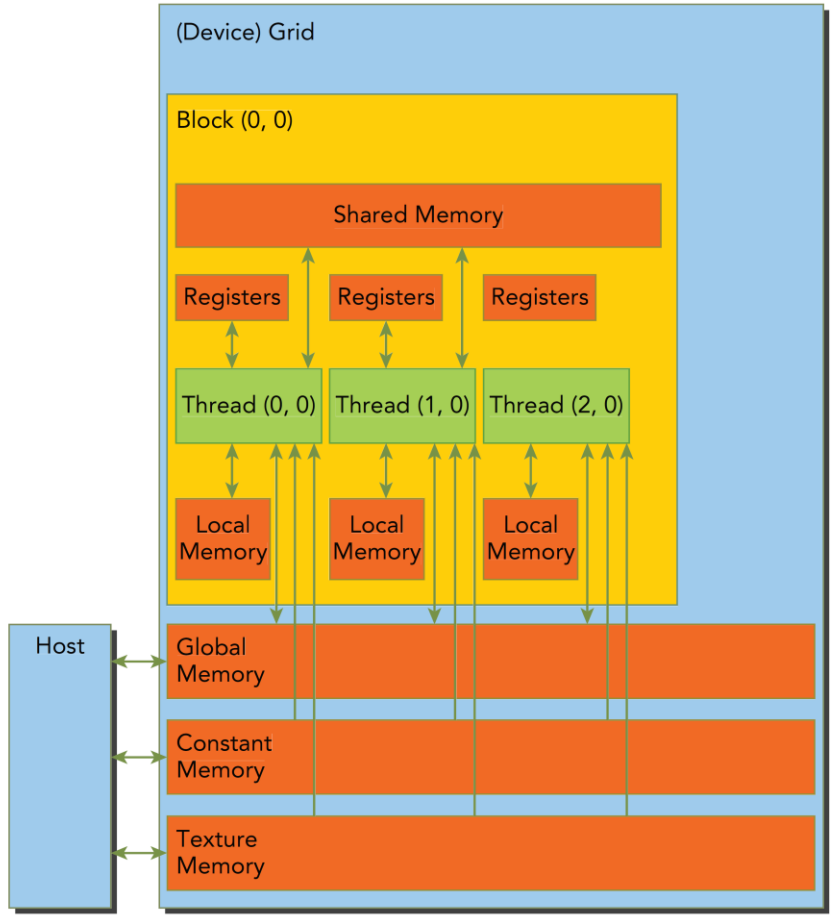
- GMEM resides in the device memory
- GMEM allocation
  - Static

```
__device__ int A
```

  - Dynamic

```
cudaMalloc
cudaFree
```

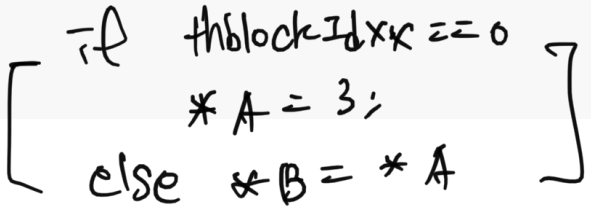
- All the treads can access GMEM, but there is no synchronization method except atomic operations
- Global atomics is very slow



Global Memory (2/3)

- Write and read to GMEM can be arbitrary order (not synchronized) even if two thread blocks can access GMEM

```
kernel12<<<2,n>>>(); // -> write GMEM (thblockIdx=0), <- read GMEM
(thblockIdx=1)
```



- Sequential Kernel calls to the same stream is always ordered
  (ex. Kernel2 can execute after kernel1 completes)

```
kernel1<<<1,n>>>(); // -> write GMEM
kernel2<<<1,n>>>(); // <- read GMEM
```

Pytorch example

- Sequential CUDA-kernel call

sequential

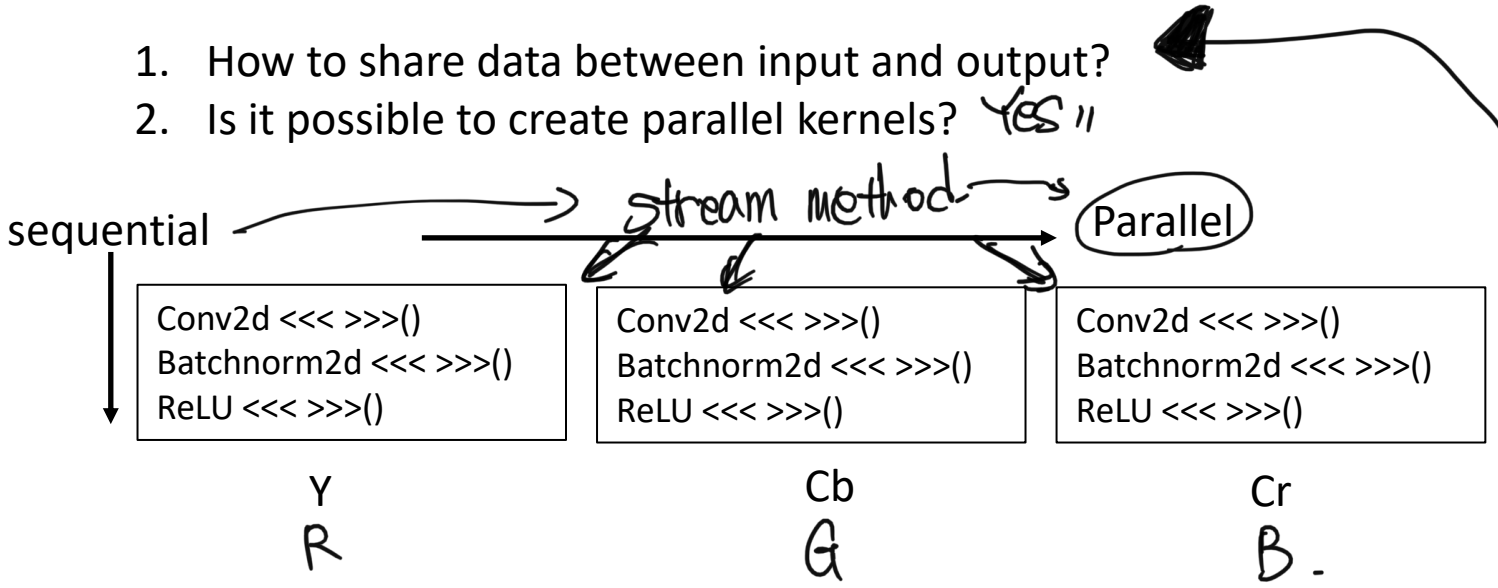
Conv2d <<< >>>()

Batchnorm2d <<< >>>()

ReLU <<< >>>()

```
def conv_block(in_f, out_f, *args, **kwargs):
    return nn.Sequential(
        nn.Conv2d(in_f, out_f, *args, **kwargs),
        nn.BatchNorm2d(out_f),
        nn.ReLU()
    )
```

1. How to share data between input and output?
2. Is it possible to create parallel kernels? Yes !!




```
class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, n_classes):
        super().__init__()
        self.conv_block1 = conv_block(in_c, 32, kernel_size=3, padding=1)

        self.conv_block2 = conv_block(32, 64, kernel_size=3, padding=1)


        self.decoder = nn.Sequential(
            nn.Linear(32 * 28 * 28, 1024),
            nn.Sigmoid(),
            nn.Linear(1024, n_classes)
        )


    def forward(self, x):
        x = self.conv_block1(x)
        x = self.conv_block2(x)

        x = x.view(x.size(0), -1) # flat

        x = self.decoder(x)

        return x
```

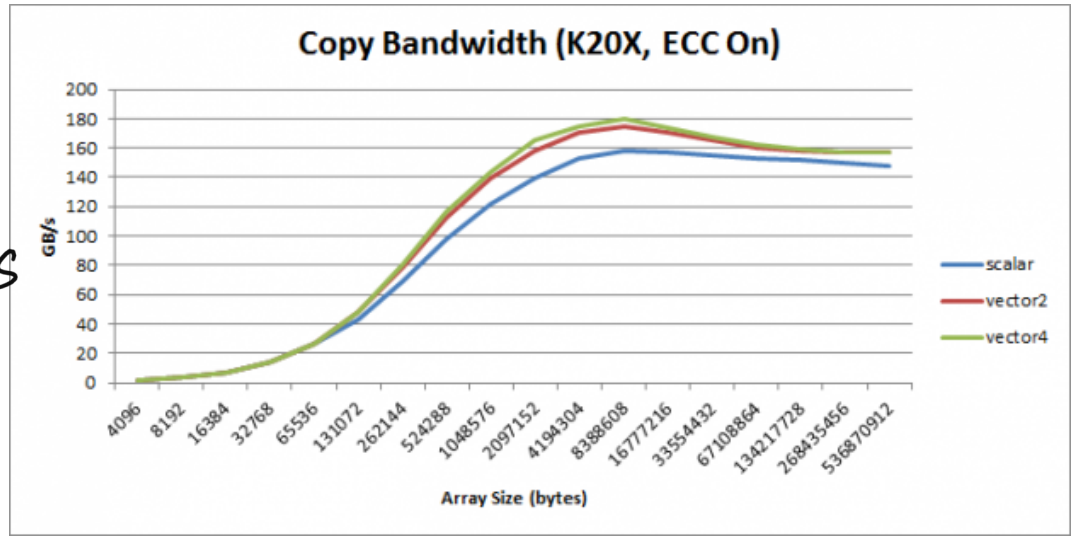
Global Memory (3/3)

- GMEM latency : 400-800 cycles → GPU clock cycle.
- Latency hiding
  - switching warps
  - Need enough warps in a thread block
- Load/Store larger word size (vectorized load/store)
  - Byte(1B) < short(2B) < Int/float (4B) < Int2/float2 (8B) < Int4/float4(16B)

*char RGB ⇒ load a set of pixels at the same time. → increase memory access

8 bit/sample

read multiple pixel data.
(4 pixel/thread)
⇒ 4 bit ...

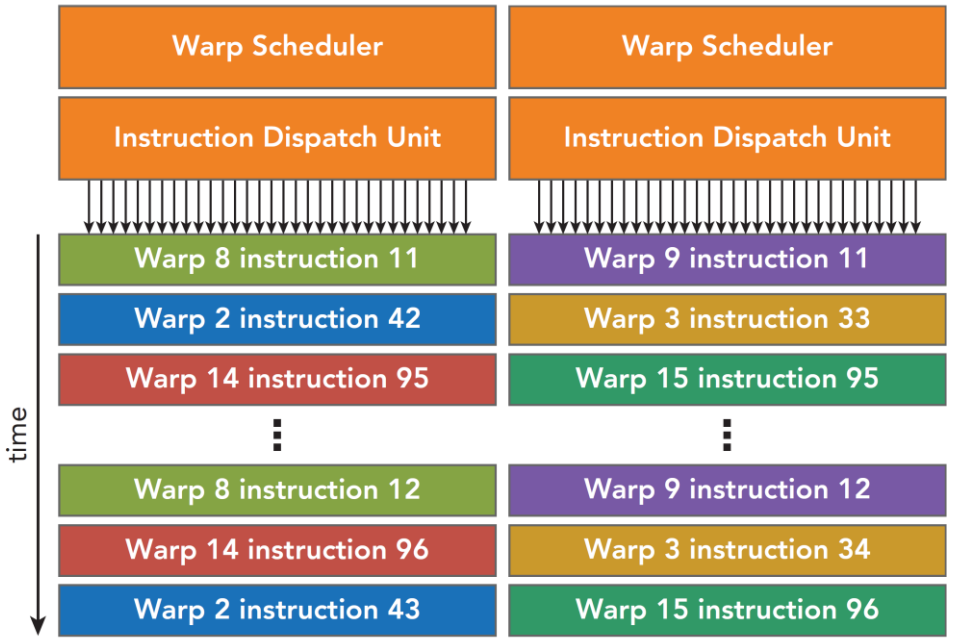


https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

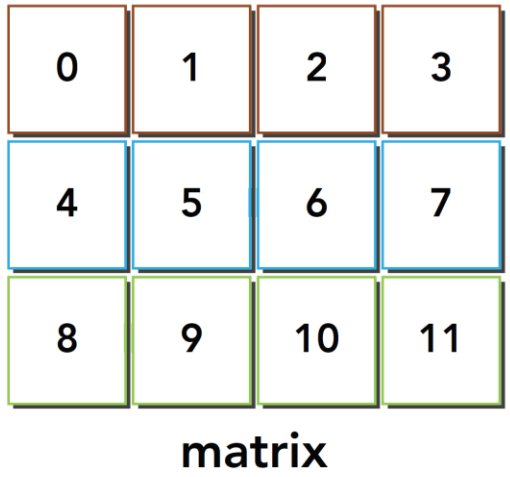
Warp Scheduling within a threadblock

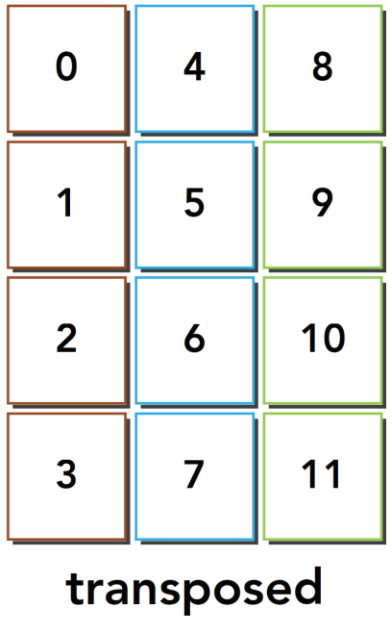
- Latency hiding of GMEM (very large latency.)

  2~8 warps per threadblock is necessary for the latency hiding



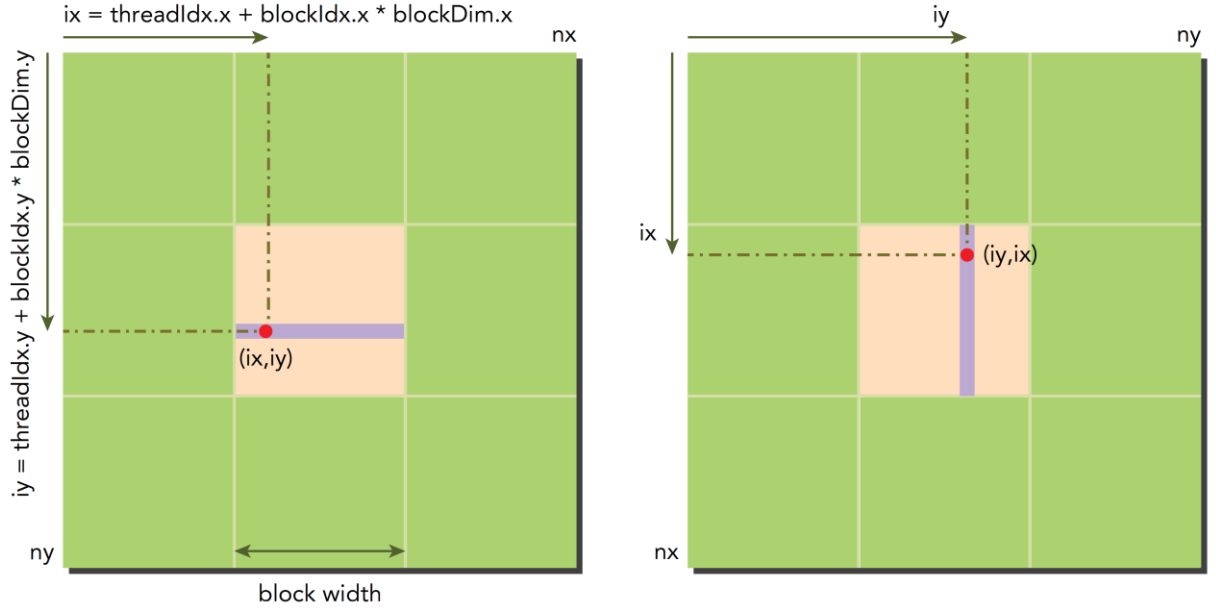
GMEM Example: Transpose of a Matrix





GMEM Example: Transpose

- A large matrix should be partitioned into threadblocks to increase parallelism
- Map a thread to an element of the matrix



"transpose operation"

```
__global__ void transposeNaiveRow(float *out, float *in, const int nx, const int ny) {
  unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
  if (ix < nx && iy < ny) {
    out[ix * ny + iy] = in[iy * nx + ix];
  }
}
```

CUDA Vector type data

- Increase bytes per (memory or PCI) transaction with vector type data
- Vector type
  int (4B) -> int2 (8B), int4 (16B)
  short (2B) -> short2 (4B), short4 (8B)
  char(1B) -> char2 (2B), char4 (4B)

  double (8B) -> double2 (16B)
  float (4B) -> float2 (8B), float4 (16B)
  half(2B) -> half2 (4B)

byte.

data bandwidth / request

⇒ performance increase.

- There are also float3 (12B), int3 (12B), char3 (3B), short3 (6B)
- However, you shouldn't use in general, which introduces memory misalignment.

CUDA Floating point

- Single precision (FP32) : 32 bits
- Double precision (FP64) : 64 bits
- Half precision (FP16) : 16 bits

- Tradeoffs accuracy vs. performance
- Half precision is sufficient for training neural networks

- CPU half precision library:
  IEEE 754-based half-precision floating point library
  http://half.sourceforge.net/

Floating point performance

- Performance of floating point is highly related to GPU HW (compute capability)
- Specially, **half precision should be used only for compute capability 6.1**
- But, Half precision is still effective to reduce transfer overheads

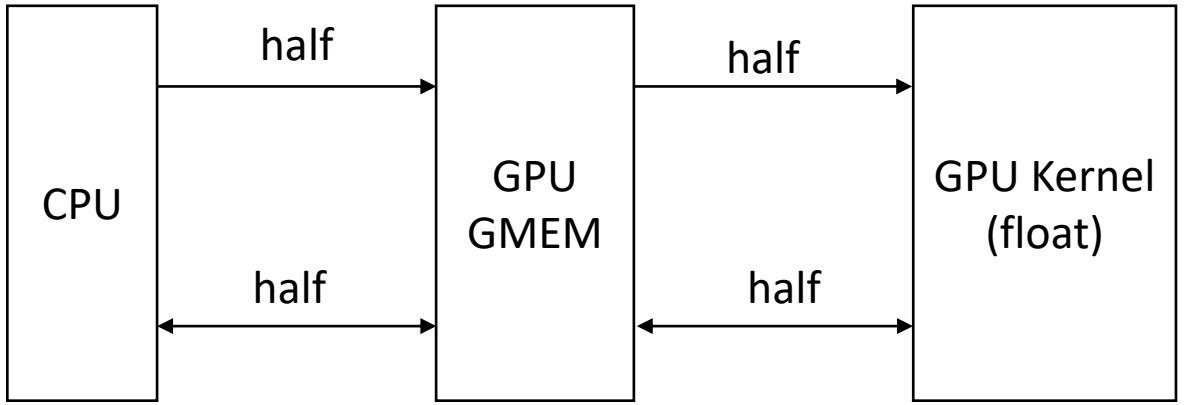
	Compute Capability							
	3.0, 3.2	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.0
16-bit floating-point add, multiply, multiply-add	N/A	N/A	N/A	256	128	2	256	128
32-bit floating-point add, multiply, multiply-add	192	192	128	128	64	128	128	64
64-bit floating-point add, multiply, multiply-add	8	64[2]	4	4	32	4	4	32
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	32	32	32	32	16	32	32	16
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	160	128	128	64	128	128	64
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	32	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	Multiple instruct.	64[3]

Pascal Hardware Numerical Throughput

GPU	DFMA (FP64 TFLOP/s)	FFMA (FP32 TFLOP/s)	HFMA2 (FP16 TFLOP/s)	DP4A (INT8 TIOP/s)	DP2A (INT16/8 TIOP/s)
GP100 (Tesla P100 NVLink)	5.3	10.6	21.2	NA	NA
GP102 (Tesla P40)	0.37	11.8	0.19	43.9	23.5
GP104 (Tesla P4)	0.17	8.9	0.09	21.8	10.9

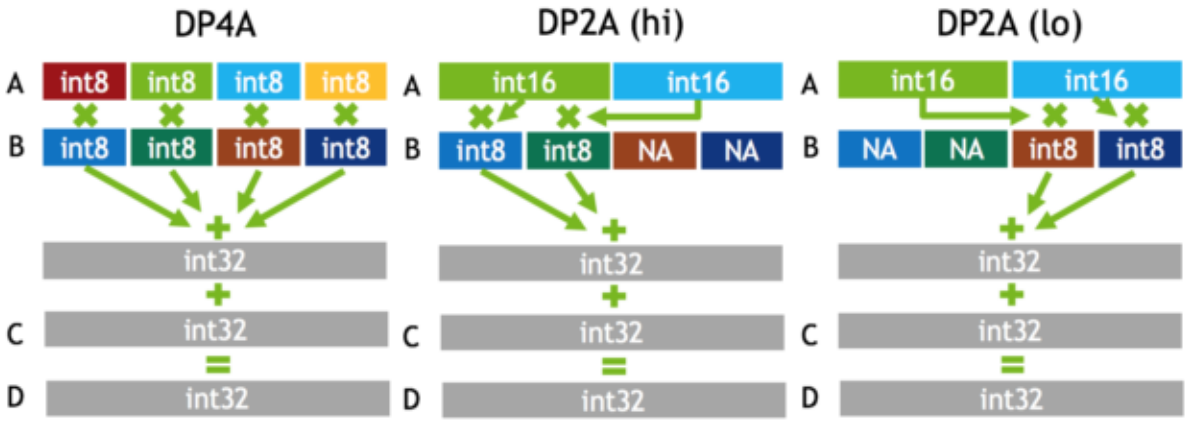
Lab example: Half precision

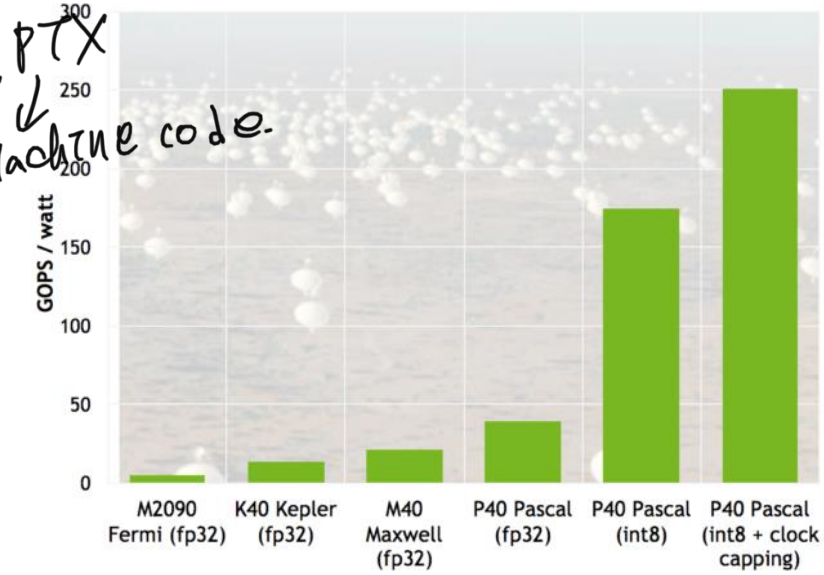
- Minimize transfer overheads
- Use half precision to transfer data (host-to-device, device-to-device)
- In GPU kernel, convert half to float or float to half
  float to half: __float2half
  half to float: __half2float
- Class lab: c3_fp16

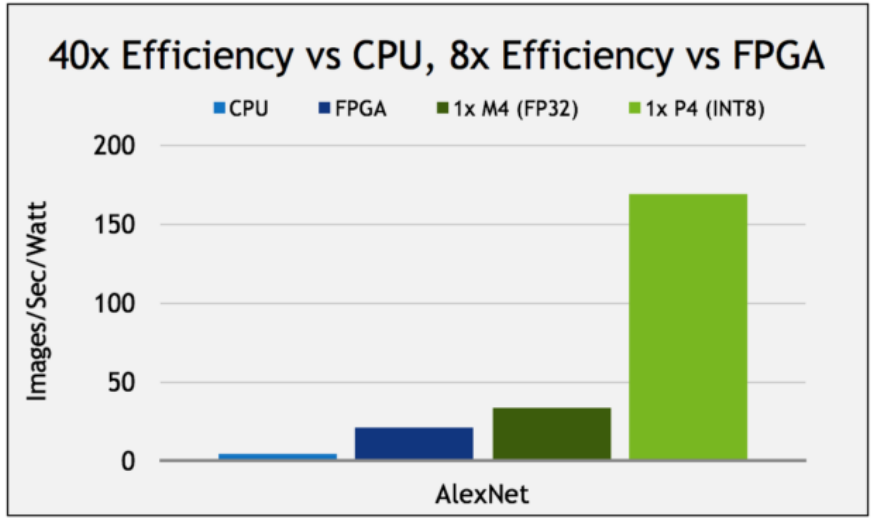


CUDA: vector operations

- CUDA supports vector operations for shorter data types
  Int8 (1B), Int16 (2B), int (4B)
- 4 samples per one instruction (one thread)





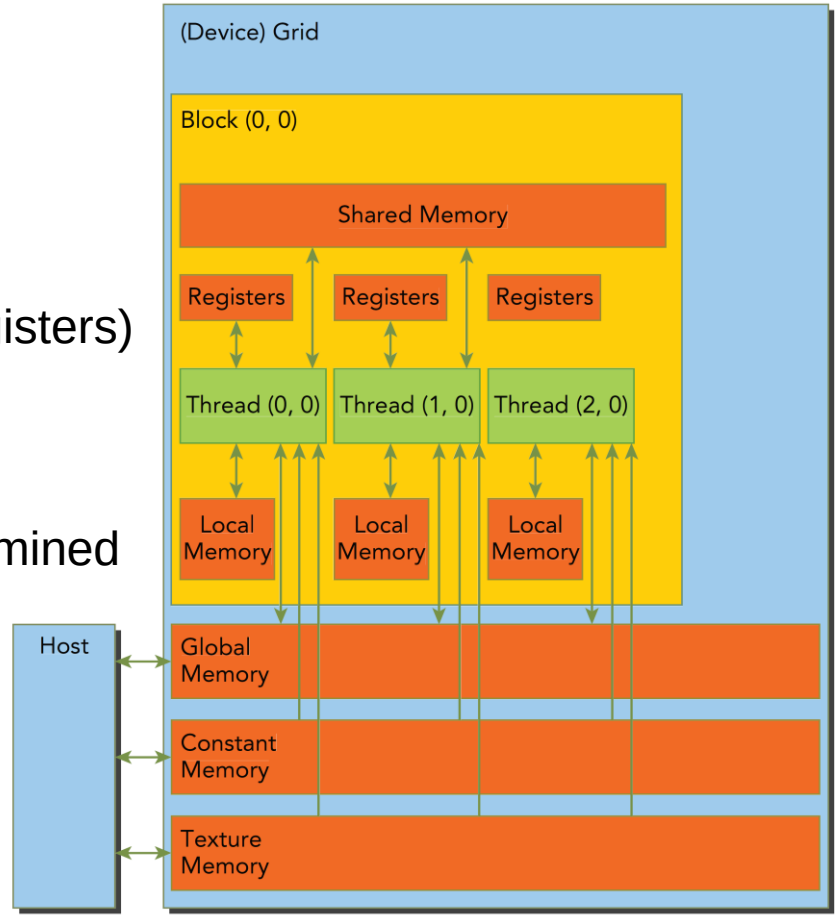


Local Memory (LMEM)

- Each thread has its own local memory
- Local memory is automatically created by CUDA compiler
- Local memory cannot be shared with any other threads
- Local memory is physically located at the off-chip memory (GDDR)
- Local memory is very slow without L1 cache
- Any variable that does not fit within the kernel register limit (256 registers)
- Large local structures or arrays consume too many registers

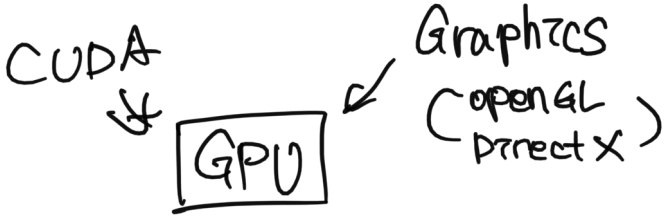
  int A[100] ⇒ × 2 warp ⇒ 64 times
- Local arrays referenced with indices whose values cannot be determined at compile time

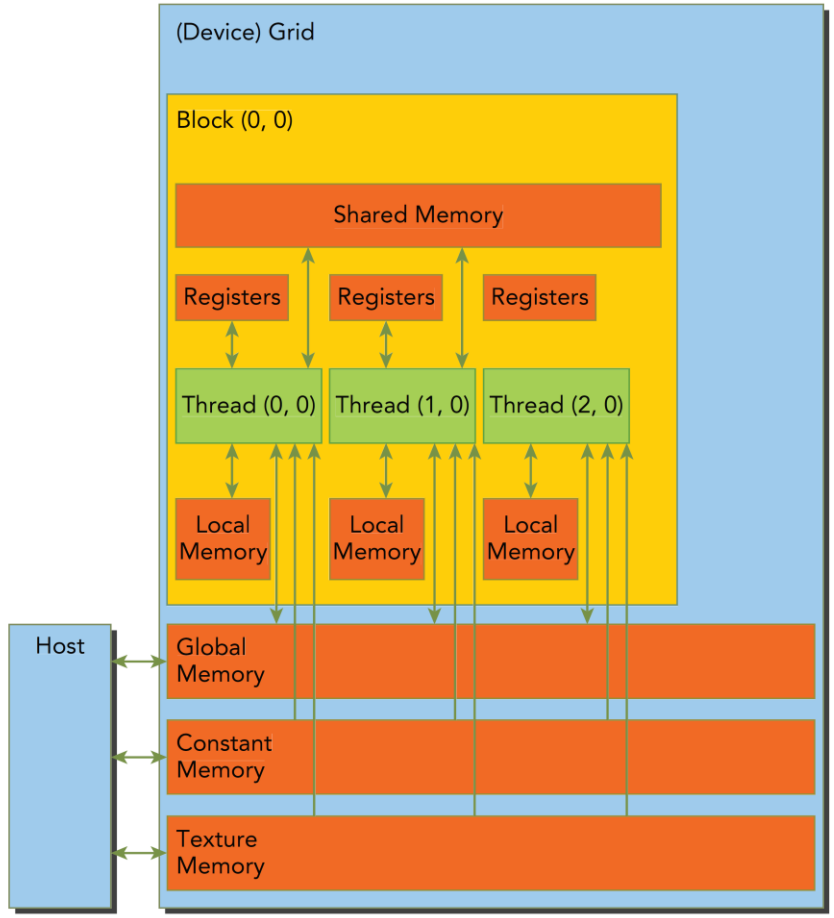
```
int A[10];
For (k=0; k<10; k++)
 A[B] = B+k; // B is only assigned into registers
```



Texture Memory (TMEM)

- Texture memory resides in device memory and is cached in a per-SM, read-only cache
- Texture memory supports dedicate HW features
  - Type conversion Int to float (HW, not CUDA)
  - Type conversion is not free. It takes time (32 cycles) which is roughly half of the arithmetic operations
  - Interpolation (HW, not CUDA)
- Texture memory is not good for general purpose since It is slower than the global memory
- Size of TMEM is limited





Constant Memory

- Very fast on-board memory area Read only (dedicated cache)
- Must be set by the host up to 64 KB
- Coalesced access if all threads of a warp read the same address (serialized otherwise)

- __constant__ qualifier in declarations

  Useful to off-load long argument lists from shared memory for coefficients and other data that is read uniformly by warps

  e.g. filter coefficients

```
__constant__  float constHueColorSpaceMat[4] = { 1.24f, 0.0f, 1.596f, 1.1644f };
```

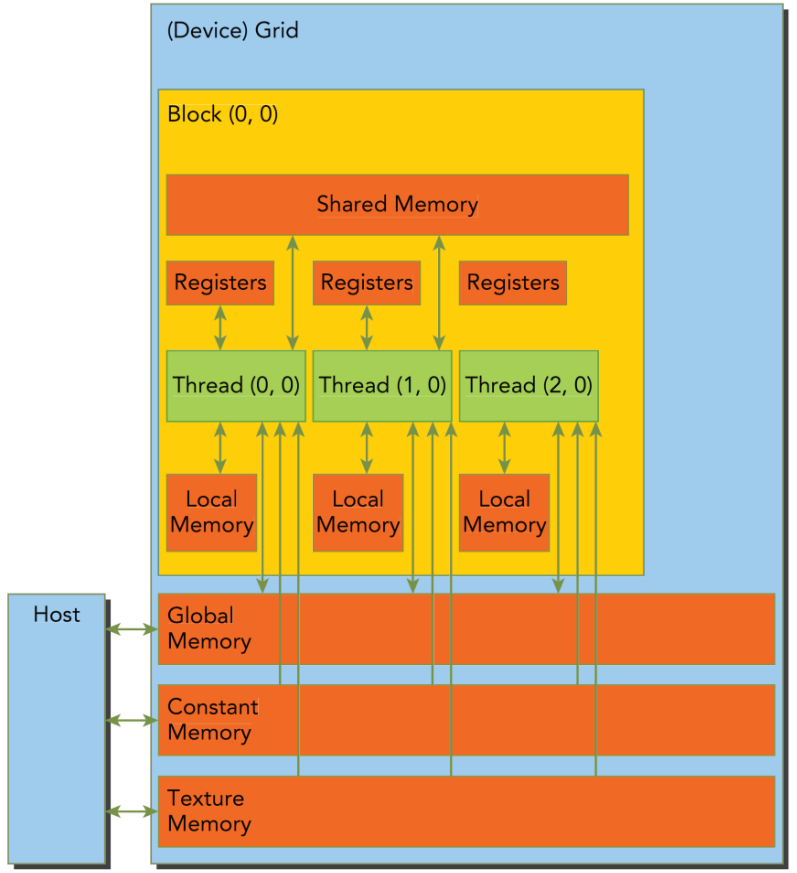
Registers (1/3)

- Fast read/write memory on GPU
- Register variables share their lifetime with the kernel.

  Once a kernel completes execution, a register variable cannot be accessed again
- Each thread has its own registers
- The other threads (even in the same warp) cannot access thread registers
- **Threads in the same warp** can exchange register values through shuffle instructions

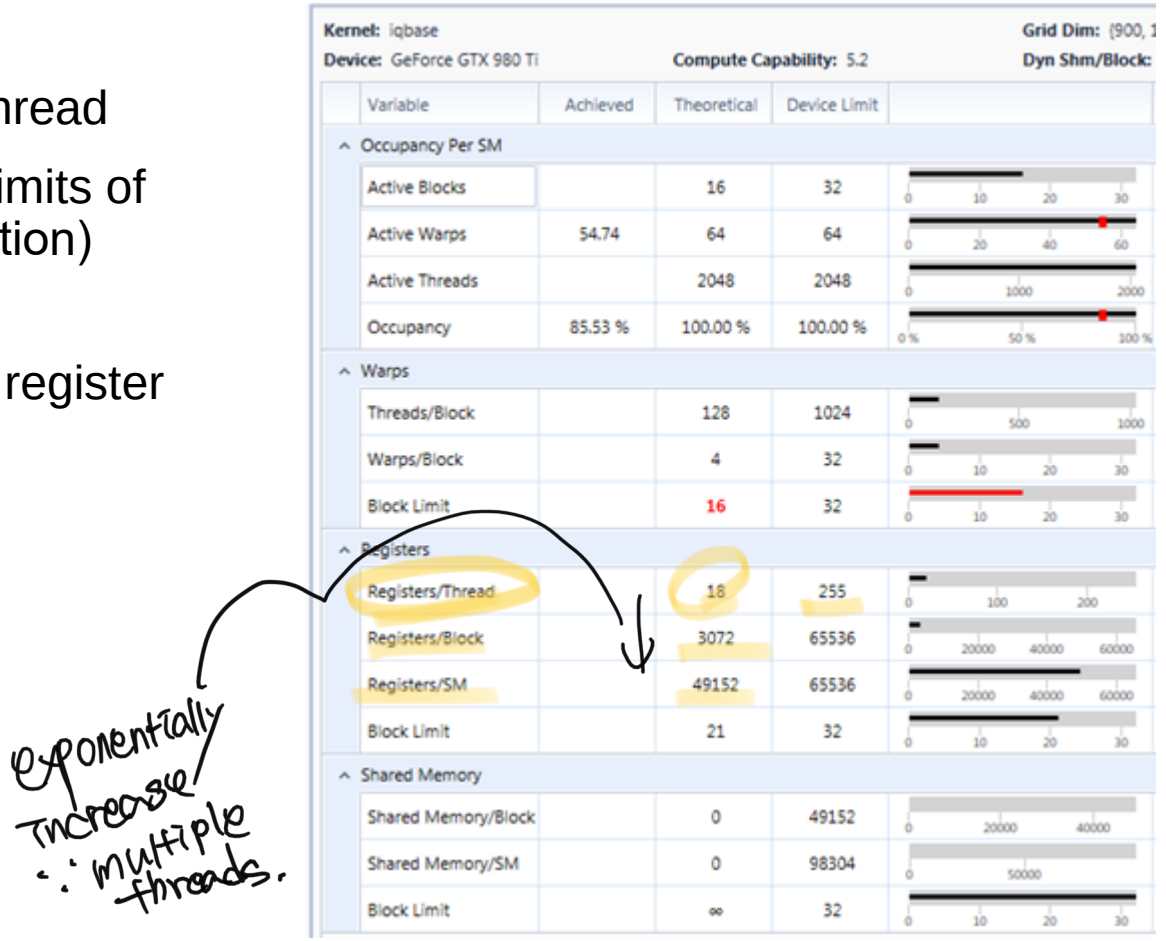
  __shfl, __shfl_down, __shfl_up, __shfl_xor

  fastest way to exchange data without any synchronization

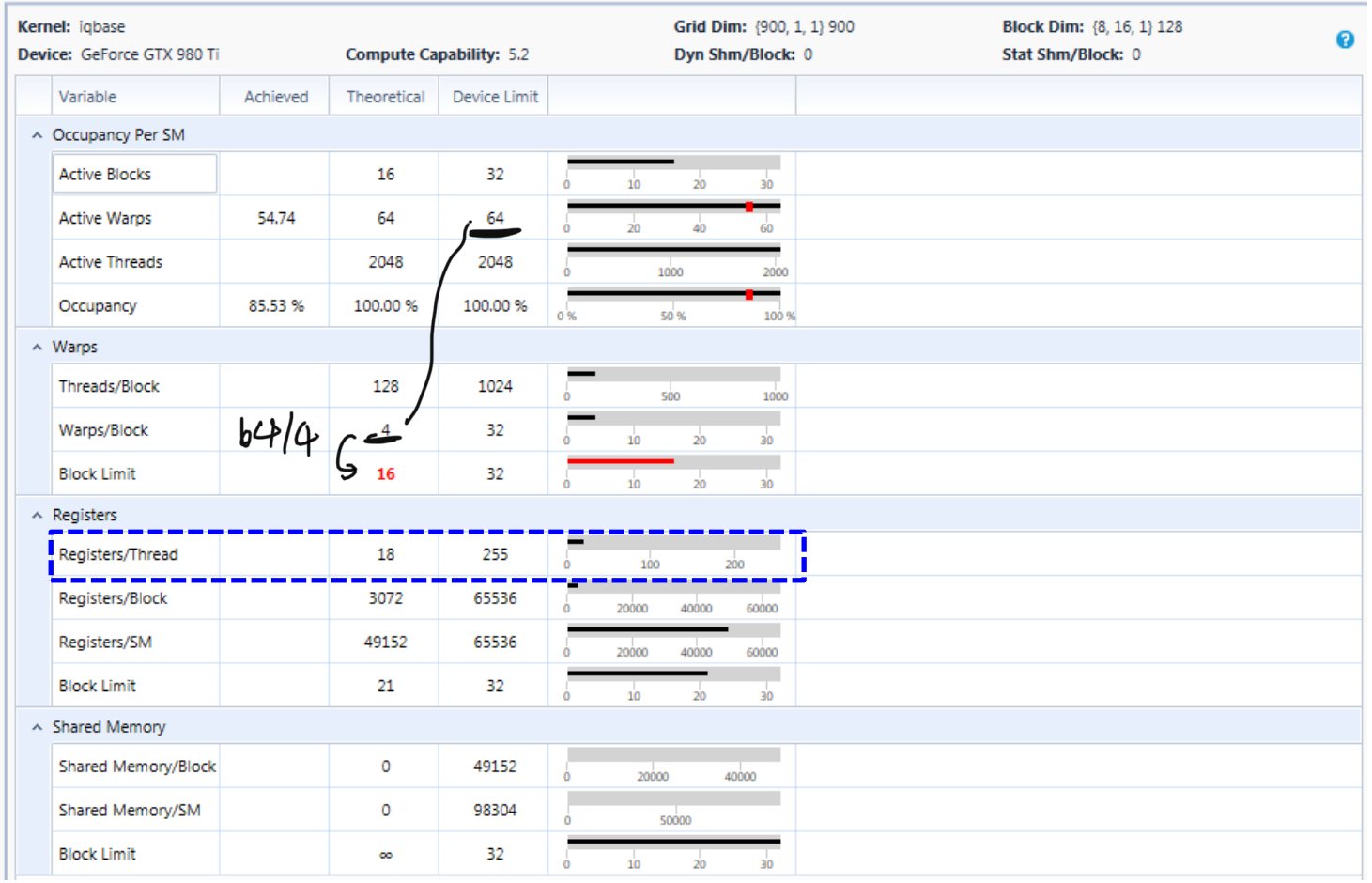


Registers (2/3)

- Registers/Thread
- Compiler automatically determine Registers/Thread
- But a programmer can specify the maximum limits of Registers/Thread (“-maxrregcount” compile option)
- Should prevent register spilling
  - LMEM is used if the source code exceeds register limit (“–Xptxas –v”)
  - Use a higher limit in –maxrregcount



MULTI_GPU_BOARD	0
MULTI_GPU_BOARD_GROUP_ID	0
MULTIPROCESSOR_COUNT	22
PAGEABLE_MEMORY_ACCESS	0
PCI_BUS_ID	1
PCI_DEVICE_ID	0
PCI_DOMAIN_ID	0
RAM_LOCATION	1
RAM_TYPE	8
SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO	32
STREAM_PRIORITIES_SUPPORTED	1
SURFACE_ALIGNMENT	512
TCC_DRIVER	0
TEXTURE_ALIGNMENT	512
TEXTURE_PITCH_ALIGNMENT	32
TOTAL_CONSTANT_MEMORY	65536
TOTAL_MEMORY	6442450944
UNIFIED_ADDRESSING	1
WARP_SIZE	32



Register allocation granularity per warp: 256

warp-level ol.

$$\text{Resiters/Warp} = \lceil 18 \times 32/256 \rceil \times 256 = 768$$

$$\text{Resiters/Block} = 768[\text{Resiters/Warp}] \times 4[\text{Warps/Block}] = 3072$$

$$\text{Block limit} = \lfloor \frac{65536}{3072} \rfloor = \lfloor 21.33 \rfloor = 21$$

$$Registers/SM = 3072 \times 16 = 49152$$

Registers (3/3)

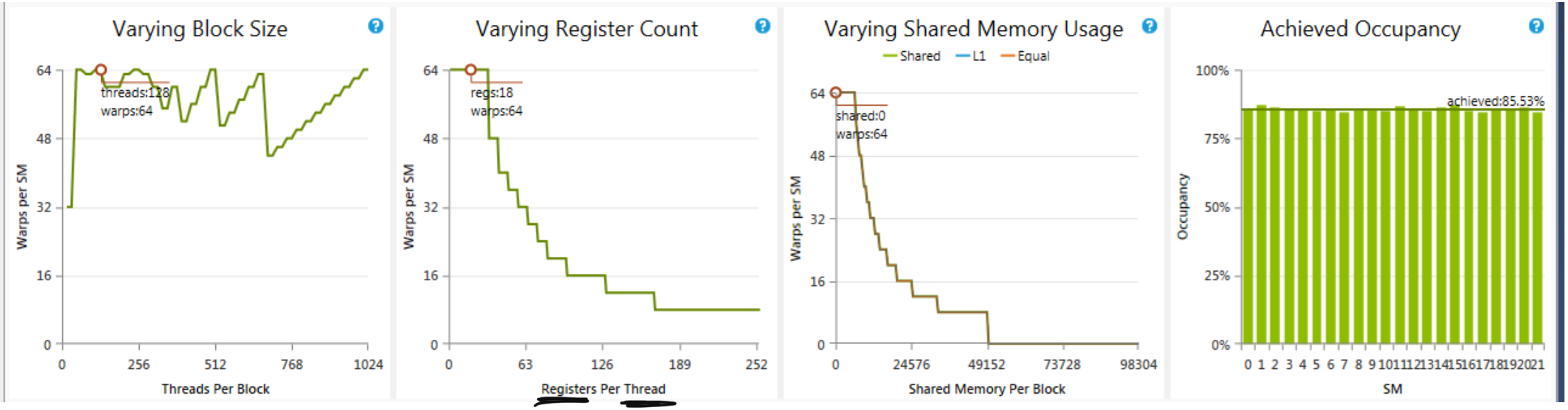
- Variables in a code are automatically assigned to registers

  A += B //(A and B are assigned to registers)

- Local arrays referenced with indices whose values are determined at the compile time are stored to registers

```
int A[10];
For (k=0; k<10; k++)
 A[k] = B+k; //A[k] and B are assigned into registers
```

Occupancy Graph



GPU Memory Hierarchy