Optimizing for CUDA II

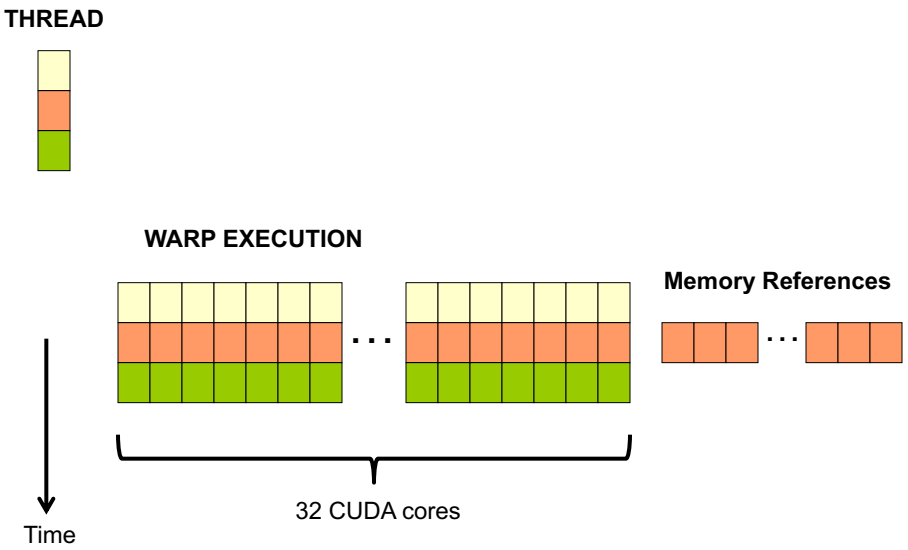
Control Divergence, Memory Coalescing, Tiling

Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green, Mark Haris and many others
Real World Techonologies by David Kanter

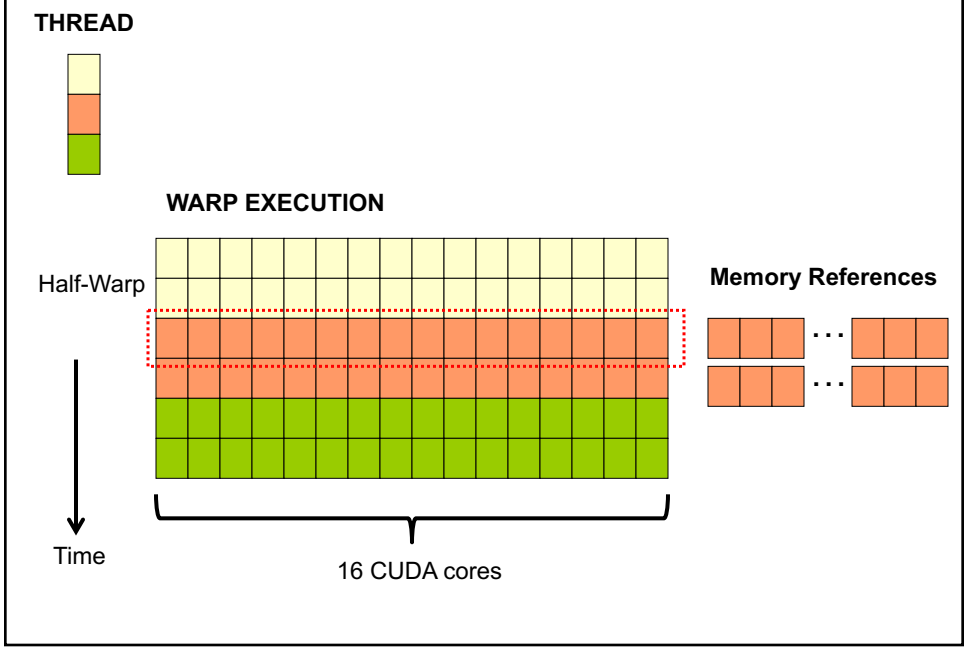
1

WARP (capability 3.0)



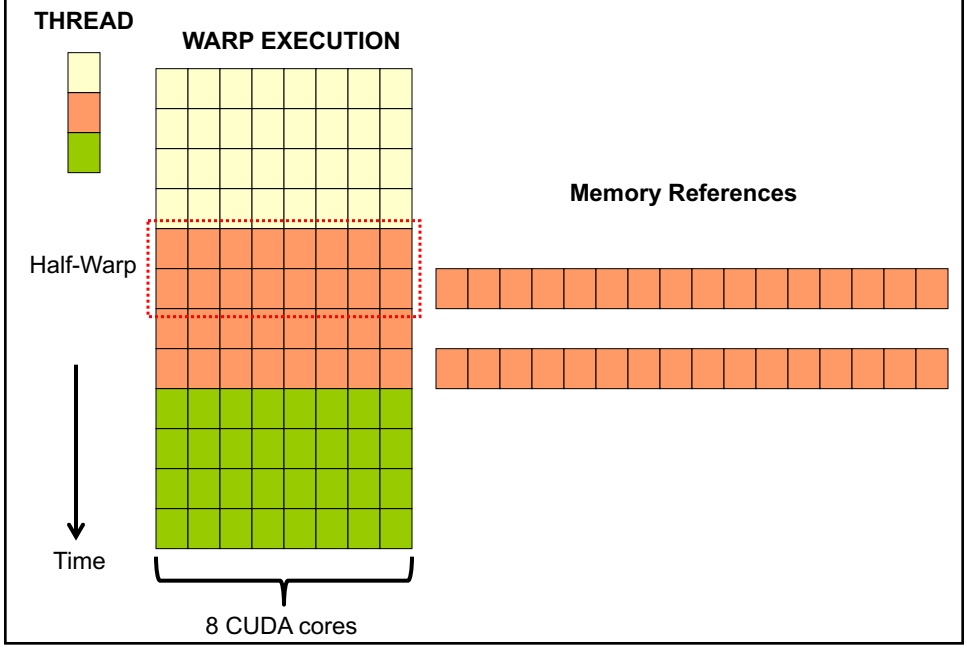
2

WARP (capability 2.0)



3

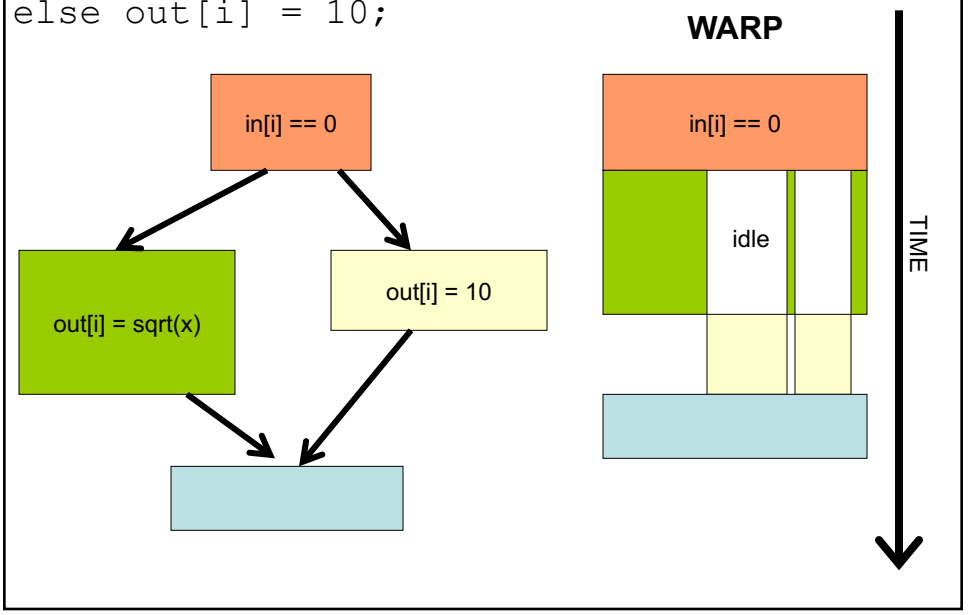
WARP (capability 1.3)



4

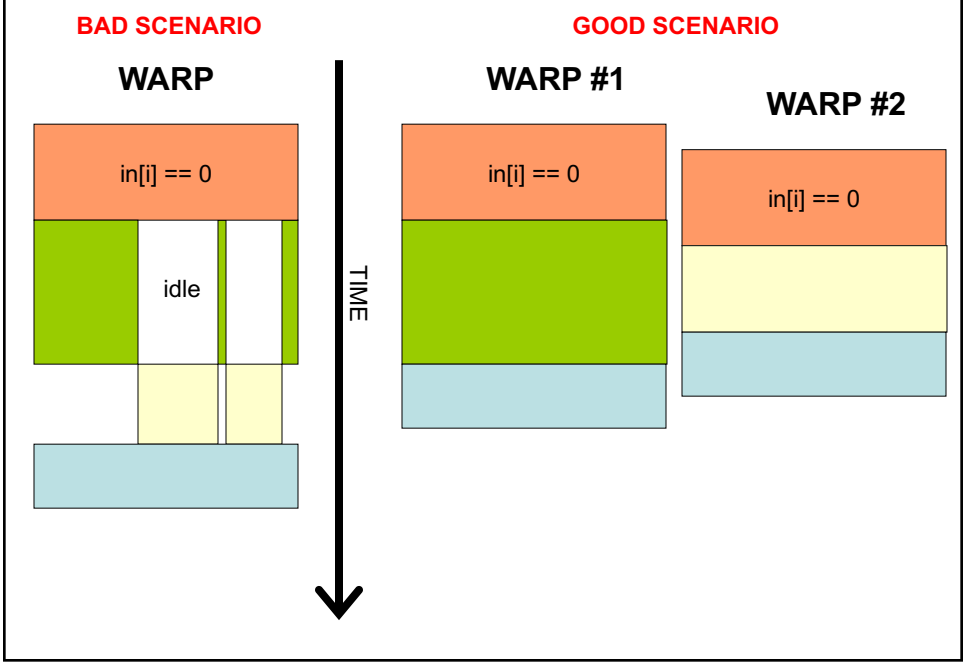
Understanding control flow divergence

```
if (in[i] == 0) out[i] = sqrt(x);
else out[i] = 10;
```



5

Control flow divergence (contd.)



6

Instruction performance

- Instruction processing steps per warp:
  - Read input operands for all threads
  - Execute the instruction
  - Write the results back
- For performance:
  - Minimize use of low throughput instructions
  - Maximize use of available memory bandwidth
  - Allow overlapping of memory accesses and computation
    - High compute/access ratio
    - Many threads (TLP)
    - Independent instructions (ILP)

7

Throughput of native arithmetic operations (I)

(Operations per Clock Cycle per Multiprocessor)

	Compute Capability					
	1.0 1.1 1.2	1.3	2.0	2.1	3.0	3.5
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192	192
64-bit floating-point add, multiply, multiply-add	1	1	16(*)	4	8	64
32-bit integer add	10	10	32	48	160	160
32-bit integer compare	10	10	32	48	160	160
32-bit integer shift	8	8	16	16	32	64
Logical operations	8	8	32	48	160	160
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	Multiple instructions	16	16	32	32
24-bit integer multiply (`__[u]mul24`)	8	8	Multiple instructions	Multiple instructions	Multiple instructions	Multiple instructions
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	2	2	4	8	32	32

8

Throughput of native arithmetic operations (II)

	Compute Capability					
	1.0 1.1 1.2	1.3	2.0	2.1	3.0	3.5
Type conversions from 8-bit and 16-bit integer to 32-bit types	8	8	16	16	128	128
Type conversions from and to 64-bit types	Multiple instructions	1	16(*)	4	8	32
All other type conversions	8	8	16	16	32	32
(*) Throughput is lower for GeForce GPUs.						

9

Optimization steps

1. Optimize Algorithms for the GPU
2. Optimize Memory Access Ordering for Coalescing
3. Take Advantage of On-Chip Shared Memory
4. Use Parallelism Efficiently

10

Optimize algorithms for the GPU

- **Maximize independent parallelism**
  - We'll see more of this with examples
  - Avoid thread synchronization as much as possible
- **Maximize arithmetic intensity (math/bandwidth)**
  - Sometimes it's better to re-compute than to cache
  - GPU spends its transistors on ALUs, not memory
  - Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host
  - Device-optimized implementations can be faster than regular ones
  - Provided as overloaded functions, nvcc option --use_fast_math
  - e.g., float divide __fdividef(a, b) is faster than a / b
  - …may not be IEEE 754 compliant, so check first if you care

11

Optimize memory access ordering for coalescing

- **Coalesced Accesses:**
  - Goal: a single access for all requests in a warp
  - One access for 4-byte accesses per thread
    - 128 contiguous bytes, i.e., one L2 cache block
    - This is the granularity of a Global Memory access
      - Even if you ask for 1 byte, you will get a 128-byte block
  - Two accesses for 8-byte accesses per thread
    - Independent accesses issued for half-warps
    - Each access: 128 contiguous bytes, i.e., one L2 cache block
- **Coalesced vs. Non-coalesced**
  - Global device memory
    - Order of magnitude performance difference
  - Shared memory
    - Coalescing plays no real role here
    - Optimization goal: avoid bank conflicts

12

Exploit the shared memory

- **Hundreds of times faster than global memory**
  - 1-2 cycles vs. 400-600 cycles
- **Threads can cooperate via shared memory**
  - __syncthreads ()
  - All threads in a block must reach this instruction before they are allowed to proceed
- **Use one / a few threads to load / compute data shared by all threads**
- **Use it to avoid non-coalesced access**
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later

13

Use parallelism efficiently

- **Partition your computation to keep the GPU multiprocessors equally busy**
  - Many threads, many thread blocks
- **Keep resource usage low enough to support multiple active thread blocks per multiprocessor**
  - Registers, shared memory

14

Global memory reads/writes

- **Highest latency instructions: 400-600 clock cycles**
- **Likely to be performance bottleneck**
  - Optimizations can greatly increase performance
  - Coalescing: up to 32x speedup
  - Latency hiding: up to 2.5x speedup

15

Reminder: How to get high performance #2

- Coalesce global memory accesses
  - 32 threads access memory together
  - Can coalesce into **single** reference, if addresses within a 128-byte block
    - Instead of issuing 32 memory accesses of 4 bytes each
    - Issue 1 memory access that is 128 bytes wide (load granularity)
    - GDDR5 provides 32 bytes/mem_clock → 128 bytes in 4 Mem cycles
    - But GDDR5 @ 6.008GHz, while cores @ 1.006GHz
      → one core cycle is 5+ memory cycles
      → get 128 bytes in 1 core cycle
  - Coalesce requests to get one access as wide as possible
    - e.g., `a[threadID]` works well
  - Ideal: 1 warp → 128 bytes of consecutive memory
    - Aligned to 128-byte boundary…

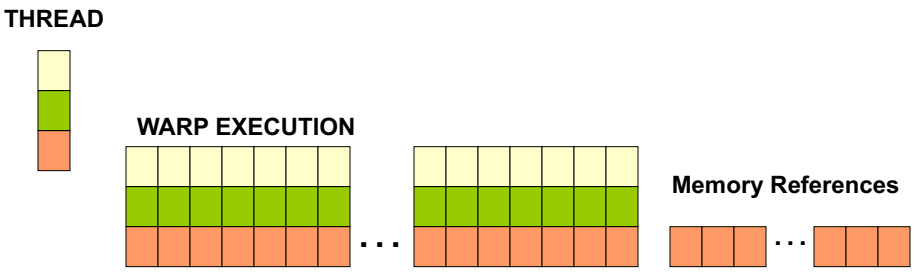
16

Coalescing

- **A coordinated read by a warp**
  - Becomes a single wide memory read
- **All accesses must fall into a continuous region of :**
  - 32 bytes – each thread reads a byte:
    - char
  - 64 bytes – each thread reads a half-word:
    - short
  - 128 bytes – each thread reads a word:
    - int, float, …
  - 256 bytes – each thread reads a double-word
    - double, int2, float2, …
  - 512 bytes – each thread reads a quad-word
    - int4, float4, …
  - Additional restrictions on G8X architecture (older gen):
    - Starting address for a region must be a multiple of region size
    - The *k*th thread in a warp must access the *k*th element in a block being read
  - Exception: not all threads must be participating
    - Predicated access, divergence within a warp

17

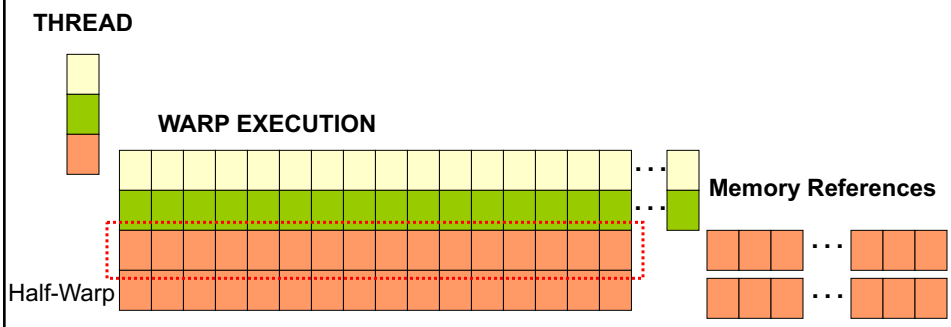
Coalescing (4-bytes/thread or less)



- Addresses must all fall into the same 128-byte region
  - each thread reads a byte: char
  - each thread reads a half-word: short
  - each thread reads a word: int, float, …

18

Coalescing (8-bytes/thread)

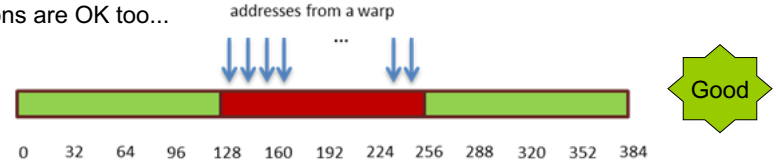


- Addresses must all fall into the same 128-byte region for each half warp
  - each thread reads a double-word (double, int2, float2,…)

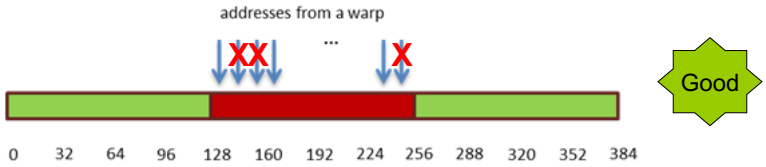
19

Coalesced access: reading floats

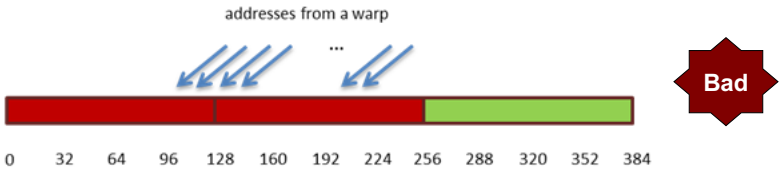
All threads participate, accesses fit within a region of 128 bytes.
Permutations are OK too...



Not all threads participate, accesses fit within a region of 128 bytes



Misaligned accesses (fit in two regions of 128 bytes):



20

Coalescing experiment code

```
for (int i = 0; i < TIMES; i++) {
    cutResetTimer(timer);
    cutStartTimer(timer);
    kernel <<<n_blocks, block_size>>> (a_d);
    cudaThreadSynchronize ();
    cutStopTimer (timer);
    total_time += cutGetTimerValue (timer);
  }
  printf ("Time %f\n", total_time / TIMES);


__global__ void kernel (float *a)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  a[i]++;

}
```

21

Coalescing experiment code

```
for (int i = 0; i < TIMES; i++) {
    cutResetTimer(timer); cutStartTimer(timer);
    kernel <<<n_blocks, block_size>>> (a_d);
    cudaThreadSynchronize (); cutStopTimer (timer);
    total_time += cutGetTimerValue (timer);
  }
  printf ("Time %f\n", total_time / TIMES);


__global__ void kernel (float *a)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
```

a[i]++;	**211 µs**
if ((i & 0x3) != 00) a[i]++;	**212 µs**
if ((i & 0x3) != 00) a[i]++; **else a[0]++;**	**5,182 µs**
if (i & 0x3) != 00) a[i]++; **else a[blockIdx.x * blockDim.x]++;**	**785 µs**

```
}
```

22

Coalescing experiment

- Kernel:
  - Read float, increment, write back: a[i]++;
  - 3M floats (12MB)
  - Times averaged over 10K runs

- 12K blocks x 256 threads/block
  - Coalesced (**a[i]++**)
    - **211 µs**
  - Coalesced / some don't participate (**if (i & 0x3 != 0) a[i]++**)
    - 3 out of 4 participate
    - **212 µs**
  - Uncoalesced / outside the region
    - Every 4 access a[0]
    - **5,182 µs** → 24.4x slowdown: 4x from not coalescing and another 8x from contention for a[0] (25% of all threads of all blocks access a[0])
      - **if (i & 0x3 != 0) a[i]++;**
        else a[0]++;
    - **785 µs** → 4x slowdown: mostly from not coalescing (only threads in block access a[startOfBlock])
      - **If (i & 0x3) != 0) a[i]++;**
        else a[startOfBlock]++;

23

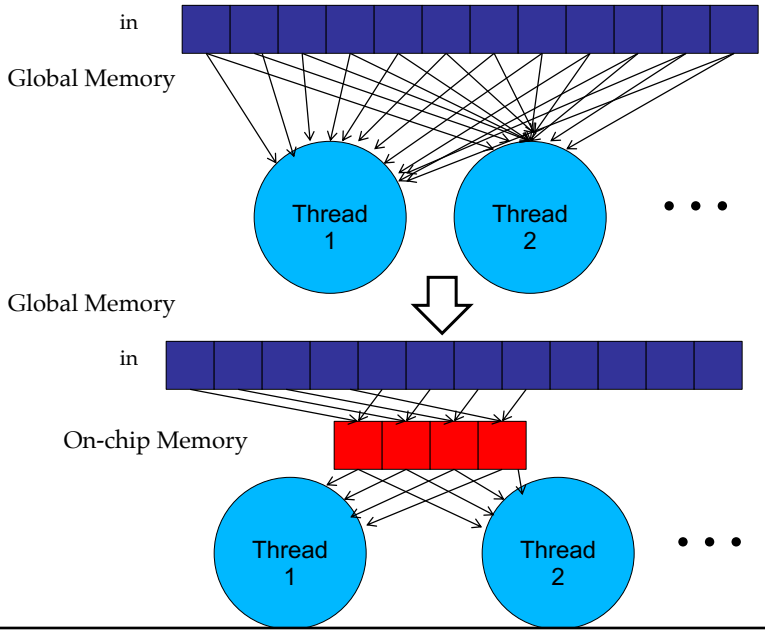
A common programming strategy

- Global memory resides in device memory (DRAM)
  - slow access

- A profitable way of performing computation on the device is: tile the input data to take advantage of fast shared memory
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

24

Shared memory blocking (or tiling) main idea



25

Basic concept of blocking/tiling

- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Blocking/Tiling for global memory accesses
    - drivers = threads
    - cars = memory requests

26

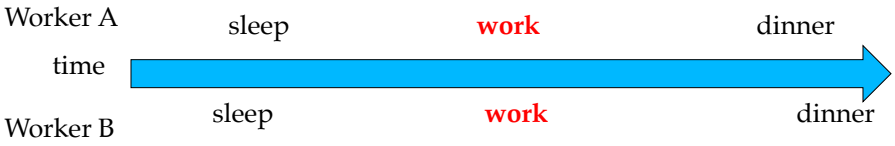
Some computations are more challenging

- Some carpools may be easier than others
  - Your co-rider should want to go where you want to go
  - Some vehicles may be more suitable for carpooling than others
- Similar variations exist in blocking/tiling

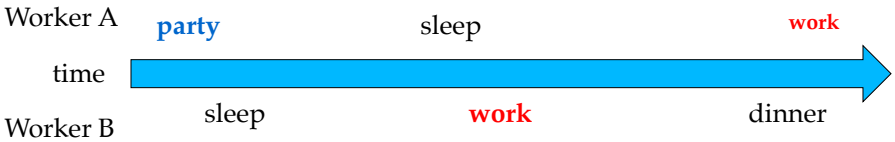
27

Carpools need synchronization

- Good – when people have similar schedule



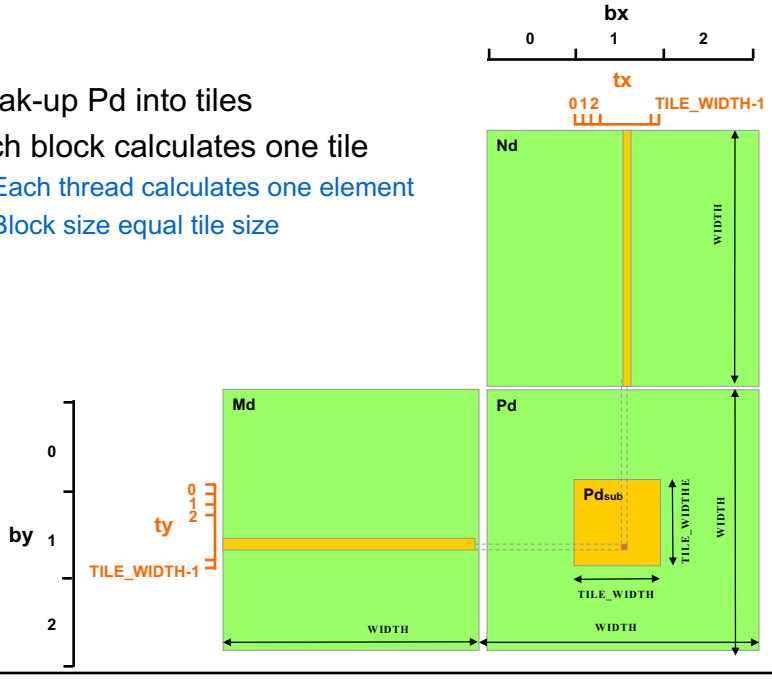
- Bad – when people have very different schedule



28

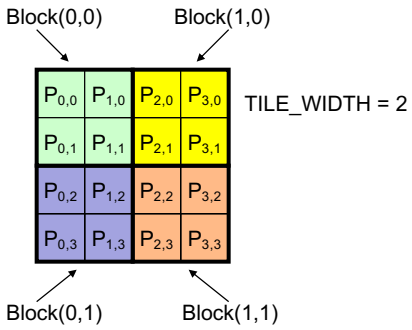
Matrix multiplication using multiple blocks

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal tile size



29

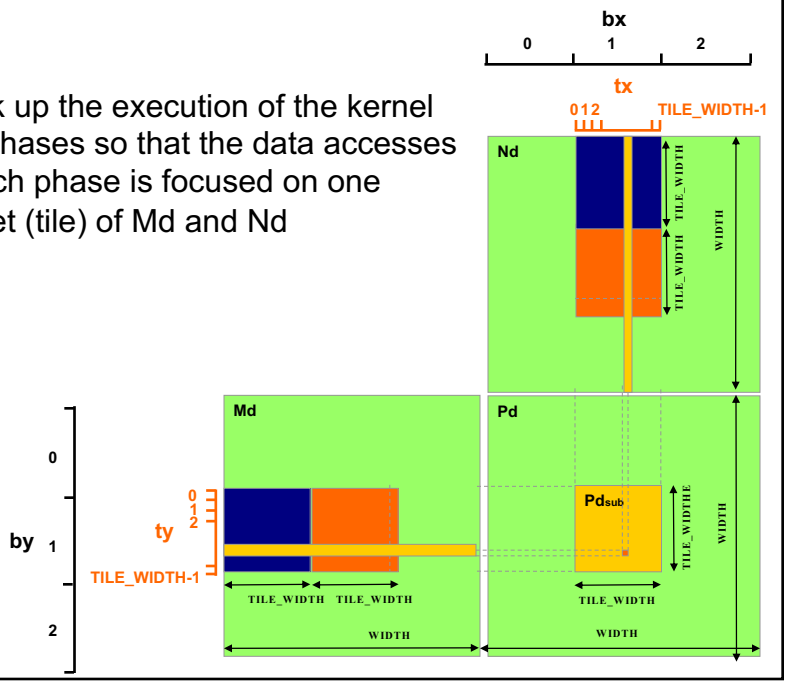
A small example



30

Tiled multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



31

Matrix multiplication kernel using multiple blocks

```
__global__ void
MatrixMulKernel(float* Md,
                float* Nd,
                float* Pd,
                int Width)
{
   // Calculate the row index of the Pd element and M
   int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;

   // Calculate the column idenx of Pd and N
   int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

   float Pvalue = 0;
   // each thread computes one element
   // of the block sub-matrix

   for (int k = 0; k < Width; ++k) {
      Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
   }
   Pd[Row * Width + Col] = Pvalue;
}
```

32

Uncoalesced float3 access code

```
__global__ void
accessFloat3(float3 *d_in, float3 d_out)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  float3 a = d_in[index];

  a.x += 2;
  a.y += 2;
  a.z += 2;

  d_out[index] = a;
}
```

This will be implemented as:

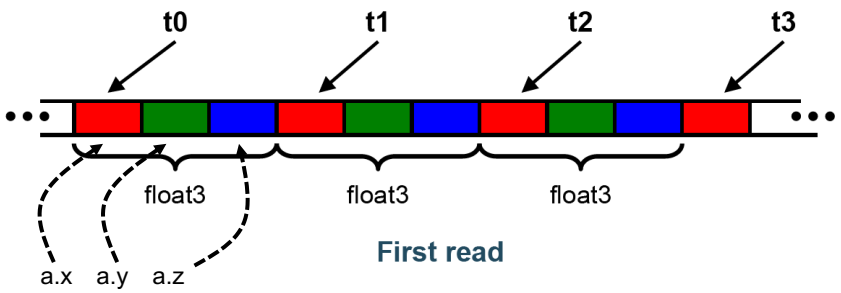
```
ld d_in[index].x
ld d_in[index].y
ld d_in[index].z
```

Execution time: 1,905 µs
12M float3
Averaged over 10K runs

33

Naïve float3 access sequence

- float3 is 12 bytes
  - Each thread ends up executing 3 32bit reads
  - sizeof(float3) = 12
  - Offsets:
    - 0, 12, 24, …, 180 (for loading the .x part)
    - 4, 16, 28, …, 184 (for loading the .y part)
    - 8, 20, 32, …, 188 (for loading the .z part)
  - Warp reads three 128-byte non-contiguous regions each time
    - 12 bytes/thread * 32 threads = 384 bytes = 3 * 128 bytes



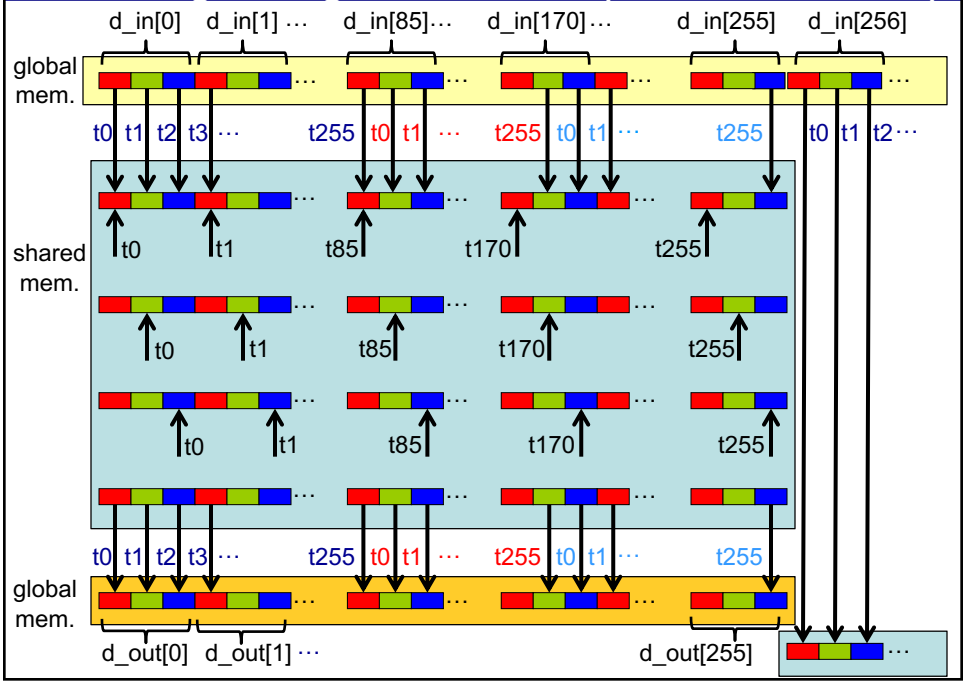
34

Coalescing + tiling float3 strategy

- **Use shared memory to allow coalescing**
  - Need sizeof(float3)*(threads/block) bytes of SMEM
- **Three Phases:**
  - Phase 1: Fetch data in shared memory
    - Each thread reads 3 scalar floats
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync
  - Phase 2: Processing
    - Each thread retrieves its float3 from SMEM array
    - Cast the SMEM pointer to (float3*)
    - Use thread ID as index
    - **Rest of the compute code does not change**
  - Phase 3: Write results back to global memory
    - Each thread writes 3 scalar floats
    - Offsets: 0, (threads/block), 2*(threads/block)

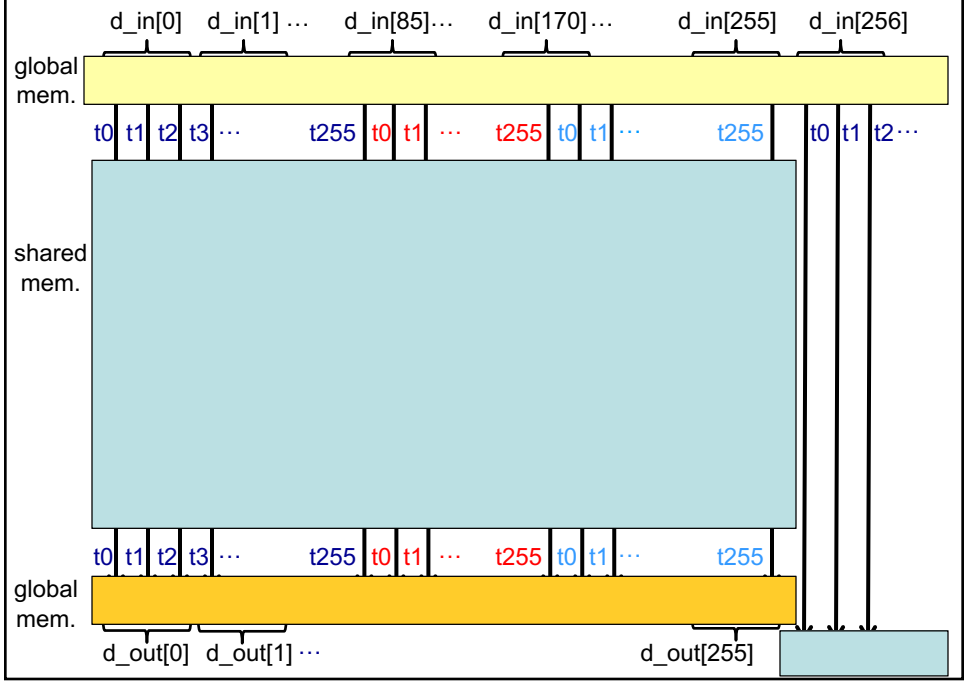
35

Coalescing + tiling float3 access (256 threads/block)



36

Coalescing + tiling float3 access (256 threads/block)



37

Coalescing + tiling float3 access (256 threads/block)

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ float s_data[256*3];
  s_data[threadIdx.x]       = g_in[index];
  s_data[threadIdx.x+256] = g_in[index+256];
  s_data[threadIdx.x+512] = g_in[index+512];
  __syncthreads();
  float3 a = ((float3*)s_data)[threadIdx.x];

  a.x += 2;
  a.y += 2;
  a.z += 2;

  ((float3*)s_data)[threadIdx.x] = a;
  __syncthreads();
  g_out[index]       = s_data[threadIdx.x];
  g_out[index+256] = s_data[threadIdx.x+256];
  g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

Compute code is not changed

Write the result through SMEM

38

Experiment: float3

- **Experiment:**
  - Kernel: read a float3, increment each element, write back
  - 1M float3s (12MB)
  - Times averaged over 10K runs
- **Results:**
  - 4K blocks x 256 threads:
    - 648µs – float3 uncoalesced
    - 245µs – float3 coalesced through shared memory

39

Global memory coalesing summary

- Coalescing greatly improves throughput
- Critical to small or memory-bound kernels
  - Accessing structs (or non-fundamental types, like float3) in global memory may break coalescing
  - Prefer Structures of Arrays (SoA) over Arrays of Structures (AoS)

```
struct {
    typeA a;
    typeB b;
    typeC c;
} array_of_structs[N];

struct {
    typeA a[N];
    typeB b[N];
    typeC c[N];
} struct_of_arrays;
```

  - If SoA is not viable, read/write through SMEM

40