

Introduction to CUDA 2

**Programming Massively Parallel Multiprocessors and
Heterogeneous Systems (Understanding and programming the
devices powering AI)**

Jonathan Appavoo

Recall: Vocabulary you should know

- **SM**: Streaming Multiprocessor
- **kernel**: function run on GPU, launched by host
- **warp**: group of threads that can execute in lockstep (SIMD)
- **block**: group of threads running on one SM
 - can be 1D, 2D, or 3D
 - partitioned into warps
 - SM runs an integral number of blocks concurrently
- **grid**: set of blocks representing the entire data set
 - can be 1D, 2D, or 3D

Recall: General Structure of simple CUDA Program

- allocate GPU memory
- copy data from CPU memory to GPU
 - memory: `cudaMemcpy(dest, src, cudaMemcpyHostToDevice);`
- invoke kernel: `kernel<<<blocks, threads_per_block>>>(...args...);`
- (synchronize: `cudaDeviceSynchronize();`)
- copy back data from GPU memory to CPU
 - memory: `cudaMemcpy(dest, src, cudaMemcpyDeviceToHost);`
- cleanup: `cudaDeviceReset();`

Recall: Partitioning Work

Programmer's task:

- partition work into Grid of Blocks of Threads
 - e.g., by partitioning loops or data
 - grid and block can be 1D, 2D, or 3D
 - structure specified at kernel launch:
 - `kernel<<<grid,block>>(...args...);`
 - have kernel identify, for each thread, what data to process
 - available local vars:
 - `gridDim.{x,y,z}`
 - `blockDim.{x,y,z}`
 - `blockIdx.{x,y,z}`
 - `threadIdx.{x,y,z}`
- e.g.: `int tid = threadIdx.x +`
`threadIdx.y * blockDim.x +`
`threadIdx.z * blockDim.x * blockDim.y`

How best to partition?

- naturally fit application
- have a block size be a multiple of warp size
- maximize # threads per SM
 - depends on resources your threads and blocks use
 - depends on compute capability (see below)
- have # threads be multiple of # cores
- conducive to coalesced global memory accesses

Remember: Amdahl's law!

Occupancy

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#kernel-launch-and-occupancy>

"The occupancy of a CUDA kernel is the ratio of the number of active warps to the maximum number of active warps supported by the SM. In general, it's a good practice to have occupancy as high as possible which hides latency and increases performance."

Rule of thumb:

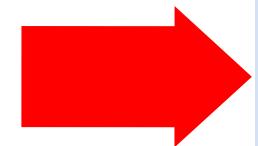
"Fill" as much work as you can into the resources of the device. Keep the cores busy and give the SM schedulers a chance to help you.



https://en.wikipedia.org/wiki/Is_the_glass_half_empty_or_half_full#/media/File:Glass-of-water.jpg

Heuristics: Number of Blocks

- The primary concern is keeping the entire GPU busy
- # blocks should be larger than the # SMs
 - each SM should have at least one block
- Since threads of a block can be waiting on each other (`__syncthreads()`)
 - good to have several small blocks than one big block to increase the likelihood of finding a warp to schedule (eg from an other block)
- But must be chosen in light of # threads in block and other constraints



Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2024	2024	2024

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. $\text{BlkSize} * j = 2048$

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. BlkSize * j = 2024
- Bad (constraints interact)
 - Too small -- Max number of Blocks per SM leaves unfilled SM Threads

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Eg. let BlkSize = 32 Thds (1 warp)
But given Max Blocks/SM = 32
at most we can fill $32 * 32 = 1024$ threads per SM

Occupancy : $1024/2048 = .5$

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. BlkSize * j = 2024
- Bad (constraints interact)
 - Too small -- Max number of Blocks per SM leaves unfilled SM Threads
 - Too big -- can't be scheduled -- "silent failure"

```
kernel<<<80, 4096>>>(...args...);  
rc = cudaGetLastError();  
if (rc != cudaSuccess) {  
    fprintf(stderr, "CUDA kernel launch failed: %s\n",  
            cudaGetStringError(rc));  
    exit(EXIT_FAILURE);  
}
```

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Eg. let BlkSize = $64 \times 64 = 4096$
BlkSize > 2048
"CUDA kernel launch failed: invalid configuration argument"

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. BlkSize * j = 2024
- Bad (constraints interact)
 - Too small -- Max number of Blocks per SM leaves unfilled SM Threads
 - Too big -- can't be scheduled -- "silent failure"
 - Not integral wrt the Max Thds/SM again leaves SM Threads unfilled

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Eg. let BlkSize = $32 \times 24 = 768$
 $2 * \text{BlkSize} = 1536 < 2048$ and
 $3 * \text{BlkSize} = 2304 > 2048$
therefore at most 2 blocks fit:

Occupancy : $1536/2048 = .75$

Registers & Occupancy

Usage interacts with Occupancy

- Registers
 - Compiler assigns a register for each local/automatic variable
 - # Reg per thread can limit the number of blocks that can fit on SM
 - Ex 1: if 64 Reg/Thd then only 1024 Thds can be assigned to an SM ($65536/64 = 1024$) = 50% occ

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#kernel-launch-and-occupancy>
<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#calculating-occupancy>

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
Registers/SM	65536	65536	65536
Shard Mem/SM	98304	167936	233472
Max Thds/SM	2048	2048	2048
Max Reg/Thd	255	255	255
Max Shared Mem/Blk	98304	167936	233472

Ex 2: Assume 512 thds/blk

If 31 reg/thread then 4 blks fit: $31 * 512 * 4 = 31 * 2048$ thds < 65536

However if pgmer add 2 variables:

$33 * 2048 = 67584 > 65536$ but $33 * 3 * 512 = 33 * 1536 = 50688 < 65536$.

So we can only have 3 blocks per SM

Occupancy = $1536 / 2048 = 0.75$

Registers & Occupancy

Usage interacts with Occupancy

- Registers
 - Compiler assigns local/automatic
 - # Reg per thread blocks that can
 - Ex 1: if 64 Regs can be assigned $\frac{64}{1024} = 50\%$

<https://docs.nvidia.com/cuda/parallel-thread-utilization/index.html>

Register and shared memory usage are reported by the compiler when compiling with the `--ptxas-options=-v` or `--resource-usage` option.

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
Registers	65536	65536	65536
Shared Memory	8304	167936	233472
Registers + Shared	2048	2048	2048
Registers / Shared	255	255	255
Registers + Shared / Shared	3304	167936	233472

Ex 2: Assume 31 Regs

If 31 reg/thread then 4 blks fit: $31 * 512 * 4 = 31 * 2048$ thds < 65536

However if pgmer add 2 variables:

$33 * 2048 = 67584 > 65536$ but $33 * 3 * 512 = 33 * 1536 = 50688 < 65536$.

So we can only have 3 blocks per SM

$$\text{Occupancy} = 1536 / 2048 = 0.75$$

Shared Mem & Occupancy

Usage interacts with Occupancy

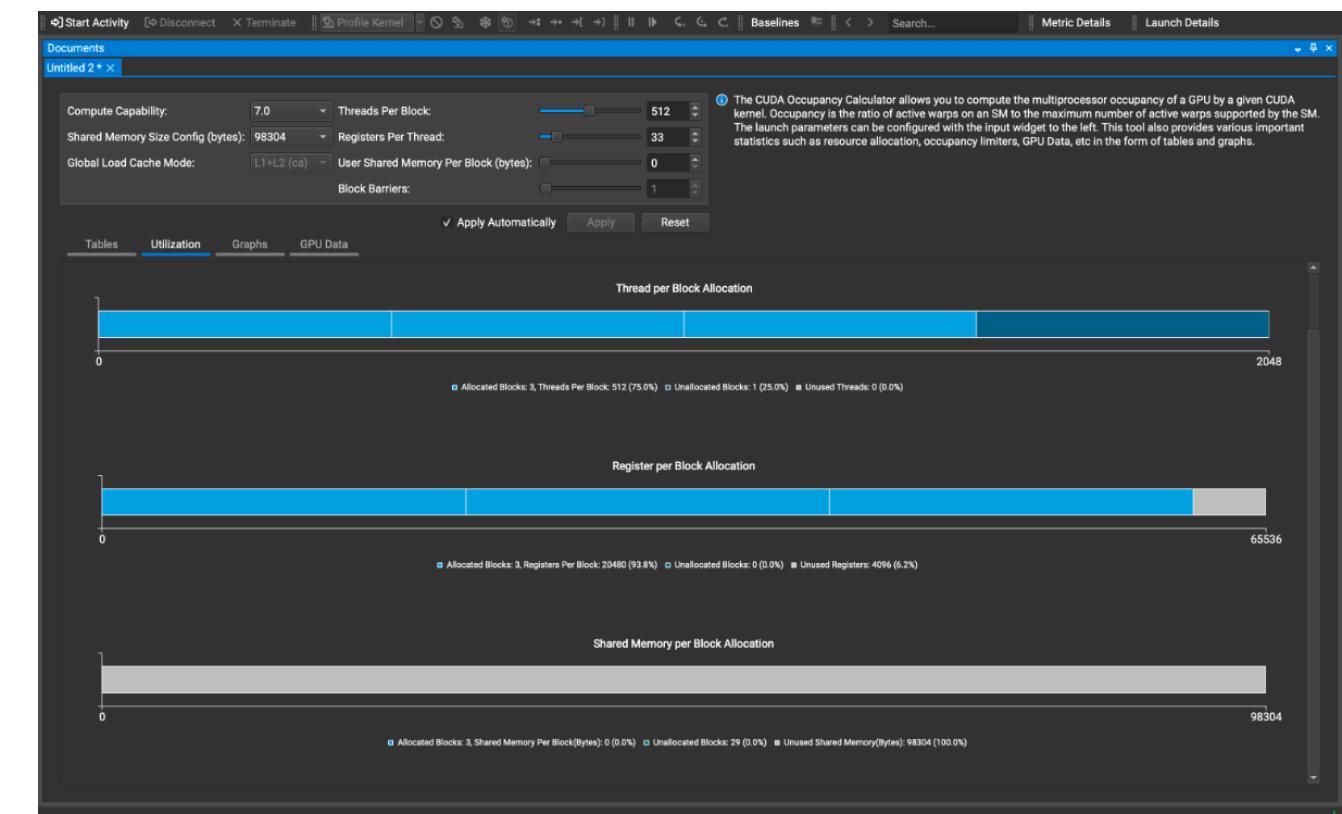
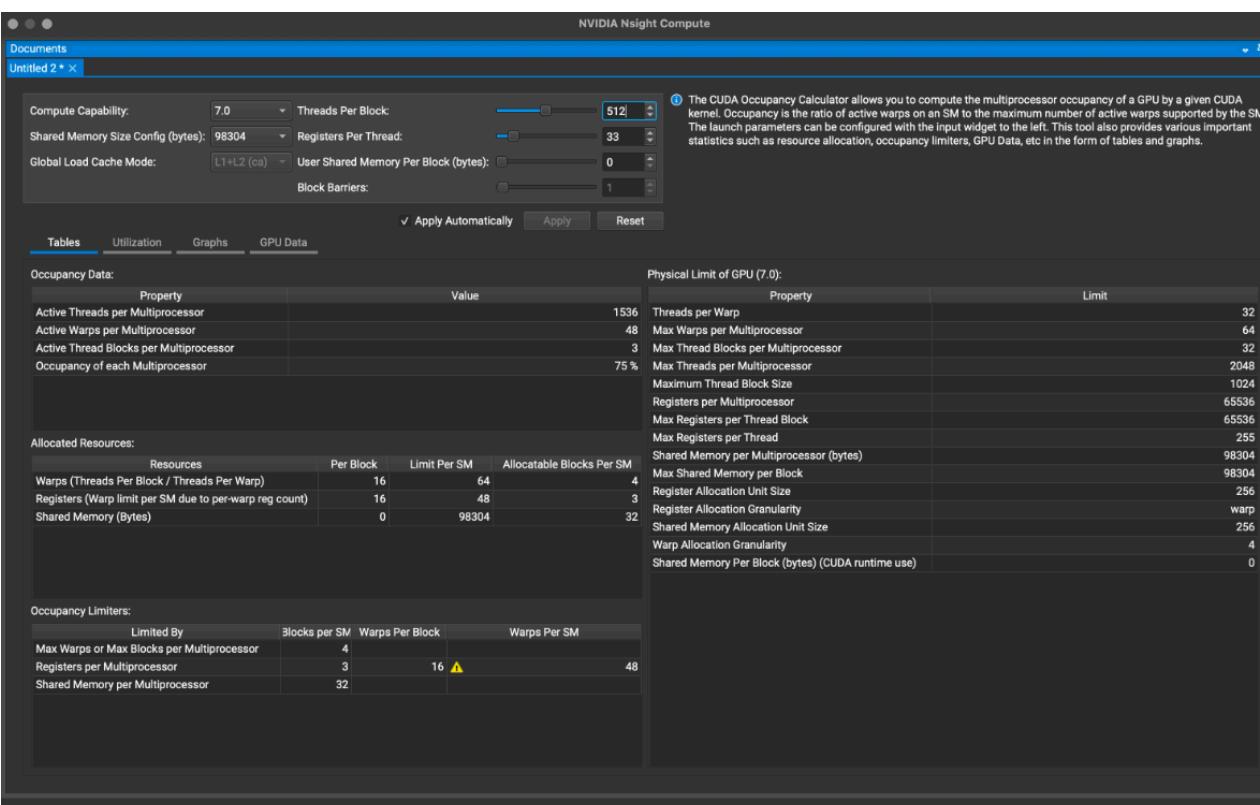
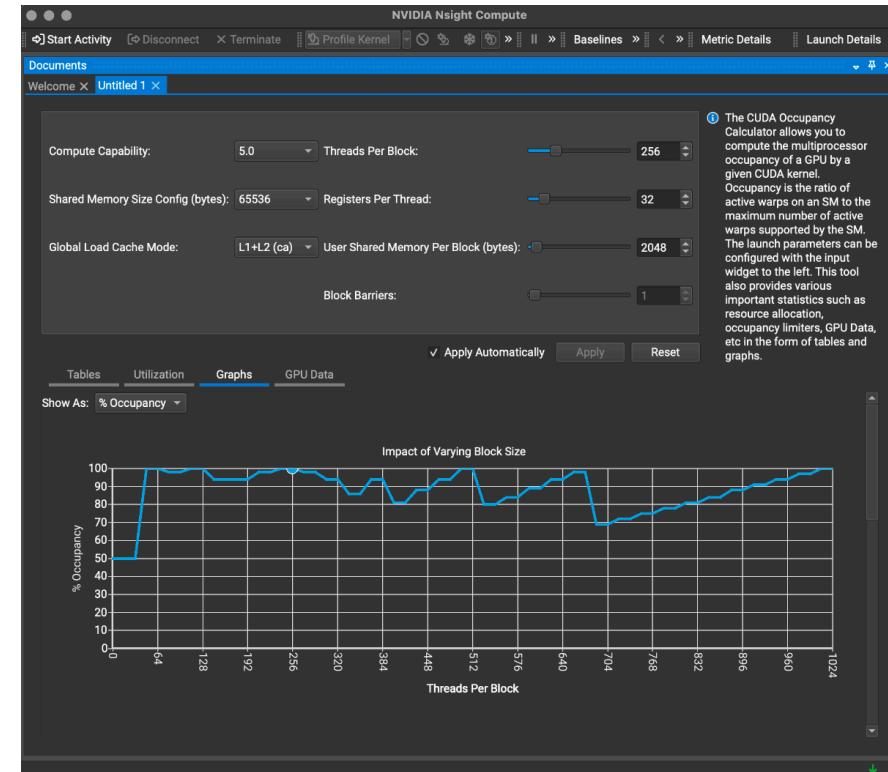
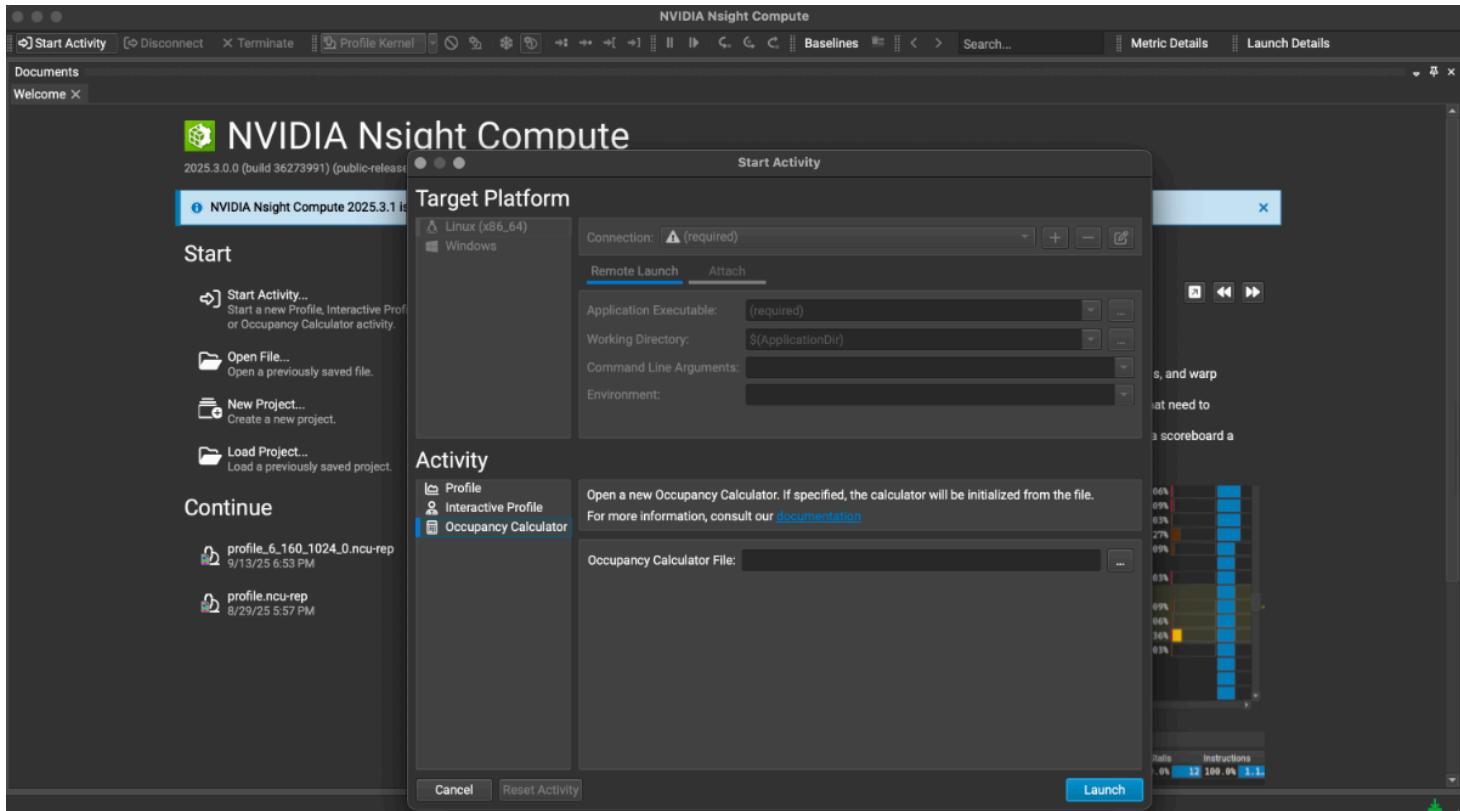
- Like Registers a thread's use of Shared Memory can constrain the number of threads/blks that can be schedule to an SM
- Note the variation in shared memory resources

Property	v100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
Registers/SM	65536	65536	65536
Shard Mem/SM	98304	167936	233472
Max Thds/SM	2048	2048	2048
Max Reg/Thd	255	255	255
Max Shared Mem/Blk	98304	167936	233472

Occupancy Calculator : Nsight Compute Tool

<https://developer.nvidia.com/nsight-compute>

<https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator>



Device Properties API

<https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html#structcudaDeviceProp>

- `cudaGetDeviceCount(&numGPUs);`
 - `cudaStatus = cudaSetDevice(0)`
- `cudaDeviceProp GPUprop ;`
`cudaGetDeviceProperties(&GPUprop, 0);`
 - `GPUprop.maxGridSize[0..2]`
 - `GPUprop.maxThreadsPerBlock`
 - and much more:

<code>char name[256];</code>	<code>int multiProcessorCount;</code>	<code>int localL1CacheSupported;</code>
<code>cudaUUID_t uuid;</code>	<code>int kernelExecTimeoutEnabled;</code>	<code>size_t sharedMemPerMultiprocessor;</code>
<code>size_t totalGlobalMem;</code>	<code>int canMapHostMemory;</code>	<code>int regsPerMultiprocessor;</code>
<code>size_t sharedMemPerBlock;</code>	<code>int computeMode;</code>	<code>int managedMemory;</code>
<code>int regsPerBlock;</code>	<code>int concurrentKernels;</code>	<code>int isMultiGpuBoard;</code>
<code>int warpSize;</code>	<code>int ECCEnabled;</code>	<code>int multiGpuBoardGroupID;</code>
<code>size_t memPitch;</code>	<code>int pciBusID;</code>	<code>int singleToDoublePrecisionPerfRatio;</code>
<code>int maxThreadsPerBlock;</code>	<code>int pciDeviceID;</code>	<code>int pageableMemoryAccess;</code>
<code>int maxThreadsDim[3];</code>	<code>int asyncEngineCount;</code>	<code>int concurrentManagedAccess;</code>
<code>int maxGridSize[3];</code>	<code>int unifiedAddressing;</code>	<code>int computePreemptionSupported;</code>
<code>int clockRate;</code>	<code>int memoryClockRate;</code>	<code>int </code>
<code>size_t totalConstMem;</code>	<code>int memoryBusWidth;</code>	<code>pageableMemoryAccessUsesHostPageTables;</code>
<code>int major;</code>	<code>int l2CacheSize;</code>	<code>int directManagedMemAccessFromHost;</code>
<code>int minor;</code>	<code>int maxThreadsPerMultiProcessor;</code>	<code>int maxTexture1D;</code>
<code>size_t textureAlignment;</code>	<code>int streamPrioritiesSupported;</code>	<code>int maxTexture1DMipmap;</code>
<code>size_t texturePitchAlignment;</code>	<code>int globalL1CacheSupported;</code>	<code>int maxTexture1DLinear;</code>

FYI: `cudaDeviceGetAttribute` has better performance:

<https://developer.nvidia.com/blog/cuda-pro-tip-the-fast-way-to-query-device-properties/>

Occupancy Calculator API

8.2.3.1. Occupancy Calculator

Several API functions exist to assist programmers in choosing thread block size and cluster size based on register and shared memory requirements.

- › The occupancy calculator API,

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`, can provide an occupancy prediction based on the block size and shared memory usage of a kernel. This function reports occupancy in terms of the number of concurrent thread blocks per multiprocessor.

- › Note that this value can be converted to other metrics. Multiplying by the number of warps per block yields the number of concurrent warps per multiprocessor; further dividing concurrent warps by max warps per multiprocessor gives the occupancy as a percentage.

- › The occupancy-based launch configurator APIs,

`cudaOccupancyMaxPotentialBlockSize` and `cudaOccupancyMaxPotentialBlockSizeVariableSMem`, heuristically calculate an execution configuration that achieves the maximum multiprocessor-level occupancy.

- › The occupancy calculator API, `cudaOccupancyMaxActiveClusters`, can provided occupancy prediction based on the cluster size, block size and shared memory usage of a kernel. This function reports occupancy in terms of

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#occupancy-calculator>

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__OCCUPANCY.html

Occupancy Summary & Subtleties

EXPERIMENT!!!!

"... higher occupancy does not always equate to better performance. For example, improving occupancy from 66 percent to 100 percent generally does not translate to a similar increase in performance. A lower occupancy kernel will have more registers available per thread than a higher occupancy kernel, which may result in less register spilling to local memory; in particular, with a high degree of exposed instruction-level parallelism (ILP) it is, in some cases, possible to fully cover latency with a low occupancy.

There are many such factors involved in selecting block size, and inevitably some experimentation is required. However, a few rules of thumb should be followed:

- ▶ Threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.
- ▶ A minimum of 64 threads per block should be used, and only if there are multiple concurrent blocks per multiprocessor.
- ▶ Between 128 and 256 threads per block is a good initial range for experimentation with different block sizes.
- ▶ Use several smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call `__syncthreads()`.

Basic Timing



https://commons.wikimedia.org/wiki/File:Louis_Moinet%27s_%22Compteur_de_Tierces%22.jpg#/media/File:Louis_Moinet's_%22Compteur_de_Tierces%22.jpg

Measuring time

CPU

- Note time measurement imprecise with lots variability
- measure n times: take best time
- See NVIDIA "CUDA C Best Practice Guide" Using CPU Timers for how time CUDA kernel calls with CPU Timers
<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#using-cpu-timers>
- See Timing Operations in CUDA Streams in the programming guide
<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#timing-operations-in-cuda-streams>

```
#include <stdio.h>
#include <assert.h>
#include <inttypes.h>
#include <time.h>

#define CLOCK_SOURCE CLOCK_MONOTONIC
#define NSEC_IN_SECOND (1000000000)

typedef struct timespec ts_t;

static inline int ts_now(ts_t *now) {
    if (clock_gettime(CLOCK_SOURCE, now) == -1) {
        perror("clock_gettime");
        assert(0);
        return 0;
    }
    return 1;
}

static inline uint64_t ts_diff(ts_t start, ts_t end)
{
    uint64_t diff =
        ((end.tv_sec - start.tv_sec) * NSEC_IN_SECOND) +
        (end.tv_nsec - start.tv_nsec);
    return diff;
}
```

This blog entry has an expanded explanation of using events for timing

https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/#timing_using_cuda_events

This seems to be the recommended approach

Measuring time GPU

1. GPU `clock()` & `clock64()`

- returns value of clock cycle counter
- 1 value per thread!
- need to convert using
`cudaDeviceProp::clockRate`

2. CUDA Timers (Event Records)

3. Using NVIDIA Profiler

- nsight compute profiler (ncu)
- nsight system profile (nsys)

```
16  __global__ void helloFromGPU()
17  {
18      printf("Hello from GPU\n");
19  }
20
21 int main(int argc, char **argv)
22 {
23     cudaEvent_t start;
24     cudaEvent_t end;
25
26     // create the events
27     CUDA_CHECK(cudaEventCreate(&start));
28     CUDA_CHECK(cudaEventCreate(&end));
29
30     // record the start event
31     CUDA_CHECK(cudaEventRecord(start, 0));
32
33     helloFromGPU <<< 1, 100 >>>();
34
35     // record the end event
36     CUDA_CHECK(cudaEventRecord(end, 0));
37
38     CUDA_CHECK(cudaGetLastError());
39     CUDA_CHECK(cudaDeviceSynchronize());
40
41     // get the timing
42     float elapsedTime;
43     CUDA_CHECK(cudaEventElapsedTime(&elapsedTime, start, end));
44     printf("Kernel execution time: %0.6f ms\n", elapsedTime);
45
46     // clean up
47     CUDA_CHECK(cudaEventDestroy(start));
48     CUDA_CHECK(cudaEventDestroy(end));
49     CUDA_CHECK(cudaDeviceReset());
50 }
```

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#timing-operations-in-cuda-streams>

The programming Guide discusses a general approach to timing with multiple streams. The above uses the default stream.

**Lets look at another code
example**

Very basic Matrix Add

MatAdd I

```
void matrixSumHost(float *A, float *B, float *C,
                    int nx, int ny) {
    float *ia=A, *ib=B, *ic=C;
    for (int iy=0; iy<ny; iy++) {
        for (int ix=0; ix<nx; ix++) {
            ic[ix] = ia[ix] + ib[ix];
        }
        ia+=nx; ib+=nx; ic +=nx;
    }
}

__global__ void matrixSumGPU(float *A, float *B, float *C,
                            int nx, int ny) {
    int ix = threadIdx.x + blockIdx.x*blockDim.x;
    int iy = threadIdx.y + blockIdx.y*blockDim.y;
    int idx = iy*nx + ix;
    if ((ix<nx) && (iy<ny)) {
        C[idx] = A[idx] + B[idx];
    }
}
```

MatAdd II

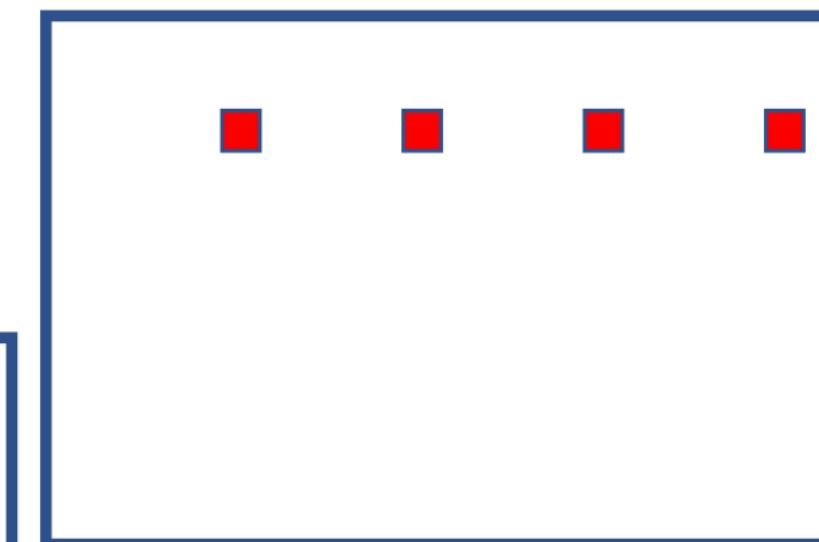
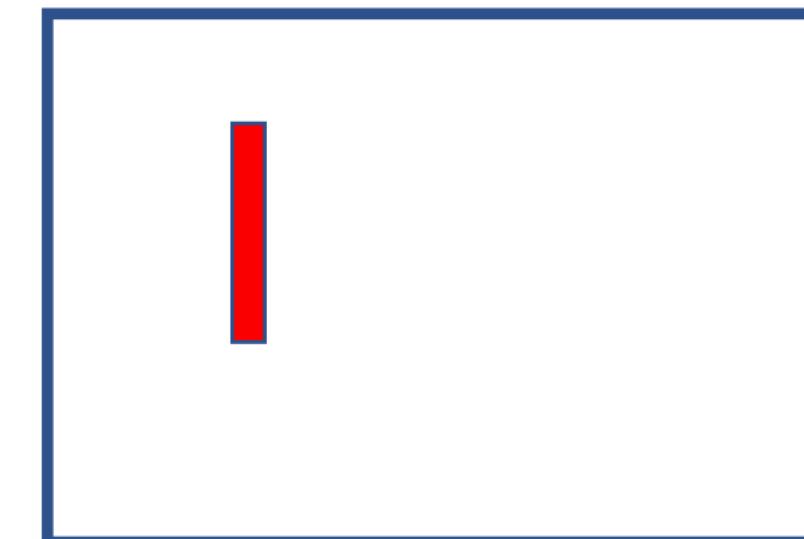
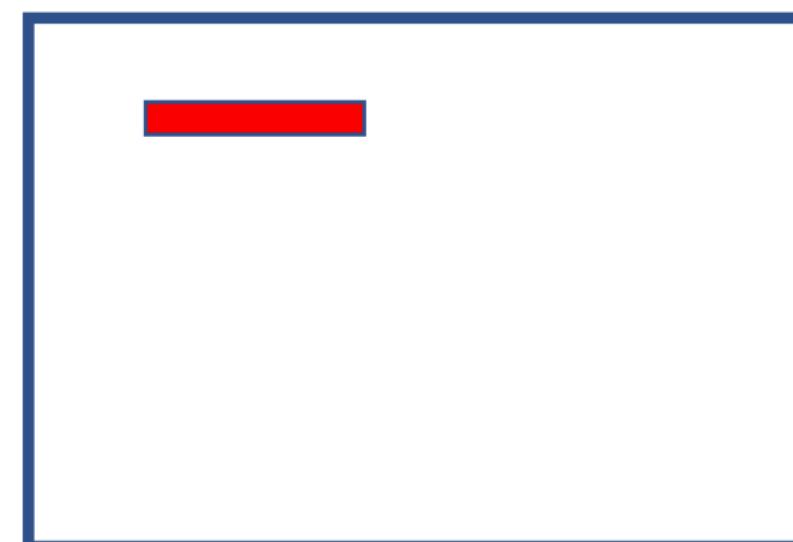
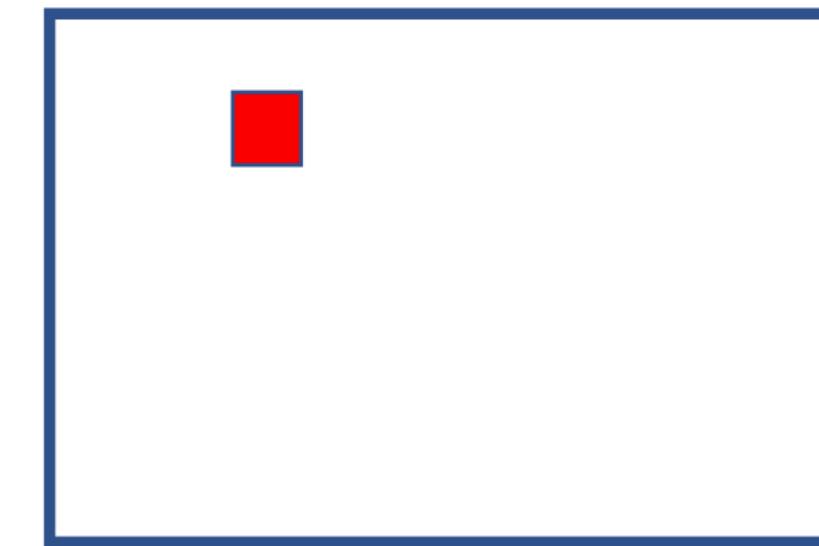
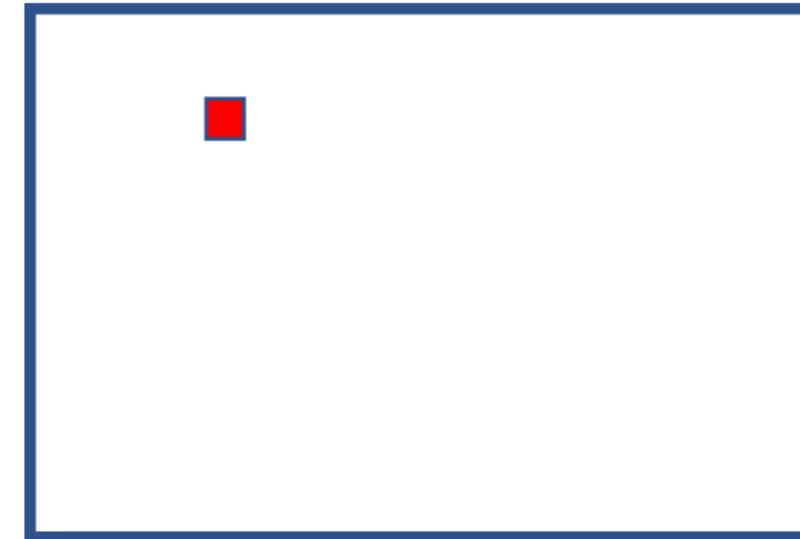
```
int main(int argc, char **argv) {
    //set matrix size
    int nx = atoi(argv[1]);
    int ny = atoi(argv[2]);
    int noElems = nx*ny;
    int bytes = noElems * sizeof(float);
    // alloc memory host-side
    float *h_A = (float *) malloc(bytes);
    float *h_B = (float *) malloc(bytes);
    float *h_hC = (float *) malloc(bytes); // host result
    float *h_dC = (float *) malloc(bytes); // gpu result
    // init matrices with random data
    initData(h_A, noElems); initData(h_B, noElems);
    // alloc memory dev-side
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, bytes);
    cudaMalloc((void **) &d_B, bytes);
    cudaMalloc((void **) &d_C, bytes);
```

MatAdd III

```
//transfer data to dev
cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);
// invoke Kernel
dim3 block(32, 32); // configure
dim3 grid((nx + block.x-1)/block.x,
           (ny + block.y-1)/block.y);
matrixSumGPU<<<grid, block>>>(d_A, d_B, d_C, nx, ny);
cudaDeviceSynchronize();
//copy data back
cudaMemcpy(h_dC, d_C, bytes, cudaMemcpyDeviceToHost);
// free GPU resources
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
cudaDeviceReset();
// check result
matrixSumHost(h_A, h_B, h_hC, nx, ny);
cmpMats(h_hC, h_dC, noElems);
return 0;
}
```

Things to try I: More operations per byte

Now consider threads computing multiple elements



Things to try II: Improve host-device copies 1

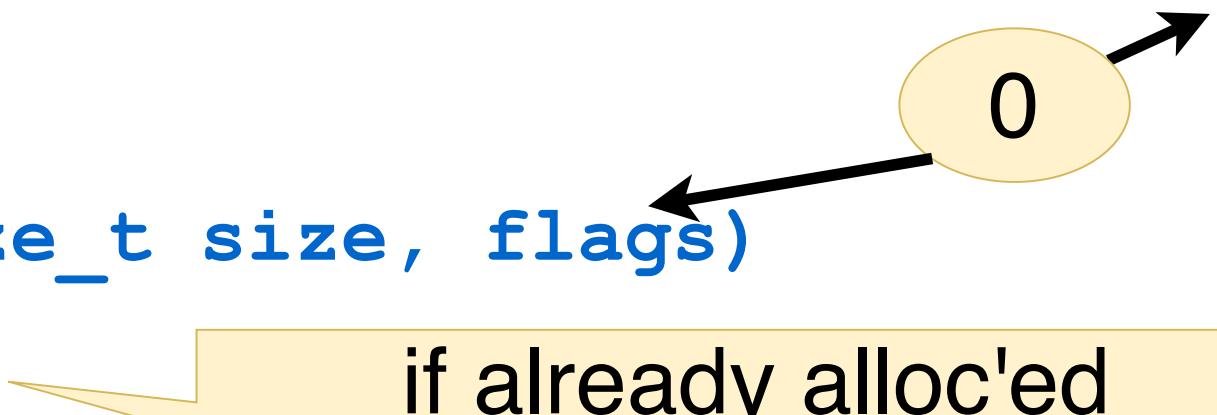
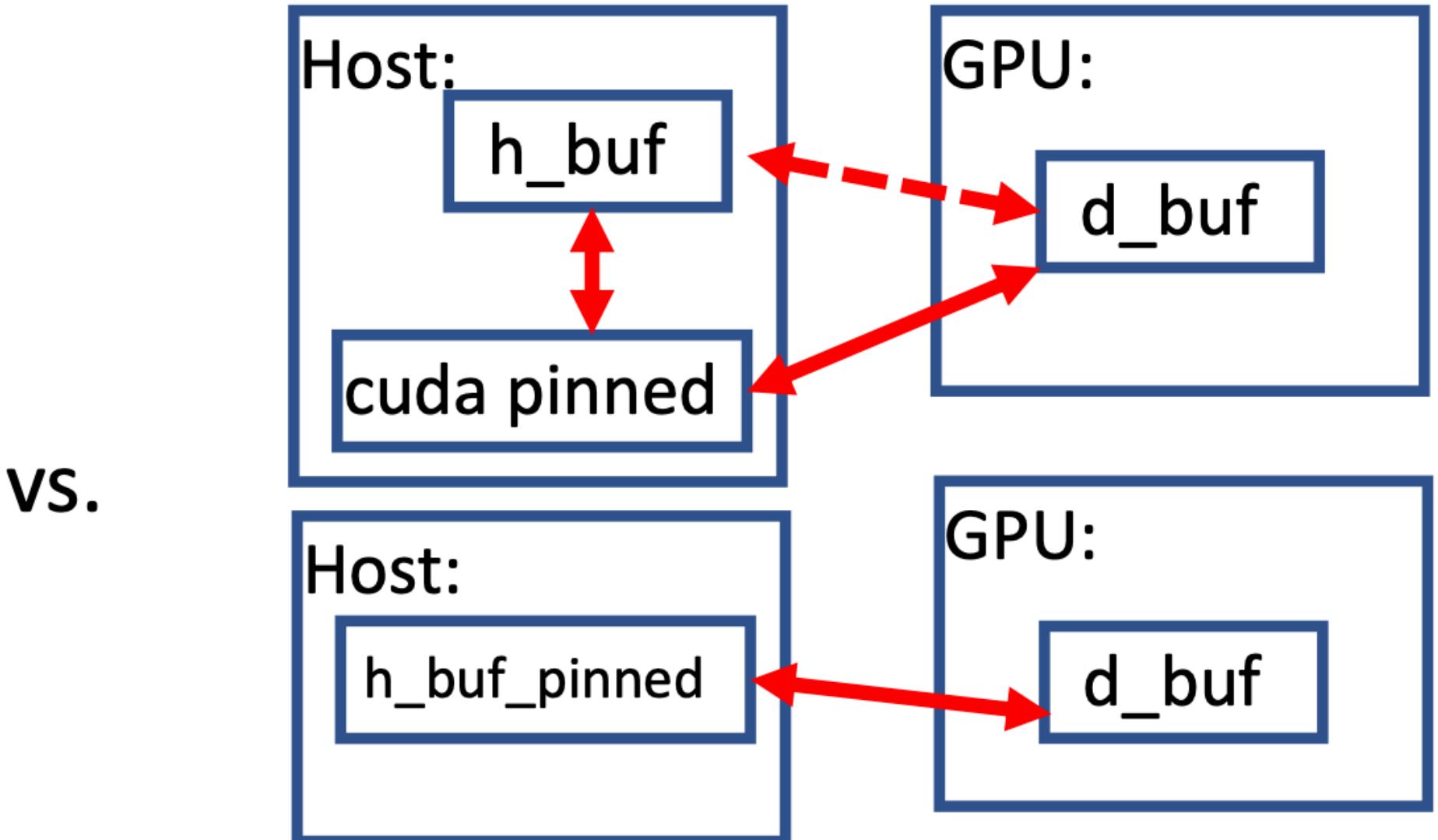
- To DMA to/from host, host memory has to be pinned
- If not pinned, extra copy
- Supporting functions:

```
cudaHostAlloc(void** pHHost, size_t size, unsigned int flags)
```

```
cudaFreeHost(void *ptr)
```

```
cudaHostRegister(void *ptr, size_t size, flags)
```

```
cudaHostUnregister(void *ptr)
```



Things to try III: Use function specifiers

	Executed On	Callable from
<code>__global__ void kernel_fct()</code>	device	host & <code>device*</code>
<code>__device__ float dev_fct()</code>	device	device
<code>__host__ float host_fct()</code>	host	host

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=cudaHostAlloc#function-execution-space-specifiers>

- **`__global__`**
 - defines a kernel function
 - must return void
 - can only call `__device__` functions
- `__device__` and `__host__` can be used together
 - two different versions of code generated

CUDA & GPU Memories

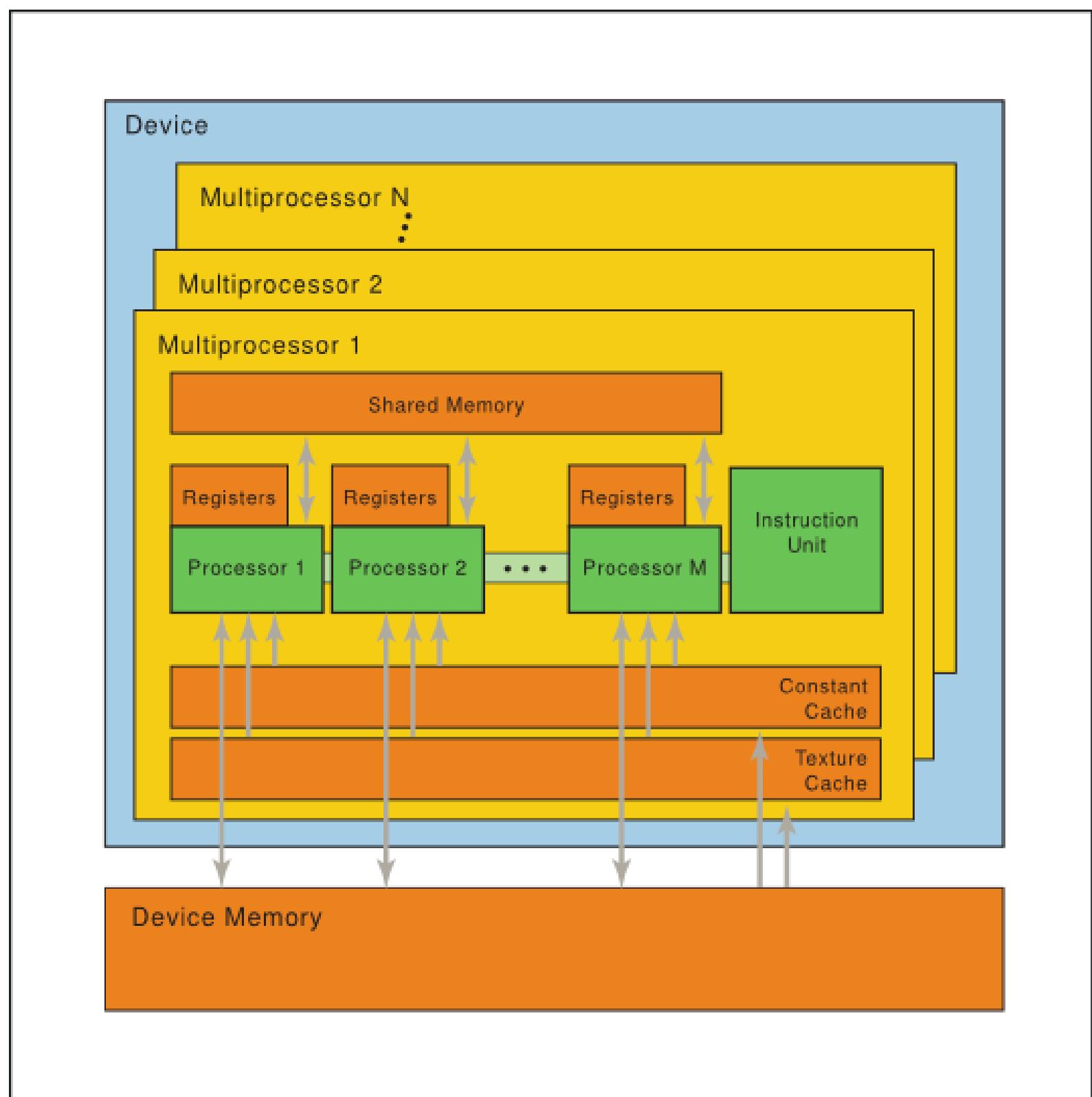
<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#gpu-device-memory-spaces>

Table 1: Memory Types, Scopes and Lifetimes

Memory Type	Scope	Lifetime	Location
Global	Grid	Application	Device
Constant	Grid	Application	Device
Shared	Block	Kernel	SM
Local	Thread	Kernel	Device
Register	Thread	Kernel	SM

Registers, Shared, & Device (Global and Local)

<https://docs.nvidia.com/cuda/parallel-thread-execution/#set-of-simt-multiprocessors-hardware-model>

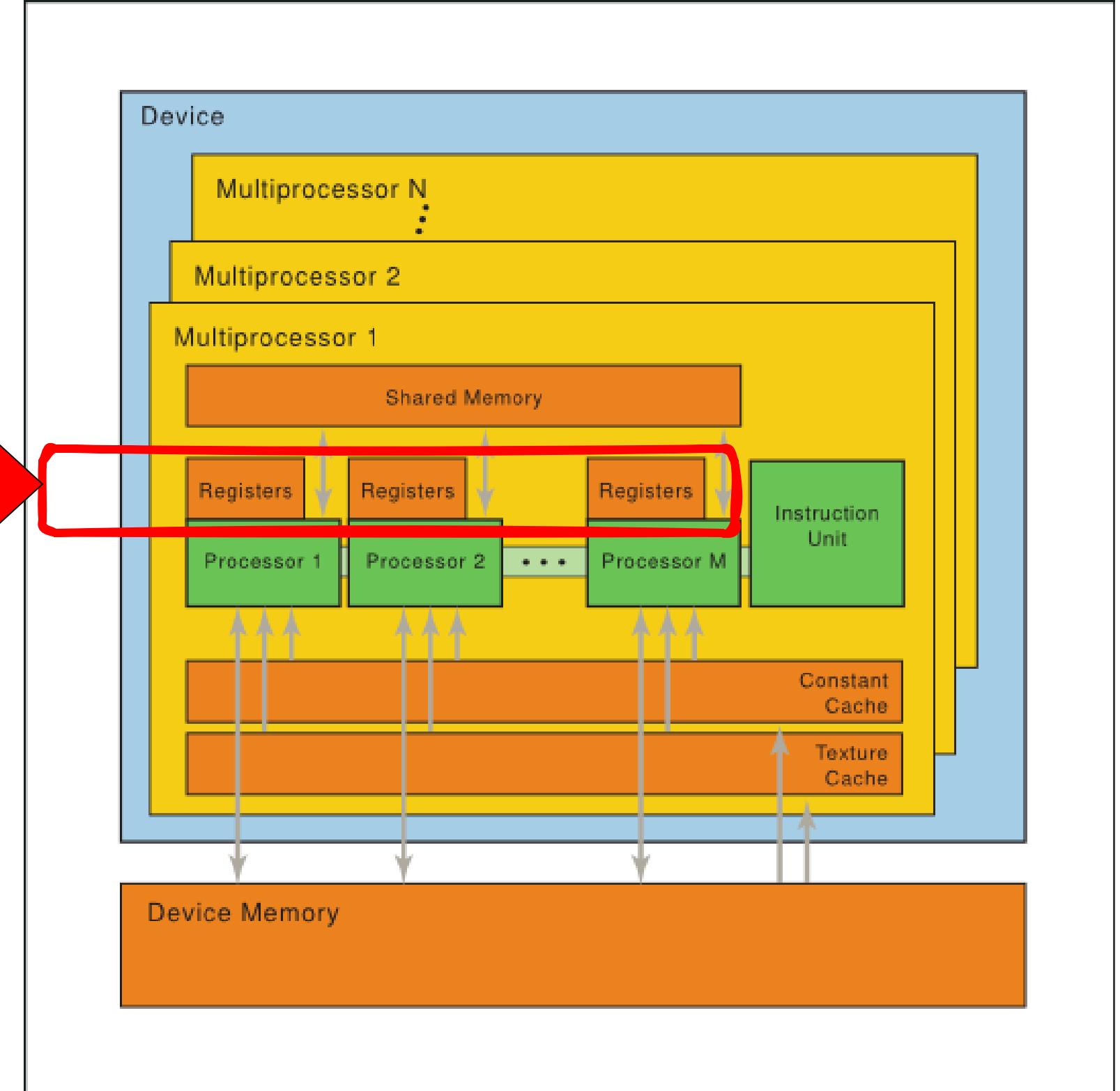
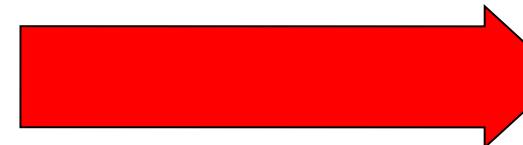


Registers

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#registers>

Fast and holds operands for instructions

1. On-chip fast and typed
2. Per-Thread only.



Registers I



Per thread registers and
local memory

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-hierarchy-memory-hierarchy-figure>

- Each SM: Set of 32-bit registers partitioned among the warps
- Provides fastest access -- directly used by instructions as operands
 - latency if: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#multiprocessor-level>
 - memory dependency: value is being loaded into register from memory
 - register dependency: value is being produced by a prior instruction
- Kernel Local/Automatic variables are placed in registers by the compiler (Some exceptions: See Local Memory later)
- as discussed # registers used by a kernel can impact occupancy
 - compiler tries to minimize register use
 - compiler can "spill" registers to further reduce register pressure
 - you can control usage: 1) **-maxrregcount** (nvcc flag), 2) **__launch_bounds__()** (kernel pragma), and 3) **__maxreg__()** qualifier.
 - spills to Local Memory with new support (CUDA 13) to Shared Memory

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#maximum-number-of-registers-per-thread>

<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#pragma-strings-enable-smem-spilling>

<https://developer.nvidia.com/blog/how-to-improve-cuda-kernel-performance-with-shared-memory-register-spilling/>

An automatic variable declared in device code without any of the `device`, `shared` and `constant` memory space specifiers described in this section generally resides in a register. However in some cases the compiler might choose to place it in local memory, which can have adverse performance consequences as detailed in Device Memory Accesses.

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/intro-to-cuda-cpp.html#variable-specifiers>

Registers II



Per thread registers and local memory

- Registers are 32 bits -- each variable consumes at least one 32-bit register
 - doubles consume 2 <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#multiprocessor-level>
- You can help the compiler to use register optimizations (eg, avoid accessing values through pointers)
 - use **__restrict__** on pointers to let the compiler know there is no aliasing
 - `void foo(const float * __restrict a, const float * __restrict b, float * __restrict c)`
 - be careful -- everything is a balancing act -- will increase register pressure and therefore affect occupancy
- New async copies can be used to avoid register use when moving data between global and shared memory <https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html#asynchronous-data-copies>
- Core register facts:
 - Number of 32 bit registers per SM 64K
 - Maximum number of registers per Thread Block 64K
 - Maximum number of registers per Thread 255

Registers II



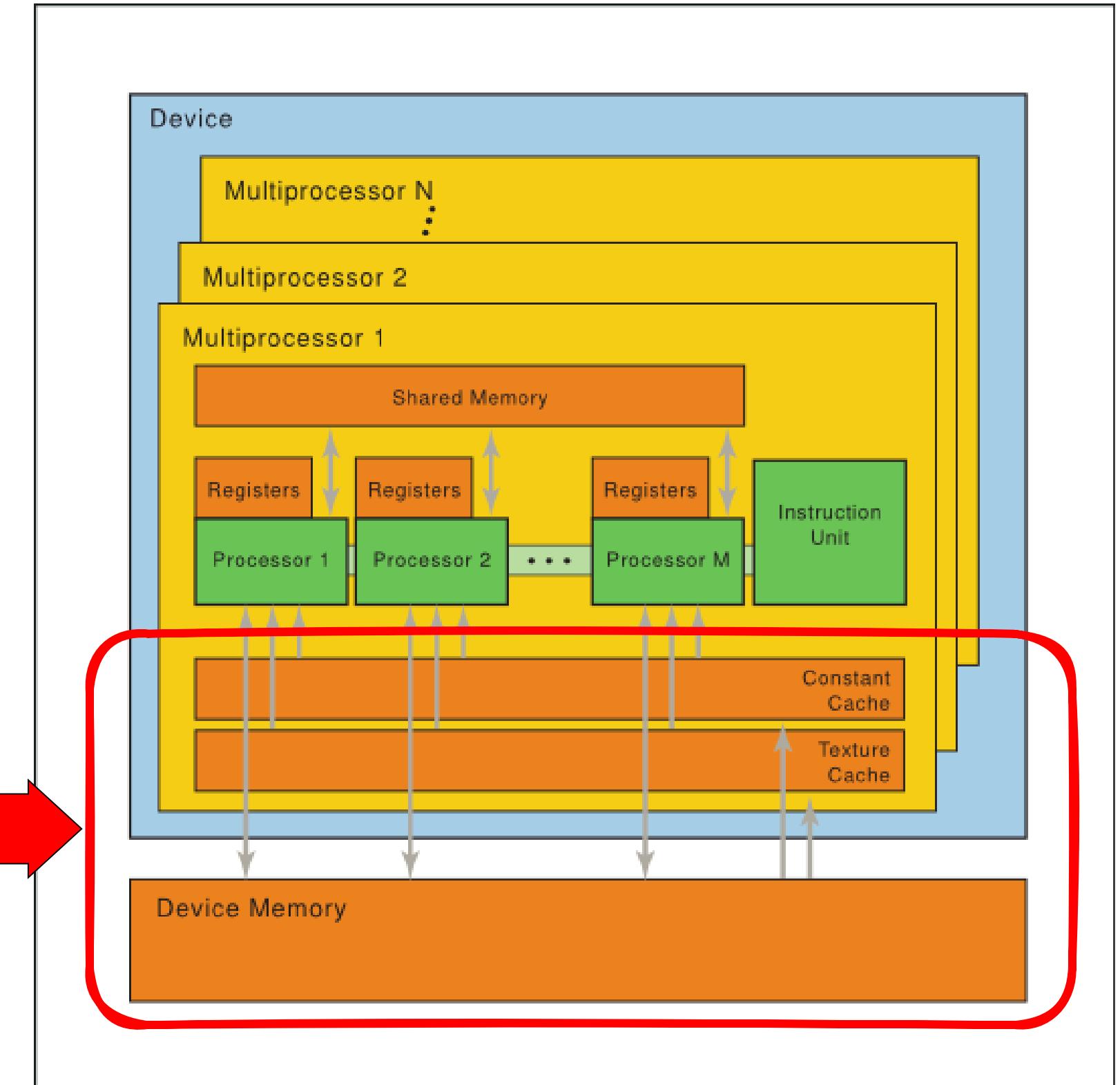
Per thread registers and local memory

- Registers are 32 bits -- each variable consumes at least one 32-bit register
 - doubles consume 2 <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#multiprocessor-level>
- You can have multiple threads per thread block. This is called **fat threads**.
 - values that are shared between threads are stored in **local memory**.
 - You may want to explore register usage tradeoffs:
 - Fat vs Thin threads with respect to register use.
 - AGAIN, it requires **experimentation** to determine what is best for performance.
- New asynchronous API for CUDA. It includes support for global and local memory. nvcc **--ptxas-options=-v** or **--resource-usage** to see register usage
- Core register limits
 - Number of registers per Thread Block 128K
 - Maximum number of registers per Thread Block 64K
 - Maximum number of registers per Thread 255

Device Memory

Big & slow

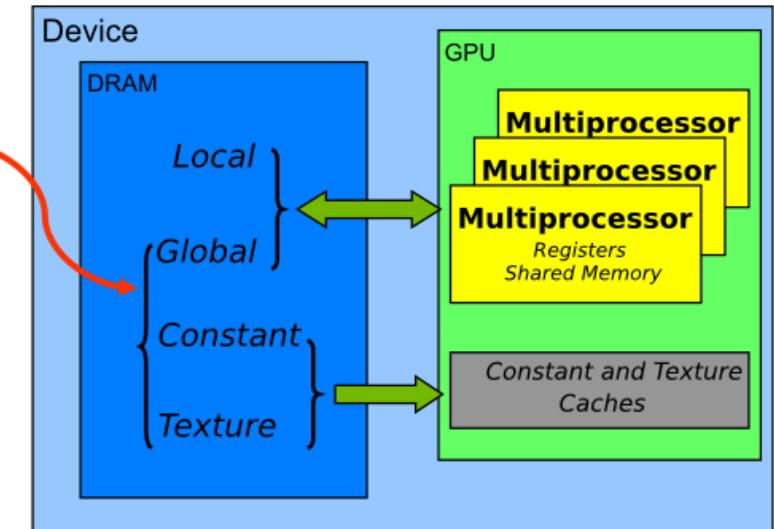
1. 100+ cycles to access
2. some data can be cache
3. exploiting coalesced access can be critical (Global Memory)



Device Memory:Local Memory

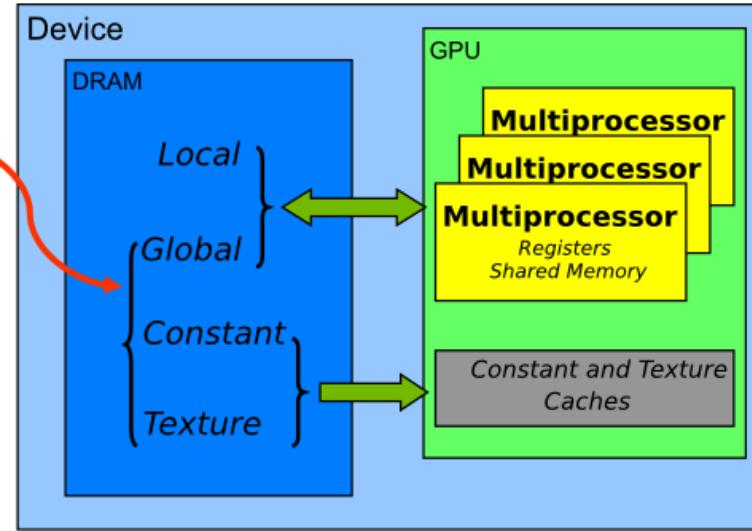
- Per thread
- In device memory: SLOW
 - organized so typical accesses are coalesced
 - cached in L2
- Compiler:
 - Arrays, it can't determine to be indexed with constant quantities
 - Large structures or arrays that would take up too many registers
 - Spills: Any variable if the kernel uses more registers than available (CUDA 13 allows spilling to local memory)
- compiler reports total local memory usage (lmem) per kernel:
 - `--ptxas-options=-v`

Device Memory: Global Memory I



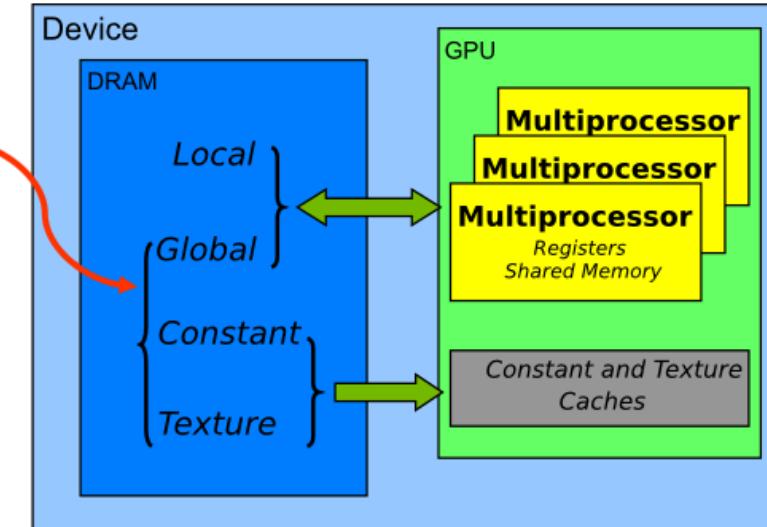
- Big & Global: accessible from all SM's and by the Host
- 100++ cycle latency
- dynamic allocation (host side):
 - `cudaMemalloc(....)`
- dynamic allocation (device side)
 - `malloc(...), new(...)`
- static allocation
 - `__device__ <type> <varname>`
 - host can read/write using special routines:
 - `cudaMemcpyToSymbol("var", *h_var, sizeof(var_type))`
 - `cudaMemcpyFromSymbol(...)`
 - or with standard `cudaMemcpy()` in combination with `cudaGetSymbolAddress(...)`
 - not used that often, but sometimes useful

Device Memory: Global Memory II



- 32-, 64- **or** 128-byte memory transactions
- Access **must** be naturally aligned 32, 64, 128 aligned respectively
- When a warp executes an instruction that accesses global memory
 - It coalesces the accesses of the individual threads of the warp
 - into one or more memory transactions, depending on
 - size of word accessed by each thread
 - distribution of the memory addresses across threads
 - **the larger the number of transactions required**
 - more unused words -- **poorer instruction throughput**
 - Number of transactions and impact is device (compute capability dependent, see compute capability doc in PG for your device)

Device Memory: Global Memory III

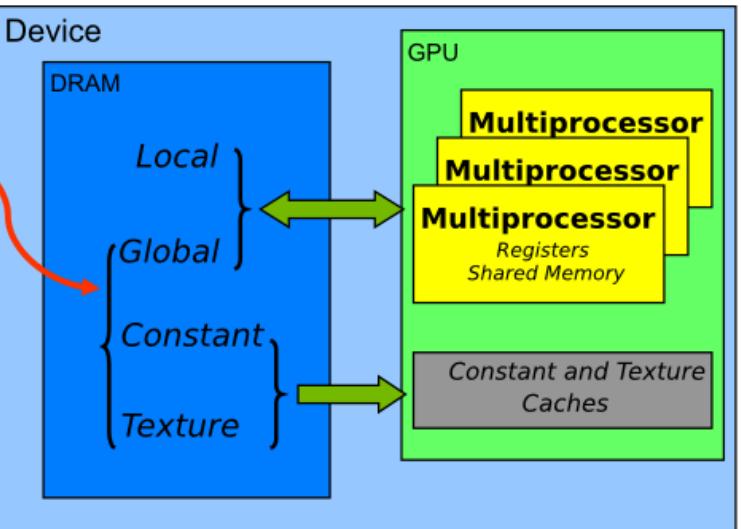
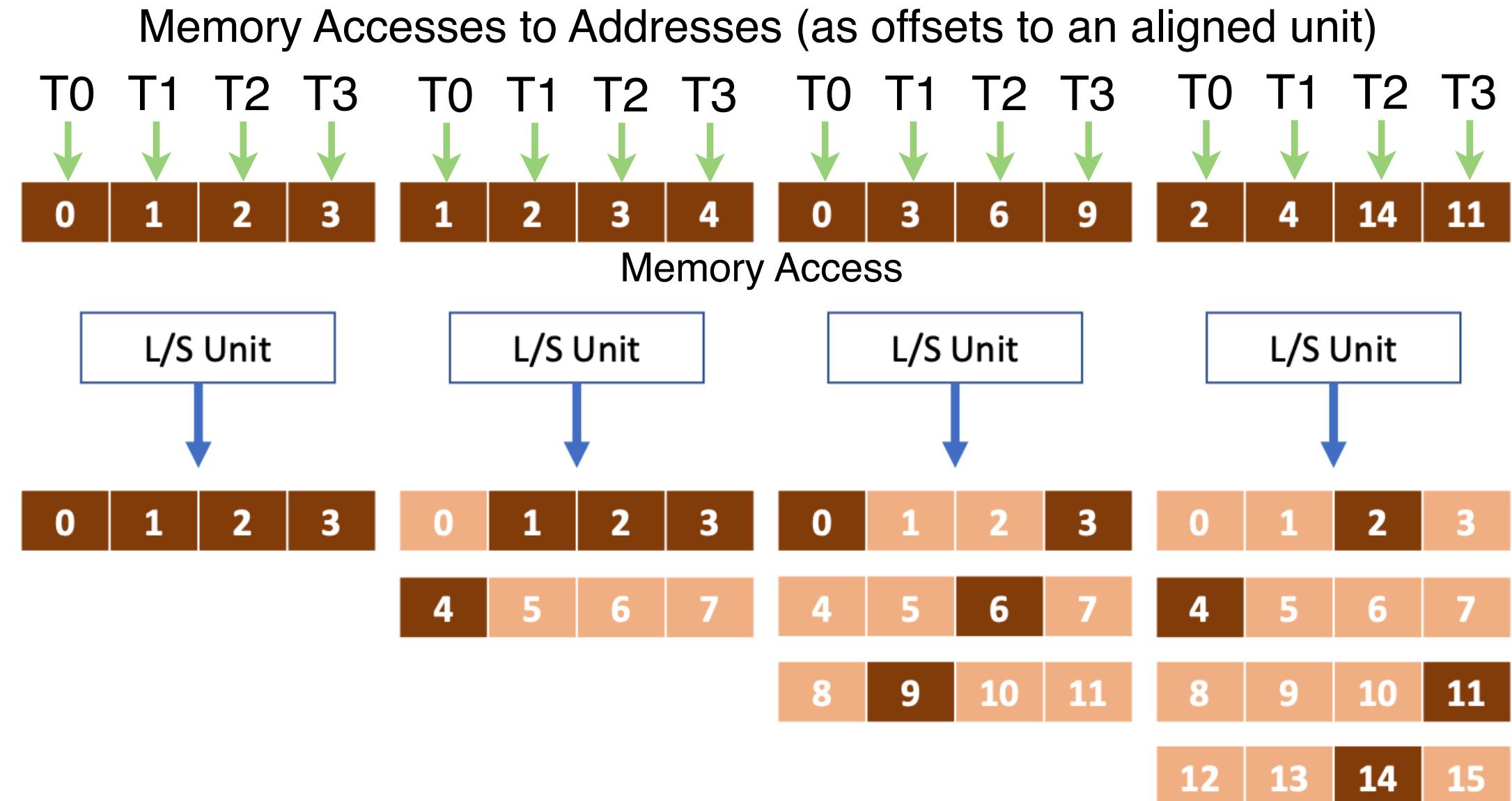


- CRITICAL OPTIMIZATION: Maximize coalescing
 1. Follow optimal access patterns based on Compute Capability (>6) is the same and we will cover now)
 2. Use a data type that meets the size and alignment requirements
 3. Padding data in some case can improve 2D array accesses

Device Memory: Global Memory IV

- Coalesced Access to Global Memory

A synthetic 4 threads per warp example



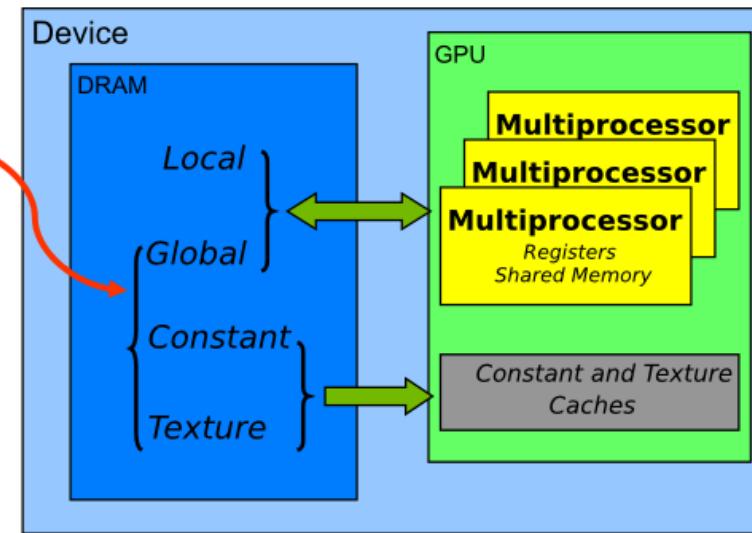
→ Your job as a programmer is to ensure most mem accesses from a warp are coalesced

Device Memory: Global Memory V

- Size and Alignment Requirements
 - coalescing of global memory access will only occur if access is with a single memory instruction (eg, ld, store of supported size) and aligned
 - supported sizes: 1, 2, 4, 8, 16 bytes (with natural alignment)
 - if not multiple instructions and less effective coalescing
- `__global__` variables and allocations are always at least 256 byte aligned
- use alignment specifiers to enforce

```
struct __align__(8) {  
    float x;  
    float y;  
};
```

```
struct __align__(16) {  
    float x;  
    float y;  
    float z;  
};
```



"Reading non-naturally aligned 8-byte or 16-byte words produces incorrect results (off by a few words)"

Device Memory: Global Memory VI

- Padding to ensure alignment is useful to improve array access
 - Eg, in a 2D array, you want both Thread Block and array width to be a multiple of warp size
 - Padding rows with extra bytes to ensure this is true will improve coalescing

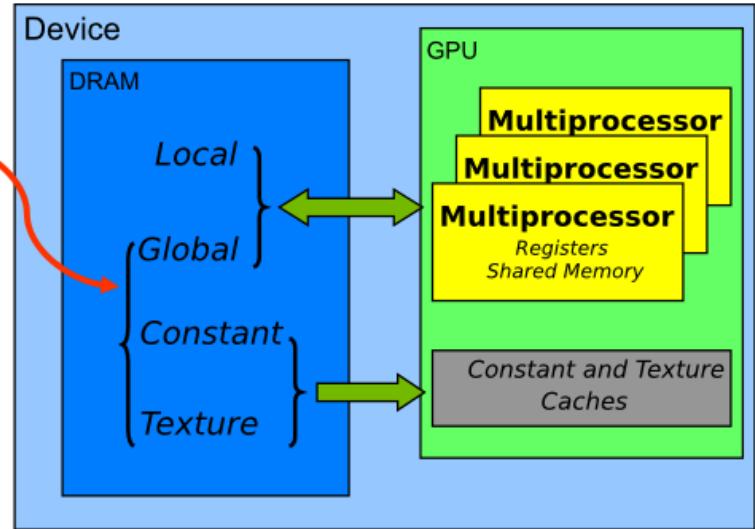
```
float M[rows][columns]; float v = M[ty][tx];
```

Base of Array
aligned for
coalesced
access

BaseAddress + width * ty + tx

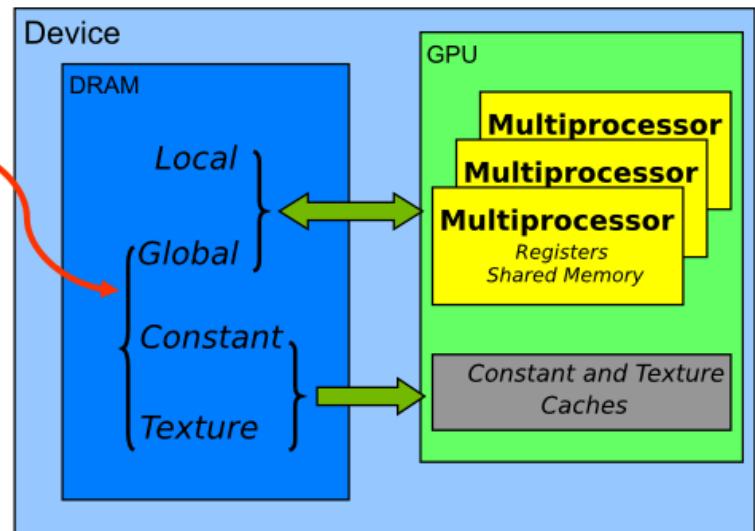
Width of array (number of columns where each element is of a type that meets the coalescing size requirements)

pad width to ensure that each row starts at a warp aligned size



<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#constant-memory>

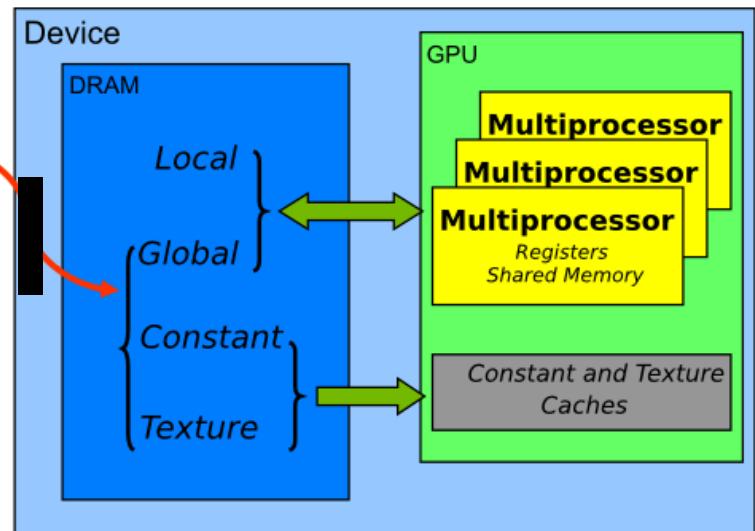
Device Memory: Constant Memory



- Device memory Device READ ONLY : slow but cached
- Size limited 64K
- Cached
 - best when all threads access the same data item
 - then fast -- like Shared Memory
 - but does not tie up a register -- can be useful to reduce register pressure
- Only static allocation
 - `__constant__ <type> <varname>`
- Read/Write by CPU:
 - `cudaMemcpyToSymbol("var", *h_var, sizeof(var_type))`
 - `cudaMemcpyFromSymbol(...)`
 - or with standard `cudaMemcpy()` in combination with `cudaGetSymbolAddress(...)`

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#constant-memory>

Device Memory: Constant Memory II



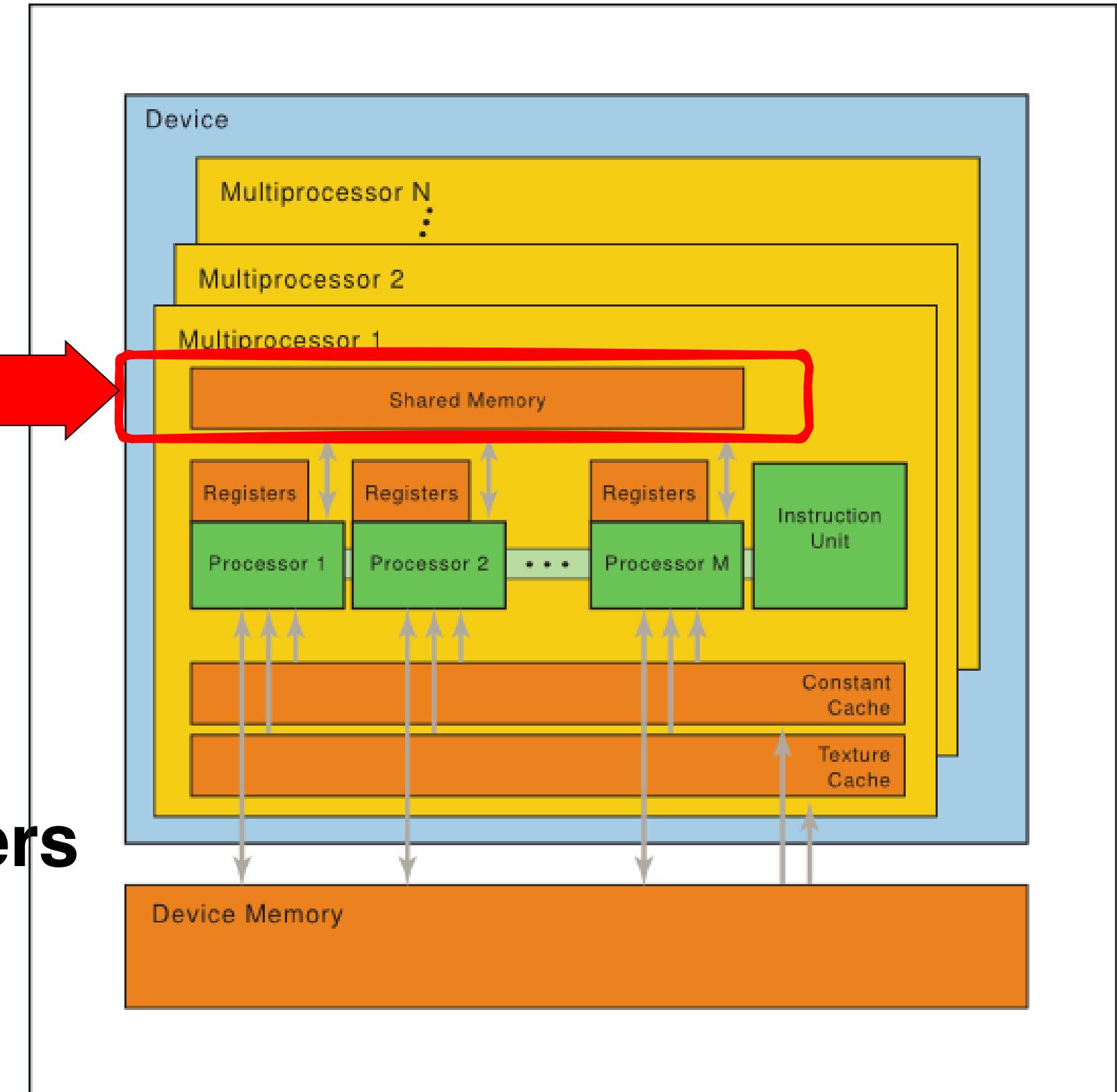
- Constant memory var values set at run-time
 - Different than standard constants
 - values known at compile time (get baked into instruction stream)
 - `#define PI 3.1415926f`
 - `a = b / (2.0f * PI);`
 - Leave these as they are, they generally won't add to register pressure

Shared Memory

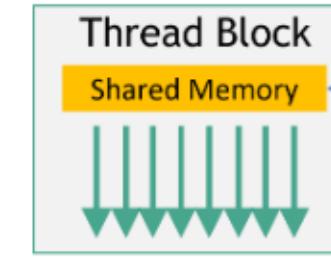
<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#shared-memory>

**Small but critical to bridging registers
and device memory**

1. On-chip" Fast & Shared
2. But Banked!



Shared Memory



Per block Shared memory

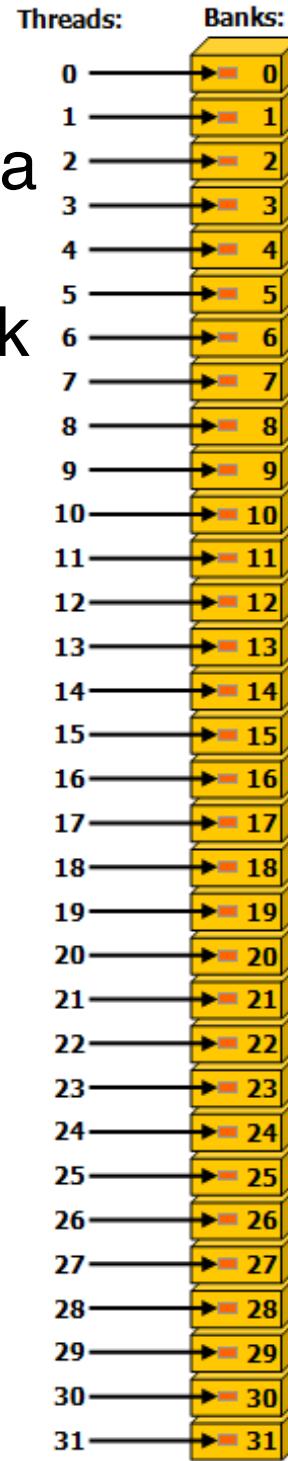
- Fast on-chip local to SM
- used for
 - operations requiring communications between threads within a block (recently CC adds support for distributed access)
 - data reuse
 - staging memory to increase degree of coalescing factor
- Limited in size : Varies with device (V100 96K, A100 164K, H100 256K)
- allocated on a per-block basis (can create occupancy constraints like registers)
- static allocation
 - `__shared__ <type> <varname>`
- dynamic allocation
 - `kernel<<<blocks, threads, shared_bytes>>(...)`
 - See programmer's guide for details on how to access

BEWARE of bank conflicts!

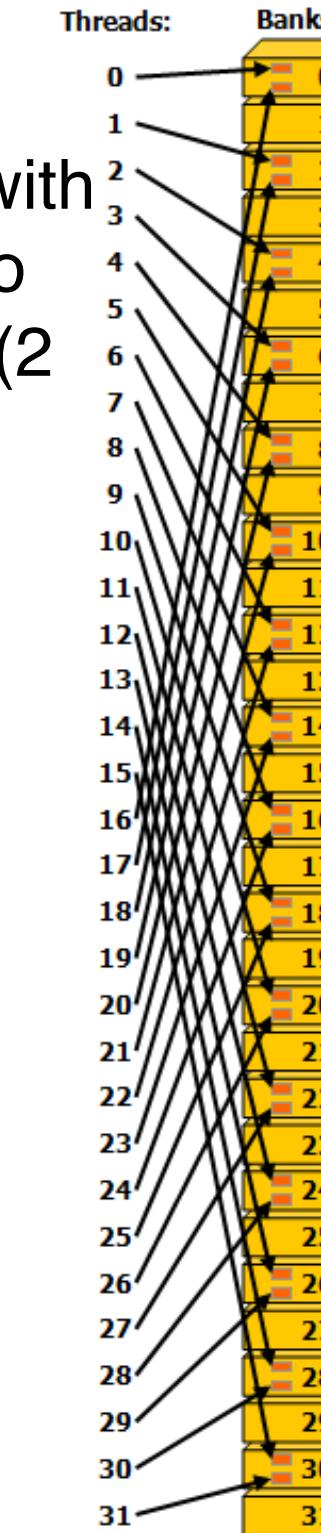
Shared Memory: Bank Conflict 1

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks.

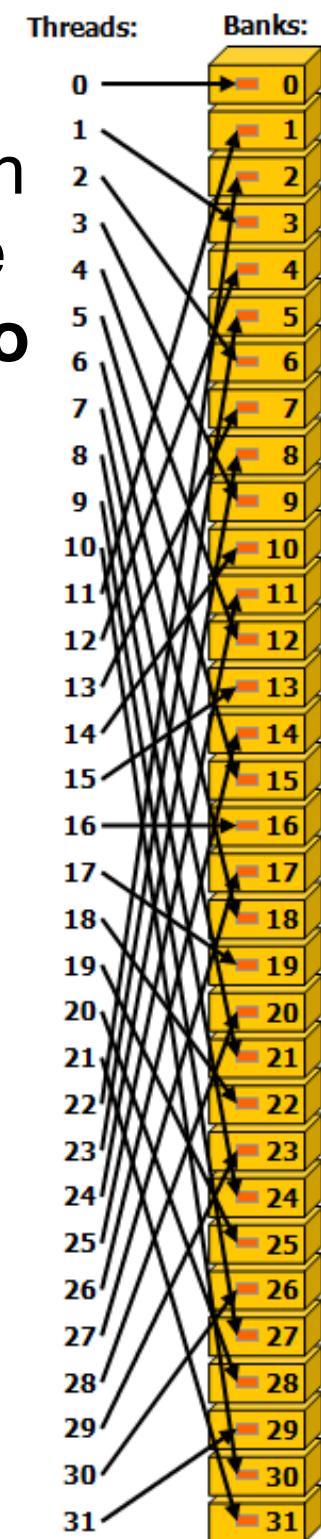
Linear addressing with a stride of one 32-bit word (no bank conflict).



Linear addressing with a stride of two 32-bit words (2 way bank conflict)



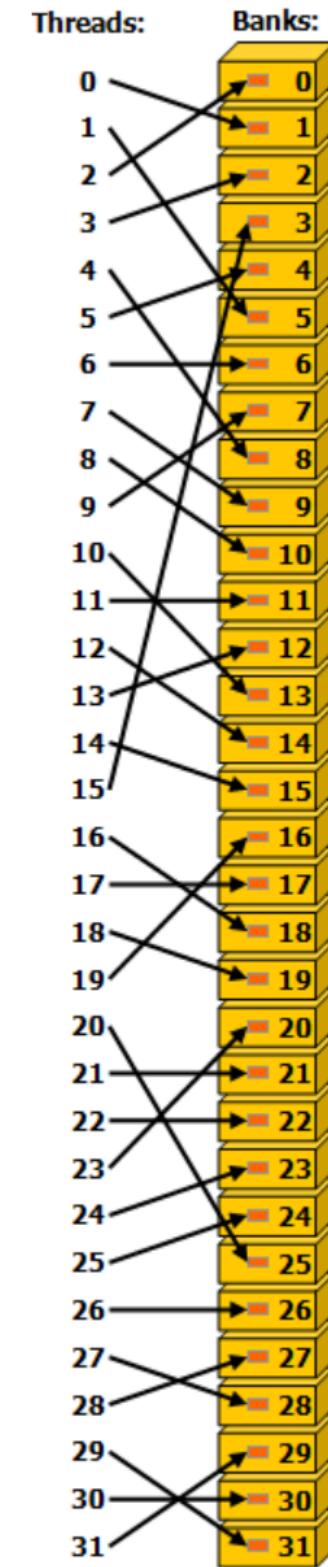
Linear addressing with a stride of three 32-bit words (**no bank conflict**)



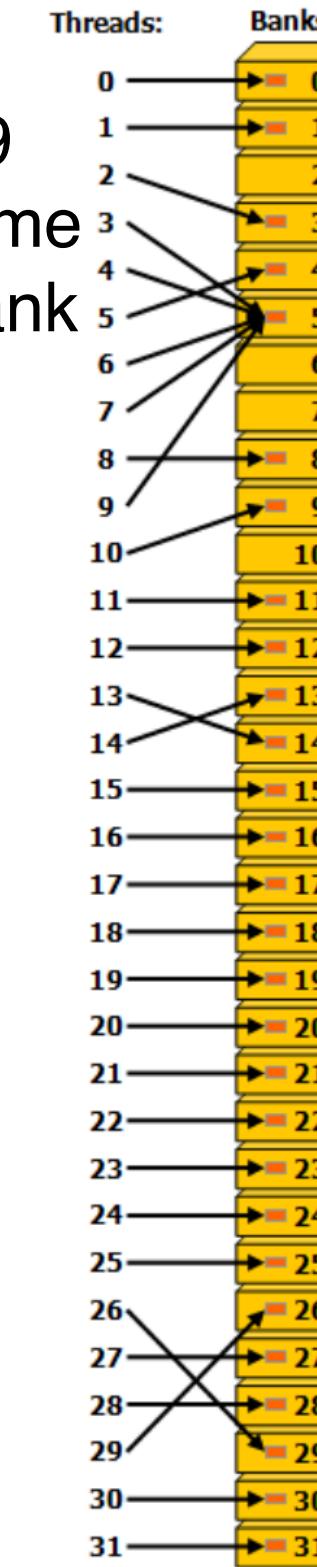
Shared Memory: Bank Conflict 2

A shared memory request for a warp does not generate a **bank conflict** between two threads that access any address within the **same 32-bit word** (even though the two addresses fall in the same bank). In that case, for read accesses, the word is broadcast to the requesting threads

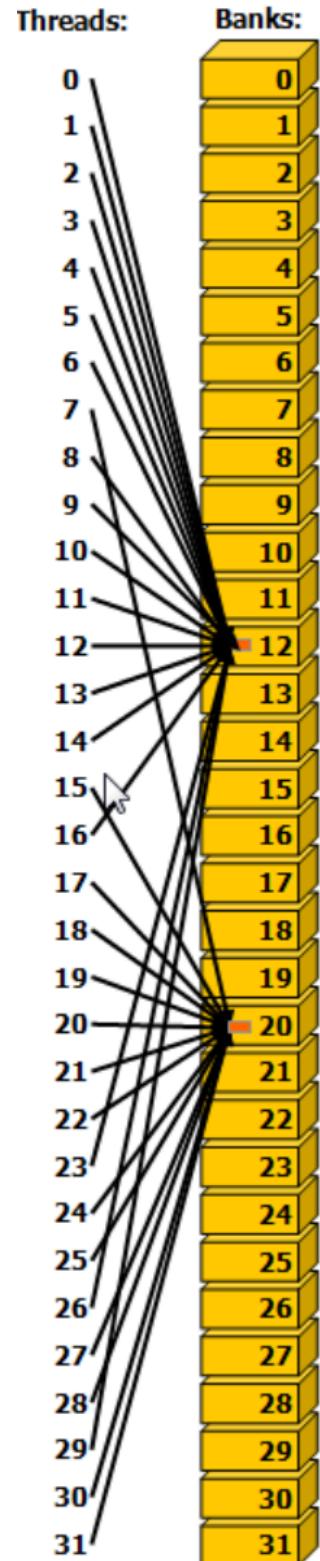
Conflict-free
access via
random
permutation



Conflict-free
since 3,4,6,7,9
access the same
word within bank
5



Conflict-free
broadcast access
(threads access
the same word
within a bank)



Shared Memory: Synchronization

- When using shared memory to share data between threads of a block, synchronization is needed (no control of order of warp execution and out-of-order memory ops)
`__syncthreads()` (and friends see manual also see `__syncwarp` for fancy warp level sync)
 - block-level synchronization barrier (recently added cluster synchronization, also now an `async-barrier` version)
 - each thread, when it reaches the statement, blocks until
 - all other threads have reached it as well
 - AND all global and shared memory written by the threads are visible to all threads (includes a memory fence)
- Note: threads in different blocks **can't** synchronize!

Q: What do you do if you want to sync threads across blocks?

Shared Memory: Synchronization

- With `__syncthreads()` watch out:
 - make sure all threads can reach `__syncthreads()`
 - otherwise: deadlock!
- The following code is a no-no:

```
if ( ... ) {  
    ...  
    __syncthreads() ;  
    ...  
} else {  
    ...  
    __syncthreads() ;  
    ...  
}
```

The `__syncthreads*` intrinsics are permitted in conditional code, but only if the condition evaluates uniformly across the entire thread block. Otherwise, execution may hang or produce unintended side effects.

Memories Summary:

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	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 in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.					
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.					

Matrix Transpose Example

Using Shared memory to optimize accesses but...

This is way more subtle than first meets the eye (but tracing some historical work teaches you a lot):

1. 2010 "Optimizing Matrix Transpose in CUDA"

<https://developer.download.nvidia.com/assets/cuda/files/MatrixTranspose.pdf>

2. 2013 "An Efficient Matrix Transpose in CUDA C/C++" no partition camping

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory-in-matrix-multiplication-c-ab>

<https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpptransposetranspose.cu>

3. 2024 "Tutorial: Matrix Tranpose in CUTLASS", includes initial look at TMA

<https://research.colfax-intl.com/tutorial-matrix-transpose-in-cutlass/>

Of course there is "stuff" on the net including more recent claims but I have not read or validated"

Matrix Transpose Example

Using
This
histo
1. 20
2. 20
camp
3. 202
at TMA

This has become such an important example for helping a developer understand memory optimization strategies that, as of CUDA 13.1, it is covered in the programming guide. But they make it seem obvious and easy. Reading about the effort that went into discovering and optimizing Matrix Transposition is still very helpful. Here are the links to the coverage in the Programming Guide

Read this first and then look at other references

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#matrix-transpose-example-using-global-memory>

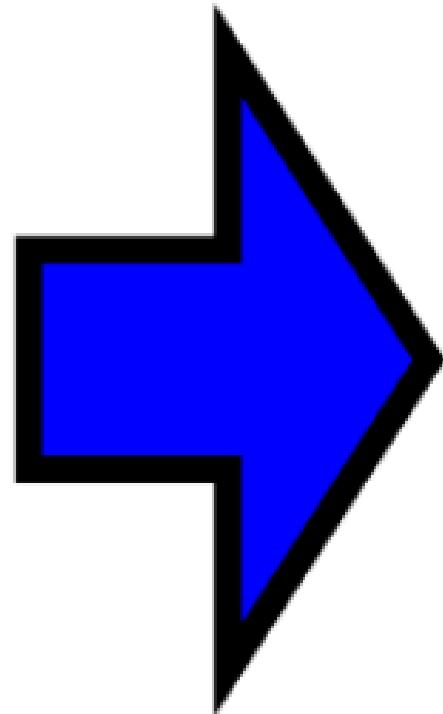
<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#matrix-transpose-example-using-shared-memory>

multiplication-c-ab
transpose.cu

Of course there is "stuff" on the net including more recent claims but I have not read or validated"

Example: Transpose: the goal

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Example: Transpose: the naive** code

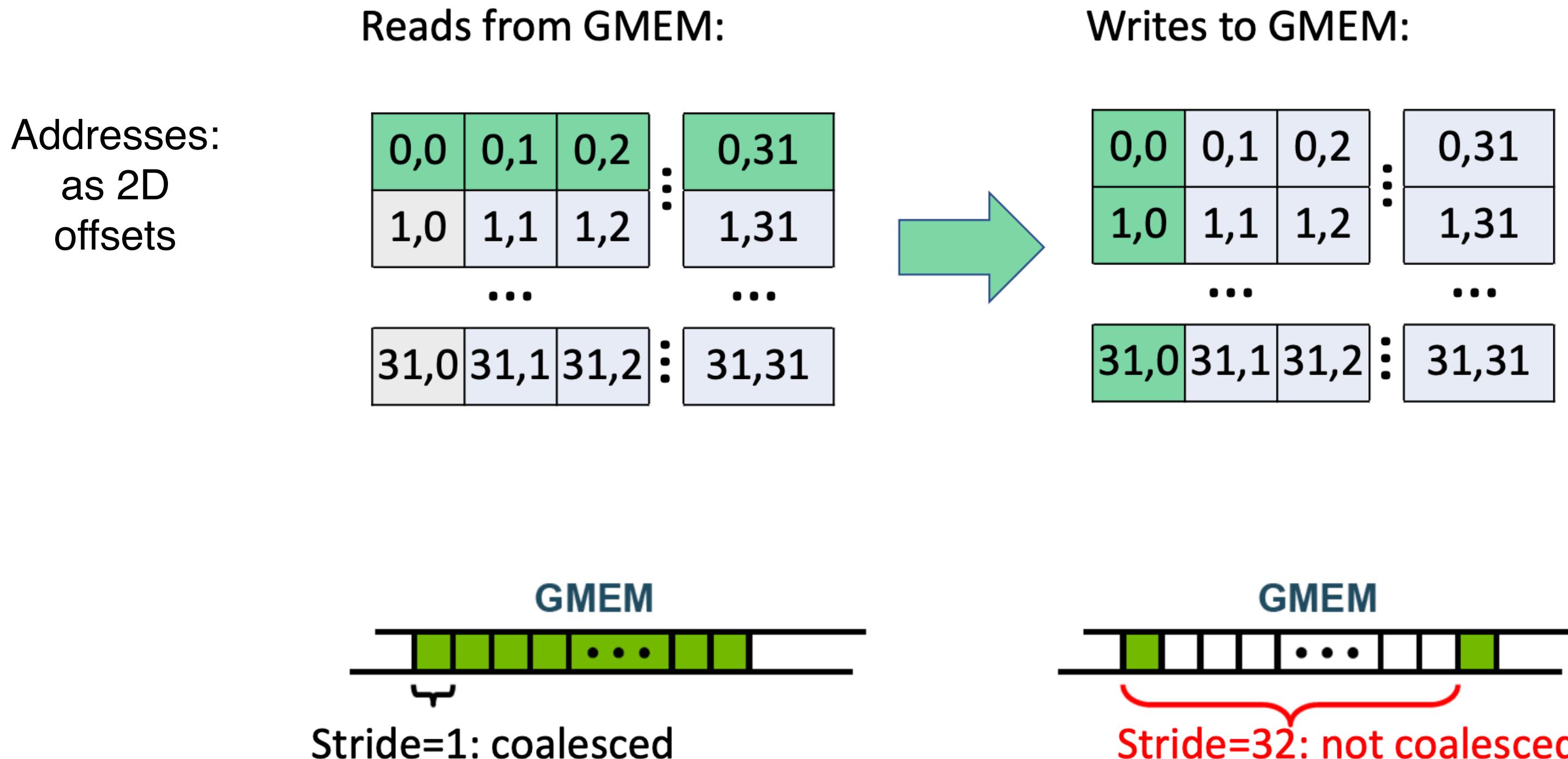
```
__global__ void transpose_naive(float *odata, float *idata,
                                int width, int height) {
    unsigned int xIndex = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int yIndex = threadIdx.y + blockIdx.y * blockDim.y;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

This is a non-tiled naive version and does one element per-thread -- different than the starting point in the NVIDIA blog and study but similar to Programming Guide

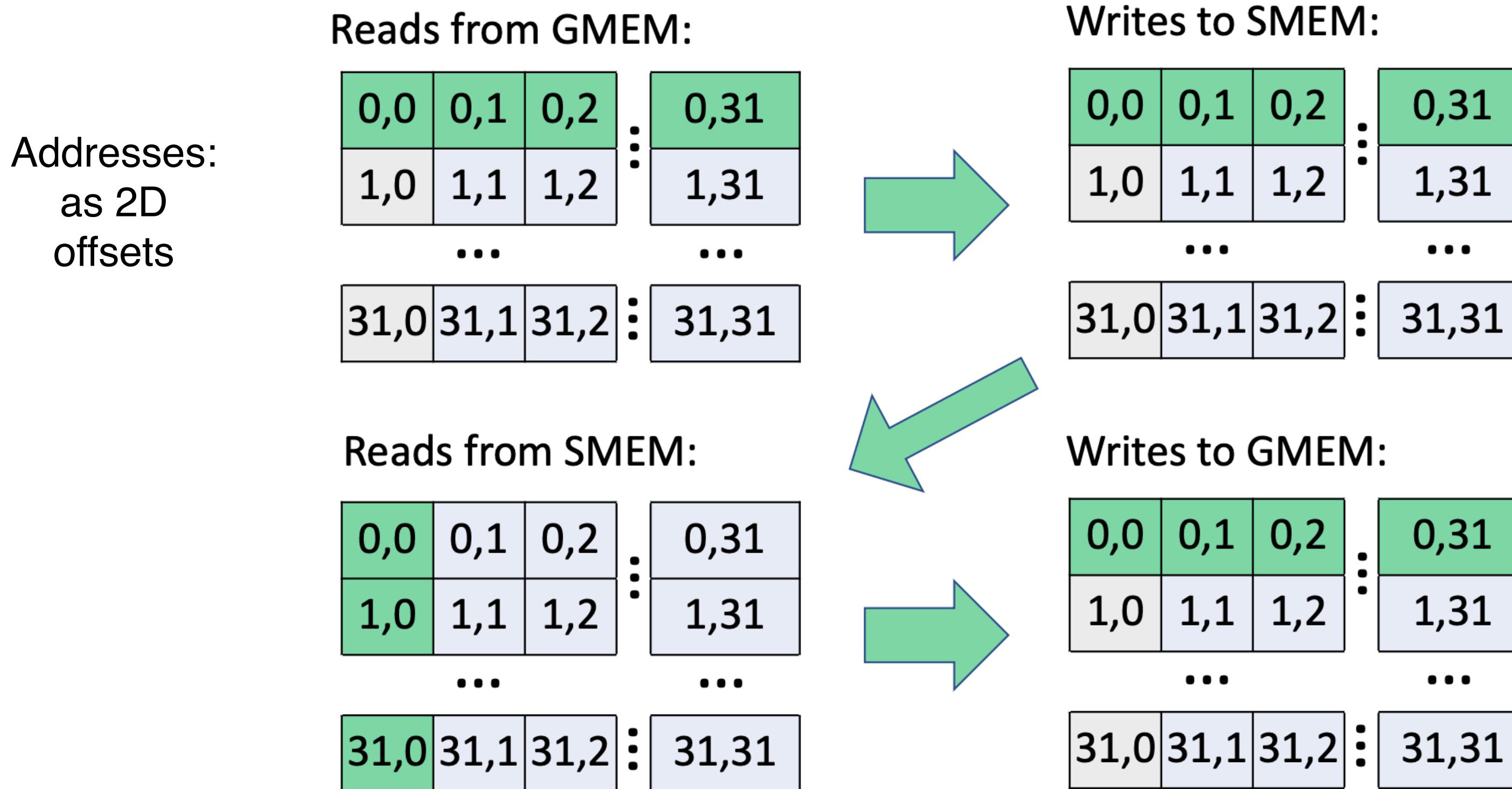
Example: Transpose: the problem

Naïve approach leads to uncoalesced global mem accesses:



Example: Transpose: the idea add SM

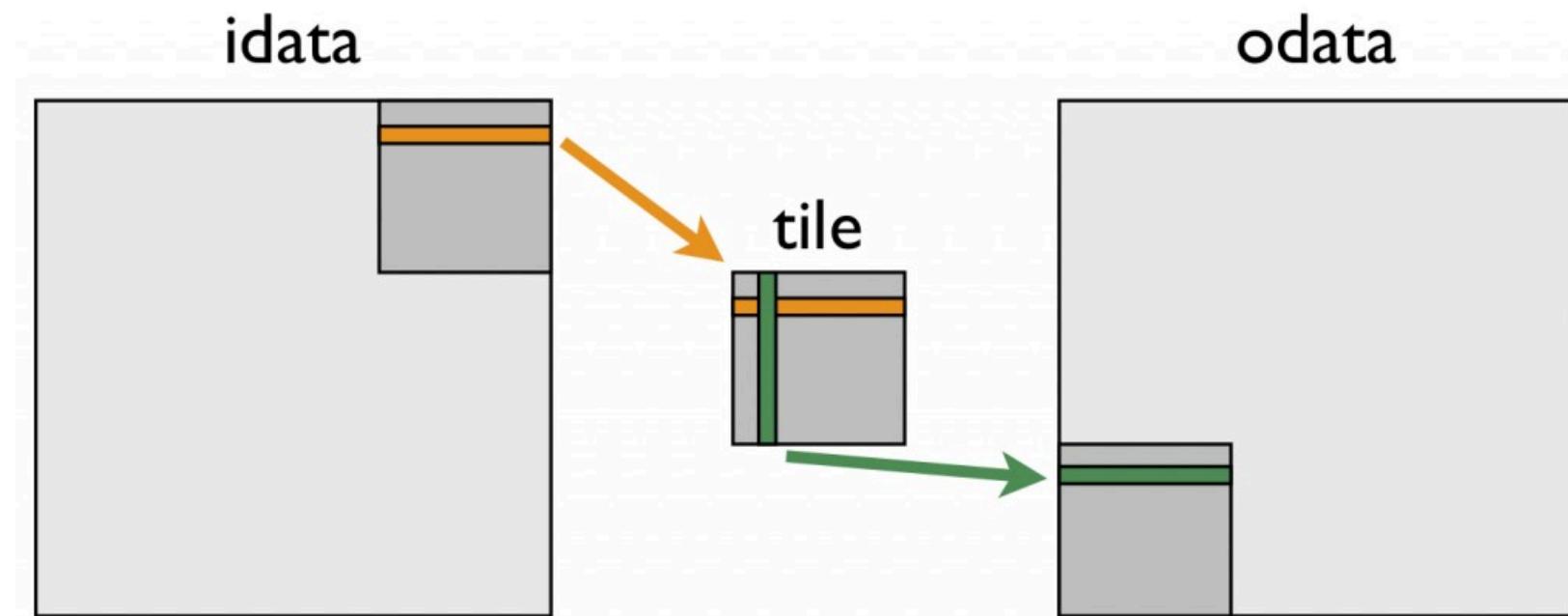
Idea: stage through shared memory:



Example: Transpose: SM + Tiling

Basic idea: conceptually partition matrix into square tiles

- Thread block (bx,by):
 - Read the (bx,by) input tile, store into SMEM
 - Write the SMEM data to (by,bx)
 - Transposing the indexing into SMEM
- Thread (tx,ty)
 - Reads element(s) (tx,ty) from input tile (possibly multiple if per-thread blocking factor used)
 - synchronization required
 - Writes element(s) into output tile
- Coalescing is achieved if:
 - Tile dimensions are multiples of 32



Example: Transpose: SM Bank Conflicts

New Problem: shared memory bank conflicts:

Reads from SMEM:

0,0	0,1	0,2	:	0,31
1,0	1,1	1,2	:	1,31
...			...	
31,0	31,1	31,2	:	31,31

Threads read SMEM with stride 32

- 32x32-way bank conflicts
- 32x slower than no conflicts

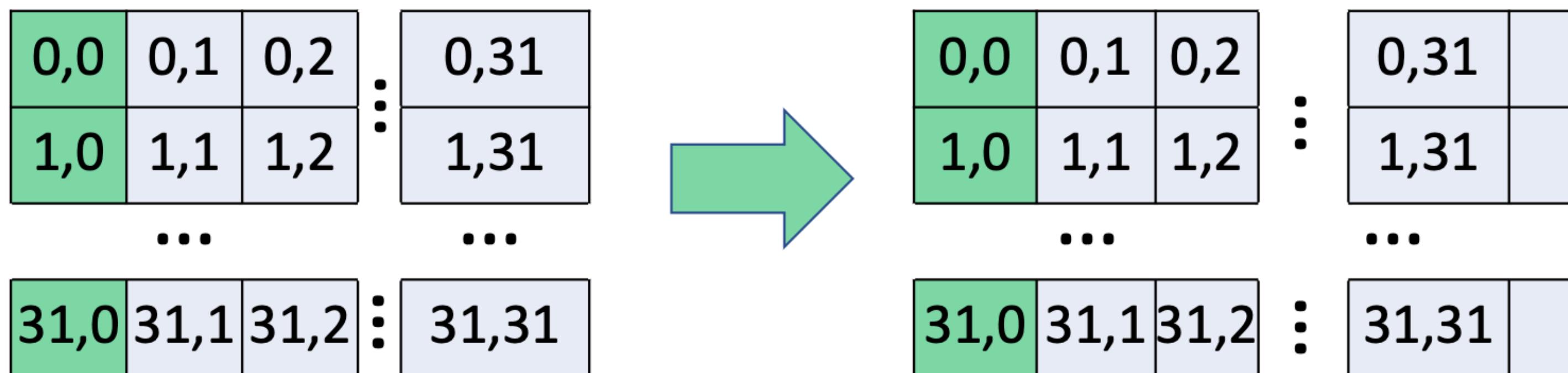
How to fix???

Example: Transpose: Add Padding

Solution: add an extra column (not used):

- Read stride = 33
 - Threads read from consecutive banks
 - No bank conflicts

Reads from SMEM:



Methodology in NVIDIA Tranpose "literature" is very useful in getting you in the right mindset.

This is way more subtle than first meets the eye (but tracing some historical work teaches you a lot):

1. 2010 "Optimizing Matrix Transpose in CUDA"
<https://developer.download.nvidia.com/assets/cuda/files/MatrixTranspose.pdf>
2. 2013 "An Efficient Matrix Transpose in CUDA C/C++" no partition camping
<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory-in-matrix-multiplication-c-ab>
<https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpptransposetranspose.cu>
3. 2024 "Tutorial: Matrix Tranpose in CUTLASS", includes initial look at TMA
<https://research.colfax-intl.com/tutorial-matrix-transpose-in-cutlass/>

Recall: Vocabulary you should know

- **SM**: Streaming Multiprocessor
- **kernel**: function run on GPU, launched by host
- **warp**: group of threads that can execute in lockstep (SIMD)
- **block**: group of threads running on one SM
 - can be 1D, 2D, or 3D
 - partitioned into warps
 - SM runs an integral number of blocks concurrently
- **grid**: set of blocks representing the entire data set
 - can be 1D, 2D, or 3D

Recall: General Structure of simple CUDA Program

- allocate GPU memory
- copy data from CPU memory to GPU
 - memory: `cudaMemcpy(dest, src, cudaMemcpyHostToDevice);`
- invoke kernel: `kernel<<<blocks, threads_per_block>>>(...args...);`
- (synchronize: `cudaDeviceSynchronize();`)
- copy back data from GPU memory to CPU
 - memory: `cudaMemcpy(dest, src, cudaMemcpyDeviceToHost);`
- cleanup: `cudaDeviceReset();`

Recall: Partitioning Work

Programmer's task:

- partition work into Grid of Blocks of Threads
 - e.g., by partitioning loops or data
 - grid and block can be 1D, 2D, or 3D
 - structure specified at kernel launch:
 - `kernel<<<grid,block>>(....args....);`
 - have kernel identify, for each thread, what data to process
 - available local vars:
 - `gridDim.{x,y,z}`
 - `blockDim.{x,y,z}`
 - `blockIdx.{x,y,z}`
 - `threadIdx.{x,y,z}`
- e.g.: `int tid = threadIdx.x +`
`threadIdx.y * blockDim.x +`
`threadIdx.z * blockDim.x * blockDim.y`

How best to partition?

- naturally fit application
- have a block size be a multiple of warp size
- maximize # threads per SM
 - depends on resources your threads and blocks use
 - depends on compute capability (see below)
- have # threads be multiple of # cores
- conducive to coalesced global memory accesses

Remember: Amdahl's law!

Occupancy

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#kernel-launch-and-occupancy>

"The occupancy of a CUDA kernel is the ratio of the number of active warps to the maximum number of active warps supported by the SM. In general, it's a good practice to have occupancy as high as possible which hides latency and increases performance."

Rule of thumb:

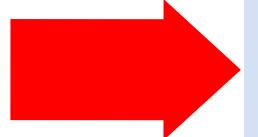
"Fill" as much work as you can into the resources of the device. Keep the cores busy and give the SM schedulers a chance to help you.



https://en.wikipedia.org/wiki/Is_the_glass_half_empty_or_half_full#/media/File:Glass-of-water.jpg

Heuristics: Number of Blocks

- The primary concern is keeping the entire GPU busy
- # blocks should be larger than the # SMs
 - each SM should have at least one block
- Since threads of a block can be waiting on each other (`__syncthreads()`)
 - good to have several small blocks than one big block to increase the likelihood of finding a warp to schedule (eg from an other block)
- But must be chosen in light of # threads in block and other constraints



Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2024	2024	2024

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. $\text{BlkSize} * j = 2048$

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. BlkSize * j = 2024
- Bad (constraints interact)
 - Too small -- Max number of Blocks per SM leaves unfilled SM Threads

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Eg. let BlkSize = 32 Thds (1 warp)
But given Max Blocks/SM = 32
at most we can fill $32 * 32 = 1024$ threads per SM

Occupancy : $1024/2048 = .5$

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. BlkSize * j = 2024
- Bad (constraints interact)
 - Too small -- Max number of Blocks per SM leaves unfilled SM Threads
 - Too big -- can't be scheduled -- "silent failure"

```
kernel<<<80, 4096>>>(...args...);  
rc = cudaGetLastError();  
if (rc != cudaSuccess) {  
    fprintf(stderr, "CUDA kernel launch failed: %s\n",  
            cudaGetStringError(rc));  
    exit(EXIT_FAILURE);  
}
```

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Eg. let BlkSize = $64 \times 64 = 4096$
BlkSize > 2048
"CUDA kernel launch failed: invalid configuration argument"

Thread Block size (BlkSize)

Good vs Bad wrt max # SM Threads

- Good
 - Multiple of warp size
 - Eg. BlkSize = $i * 32$
 - An integral number of blocks can fill max Thds/SM
 - Eg. BlkSize * j = 2024
- Bad (constraints interact)
 - Too small -- Max number of Blocks per SM leaves unfilled SM Threads
 - Too big -- can't be scheduled -- "silent failure"
 - Not integral wrt the Max Thds/SM again leaves SM Threads unfilled

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
threads/warp	32	32	32
Max Thds/Block	1024	1024	1024
Max warps/SM	64	64	64
Max Blocks/SM	32	32	32
Max Thds/SM	2048	2048	2048

Eg. let BlkSize = $32 \times 24 = 768$
 $2 * \text{BlkSize} = 1536 < 2048$ and
 $3 * \text{BlkSize} = 2304 > 2048$
therefore at most 2 blocks fit:

Occupancy : $1536/2048 = .75$

Registers & Occupancy

Usage interacts with Occupancy

- Registers
 - Compiler assigns a register for each local/automatic variable
 - # Reg per thread can limit the number of blocks that can fit on SM
 - Ex 1: if 64 Reg/Thd then only 1024 Thds can be assigned to an SM ($65536/64 = 1024$) = 50% occ

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#kernel-launch-and-occupancy>
<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#calculating-occupancy>

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
Registers/SM	65536	65536	65536
Shard Mem/SM	98304	167936	233472
Max Thds/SM	2048	2048	2048
Max Reg/Thd	255	255	255
Max Shared Mem/Blk	98304	167936	233472

Ex 2: Assume 512 thds/blk

If 31 reg/thread then 4 blks fit: $31 * 512 * 4 = 31 * 2048$ thds < 65536

However if pgmer add 2 variables:

$33 * 2048 = 67584 > 65536$ but $33 * 3 * 512 = 33 * 1536 = 50688 < 65536$.

So we can only have 3 blocks per SM

Occupancy = $1536 / 2048 = 0.75$

Registers & Occupancy

Usage interacts with Occupancy

- Registers
 - Compiler assigns local/automatic
 - # Reg per thread blocks that can
 - Ex 1: if 64 Regs can be assigned $\frac{64}{1024} = 50\%$

<https://docs.nvidia.com/cuda/parallel-thread-blocks/index.html#register-usage>

Register and shared memory usage are reported by the compiler when compiling with the `--ptxas-options=-v` or `--resource-usage` option.

Property	V100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
Registers	65536	65536	65536
Shared Memory	8304	167936	233472
Registers + Shared	2048	2048	2048
Registers / Shared	255	255	255
Registers + Shared / Shared	3304	167936	233472

Ex 2: Assume 31 Regs

If 31 reg/thread then 4 blks fit: $31 * 512 * 4 = 31 * 2048$ thds < 65536

However if pgmer add 2 variables:

$33 * 2048 = 67584 > 65536$ but $33 * 3 * 512 = 33 * 1536 = 50688 < 65536$.

So we can only have 3 blocks per SM

$$\text{Occupancy} = 1536 / 2048 = 0.75$$

Shared Mem & Occupancy

Usage interacts with Occupancy

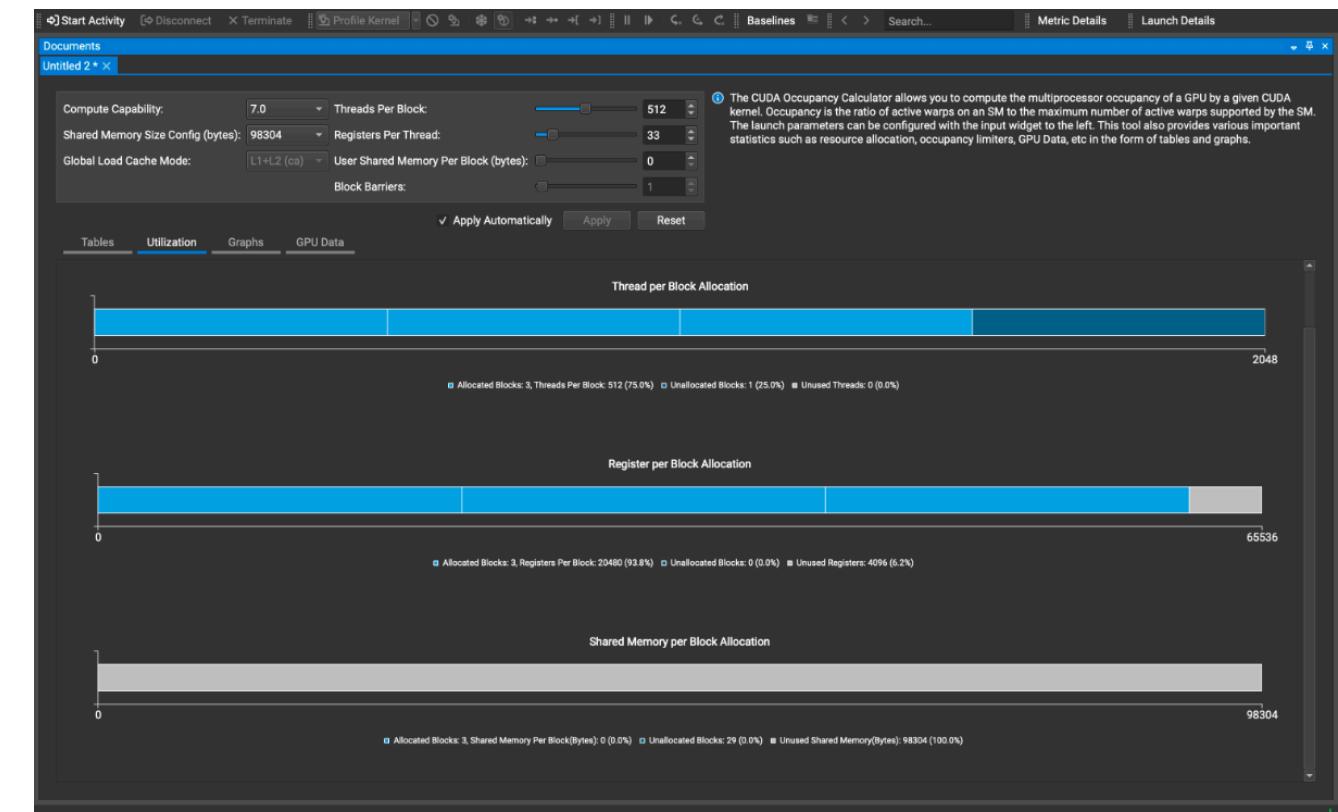
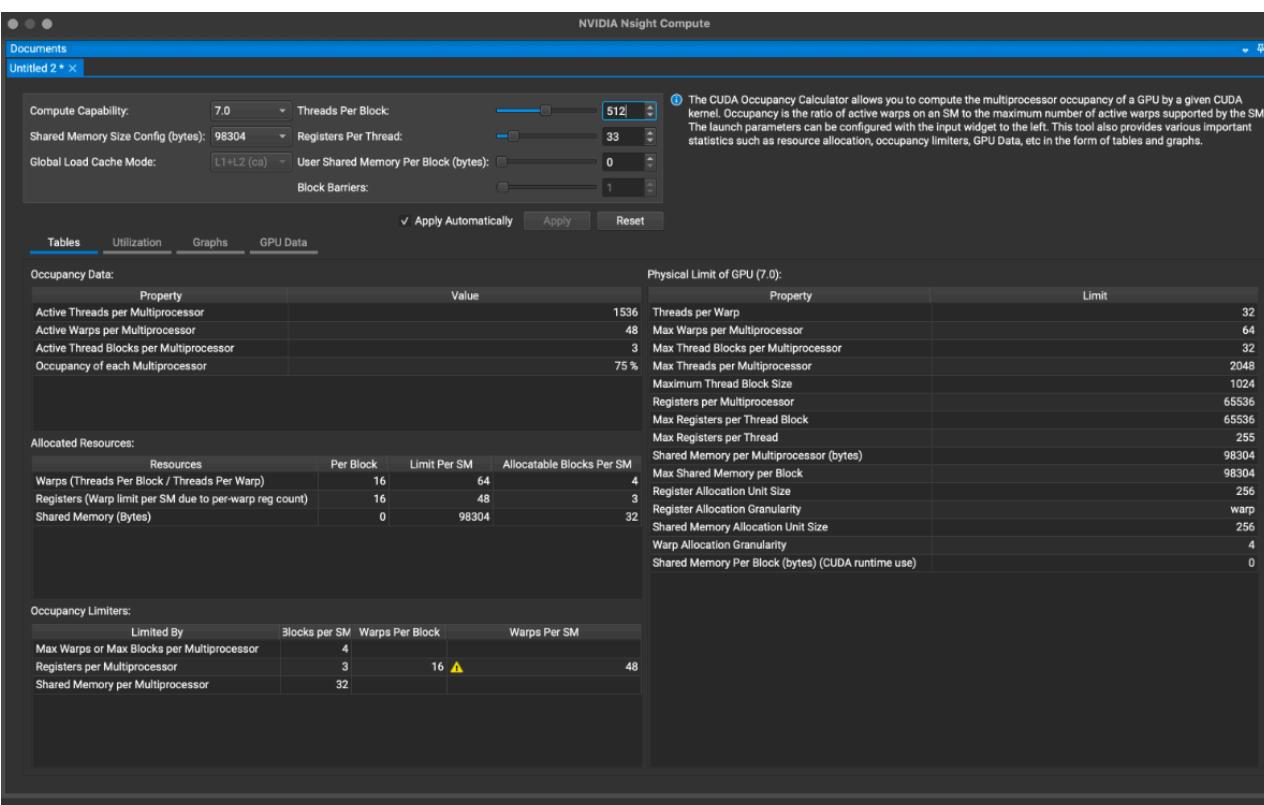
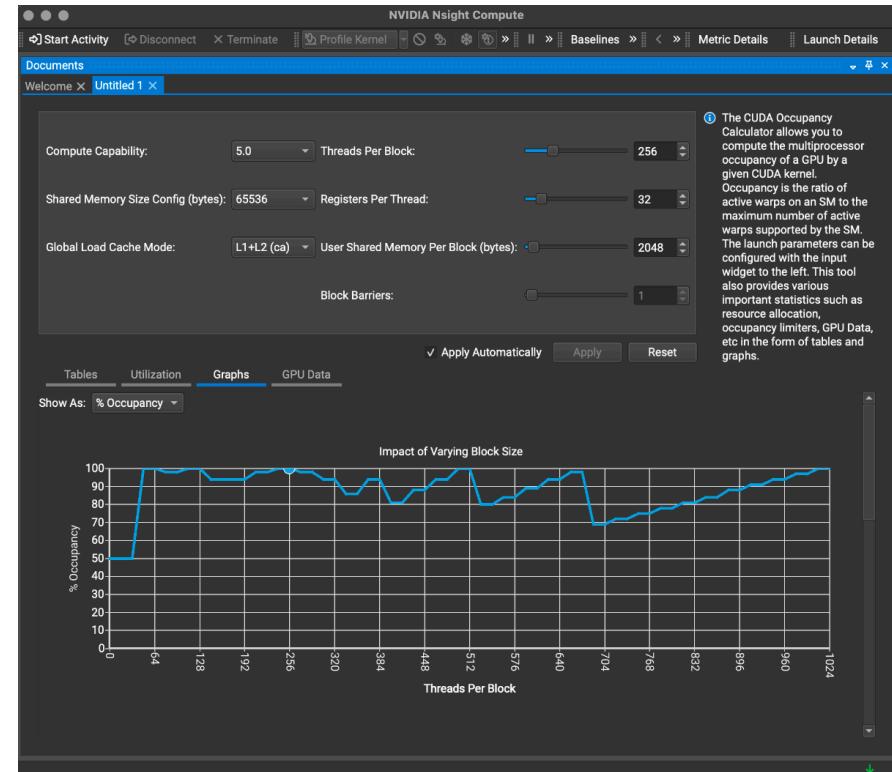
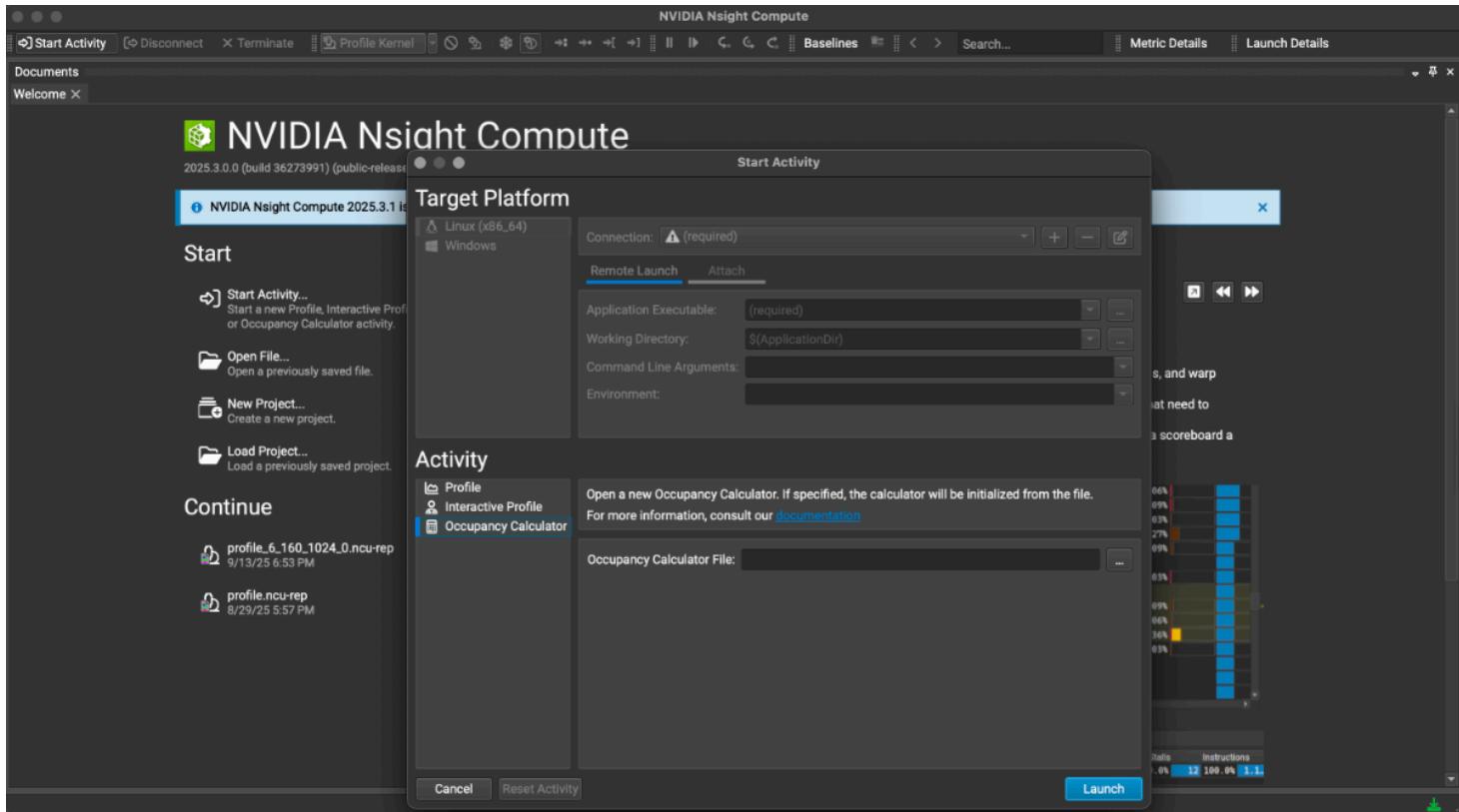
- Like Registers a thread's use of Shared Memory can constrain the number of threads/blks that can be schedule to an SM
- Note the variation in shared memory resources

Property	v100 (cc 7.0)	A100 (cc 8.0)	H100 (cc 9.0)
# SMs	80	108	114
Registers/SM	65536	65536	65536
Shard Mem/SM	98304	167936	233472
Max Thds/SM	2048	2048	2048
Max Reg/Thd	255	255	255
Max Shared Mem/Blk	98304	167936	233472

Occupancy Calculator : Nsight Compute Tool

<https://developer.nvidia.com/nsight-compute>

<https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator>



Device Properties API

<https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html#structcudaDeviceProp>

- `cudaGetDeviceCount(&numGPUs);`
 - `cudaStatus = cudaSetDevice(0)`
- `cudaDeviceProp GPUprop ;`
`cudaGetDeviceProperties(&GPUprop, 0);`
 - `GPUprop.maxGridSize[0..2]`
 - `GPUprop.maxThreadsPerBlock`
 - and much more:

<code>char name[256];</code>	<code>int multiProcessorCount;</code>	<code>int localL1CacheSupported;</code>
<code>cudaUUID_t uuid;</code>	<code>int kernelExecTimeoutEnabled;</code>	<code>size_t sharedMemPerMultiprocessor;</code>
<code>size_t totalGlobalMem;</code>	<code>int canMapHostMemory;</code>	<code>int regsPerMultiprocessor;</code>
<code>size_t sharedMemPerBlock;</code>	<code>int computeMode;</code>	<code>int managedMemory;</code>
<code>int regsPerBlock;</code>	<code>int concurrentKernels;</code>	<code>int isMultiGpuBoard;</code>
<code>int warpSize;</code>	<code>int ECCEnabled;</code>	<code>int multiGpuBoardGroupID;</code>
<code>size_t memPitch;</code>	<code>int pciBusID;</code>	<code>int singleToDoublePrecisionPerfRatio;</code>
<code>int maxThreadsPerBlock;</code>	<code>int pciDeviceID;</code>	<code>int pageableMemoryAccess;</code>
<code>int maxThreadsDim[3];</code>	<code>int asyncEngineCount;</code>	<code>int concurrentManagedAccess;</code>
<code>int maxGridSize[3];</code>	<code>int unifiedAddressing;</code>	<code>int computePreemptionSupported;</code>
<code>int clockRate;</code>	<code>int memoryClockRate;</code>	<code>int </code>
<code>size_t totalConstMem;</code>	<code>int memoryBusWidth;</code>	<code>pageableMemoryAccessUsesHostPageTables;</code>
<code>int major;</code>	<code>int l2CacheSize;</code>	<code>int directManagedMemAccessFromHost;</code>
<code>int minor;</code>	<code>int maxThreadsPerMultiProcessor;</code>	<code>int maxTexture1D;</code>
<code>size_t textureAlignment;</code>	<code>int streamPrioritiesSupported;</code>	<code>int maxTexture1DMipmap;</code>
<code>size_t texturePitchAlignment;</code>	<code>int globalL1CacheSupported;</code>	<code>int maxTexture1DLinear;</code>

FYI: `cudaDeviceGetAttribute` has better performance:

<https://developer.nvidia.com/blog/cuda-pro-tip-the-fast-way-to-query-device-properties/>

Occupancy Calculator API

8.2.3.1. Occupancy Calculator

Several API functions exist to assist programmers in choosing thread block size and cluster size based on register and shared memory requirements.

- › The occupancy calculator API,

`cudaOccupancyMaxActiveBlocksPerMultiprocessor`, can provide an occupancy prediction based on the block size and shared memory usage of a kernel. This function reports occupancy in terms of the number of concurrent thread blocks per multiprocessor.

- › Note that this value can be converted to other metrics. Multiplying by the number of warps per block yields the number of concurrent warps per multiprocessor; further dividing concurrent warps by max warps per multiprocessor gives the occupancy as a percentage.

- › The occupancy-based launch configurator APIs,

`cudaOccupancyMaxPotentialBlockSize` and `cudaOccupancyMaxPotentialBlockSizeVariableSMem`, heuristically calculate an execution configuration that achieves the maximum multiprocessor-level occupancy.

- › The occupancy calculator API, `cudaOccupancyMaxActiveClusters`, can provided occupancy prediction based on the cluster size, block size and shared memory usage of a kernel. This function reports occupancy in terms of

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#occupancy-calculator>

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__OCCUPANCY.html

Occupancy Summary & Subtleties

EXPERIMENT!!!!

"... higher occupancy does not always equate to better performance. For example, improving occupancy from 66 percent to 100 percent generally does not translate to a similar increase in performance. A lower occupancy kernel will have more registers available per thread than a higher occupancy kernel, which may result in less register spilling to local memory; in particular, with a high degree of exposed instruction-level parallelism (ILP) it is, in some cases, possible to fully cover latency with a low occupancy.

There are many such factors involved in selecting block size, and inevitably some experimentation is required. However, a few rules of thumb should be followed:

- ▶ Threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.
- ▶ A minimum of 64 threads per block should be used, and only if there are multiple concurrent blocks per multiprocessor.
- ▶ Between 128 and 256 threads per block is a good initial range for experimentation with different block sizes.
- ▶ Use several smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call `__syncthreads()`.

Basic Timing



https://commons.wikimedia.org/wiki/File:Louis_Moinet%27s_%22Compteur_de_Tierces%22.jpg#/media/File:Louis_Moinet's_%22Compteur_de_Tierces%22.jpg

Measuring time

CPU

- Note time measurement imprecise with lots variability
- measure n times: take best time
- See NVIDIA "CUDA C Best Practice Guide" Using CPU Timers for how time CUDA kernel calls with CPU Timers
- See Timing Operations in CUDA Streams in the programming guide

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#using-cpu-timers>

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#timing-operations-in-cuda-streams>

```
#include <stdio.h>
#include <assert.h>
#include <inttypes.h>
#include <time.h>

#define CLOCK_SOURCE CLOCK_MONOTONIC
#define NSEC_IN_SECOND (1000000000)

typedef struct timespec ts_t;

static inline int ts_now(ts_t *now) {
    if (clock_gettime(CLOCK_SOURCE, now) == -1) {
        perror("clock_gettime");
        assert(0);
        return 0;
    }
    return 1;
}

static inline uint64_t ts_diff(ts_t start, ts_t end)
{
    uint64_t diff =
        ((end.tv_sec - start.tv_sec) * NSEC_IN_SECOND) +
        (end.tv_nsec - start.tv_nsec);
    return diff;
}
```

This blog entry has an expanded explanation of using events for timing

https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/#timing_using_cuda_events

This seems to be the recommended approach

Measuring time GPU

1. GPU `clock()` & `clock64()`

- returns value of clock cycle counter
- 1 value per thread!
- need to convert using
`cudaDeviceProp::clockRate`

2. CUDA Timers (Event Records)

3. Using NVIDIA Profiler

- nsight compute profiler (ncu)
- nsight system profile (nsys)

```
16  __global__ void helloFromGPU()
17  {
18      printf("Hello from GPU\n");
19  }
20
21 int main(int argc, char **argv)
22 {
23     cudaEvent_t start;
24     cudaEvent_t end;
25
26     // create the events
27     CUDA_CHECK(cudaEventCreate(&start));
28     CUDA_CHECK(cudaEventCreate(&end));
29
30     // record the start event
31     CUDA_CHECK(cudaEventRecord(start, 0));
32
33     helloFromGPU <<< 1, 100 >>>();
34
35     // record the end event
36     CUDA_CHECK(cudaEventRecord(end, 0));
37
38     CUDA_CHECK(cudaGetLastError());
39     CUDA_CHECK(cudaDeviceSynchronize());
40
41     // get the timing
42     float elapsedTime;
43     CUDA_CHECK(cudaEventElapsedTime(&elapsedTime, start, end));
44     printf("Kernel execution time: %0.6f ms\n", elapsedTime);
45
46     // clean up
47     CUDA_CHECK(cudaEventDestroy(start));
48     CUDA_CHECK(cudaEventDestroy(end));
49     CUDA_CHECK(cudaDeviceReset());
50 }
```

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#timing-operations-in-cuda-streams>

The programming Guide discusses a general approach to timing with multiple streams. The above uses the default stream.

**Lets look at another code
example**

Very basic Matrix Add

MatAdd I

```
void matrixSumHost(float *A, float *B, float *C,
                    int nx, int ny) {
    float *ia=A, *ib=B, *ic=C;
    for (int iy=0; iy<ny; iy++) {
        for (int ix=0; ix<nx; ix++) {
            ic[ix] = ia[ix] + ib[ix];
        }
        ia+=nx; ib+=nx; ic +=nx;
    }
}

__global__ void matrixSumGPU(float *A, float *B, float *C,
                            int nx, int ny) {
    int ix = threadIdx.x + blockIdx.x*blockDim.x;
    int iy = threadIdx.y + blockIdx.y*blockDim.y;
    int idx = iy*nx + ix;
    if ((ix<nx) && (iy<ny)) {
        C[idx] = A[idx] + B[idx];
    }
}
```

MatAdd II

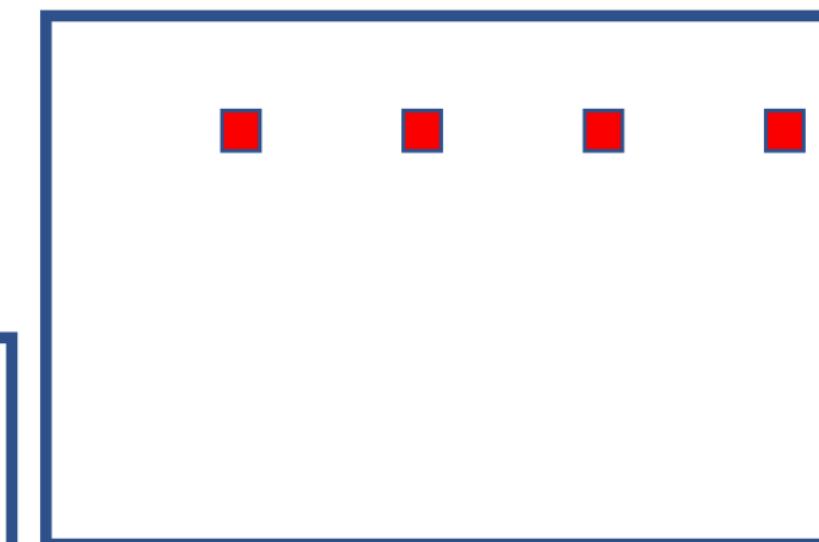
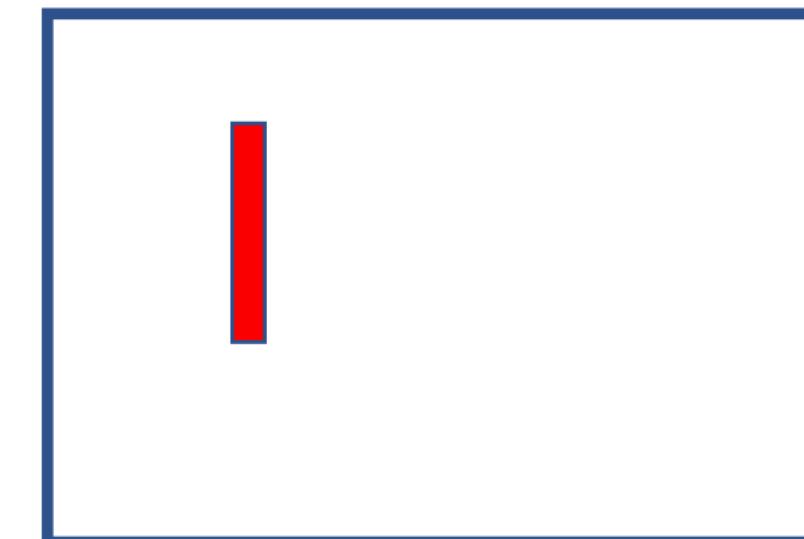
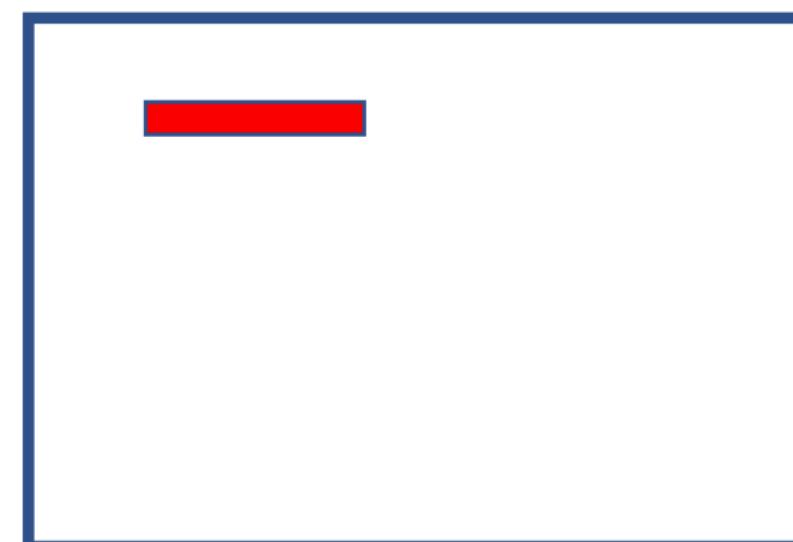
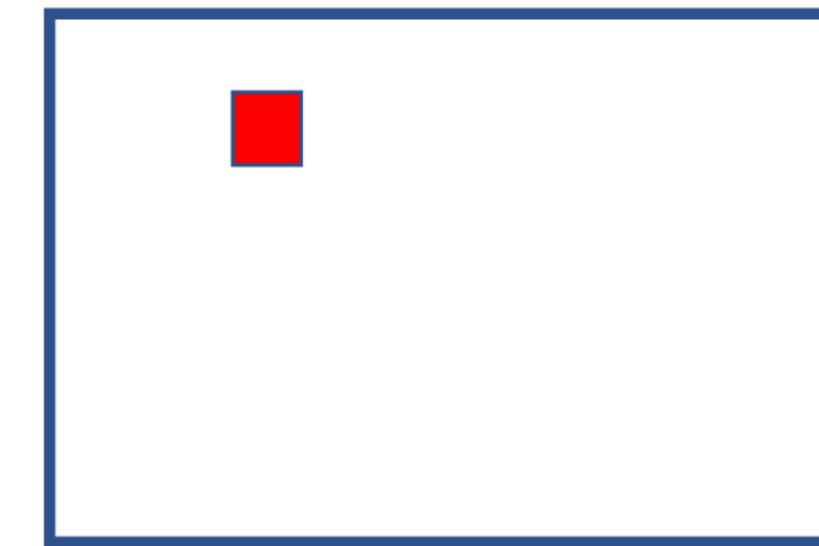
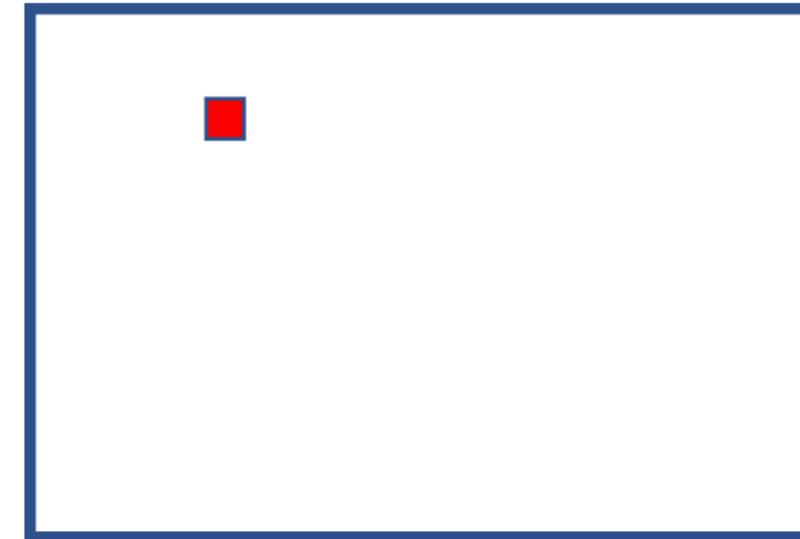
```
int main(int argc, char **argv) {
    //set matrix size
    int nx = atoi(argv[1]);
    int ny = atoi(argv[2]);
    int noElems = nx*ny;
    int bytes = noElems * sizeof(float);
    // alloc memory host-side
    float *h_A = (float *) malloc(bytes);
    float *h_B = (float *) malloc(bytes);
    float *h_hC = (float *) malloc(bytes); // host result
    float *h_dC = (float *) malloc(bytes); // gpu result
    // init matrices with random data
    initData(h_A, noElems); initData(h_B, noElems);
    // alloc memory dev-side
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, bytes);
    cudaMalloc((void **) &d_B, bytes);
    cudaMalloc((void **) &d_C, bytes);
```

MatAdd III

```
//transfer data to dev
cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);
// invoke Kernel
dim3 block(32, 32); // configure
dim3 grid((nx + block.x-1)/block.x,
           (ny + block.y-1)/block.y);
matrixSumGPU<<<grid, block>>>(d_A, d_B, d_C, nx, ny);
cudaDeviceSynchronize();
//copy data back
cudaMemcpy(h_dC, d_C, bytes, cudaMemcpyDeviceToHost);
// free GPU resources
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
cudaDeviceReset();
// check result
matrixSumHost(h_A, h_B, h_hC, nx, ny);
cmpMats(h_hC, h_dC, noElems);
return 0;
}
```

Things to try I: More operations per byte

Now consider threads computing multiple elements



Things to try II: Improve host-device copies 1

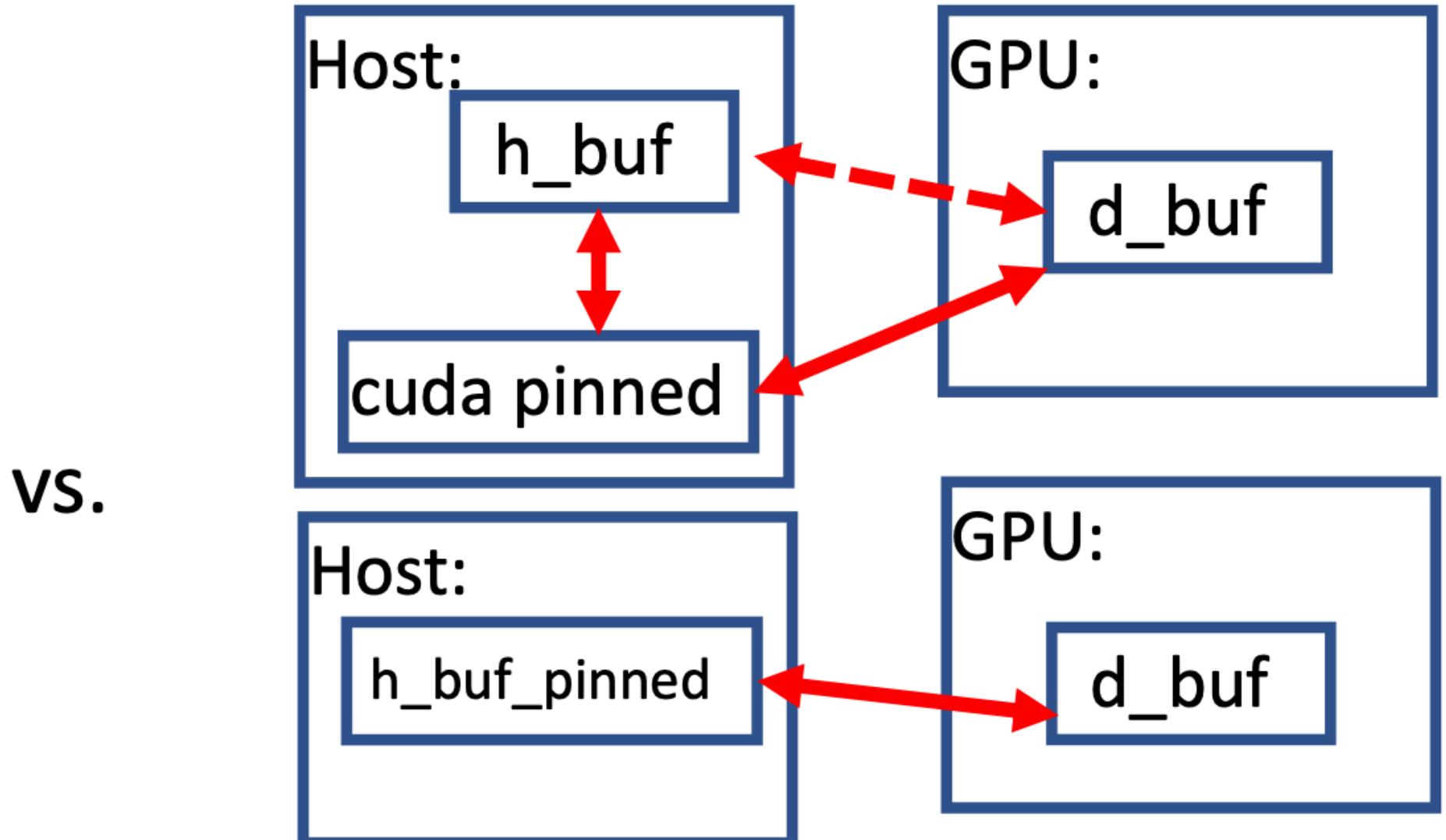
- To DMA to/from host, host memory has to be pinned
- If not pinned, extra copy
- Supporting functions:

```
cudaHostAlloc(void** pHHost, size_t size, unsigned int flags)
```

```
cudaFreeHost(void *ptr)
```

```
cudaHostRegister(void *ptr, size_t size, flags)
```

```
cudaHostUnregister(void *ptr)
```



if already alloc'ed

Things to try III: Use function specifiers

	Executed On	Callable from
<code>__global__ void kernel_fct()</code>	device	host & <code>device*</code>
<code>__device__ float dev_fct()</code>	device	device
<code>__host__ float host_fct()</code>	host	host

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=cudaHostAlloc#function-execution-space-specifiers>

- **`__global__`**
 - defines a kernel function
 - must return void
 - can only call `__device__` functions
- `__device__` and `__host__` can be used together
 - two different versions of code generated

CUDA & GPU Memories

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#gpu-device-memory-spaces>

Table 1: Memory Types, Scopes and Lifetimes

Memory Type	Scope	Lifetime	Location
Global	Grid	Application	Device
Constant	Grid	Application	Device
Shared	Block	Kernel	SM
Local	Thread	Kernel	Device
Register	Thread	Kernel	SM

Registers, Shared, & Device (Global and Local)

Registers

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#registers>

Fast and holds operands for instructions

- 1. On-chip fast and typed**
- 2. Per-Thread only.**



Registers I



Per thread registers and
local memory

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-hierarchy-memory-hierarchy-figure>

- Each SM: Set of 32-bit registers partitioned among the warps
- Provides fastest access -- directly used by instructions as operands
 - latency if: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#multiprocessor-level>
 - memory dependency: value is being loaded into register from memory
 - register dependency: value is being produced by a prior instruction
- Kernel Local/Automatic variables are placed in registers by the compiler (Some exceptions: See Local Memory later)
- as discussed # registers used by a kernel can impact occupancy
 - compiler tries to minimize register use
 - compiler can "spill" registers to further reduce register pressure
 - you can control usage: 1) **-maxrregcount** (nvcc flag), 2) **__launch_bounds__()** (kernel pragma), and 3) **__maxreg__()** qualifier.
 - spills to Local Memory with new support (CUDA 13) to Shared Memory

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#maximum-number-of-registers-per-thread>

<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#pragma-strings-enable-smem-spilling>

<https://developer.nvidia.com/blog/how-to-improve-cuda-kernel-performance-with-shared-memory-register-spilling/>

An automatic variable declared in device code without any of the `device`, `shared` and `constant` memory space specifiers described in this section generally resides in a register. However in some cases the compiler might choose to place it in local memory, which can have adverse performance consequences as detailed in Device Memory Accesses.

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/intro-to-cuda-cpp.html#variable-specifiers>

Registers II



Per thread registers and local memory

- Registers are 32 bits -- each variable consumes at least one 32-bit register
 - doubles consume 2 <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#multiprocessor-level>
- You can help the compiler to use register optimizations (eg, avoid accessing values through pointers)
 - use **__restrict__** on pointers to let the compiler know there is no aliasing
 - `void foo(const float * __restrict a, const float * __restrict b, float * __restrict c)`
 - be careful -- everything is a balancing act -- will increase register pressure and therefore affect occupancy
- New async copies can be used to avoid register use when moving data between global and shared memory <https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html#asynchronous-data-copies>
- Core register facts:
 - Number of 32 bit registers per SM 64K
 - Maximum number of registers per Thread Block 64K
 - Maximum number of registers per Thread 255

Registers II



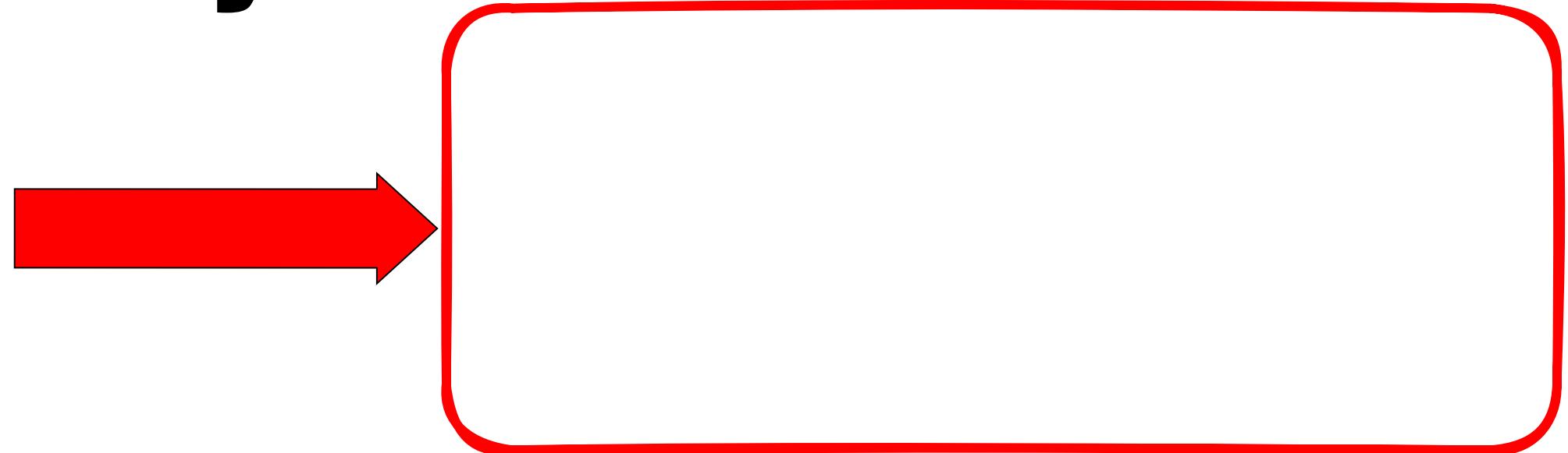
Per thread registers and local memory

- Registers are 32 bits -- each variable consumes at least one 32-bit register
 - doubles consume 2 <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#multiprocessor-level>
- You can have multiple threads per thread block. This is called **fat threads**.
 - values that are shared between threads are stored in **local memory**.
 - You may want to explore register usage tradeoffs:
 - Fat vs Thin threads with respect to register use.
 - AGAIN, it requires **experimentation** to determine what is best for performance.
- New asynchronous API for CUDA. It includes support for global and local memory. nvcc **--ptxas-options=-v** or **--resource-usage** to see register usage
- Core register limits
 - Number of registers per Thread Block 128K
 - Maximum number of registers per Thread Block 64K
 - Maximum number of registers per Thread 255

Device Memory

Big & slow

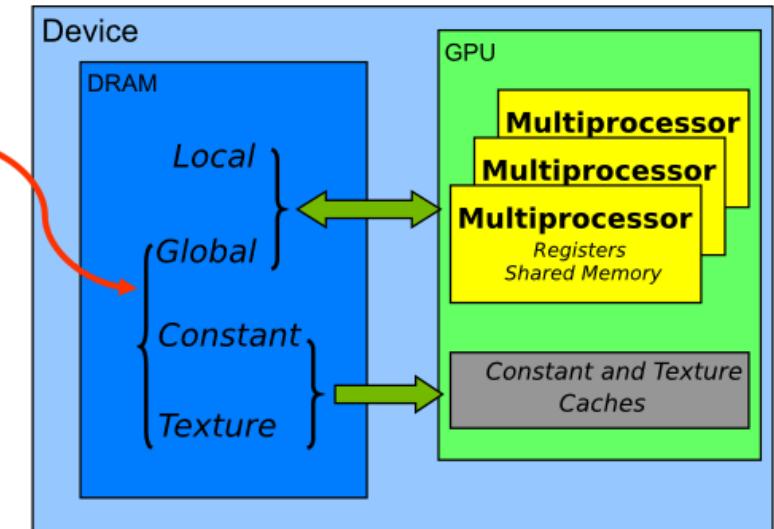
1. 100+ cycles to access
2. some data can be cache
3. exploiting coalesced access can be critical (**Global Memory**)



Device Memory:Local Memory

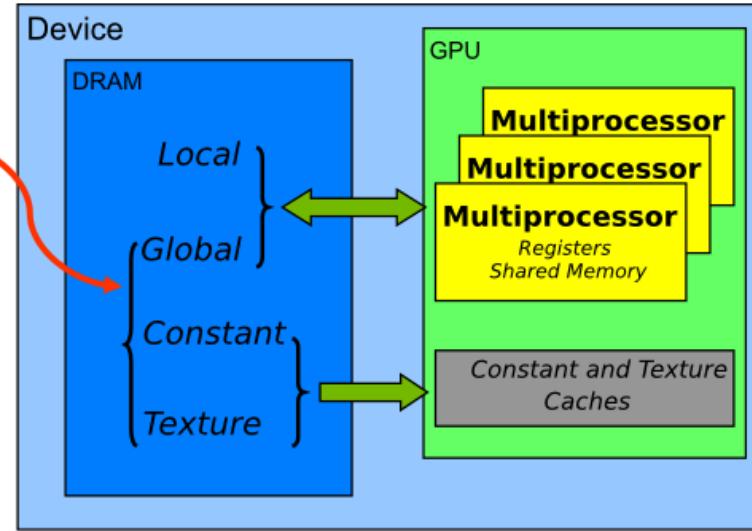
- Per thread
- In device memory: SLOW
 - organized so typical accesses are coalesced
 - cached in L2
- Compiler:
 - Arrays, it can't determine to be indexed with constant quantities
 - Large structures or arrays that would take up too many registers
 - Spills: Any variable if the kernel uses more registers than available (CUDA 13 allows spilling to local memory)
- compiler reports total local memory usage (lmem) per kernel:
 - `--ptxas-options=-v`

Device Memory: Global Memory I



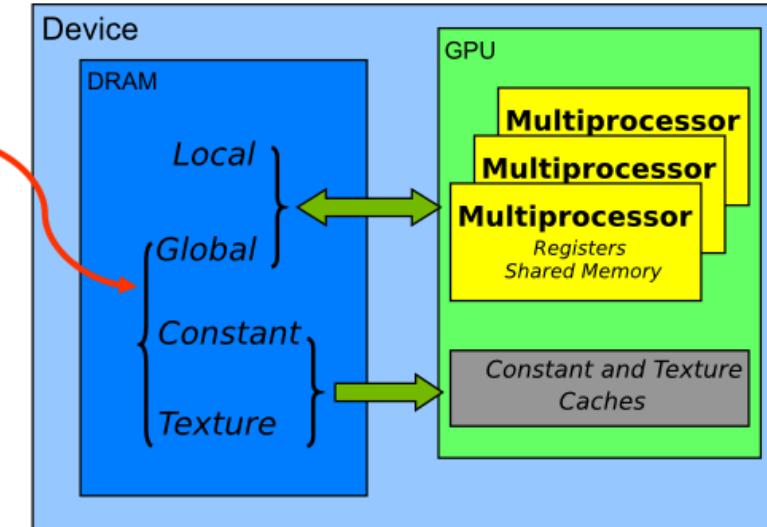
- Big & Global: accessible from all SM's and by the Host
- 100++ cycle latency
- dynamic allocation (host side):
 - `cudaMemalloc(....)`
- dynamic allocation (device side)
 - `malloc(...), new(...)`
- static allocation
 - `__device__ <type> <varname>`
 - host can read/write using special routines:
 - `cudaMemcpyToSymbol("var", *h_var, sizeof(var_type))`
 - `cudaMemcpyFromSymbol(...)`
 - or with standard `cudaMemcpy()` in combination with `cudaGetSymbolAddress(...)`
 - not used that often, but sometimes useful

Device Memory: Global Memory II



- 32-, 64- **or** 128-byte memory transactions
- Access **must** be naturally aligned 32, 64, 128 aligned respectively
- When a warp executes an instruction that accesses global memory
 - It coalesces the accesses of the individual threads of the warp
 - into one or more memory transactions, depending on
 - size of word accessed by each thread
 - distribution of the memory addresses across threads
 - **the larger the number of transactions required**
 - more unused words -- **poorer instruction throughput**
 - Number of transactions and impact is device (compute capability dependent, see compute capability doc in PG for your device)

Device Memory: Global Memory III

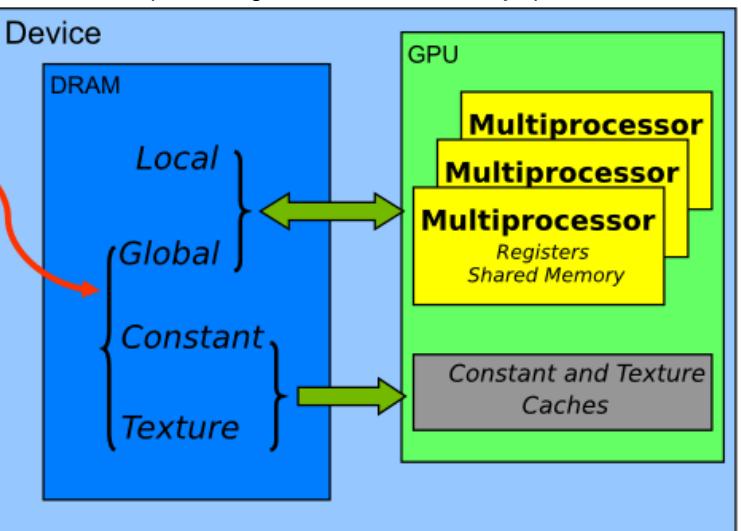
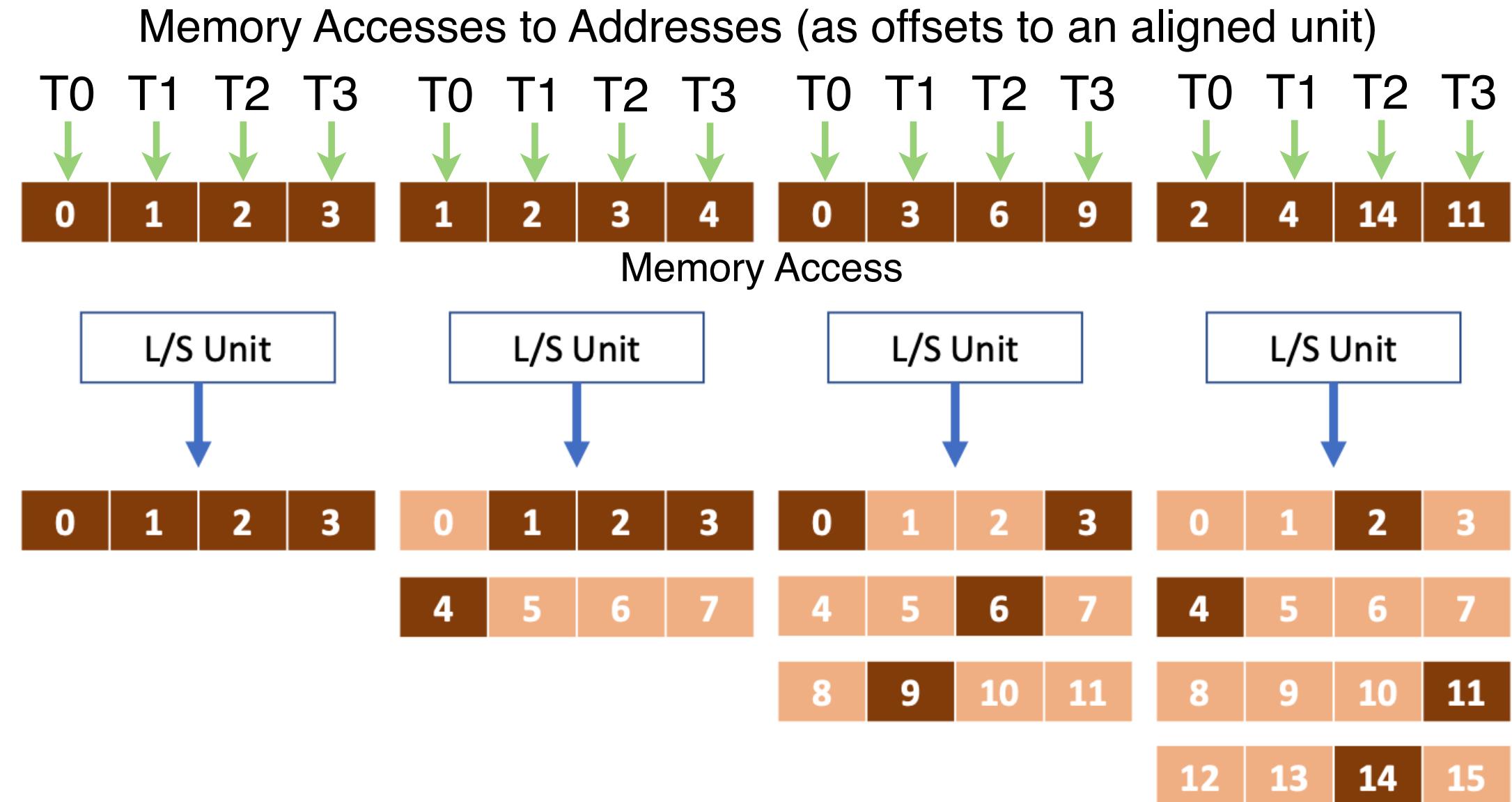


- CRITICAL OPTIMIZATION: Maximize coalescing
 1. Follow optimal access patterns based on Compute Capability (>6) is the same and we will cover now)
 2. Use a data type that meets the size and alignment requirements
 3. Padding data in some case can improve 2D array accesses

Device Memory: Global Memory IV

- Coalesced Access to Global Memory

A synthetic 4 threads per warp example



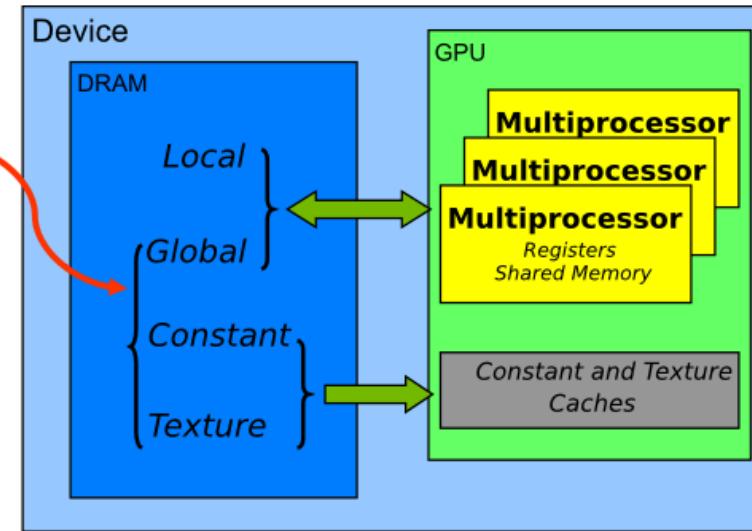
→ Your job as a programmer is to ensure most mem accesses from a warp are coalesced

Device Memory: Global Memory V

- Size and Alignment Requirements
 - coalescing of global memory access will only occur if access is with a single memory instruction (eg, ld, store of supported size) and aligned
 - supported sizes: 1, 2, 4, 8, 16 bytes (with natural alignment)
 - if not multiple instructions and less effective coalescing
- `__global__` variables and allocations are always at least 256 byte aligned
- use alignment specifiers to enforce

```
struct __align__(8) {  
    float x;  
    float y;  
};
```

```
struct __align__(16) {  
    float x;  
    float y;  
    float z;  
};
```



"Reading non-naturally aligned 8-byte or 16-byte words produces incorrect results (off by a few words)"

Device Memory: Global Memory VI

- Padding to ensure alignment is useful to improve array access
 - Eg, in a 2D array, you want both Thread Block and array width to be a multiple of warp size
 - Padding rows with extra bytes to ensure this is true will improve coalescing

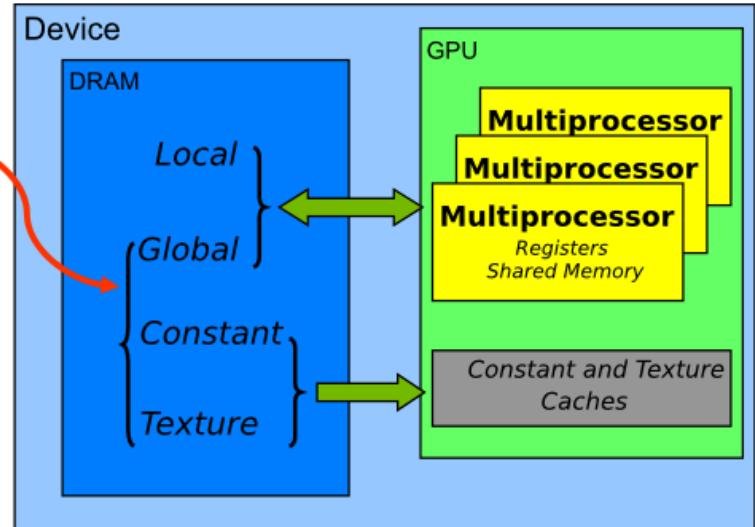
```
float M[rows][columns]; float v = M[ty][tx];
```

Base of Array
aligned for
coalesced
access

BaseAddress + width * ty + tx

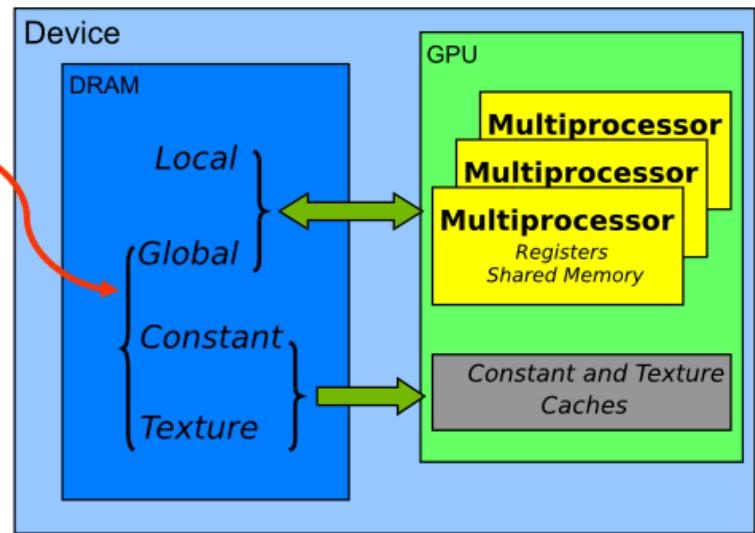
Width of array (number of columns where each element is of a type that meets the coalescing size requirements)

pad width to ensure that each row starts at a warp aligned size



<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#constant-memory>

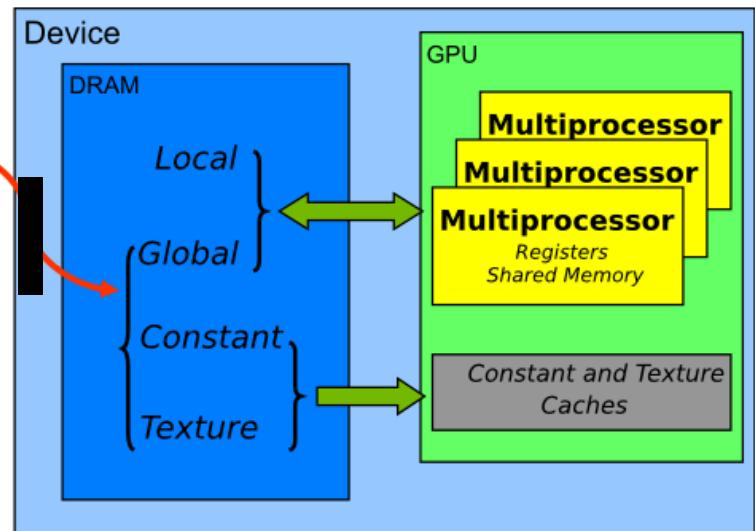
Device Memory: Constant Memory



- Device memory Device READ ONLY : slow but cached
- Size limited 64K
- Cached
 - best when all threads access the same data item
 - then fast -- like Shared Memory
 - but does not tie up a register -- can be useful to reduce register pressure
- Only static allocation
 - `__constant__ <type> <varname>`
- Read/Write by CPU:
 - `cudaMemcpyToSymbol("var", *h_var, sizeof(var_type))`
 - `cudaMemcpyFromSymbol(...)`
 - or with standard `cudaMemcpy()` in combination with `cudaGetSymbolAddress(...)`

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#constant-memory>

Device Memory: Constant Memory II



- Constant memory var values set at run-time
 - Different than standard constants
 - values known at compile time (get baked into instruction stream)
 - `#define PI 3.1415926f`
 - `a = b / (2.0f * PI);`
 - Leave these as they are, they generally won't add to register pressure



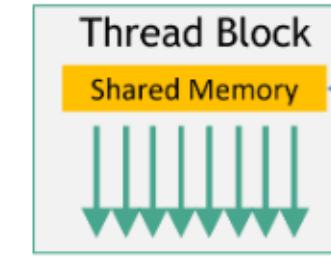
Shared Memory

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#shared-memory>

**Small but critical to bridging registers
and device memory**

- 1. On-chip" Fast & Shared**
- 2. But Banked!**

Shared Memory



Per block Shared memory

- Fast on-chip local to SM
- used for
 - operations requiring communications between threads within a block (recently CC adds support for distributed access)
 - data reuse
 - staging memory to increase degree of coalescing factor
- Limited in size : Varies with device (V100 96K, A100 164K, H100 256K)
- allocated on a per-block basis (can create occupancy constraints like registers)
- static allocation
 - `__shared__ <type> <varname>`
- dynamic allocation
 - `kernel<<<blocks, threads, shared_bytes>>(...)`
 - See programmer's guide for details on how to access

BEWARE of bank conflicts!

Shared Memory: Bank Conflict 1

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks.

Linear addressing with a stride of one 32-bit word (no bank conflict).

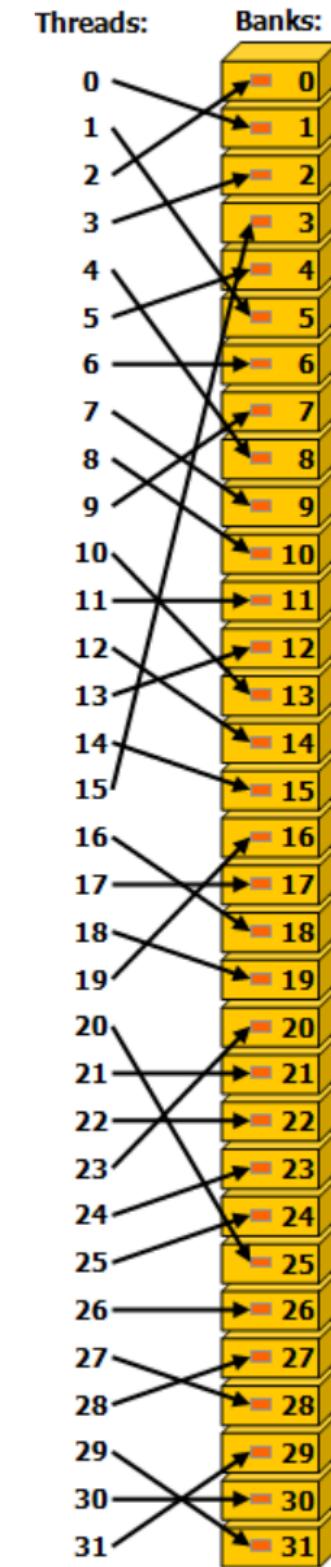
Linear addressing with a stride of two 32-bit words (2 way bank conflict)

Linear addressing with a stride of three 32-bit words (**no bank conflict**)

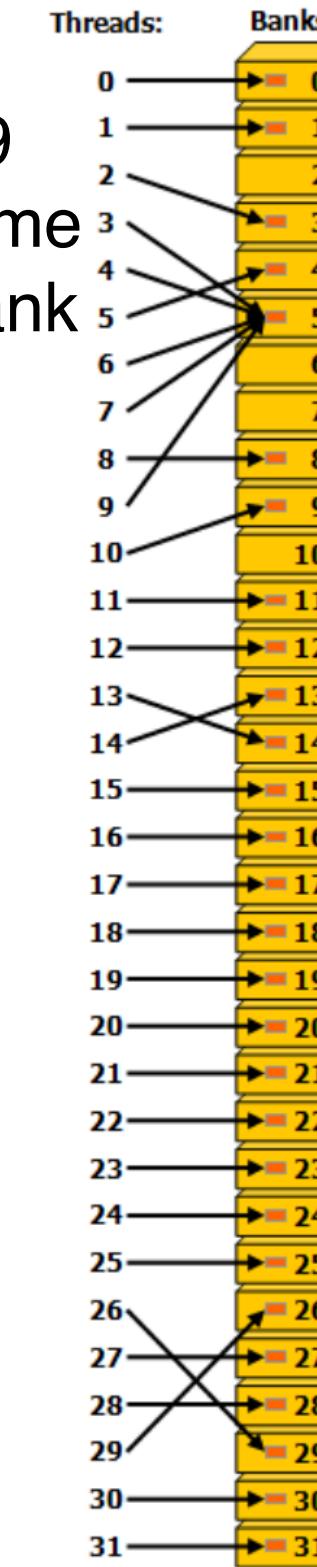
Shared Memory: Bank Conflict 2

A shared memory request for a warp does not generate a **bank conflict** between two threads that access any address within the **same 32-bit word** (even though the two addresses fall in the same bank). In that case, for read accesses, the word is broadcast to the requesting threads

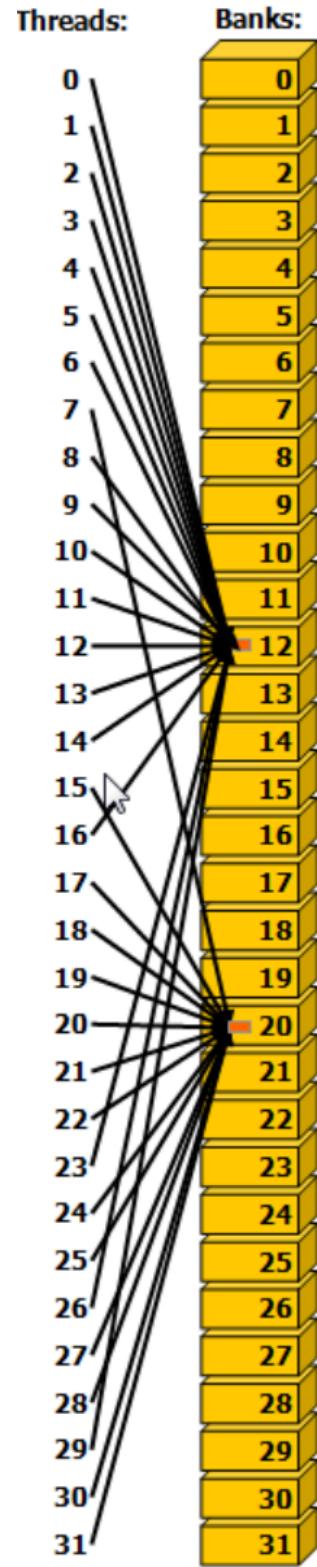
Conflict-free
access via
random
permutation



Conflict-free
since 3,4,6,7,9
access the same
word within bank



Conflict-free
broadcast access
(threads access
the same word
within a bank)



Shared Memory: Synchronization

- When using shared memory to share data between threads of a block, synchronization is needed (no control of order of warp execution and out-of-order memory ops)
`__syncthreads()` (and friends see manual also see `__syncwarp` for fancy warp level sync)
 - block-level synchronization barrier (recently added cluster synchronization, also now an `async-barrier` version)
 - each thread, when it reaches the statement, blocks until
 - all other threads have reached it as well
 - AND all global and shared memory written by the threads are visible to all threads (includes a memory fence)
- Note: threads in different blocks **can't** synchronize!

Q: What do you do if you want to sync threads across blocks?

Shared Memory: Synchronization

- With `__syncthreads()` watch out:
 - make sure all threads can reach `__syncthreads()`
 - otherwise: deadlock!
- The following code is a no-no:

```
if ( ... ) {  
    ...  
    __syncthreads() ;  
    ...  
} else {  
    ...  
    __syncthreads() ;  
    ...  
}
```

The `__syncthreads*` intrinsics are permitted in conditional code, but only if the condition evaluates uniformly across the entire thread block. Otherwise, execution may hang or produce unintended side effects.

Memories Summary:

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	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 in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.					
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.					

Matrix Transpose Example

Using Shared memory to optimize accesses but...

This is way more subtle than first meets the eye (but tracing some historical work teaches you a lot):

1. 2010 "Optimizing Matrix Transpose in CUDA"

<https://developer.download.nvidia.com/assets/cuda/files/MatrixTranspose.pdf>

2. 2013 "An Efficient Matrix Transpose in CUDA C/C++" no partition camping

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory-in-matrix-multiplication-c-ab>

<https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpptranspose/transpose.cu>

3. 2024 "Tutorial: Matrix Tranpose in CUTLASS", includes initial look at TMA

<https://research.colfax-intl.com/tutorial-matrix-transpose-in-cutlass/>

Of course there is "stuff" on the net including more recent claims but I have not read or validated"

Matrix Transpose Example

Using
This
histo
1. 20
2. 20
camp
3. 202
at TMA

This has become such an important example for helping a developer understand memory optimization strategies that, as of CUDA 13.1, it is covered in the programming guide. But they make it seem obvious and easy. Reading about the effort that went into discovering and optimizing Matrix Transposition is still very helpful. Here are the links to the coverage in the Programming Guide

Read this first and then look at other references

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#matrix-transpose-example-using-global-memory>

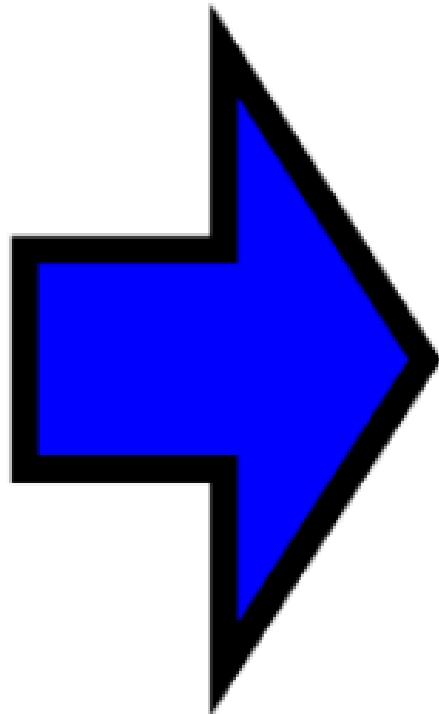
<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#matrix-transpose-example-using-shared-memory>

on
ultiplication-c-ab
ranspose.cu

Of course there is "stuff" on the net including more recent claims but I have not read or validated"

Example: Transpose: the goal

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Example: Transpose: the naive** code

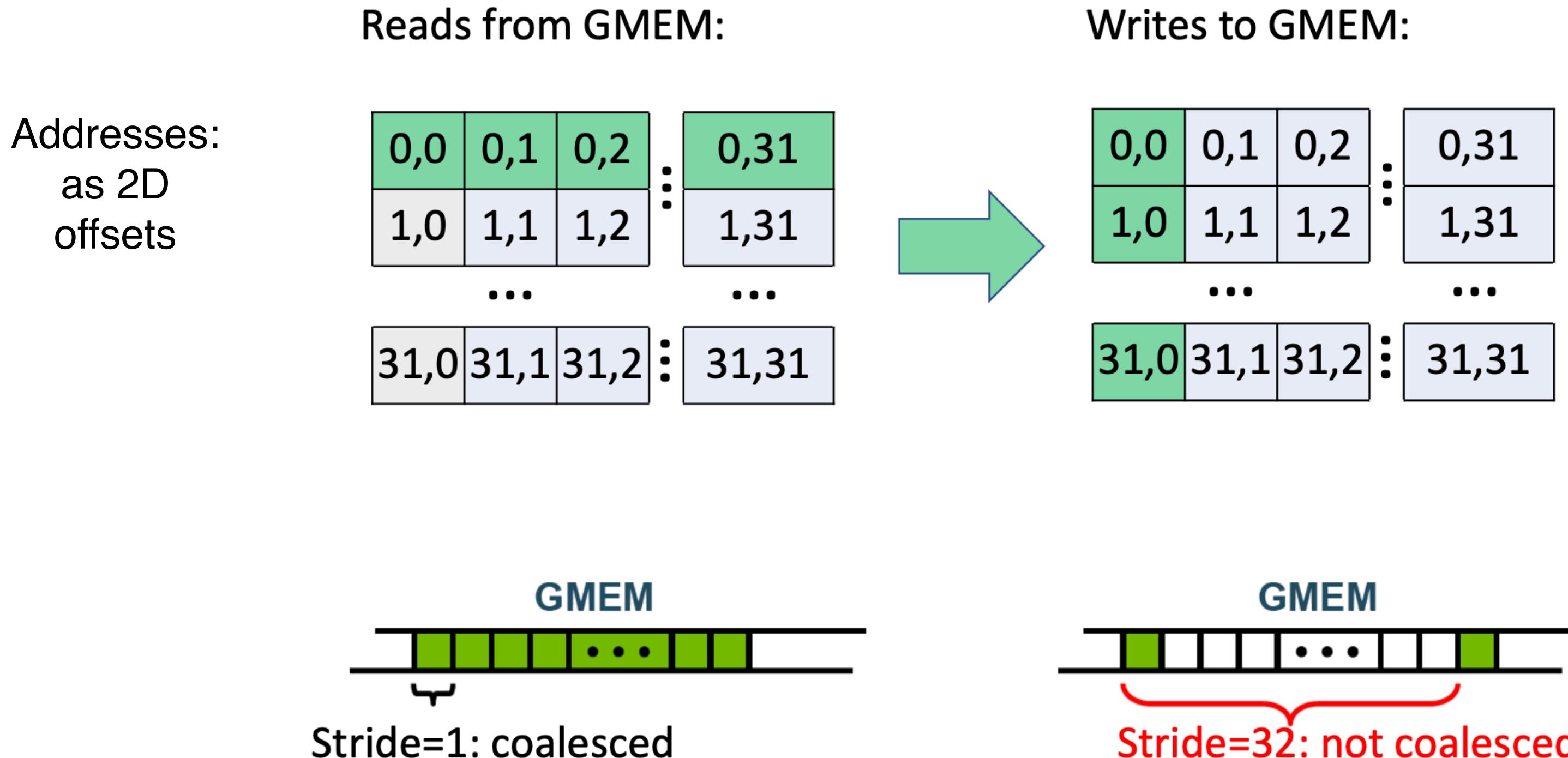
```
__global__ void transpose_naive(float *odata, float *idata,
                                int width, int height) {
    unsigned int xIndex = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int yIndex = threadIdx.y + blockIdx.y * blockDim.y;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

This is a non-tiled naive version and does one element per-thread -- different than the starting point in the NVIDIA blog and study but similar to Programming Guide

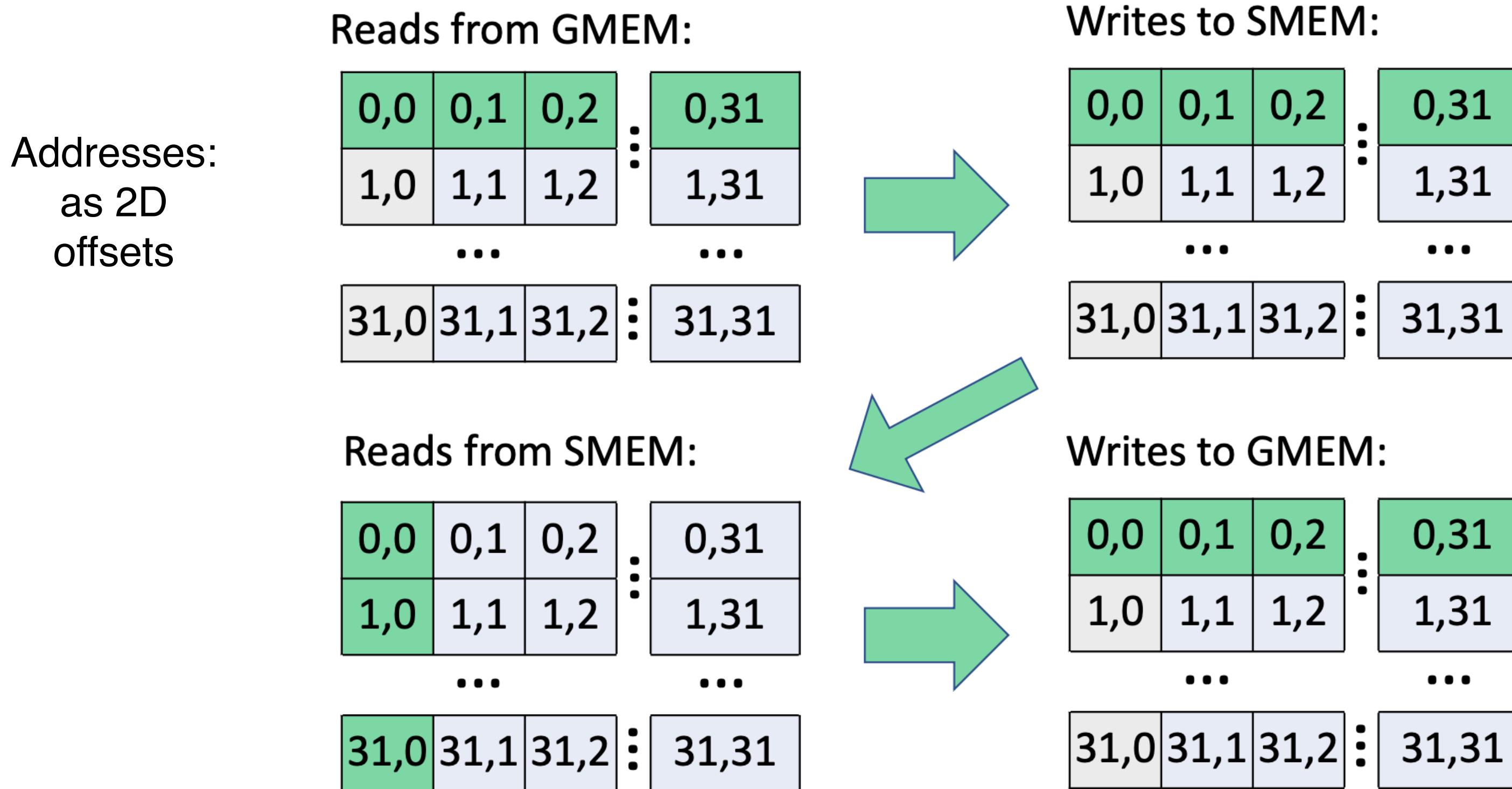
Example: Transpose: the problem

Naïve approach leads to uncoalesced global mem accesses:



Example: Transpose: the idea add SM

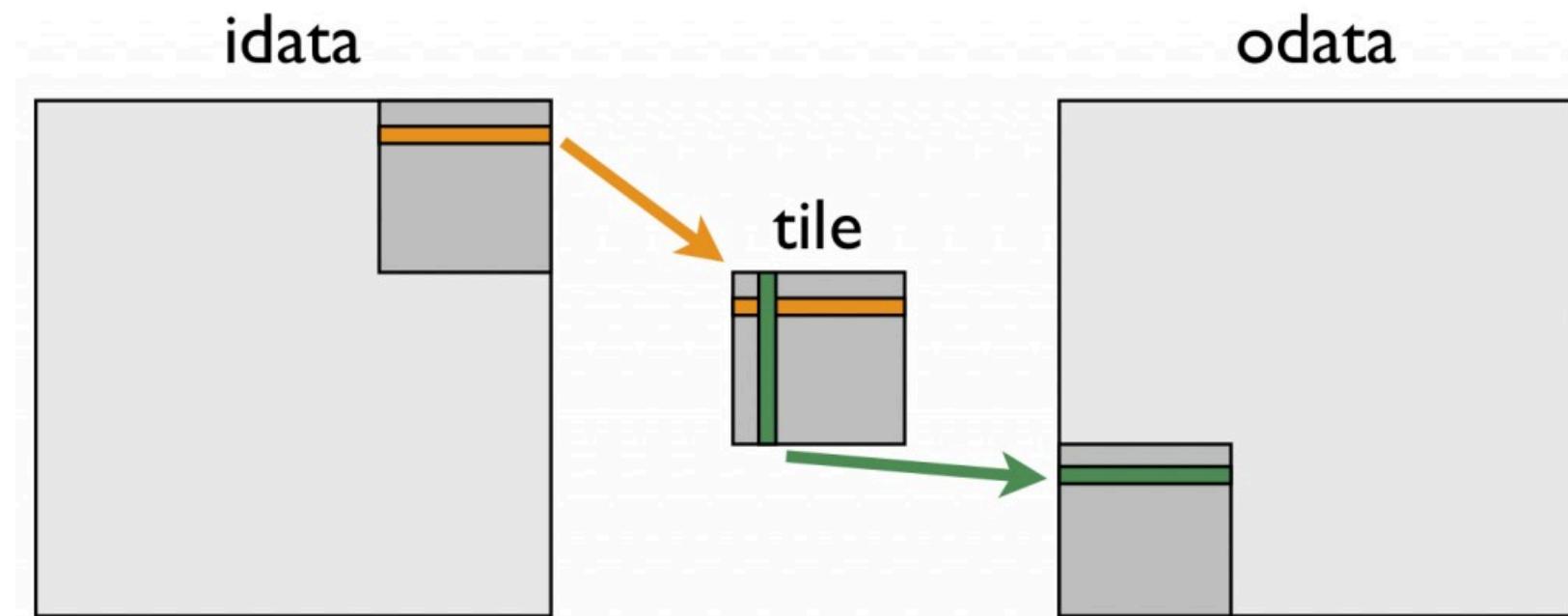
Idea: stage through shared memory:



Example: Transpose: SM + Tiling

Basic idea: conceptually partition matrix into square tiles

- Thread block (bx,by):
 - Read the (bx,by) input tile, store into SMEM
 - Write the SMEM data to (by,bx)
 - Transposing the indexing into SMEM
- Thread (tx,ty)
 - Reads element(s) (tx,ty) from input tile (possibly multiple if per-thread blocking factor used)
 - synchronization required
 - Writes element(s) into output tile
- Coalescing is achieved if:
 - Tile dimensions are multiples of 32



Example: Transpose: SM Bank Conflicts

New Problem: shared memory bank conflicts:

Reads from SMEM:

0,0	0,1	0,2	:	0,31
1,0	1,1	1,2	:	1,31
...			...	
31,0	31,1	31,2	:	31,31

Threads read SMEM with stride 32

- 32x32-way bank conflicts
- 32x slower than no conflicts

How to fix???

Example: Transpose: Add Padding

Solution: add an extra column (not used):

- Read stride = 33
 - Threads read from consecutive banks
 - No bank conflicts

Reads from SMEM:

