Scheduling Issues in CUDA
Global Memory Access Patterns

October 4, 2013

© Dan Negrut, 2013
ECE/ME/EMA/CS 759 UW-Madison

Computers are useless. They can only give you answers.
Pablo Picasso
Before We Get Started…

- Last time
  - Example: tiled matrix multiplication → introduced the concept of Shared Memory
  - Execution scheduling

- Today
  - Wrap up discussion on execution scheduling
  - Discuss global memory access patterns

- Miscellaneous
  - Fourth assignment due on Monday, October 7, at 11:59 PM
    - GPU computing related
    - Kicks off a series of four challenging assignments
  - Read pages 73 through its end: [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
    - Please post suggestions for improvement
  - Half way through the semester: I’m asking for your feedback
    - To be provided anonymous on Wednesday, Oct 9 – details to follow in an email on Monday
    - I’ll compile all of them and upload on the class website
  - Syllabus updated on the course website
## Technical Specifications and Features

[Short Detour]

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**

“multiprocessor” stands for Stream Multiprocessor (what we called SM)

---

**Feature Support**
(Unlisted features are supported for all compute capabilities)

<table>
<thead>
<tr>
<th>Feature</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer atomic functions operating on 32-bit words in global memory</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>(Section B.11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer atomic functions operating on 64-bit words in global memory</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Section B.11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer atomic functions operating on 32-bit words in shared memory</td>
<td>No</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>(Section B.11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp vote functions (Section B.12)</td>
<td>No</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>_ballot() (Section B.12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_thread_fence_system() (Section B.5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_syncthreads_count(), _syncthreads_and(), _syncthreads_or() (Section B.6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surface functions (Section B.9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Threads are Executed in Warps

- Each thread block split into one or more warps
- When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
- Warps within a thread block can execute independently
Organizing Threads into Warps

- Thread IDs within a warp are consecutive and increasing
  - This goes back to the 1D projection from thread index to thread ID
  - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.

- Partitioning of threads in warps is always the same
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

- While you can rely on ordering among threads, DO NOT rely on any ordering among warps since there is no such thing
  - Warp scheduling is not something you control in CUDA
Thread and Warp Scheduling

- An SM can switch between warps with no apparent overhead
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling
- When a warp is selected for execution, all [active] threads execute the same instruction in lockstep fashion
Revisiting the Concept of Execution Configuration

- Prefer thread block sizes that result in mostly full warps

**Bad:**  \texttt{kernel}^{<<<N, 1>>>( \ldots )}

**Okay:**  \texttt{kernel}^{<<<(N+31) / 32, 32>>>( \ldots )}

**Better:**  \texttt{kernel}^{<<<(N+127) / 128, 128>>>( \ldots )}

- Prefer to have enough threads per block to provide hardware with many warps to switch between
  - This is how the GPU hides memory access latency

- Resource like \texttt{__shared__} may constrain number of threads per block

- Algorithm and decomposition of problem will reveal the preferred amount of shared data and \texttt{__shared__} allocation
  - We often have to take a step back and come up with a new algorithm that exposes parallelism
Scheduling: Summing It Up…

- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity.

- Up to 8 blocks (on Fermi) can be executed at the same time by an SM.

- When a block of threads is executed on an SM, its threads are grouped in warps. The SM executes several warps at the same time.

- When a thread block finishes, a new block is launched on the vacated SM.
Granularity Considerations

[NOTE: Specific to Fermi]

- For Matrix Multiplication example (with shared memory) of last lecture, should I use 8X8, 16X16 or 64X64 threads per blocks?
  - For 8X8, we have 64 threads per Block. Since each Fermi SM can manage up to 1536 resident threads, it could take up to 24 Blocks. However, each SM is limited to 8 resident Blocks, so only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each Fermi SM can take up to 1536 resident threads, it can take up to 6 Blocks unless other resource considerations overrule.
    - Next you need to see how much shared memory and how many registers get used in order to understand whether you can actually have four blocks per SM
  - 64X64 is a no starter, you can only have up to 1024 threads in a block, the tile cannot be this big

- NOTE: this is the “computational thinking” we discussed last time
Example: More Warps Not Always Better
[C1060 specific]

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 20 registers
- Also, assume global loads have an associated overhead of 400 cycles
  - 3 Blocks can run on each SM; i.e., 24 warps

- If a compiler can use two more registers to change the dependence pattern so that 8 independent instructions exist (instead of 4) for each global memory load
  - Only two blocks can now run on each SM
  - However, one only needs $400 \text{ cycles}/(8 \text{ instructions} \times 4 \text{ cycles/instruction}) \approx 13$ Warps to tolerate the memory latency
  - Two Blocks have 16 Warps. The performance can be actually higher!
A Word on HTT

[Detour: slide 1/2]

- The traditional host processor (CPU) may stall due to a cache miss, branch misprediction, or data dependency

- Hyper-threading Technology (HTT): an Intel-proprietary technology used to improve parallelization of computations

- For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible.

- HT works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources.
  - This allows a hyper-threading processor to appear as two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously.

- Similar to the use of multiple warps on the GPU to hide latency
  - The GPU has an edge, since it can handle simultaneously up to 32 warps (on Tesla C1060)
Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors in response to AMD's 3DNow!

- SSE contains 70 new instructions

**Example**

- Old school, adding two vectors. Corresponds to four x86 FADD instructions in the object code

```c
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

- SSE pseudocode: a single 128 bit 'packed-add' instruction can replace the four scalar addition instructions

```c
movaps xmm0, address-of-v1 ; xmm0=v1.w | v1.y | v1.x
addps xmm0, address-of-v2 ; xmm0=v1.w+v2.w | v1.y+v2.y | v1.x+v2.x
movaps address-of-vec_res, xmm0
```
Thread Divergence

Consider the following code:

```c
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads in the warp execute the `if` clause, the other half the `else` clause
Thread Divergence

[2/4]

- The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel.

- Often, this requires that the hardware execute multiple paths through a kernel for a warp:
  - For example, both the if clause and the corresponding else clause.
Thread Divergence

[3/4]

```c
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if( b )
    {
        // g(x)
        t = g(x[i]);
    }
    else
    {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
```
Thread Divergence

[4/4]

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled

- In general, one **does not** need to consider divergence when reasoning about the correctness of a program
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock

- In general, one **does** need to consider divergence when reasoning about the performance of a program

- NVIDIA calls execution model SIMT (Single Instruction Multiple Threads) to differentiate from actual SIMD where threads really are in lockstep
Performance of Divergent Code

- Performance decreases with degree of divergence in warps
- Here’s an extreme example...

```c
__global__ void dv(int* x) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32) {
    case 0 : x[i] = a(x[i]);
             break;
    case 1 : x[i] = b(x[i]);
             break;
    ...
    case 31: x[i] = v(x[i]);
             break;
    }
}
```
Compiler and hardware can detect when all threads in a warp branch in the same direction

- Example: all take the if clause, or all take the else clause
- The hardware is optimized to handle these cases without loss of performance
- In other words, use of if or switch does not automatically translate into disaster:

  ```c
  if (threadIdx.x / WARP_SIZE >= 2) { }
  ```

  - Creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path. There is no warp divergence...

- The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)
  - Avoids some branch divergence overheads, and is more efficient
  - Often acceptable performance with short conditional clauses
Global Memory Access Issues
Global Memory and Memory Bandwidth

- Memory attributes change from card to card. On Tesla C1060:
  - 4 GB in GDDR3 RAM
  - Memory clock speed: 800 MHz
  - Memory interface: 512 bits
  - Peak Bandwidth: \( 800 \times 10^6 \times \frac{512}{8} \times 2 = 102.4 \text{ GB/s} \)

- When reporting effective bandwidth of your application:
  - Formula, effective bandwidth \((B_r - \text{bytes read}, B_w - \text{bytes written})\) [measured in GB/s]
    \[
    \text{Effective bandwidth} = \frac{((B_r + B_w)/10^9)}{\text{time}}
    \]
  - Example: kernel copies a \(2048 \times 2048\) matrix from global memory, then copies matrix back to global memory. Does it in a certain amount of \text{time} [measured in seconds]
    \[
    \text{Effective bandwidth} = \frac{((2048^2 \cdot 4 \cdot 2)/10^9)}{\text{time}}
    \]
  - 4 above comes from four bytes per float, 2 from the fact that the matrix is both read from and written to the global memory. The \(10^9\) used to get an answer in GB/s.
Data Access “Divergence”

- Concept is similar to thread divergence and often conflated

- Hardware is optimized for accessing contiguous blocks of global memory when performing loads and stores

- If a warp doesn’t access a contiguous block of global memory the effective bandwidth is getting reduced

- Remember this: when you look at a kernel you see what a collection of threads; i.e., a warp, is supposed to do in lockstep fashion
Global Memory

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers
  - The layout of the access to global memory (the pattern of the access)
  - The size/alignment of the data you try to fetch from global memory
“Memory Access Layout”
What is it?

- The basic idea:
  - Suppose each thread in a warp accesses a global memory address for a load operation at some point in the execution of the kernel.
  
  - These threads can access global memory data that is either (a) neatly grouped, or (b) scattered all over the place.

  - Case (a) is called a “coalesced memory access”
    - If you end up with (b) this will adversely impact the overall program performance.

- Analogy
  - Can send one truck on six different trips to bring back each time a bundle of wood.
  - Alternatively, can send truck to one place and get it back fully loaded with wood.
Memory Facts, Fermi GPUs

- There is 64 KB of fast memory on each SM that gets split between L1 cache and Shared Memory
  - You can split 64 KB as “L1/Sh: 16/48” or “L1/Sh: 48/16”

- L2 cache: 768 KB – one big pot available to *all* SMs on the device

- L1 and L2 cache used to cache accesses to
  - Local memory, including register spill
  - Global memory

- Whether reads are cached in [L1 & L2] or in [L2 only] can be partially configured on a per-access basis using modifiers to the load or store instruction
Fermi Memory Layout

[credits: NVIDIA]
# GPU-CPU Face Off

<table>
<thead>
<tr>
<th></th>
<th>GPU – NVIDIA Tesla C2050 (Fermi)</th>
<th>CPU – Intel core i7 975 Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Cores</td>
<td>448</td>
<td>4 (8 threads)</td>
</tr>
<tr>
<td>Memory</td>
<td>64* KB L1, per SM, 768 KB L2, all SMs, 3 GB Device Mem.</td>
<td>- 32 KB L1 cache / core, - 256 KB L2 (I&amp;D) cache / core, - 8 MB L3 (I&amp;D) shared, all cores</td>
</tr>
<tr>
<td>Clock speed</td>
<td>1.15 GHz</td>
<td>3.20 GHz</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>140 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td>Floating point operations/s</td>
<td>$515 \times 10^9$ Double Precision</td>
<td>$70 \times 10^9$ Double Precision</td>
</tr>
</tbody>
</table>

* - split 48/16
More Memory Facts
[Fermi GPUs]

- All global memory accesses are cached

- A cache line is 128 bytes
  - It maps to a 128-byte aligned segment in device memory
  - Note: it so happens that 128 bytes = 32 (warp size) * 4 bytes
    - In other words, 32 floats or 32 ints can be brought over in fell swoop

- You can determine at *compile* time (through flags: `-dlcm=ca/cg`) if you double cache [L1 & L2] or only cache [L2 only]
  - If [L1 & L2], a memory access is serviced with a 128-byte memory transaction
  - If [L2 only], a memory access is serviced with a 32-byte memory transaction
    - This can reduce over-fetch in the case of scattered memory accesses
    - Good for irregular pattern access (sparse linear algebra)
More Memory Facts
[Fermi GPUs]

- If the size of the type accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently.

- The memory access schema is as follows:
  - Two memory requests, one for each half-warp, if the size is 8 bytes.
  - Four memory requests, one for each quarter-warp, if the size is 16 bytes.

- Each memory request is then broken down into cache line requests that are issued independently.

- NOTE: a cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.
More Memory Facts
[Fermi GPUs]

- When it comes to memory store transactions to global memory:
  - First, the L1 cache is invalidated if need be
  - Next, the data is stored in L2
  - The data is actually written to global memory only if/when the data gets evicted from L2

- This strategy works since L2 is visible to all SMs on the device (unlike L1)

- How about read-before-write issues?
  - Use atomic operations (discussed next lecture)
How to Use L1 and L2

- Should you start programming to leverage L1 and L2 cache?

- The answer is: NO
  - GPU caches are not intended for the same use as CPU caches
    - Smaller sizes (on a per-thread basis, that is), not aimed at temporal reuse
      - Intended to smooth out some access patterns, help with spilled registers, etc.

- Don’t try to block for L1/L2 like you would on CPU
  - You have 100s to 1000s of run-time scheduled thread hitting the caches
  - Instead of L1, you should start thinking how to leverage Shared Memory
    - Same bandwidth (they *physically* share the same memory banks)
    - Hardware will not evict behind your back

- Conclusions
  1. Optimize as if no caches were there
  2. The reason why we talk about this: it helps you understand when the GPU is good and when it’s not
Examples of Global Mem. Access by a Warp

- **Setup:**
  - You want to access floats or integers
  - In other words, each thread is requesting a **4-Byte word**

- **Scenario A:** access is aligned and sequential

<table>
<thead>
<tr>
<th>Addresses</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Good to know:** any address of memory allocated with `cudaMalloc` is a multiple of 256
  - That is, the addressed is 256 byte aligned, which is stronger than 128 byte aligned
Examples of Global Mem. Access by a Warp

[Cntd.]

- **Scenario B: Aligned but non-sequential**

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Scenario C: Misaligned and sequential**

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why is this important?

- Compare Scenario B to Scenario C

- Basically, you have in Scenario C half the effective bandwidth you get in Scenario B
  - Just because of the alignment of your data access

- If your code is memory bound and dominated by this type of access, you might see a doubling of the run time…

- The moral of the story:
  - When you reach out to fetch data from global memory, visualize how a full warp reaches out for access. Is the access coalesced and well aligned?

- Scenarios A and B: illustrate what is called a coalesced memory access
Test Your Understanding

- Say you use in your program complex data constructs that could be organized using C-structures

- Based on what we’ve discussed so far today, how is it more advantageous to store data in global memory?
  - Alternative A: as an array of structures
  - Alternative B: as a structure of arrays
Example: Adding Two Matrices

- You have two matrices A and B of dimension $N \times N$ ($N=32$)
- You want to compute $C = A + B$ in parallel
- Code provided below (some details omitted, such as `#define N 32`)

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of $N \times N \times 1$ threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```
Test Your Understanding

- Given that the x field of a thread index changes the fastest, is the array indexing scheme on the previous slide good or bad?

- The “good or bad” refers to how data is accessed in the device’s global memory

- In other words should we have

\[
C[i][j] = A[i][j] + B[i][j]
\]

or...

\[
C[j][i] = A[j][i] + B[j][i]
\]