Advanced Computing for Engineering Applications

Dan Negrut

Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Before we get started

- **Yesterday**
  - Discussed parallel computing issues
    - Intel Haswell, NVIDIA Fermi, “big iron” HPC
    - The “black silicon” and the falling apart of Dennard’s scaling
  - Started discussion about GPU computing
    - Basic examples highlighting use of GPU computing w/ CUDA
    - Execution configuration
      - Grids, Blocks, Threads
      - Going from the thread index to the thread id
      - Figuring out a global index from a thread/block index combo
  - Discussion CUDA API
Before we get started

- Today
  - The NVIDIA GPU memory ecosystem
  - Scheduling issues
  - Atomic operations
  - Optimization issues

- Today’s challenge problem: sum up all the integers in a large array of up to 10 million elements
Why Learn More about GPUs?

- Hone our “Computational Thinking” skills

- “Computational Thinking” gained by
  - Improving your programming skills
    and more importantly,
  - Gaining a good understanding of how the hardware supports the execution of your code (the hardware/software interplay)

- Good programming skills ensures we get correct results
- Computational thinking allows us to get correct results fast
The Memory Ecosystem
Fermi: Global Memory

- Up to 6 GB of “global memory”
- “Global” in the sense that it doesn’t belong to an SM but rather all SM can access it
The Fermi Architecture

- 64 KB L1 cache & shared memory
- 768 KB L2 uniform cache (shared by all SMs)
- Memory operates at its own clock rate
- High memory bandwidth
  - Close to 200 GB/s
CUDA Device Memory Space Overview
[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on an SM

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memory

**IMPORTANT NOTE**: Global, constant, and texture memory spaces are **persistent** across kernels called by the same host application.

HK/UIUC
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads

NOTE: We will not emphasize texture here.
The Concept of Local Memory

- Local memory does not exist physically
  - “Local” in scope but not in location
- Data that is stored in “local memory” is actually placed in cache or the global memory at runtime or by the compiler.
  - If too many registers are needed for computation (“high register pressure”) the ensuing data overflow is stored in local memory
  - “Local” means that it’s got local scope; i.e., it’s specific to one thread
  - Long access times for local memory (Fermi, local memory is cached)
## Storage Locations

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

Off-chip means on-device; i.e., slow access time.
Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM: *slow* (unless if cached, which is very likely)
- Global Memory – DRAM: *slow* (unless if cached)
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached
The Three Most Important Parallel Memory Spaces

- **Register**: per-thread basis
  - Private per thread
  - Can spill into local memory (potential performance hit unless cached)

- **Shared Memory**: per-block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication

- **Global Memory**: per-application basis
  - Available for use to all threads
  - Used for: Inter-thread communication
  - Also used for inter-grid communication
Programmer View of Register File

- Number of **32 bit** registers in one **SM**:
  - 8K registers in each SM in G80
  - 16K on Tesla
  - 32K on Fermi
  - 64K on Kepler and Maxwell

- Registers are **dynamically partitioned** across all Blocks assigned to the SM

- Once assigned to a Block, these registers are **NOT accessible** by threads in other Blocks

- A thread in a Block can only access registers assigned to itself
  - Kepler and Maxwell: a thread can have assigned by the compiler up to 255 registers

Possible per-block partitioning scenarios of the RF available on the SM
Matrix Multiplication Example [Revisited]

- Purpose
  - See an example where the use of multiple blocks of threads plays a central role
  - Emphasize the role of the shared memory
  - Emphasize the need for the _syncthreads() function call

- NOTE: A one dimensional array stores the entries in the matrix
Why Revisit the Matrix Multiplication Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction ("arithmetic intensity") very low
  - Each arithmetic computation required one fetch from global memory
  - The matrix M (its entries) is copied from global memory to the device $N$.width times
  - The matrix N (its entries) is copied from global memory to the device $M$.height times

- When solving a numerical problem the goal is to go through the chain of computations as fast as possible
  - You don’t get brownie points moving data around but only computing things
A Common Programming Pattern
BRINGING THE SHARED MEMORY INTO THE PICTURE

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory

- An advantageous way of performing computation on the device is to partition (“tile”) data to take advantage of fast shared memory:
  - Partition data into data subsets (tiles) that each fits into shared memory
  - Handle each data subset (tile) with one thread block by:
    - Loading the tile from global memory into shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
Simple Test, Shared Memory Relevance

- Test whether shared memory is relevant or not
  - Imagine you are a thread and execute the kernel
  - If data that you use turns out that can be used by any other thread in your block then you should consider using shared memory

- Note: you can use shared memory as scratch pad memory
  - Don’t let it go wasted… use it just like you’d use registers
Multiply Using Several Blocks

- One **block** computes one square sub-matrix $C_{\text{sub}}$ of size $\text{Block}_\text{Size}$

- One **thread** computes one entry of $C_{\text{sub}}$

- **Assumption:** $\mathbf{A}$ and $\mathbf{B}$ are square matrices and their dimensions are multiples of $\text{Block}_\text{Size}$
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
  - In this example work with $\text{Block}_\text{Size}=16 \times 16$

---

**NOTE:** A similar technique is used on CPUs to improve cache hits. See slide “Blocking Example” at http://cseweb.ucsd.edu/classes/fa10/cse240a/pdf/08/CSE240A-MBT-L15-Cache.ppt.pdf
A Block of 16 X 16 Threads
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;  // the B (and C) matrix sub-block column index
    int by = blockIdx.y;  // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x;  // the column index in the sub-block
    int ty = threadIdx.y;  // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Shared memory for the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Shared memory for the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep, b += bStep) {
        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}

(continues with next block…)

// Shared memory for the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

// Shared memory for the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
Synchronization Function

- It’s a device lightweight runtime API function
  - `void __syncthreads();`

- Synchronizes all threads in a block (acts as a barrier for all threads of a block)
  - Does not synchronize threads from two blocks

- Once all threads have reached this point, execution resumes normally

- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory

- Allowed in conditional constructs only if the conditional is uniform across the entire thread block
The Cache vs. Shared Mem. Conundrum

- On Fermi and Kepler you can split some fast memory between shared memory and cache

  - Fermi: you can go 16/48 or 48/16 KB for ShMem/Cache

  - Lots of Cache & Little ShMem:
    - Cache handled for you by the scheduler
    - No control over it
    - Can’t have too many blocks of threads running if blocks use ShMem

  - Lots of ShMem & Little Cache:
    - Good in tiling, if you want to have full control
    - ShMem pretty cumbersome to manage
Memory Issues Not Addressed Yet...

- Not all *global* memory accesses are equivalent
  - How can you optimize memory accesses?
  - Very relevant question

- Not all *shared* memory accesses are equivalent
  - How can optimize shared memory accesses?
  - Moderately relevant questions

- To do justice to these topics we’ll need to talk first about scheduling threads for execution
  - Coming up next…
Execution Scheduling Issues
[NVIDIA cards specific]
Thread Execution Scheduling

- Topic we are about to discuss:
  - You launch on the device many blocks, each containing many threads
  - Several blocks can get executed simultaneously on one SM. How is this possible?
The 30,000 Feet Perspective

- There are two schedulers at work in GPU computing
  - A device-level scheduler: assigns blocks to SM that indicate at a given time “excess capacity”
  
  - An SM-level scheduler, which schedules the execution of the threads in a block onto the functional units available in an SM

- The more interesting is the SM-level scheduler
Device-Level Scheduler

- Grid is launched on the device

- Thread Blocks are serially distributed to all the SMs
  - Potentially more than one block per SM

- As Thread Blocks complete kernel execution, resources are freed
  - Device-level scheduler can launch next Block[s] in line

- This is the first levels of scheduling:
  - For running [desirably] a large number of blocks on a relatively small number of SMs (16/14/etc.)

- Limits for resident blocks:
  - 32 blocks resident on a Maxwell SM
  - 16 blocks can be resident on a Kepler SM
  - 8 blocks can be resident on a Fermi & Tesla SM
SM-Level Scheduler[s]

- Each Thread Block divided in 32-thread “Warps”
  - This is an implementation decision, not part of the CUDA programming model

- Warps are the basic scheduling unit in SM

- Limits, number of resident warps on an SM:
  - 64 warps on Kepler & Maxwell (i.e., 2048 resident threads)
  - 48 warps on Fermi (i.e., 1536 resident threads)
  - 32 warps on Tesla (i.e., 1024 resident threads)

- EXAMPLE: If 3 blocks are processed by an SM and each Block has 256 threads, how many Warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

- Cycles needed to dispatch the same instruction for all threads in a warp
  - On Tesla: 4 cycles
  - On Fermi: 1 cycle

- How is this relevant?
  - Suppose you use a Tesla card AND our code has one global memory access every six simple instructions
  - Then, a minimum of 17 Warps are needed to fully tolerate 400-cycle memory latency:

\[
\frac{400}{(6 \times 4)} = 16.6667 \rightarrow 17 \text{ Warps}
\]
Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads.
- One warp issued at each clock cycle by each scheduler.
- During no cycle can more than 2 warps be dispatched for execution on the four functional units.
- Scoreboarding is used to figure out which warp is ready.
Example: Fermi Related

- Scheduler works at 607 MHz
- Functional units work at 1215 MHz

Question:
- What is the peak flop rate of GTX480?
  - 15 SMs * 32 SPs * 1215 * 2 (Fused Multiplied Add) = 1166400 Mflops
  - That is, 1.166 Tflops, single precision
Fermi Specifics

- As illustrated in the picture, at no time can we see more than 2 warps being dispatched for execution during a cycle.
- Note that at any given time we might have more than two functional units working though (which is actually very good, device kept busy).
### NVIDIA Architecture Specifications

**[Information regarding one SM]**

<table>
<thead>
<tr>
<th>Architecture specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Number of cores for integer and floating-point arithmetic functions operations</td>
<td>8[^17] 32 48 192 192 128</td>
</tr>
<tr>
<td>Number of special function units for single-precision floating-point transcendental functions</td>
<td>2 4 8 32 32 32</td>
</tr>
<tr>
<td>Number of texture filtering units for every texture address unit or render output unit (ROP)</td>
<td>2 4 8 16 16 8</td>
</tr>
<tr>
<td>Number of warp schedulers</td>
<td>1 2 2 4 4 4</td>
</tr>
<tr>
<td>Number of instructions issued at once by scheduler</td>
<td>1 1 2[^18] 2 2 2</td>
</tr>
</tbody>
</table>
Threads are Executed in Warps

- Each thread block split into one or more warps based on thread ID
- 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 can control in CUDA
Thread and Warp Scheduling

- An SM can switch between warps with almost no 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

![Diagram showing warps and their states: executing, waiting for data, and ready to execute.](image-url)
Revisiting the Concept of Execution Configuration

- Prefer thread block sizes that result in mostly full warps
  
  **Bad:**  \texttt{kernel<<<N, 1>>>( ... )}  
  **Okay:**  \texttt{kernel<<<(N+31)/32, 32>>>( ... )}  
  **Better:**  \texttt{kernel<<<(N+127)/128, 128>>>( ... )}

- 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

- Register pressure might also reduce the number of threads per block
  - Because of limited register size per SM
Scheduling: Summing It Up…

- When host 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
  - Up to 16 on Kepler, up to 32 on Maxwell

- When a block of threads is executed on an SM, its threads are grouped in warps. The SM manages several warps at the same time
  - Up to 64 warps can be managed on Kepler and Maxwell

- When a thread block finishes; i.e., when all threads in the block finish executing the kernel, the block is retired and a new block is launched on the vacated SM
## Technical Specifications and Features

### [Short Detour]

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>2</td>
</tr>
<tr>
<td>Maximum x-, y-, or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y dimension of a block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>758</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>128</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
</tr>
</tbody>
</table>
Granularity Considerations
[NOTE: Specific to Fermi]

- For Matrix Multiplication example (with shared memory), 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 32 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 earlier
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

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

Often, this requires for a warp that the hardware execute multiple paths through a kernel.

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;
    }
}
```
Performance of Divergent Code

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
Data Access “Divergence”

- Concept is similar to thread divergence and they’re 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 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
    - Less important
“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]
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

- 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
More Memory Facts
[Fermi GPUs]

- 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-warps, 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
Examples of Global Mem. Access by a Warp

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

- Scenario A: access is aligned and sequential

- 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>Address</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>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>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td></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>Address</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>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 32B at 256</td>
<td>1 x 128B 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 NxN (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 * N * 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...

\[
C[j][i] = A[j][i] + B[j][i]
\]
Accesses to shared locations (global memory & shared memory) need to be correctly coordinated (orchestrated) to avoid race conditions.

In many common shared memory multithreaded programming models, one uses coordination objects such as locks to synchronize accesses to shared data.

CUDA provides several scalable synchronization mechanisms, such as efficient barriers and atomic memory operations.

Whenever possible, try hard to design algorithms with few synchronizations.
  - Coordination between threads impacts execution speed.
Race Condition

- A contrived (artificial) example...

```c++
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x;
    if (i < 2)
        *x = *y;
    else
        *x += 2*i;
}

// main.cpp
update_race<<<1,5>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Relevant Issue: Thread Divergence in “if-then-else”

- Handling of an if-then-else construct in CUDA
  - First a subset of threads of the warp execute the “then” branch
  - Next, the rest of the threads in the warp execute the “else” branch

- Question: what happens if in the previous slide if you change the “if” to “if(i>=3) …” and swap the then and the else parts?
  - How are things different compared to sequential computing?
#include <cuda.h>
#include "stdio.h"

__global__ void testKernel(int *x, int *y) {
    int i = threadIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}

int main() {
    int* dArr;
    int hArr[2] = {23, -5};
    cudaMalloc(&dArr, 2 * sizeof(int));
    cudaMemcpy(dArr, hArr, 2 * sizeof(int), cudaMemcpyHostToDevice);
    testKernel <<<1, 2 >>>(dArr, dArr + 1);
    cudaMemcpy(hArr, dArr, 2 * sizeof(int), cudaMemcpyDeviceToHost);
    printf("x = %d\n", hArr[0]);
    printf("y = %d\n", hArr[1]);
    return 0;
}
C:\Users\negrut\Bin\US13Projects\CudaSandBox\Darmstadt\Debug>testRace
x = 1
y = 1
C:\Users\negrut\Bin\US13Projects\CudaSandBox\Darmstadt\Debug>
Example: Inter-Block Issue

- Would this fly?

```cpp
// update.cu
__global__ void update(int* x, int* y)
{
    int i = threadIdx.x;
    if (i == 0) *x = blockIdx.x;
    if (i == 1) *y = *x;
}

// main.cpp
update<<<2,5>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Synchronization within Grid
[The Need for Atomics]

- Often not reasonable to split kernels to synchronize reads and writes from different threads to common locations. Here’re two reasons:
  - Values of `__shared__` variables are lost unless explicitly saved
  - Kernel launch overhead is nontrivial – extra launches can degrade performance

- CUDA provides atomic functions (commonly called atomic memory operations) to enforce atomic accesses to variables that may be accessed by multiple threads

- Programmers can synthesize various coordination objects and synchronization schemes using atomic functions.
Atomics
Atomics, Introduction

- Atomic memory operations (atomic functions) are used to solve coordination problems in parallel computer systems.

- General concept: provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads.

- This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.
Atomic Functions

[1/3]

- Atomic functions perform read-modify-write operations on data residing in global and shared memory

```c
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = atomicAdd(x, i);  // j is now old value of x;
}

// snippet of code in main.cpp
int x = 0;
cudaMemcpy(&d_x, &x, cudaMemcpyHostToDevice);
update<<<1,128>>>(x_d);
cudaMemcpy(&x, &d_x, cudaMemcpyDeviceToHost);
```

- Atomic functions guarantee that only one thread may access a memory location while the operation completes
- Order in which threads get to write is not specified though…
Atomic Functions

Atomic functions perform read-modify-write operations on data that can reside in global or shared memory.

Synopsis of atomic function atomicOP(a,b) is typically

\[
\begin{align*}
    t1 &= *a; \quad \text{// read} \\
    t2 &= (*a) \text{ OP } (*b); \quad \text{// modify} \\
    *a &= t2; \quad \text{// write} \\
    \text{return } t1;
\end{align*}
\]

The hardware ensures that all statements are executed atomically without interruption by any other atomic functions.

The atomic function returns the initial value, *not* the final value, stored at the memory location.
Atomic Functions

The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates.

The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary.

However, none of the atomic updates will be lost.

Many different kinds of atomic operations:
- Add (add), Sub (subtract), Inc (increment), Dec (decrement)
- And (bit-wise and), Or (bit-wise or), Xor (bit-wise exclusive or)
- Exch (Exchange)
- Min (Minimum), Max (Maximum)
- Compare-and-Swap
// Compute histogram of colors in an image
//
// color – pointer to picture color data
// bucket – pointer to histogram buckets, one per color
//
__global__ void histogram(int n, int* color, int* bucket)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
    {
        int c = colors[i];
        atomicAdd(&bucket[c], 1);
    }
}
Performance Notes

- Atomics are slower than normal accesses (loads, stores)

- Performance can degrade when **many** threads attempt to perform atomic operations on a **small** number of locations

- Possible to have all threads on the machine stalled, waiting to perform atomic operations on a single memory location

- Atomics: convenient to use, come at a typically high efficiency loss…
Important note about Atomics

- Atomic updates are not guaranteed to appear atomic to concurrent accesses using loads and stores

```c
__global__ void broken(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0)
    {
        *x = *x + 1;
    }
    else
    {
        int j = atomicAdd(x, 1); // j = *x; *x += i;
    }
}

// main.cpp
broken<<<1,128>>>(128, d_x); // d_x = d_x + {1, 127, 128}
```
Summary of Atomics

- When to use: Cannot use normal load/store because of possible race conditions

- Use atomic functions for infrequent, sparse, and/or unpredictable global communication

- Attempt to use shared memory and structure algorithms to avoid synchronization whenever possible
CUDA GPU Programming
~ Resource Management Considerations ~
What Do I Mean By “Resource Management”?

- The GPU is a resourceful device

- What do you have to do to make sure you capitalize on these resources?
  - In other words, how can you ensure that all the SPs are busy all the time?

- The three factors that come into play are
  - How many threads you decide to use in each block
  - What register requirements end up associated with a thread
  - How much shared memory you assign to one block of threads
The Bottom Line

- You want to have as many warps resident on one SM as possible
  - 32 warps on Tesla
  - 48 warps on Fermi
  - 64 warps on Kepler and Maxwell

- Why?
  - If you have many warps you stand a better chance of hiding latency of memory accesses with useful execution
  - In other words, you don’t see memory as a bottleneck anymore
Resource Management – The Key Actors: Threads, Warps, Blocks [Review]

- A collection of 32 Threads makes up a Warp
  - A warp is made up of threads with consecutive IDs

- A Block has at the most 1024 threads
  - Threads are organized in a 3D fashion; each thread has an \( (T_x, T_y, T_z) \) unique thread ID
  - Threads in a block get to use together the shared memory

- Each Block of threads gets assigned to a SM and then is executed on that SM
  - One SM can be assigned by the scheduler to handle more blocks at the same time
    - Done through time slicing
Execution Model, Key Observations [1 of 2]

- Each block is executed on *one* Stream Multiprocessor (SM)
  - “block is executed” above means that all the threads that make up that block execute a kernel
  - Each block is split into warps of threads. The warps are executed one at a time by the SPs of the SM (time slicing in warp execution is the norm)
A Stream Multiprocessor can execute multiple blocks concurrently

- If N blocks get executed simultaneously on the same SM, then:
  - The total amount of shared memory available on SM should accommodate the N blocks
  - NOTE: The shared memory is an attribute of the block, not of a thread

- If N blocks get executed simultaneously on the same SM, then
  - The register files associated with the SM is split amongst the union of threads associated with the N blocks
  - NOTE: The less registers get used per thread, the better (you potentially can squeeze more blocks on a SM ) better resource utilization
Some Hard Constraints  [1 of 2]

- Max number of warps that one SM can service simultaneously:
  - 32 on Tesla C1060, 48 on Fermi, 64 on Kepler

- Max number of blocks that one SM can process simultaneously:
  - 8 (Fermi), 16 (Kepler), 32 (Maxwell)
Some Hard Constraints [2 of 2]

- The number of 32 bit registers available on each SM is limited:
  - 16,384 registers (on Tesla C1060)
  - Roughly 48,000 on Fermi
  - Roughly 64,000 on Kepler and Maxwell

- The amount of shared memory available to each SM is limited
  - 16 KB on Tesla 1060
  - 64 KB on Fermi (16/48 or 48/16 configurable between L1$ and shared memory)
  - 64 KB on Kepler (16/48 or 48/16 or 32/32 configurable)
## Technical Specifications and Features

[Short Detour]

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td></td>
</tr>
<tr>
<td>Maximum x-, y-, or z-dimension of a grid of thread blocks</td>
<td></td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>758</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>128</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
</tr>
</tbody>
</table>
The Concept of Occupancy
[Discussion Focused on Tesla]

- Ideally, you want to have 32 warps serviced at the same time by one SM
  - That is, you’d like to have the SM end up executing the max number that it is designed to execute
  - This keeps the SM busy and hides latencies associated with memory access

- Examples:
  - Two blocks with 512 threads running together on one SM: 100% occupancy
  - Four blocks of 256 threads each running on one SM: 100% occupancy
  - 16 blocks with 64 threads each – not good, can’t have more than 8 blocks running on a SM
    - Effectively this scenario gives you at most 50% occupancy
The Concept of Occupancy [Cntd.]

- What prevents you from getting high occupancy?
  - Many warps means many threads and potentially many blocks
    - Many blocks - you can’t have too much shared mem allocated to each one of them
      - Total amount of shared memory in one SM is limited
    - Many threads - you can’t have too many registers used by each thread
      - Size of the register file in one SM is limited
Resource Utilization

- There is an “occupancy calculator” that can tell you what percentage of the HW gets utilized by your kernel.

- Assumes the form of an Excel spreadsheet.

- Requires the following input:
  - Threads per block
  - Registers per thread
  - Shared memory per block

- Google “occupancy calculator cuda” to access it.
NVIDIA CUDA Occupancy Calculator

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA visit [http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

![Graphs showing varying block size, varying register count, and varying shared memory usage.](image-url)
Granularity Considerations
[NOTE: Specific to Fermi]

- For Matrix Multiplication example (with shared memory), 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 32 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
Premature Optimization is the Root of All Evil. Yet,…

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

Donald Knuth

In “Structured Programming With Go To Statements”
Computing Surveys, Vol. 6, No. 4, December 1974
Available on class website.
Next, the discussion focuses on tools you can use to find that 3% of the code worth optimizing...
Code Timing/Profiling

- Lazy man’s solution
  - Do nothing, instruct the executable to register crude profiling info

- Advanced approach: use NVIDIA’s nvvp Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Allows for automated analysis
  - **nvvp** is a cross platform tool (linux, mac, windows)
Lazy Man’s Solution…

- Set the right environment variable and run your executable [illustrated on Euler]:

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> export CUDA_PROFILE=1
>> ./testV4_20
>> cat cuda_profile_0.log
```

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```
Lazy Man’s Solution...

```
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

```
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
Lazy Man’s Solution...

```plaintext
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

```plaintext
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
Lazy Man’s Solution...

```bash
$ nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
$ ./testV4_20

# CUDA PROFILE LOG VERSION 2.0
# CUDA DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR ffffffff6c689a404a8
method,gputime,cputime,occupancy
 method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
 method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

Compute capability 2.0 (Fermi)

```bash
$ nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
$ ./testV4_10

# CUDA PROFILE LOG VERSION 2.0
# CUDA DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
 method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
 method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
 method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```

Compute capability 1.0 (Tesla/G80)
nvvp: NVIDIA Visual Profiler

- Available on Euler

- Provides a nice GUI and ample information regarding your run

- Many bells & whistles
  - Covering here the basics through a 1D stencil example

- Acknowledgement: Discussion on nvvp uses material from NVIDIA (S. Satoor).
  - Slides that include this material marked by “NVIDIA [S. Satoor]→” sign at bottom of slide
1D Stencil: A Common Algorithmic Pattern

[Problem Used to Introduce Profiling Tool]

- Applying a 1D stencil to a 1D array of elements
  - Function of input elements within a radius

- Fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering,…

- This example will use weighted arithmetic mean
Serial Algorithm

(radius = 3)

\[ f \]

\[ \sim \] = CPU Thread

NVIDIA [S. Satoor]→
Serial Algorithm

\[ \text{(radius} = 3) \]

\[ f \]

\[ \Downarrow \]

Repeat for each element

\[ \Downarrow = \text{CPU Thread} \]
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS,N-RADIUS,weights,in,out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);
    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
Serial Implementation

```c
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);
    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

For each element...

Weighted mean over radius
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

<table>
<thead>
<tr>
<th>CPU</th>
<th>MEElements/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930</td>
<td>30</td>
</tr>
</tbody>
</table>

NVIDIA [S. Satoor]→
Parallel Algorithm

\[\rightarrow\] = Thread

Serial: One element at a time

Parallel: Many elements at a time
The GPU kernel

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights;  cudaMalloc(&d_weights, wsize);
    float *d_in;  cudaMalloc(&d_in, size);
    float *d_out;  cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if(i < eIdx) {
        out[i] = 0;
        //loop over all elements in the stencil
        for(int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

Copy GPU Inputs

The Parallel Implementation
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cuMemAlloc(&d_weights, wsize);
    float *d_in; cuMemAlloc(&d_in, size);
    float *d_out; cuMemAlloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cuMemFree(d_weights); cuMemFree(d_in); cuMemFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x * blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
void main() {
    int size = N * sizeof(float);
    int ssize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(ssize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, ssize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

cudamemcpy(d_weights, weights, ssize, cudaMemcpyHostToDevice);
cudamemcpy(d_in, in, size, cudaMemcpyHostToDevice);
applyStencillD<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
cudamemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
cudafree(d_weights); cudafree(d_in); cudafree(d_out);
}

__global__ void applyStencillD(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

Copy results from GPU

The Parallel Implementation

NVIDIA [S. Satoor]→
### The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N - RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
</tbody>
</table>

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```
Application Optimization Process
[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - `cuda-gdb`, `cuda-memcheck`

- Optimize
  - …dealing with this next
NVIDIA Visual Profiler

Timeline of CPU and GPU activity

Kernel and memcpy details

<table>
<thead>
<tr>
<th>Process: 8058</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread: 2127574912</td>
</tr>
<tr>
<td>RunTime API</td>
</tr>
<tr>
<td>[0] Tesla C2075</td>
</tr>
<tr>
<td>Context 1 (CUDA)</td>
</tr>
<tr>
<td>MemCPy (HtoD)</td>
</tr>
<tr>
<td>MemCPy (DtoH)</td>
</tr>
<tr>
<td>Compute</td>
</tr>
<tr>
<td>5.4% [1] applySt...</td>
</tr>
<tr>
<td>Streams</td>
</tr>
<tr>
<td>Stream 1</td>
</tr>
<tr>
<td>apply...</td>
</tr>
</tbody>
</table>

applyStencil1D_gpu(int, int, float ...)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>69.628 ms</td>
</tr>
<tr>
<td>Duration</td>
<td>8.177 ms</td>
</tr>
<tr>
<td>Grid Size</td>
<td>[32768,1,1]</td>
</tr>
<tr>
<td>Block Size</td>
<td>[512,1,1]</td>
</tr>
<tr>
<td>Registers/Thread</td>
<td>20</td>
</tr>
<tr>
<td>Shared Memory/Block</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Occupancy</td>
<td>100%</td>
</tr>
<tr>
<td>Theoretical</td>
<td>100%</td>
</tr>
<tr>
<td>L1 Cache Configuration</td>
<td></td>
</tr>
<tr>
<td>Shared Memory Request</td>
<td>48 KB</td>
</tr>
<tr>
<td>Shared Memory Execution</td>
<td>48 KB</td>
</tr>
</tbody>
</table>

GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s
## NVIDIA Visual Profiler

### CUDA API activity on CPU

- Process: 8058
- Thread: 2127574912
- Runtime API
- Driver API

### Memcpy and kernel activity on GPU

- Context 1 (CUDA)
  - MemCpy (HtoD)
  - MemCpy (DtoH)
  - Compute
    - 5.4% [1] applyStencil
  - Streams
    - Stream 1

### GPU Performance

```
GPU: 0.058926 seconds, 2.27773 GBytes/s, 0.284716 GElements/s
```
Detecting Low Memory Throughput

- Spend majority of time in data transfer
  - Often can be overlapped with preceding or following computation

- From timeline can see that throughput is low
  - PCIe x16 can sustain > 5GB/s

NVIDIA [S. Satoor]→
Visual Profiler Analysis

- How do we know when there is an optimization opportunity?
  - Timeline visualization seems to indicate an opportunity
  - Documentation gives guidance and strategies for tuning
    - CUDA Best Practices Guide – link on the website
    - CUDA Programming Guide – link on the website

- Visual Profiler analyzes your application
  - Uses timeline and other collected information
  - Highlights specific guidance from Best Practices
  - Like having a customized Best Practices Guide for your application
Visual Profiler Analysis

Several types of analysis are provided

Analysis pointing out low memcpy throughput

The amount of time performing compute is low relative to the amount of time required for memcpy:

- Low Memcpy/Compute Overlap [0 ns / 8.176 ms = 0%]
  The percentage of time when memcpy is being performed in parallel with compute is low.

- Low Memcpy Throughput [997.19 MB/s avg, for memcpys accounting for 68.1% of a]
  The memory copies are not fully using the available host to device bandwidth.

- Low Memcpy Overlap [0 ns / 15.79 ms = 0%]
Online Optimization Help

Low Memcpy Throughput [ 997.19 MB/s avg, for memcpys accounting for 68.1% of all memcpy time ]
The memory copies are not fully using the available host to device bandwidth.

Each analysis has link to Best Practices documentation
Pinned CPU Memory Implementation

```c
int main()
{
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights; cudaMallocHost(&weights, wsize);
    float *in; cudaMallocHost(&in, size);
    float *out; cudaMallocHost(&out, size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights);
    float *d_in; cudaMalloc(&d_in);
    float *d_out; cudaMalloc(&d_out);
    ...
}
```
**Pinned CPU Memory Result**

<table>
<thead>
<tr>
<th>Process: 8527</th>
<th>0.1 s</th>
<th>0.11 s</th>
<th>0.12 s</th>
<th>0.13 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread: -1613166720</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime API</td>
<td>cudaMemcpy</td>
<td>cudaMemcpy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver API</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context 1 (CUDA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemCpy (HtoD)</td>
<td>Memcpy HtoD [sy...</td>
<td></td>
<td>MemcpyDtoH [s...</td>
<td></td>
</tr>
<tr>
<td>MemCpy (DtoH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td>applyStencil...</td>
<td>applyStencil...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream 1</td>
<td>Memcpy HtoD [sy...</td>
<td>applyStencil...</td>
<td>MemcpyDtoH [s...</td>
<td></td>
</tr>
</tbody>
</table>

**Memcpy DtoH [sync]**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>117.622 ms</td>
</tr>
<tr>
<td>Duration</td>
<td>10.469 ms</td>
</tr>
<tr>
<td>Size</td>
<td>64 MB</td>
</tr>
<tr>
<td>Throughput</td>
<td>5.97 GB/s</td>
</tr>
</tbody>
</table>

GPU PINNED: 0.0297912 seconds, 4.50528 GBytes/s, 0.563158 GElements/s
Pinned CPU Memory Result

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Pinned Memory</td>
<td>560</td>
<td>4.3x</td>
</tr>
</tbody>
</table>

*4 cores + hyperthreading
CUDA Optimization: Wrap Up...
Performance Optimization

[Wrapping Up…]

- We discussed many rules and ways to write better CUDA code
- The next several slides sort this collection of recommendations based on their importance
- Writing CUDA software is a craft/skill that is learned
  - Just like playing a game well: know the rules and practice
  - A list of high, medium, and low priority recommendations wraps up discussion on CUDA optimization
    - For more details, check the CUDA C Best Practices Guide:

1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code.

2. Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits.

3. Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU.

Writing CUDA Software: High-Priority Recommendations

4. Strive to have aligned and coalesced global memory accesses

5. Minimize the use of global memory. Prefer shared memory access where possible (consider tiling as a design solution)

6. Avoid different execution paths within the same warp
Writing CUDA Software: Medium-Priority Recommendations

1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts

2. To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy)

3. The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing

4. Use the fast math library whenever speed is very important and you can live with a tiny loss of accuracy

5. Prefer faster, more specialized math functions over slower, more general ones when possible
Writing CUDA Software: Low-Priority Recommendations

1. For kernels with long argument lists, place some arguments into constant memory to save shared memory

2. Use shift operations to avoid expensive division and modulo calculations

3. Avoid automatic conversion of doubles to floats

Advanced Memory Issues
Issues Discussed

- Zero-copy memory in CUDA
- Unified Virtual Addressing
- Managed Memory

Based on Dr. Dobbs article of Sept 30, 2014
Summary / Objective

- Premise: Managing and optimizing host-device data transfers has been challenging

- Key point: Unified Memory (UM) support in CUDA 6 simplifies the programmer’s job

- This segment’s two goals:
  - Briefly review history of CUDA host/device memory management
  - Explain how UM makes host/device memory management easier and more efficient
cudaMemcpy

- A staple of CUDA, available in release 1.0

- Setup was simple: one CPU thread dealt with one GPU
  - The drill:
    - Data transferred from host memory into device memory with cudaMemcpy
    - Data was processed on the device by invoking a kernel
    - Results transferred from device memory into host memory with cudaMemcpy

- Memory allocated on the host with malloc
- Memory allocated on the device using the CUDA runtime function cudaMemcpy
- The bottleneck: data movement over the PCI-E link
The PCI-E Pipe, Putting Things in Perspective

- PCI-E
  - V1: 3 GB/s (per direction)
  - V2: 6 GB/s
  - V3 (today): 12 GB/s

- Bandwidths above pretty small, see for instance
  - Host memory bus (25 – 51.2 GB/s per socket)
  - GMEM bandwidth 100 – 200 GB/s
cudaHostAlloc: A friend, with its pluses and minuses

- Host/Device data transfer speeds could be improved if host memory was not pageable
  - Rather than allocating with malloc, host memory was allocated using CUDA’s cudaHostAlloc()
  - No magic on the hardware side, data still moves back-and-forth through same PCI-E connection

- cudaHostAlloc cons
  - cudaHostAlloc-ing large amounts of memory can negatively impact overall system performance
    - Why? It reduces the amount of system memory available for paging
    - How much is too much? Not clear, dependent on the system and the applications running on the machine
  - cudaHostAlloc is slow - ballpark 5 GB/s
    - Allocating 5 GB of memory is timewise comparable to moving that much memory over the PCI-E bus
Key Benefits, cudaHostAlloc-ing Memory

- Three benefits to replacing host malloc call with CUDA cudaHostAlloc call

  1. Enables faster device/host back-and-forth transfers

  2. Enables the use of asynchronous memory transfer and kernel execution
     - Draws on the concept of CUDA stream, a topic not covered here

  3. Enables the mapping of the pinned memory into the memory space of the device
     - Device now capable to access data on host while executing a kernel or other device function

- Focus next is on 3 above
Zero-Copy (Z-C) GPU-CPU Interaction

- Last argument ("flag") controls the magic:
  ```c
  cudaError_t cudaHostAlloc ( void** pHost, size_t size, unsigned int flag)
  ```

- "flag" values: cudaHostAllocPortable, cudaHostAllocWriteCombined, etc.

- The "flag" of most interest is "cudaHostAllocMapped"
  - Maps the memory allocated on the host in the memory space of the device for direct access

- What's gained:
  - The ability to access a piece of data from pinned and mapped host memory by a thread running on the GPU without a CUDA runtime copy call to explicitly move data onto the GPU
  - This is called zero-copy GPU-CPU interaction, from where the name "zero-copy memory"
  - Note that data is still moved through the PCI-E pipe, but it's done in a transparent fashion
Z-C, Further Comments

- More on the “flag” argument, which can take four values:
  - Use cudaMemcpyDefault argument for getting plain vanilla pinned host memory (call becomes identical in this case to cudaMemcpyHost call)
  - Use cudaMemcpyHostAlloc to pick up the Z-C functionality
  - See documentation for cudaMemcpyHostAllocWriteCombined and cudaMemcpyHostAllocPortable
    - These two flags provide additional tweaks, irrelevant here

- The focus should not be on cudaMemcpy() and the “flag”
  - This function call is only a means to an end

- Focus should be on the fact that a device thread can directly access host memory
From Z-C to UVA: CUDA 2.2 to CUDA 4.0

- Z-C enabled access of data on the host from the device required one additional runtime call to `cudaHostGetDevicePointer()`
  - `cudaHostGetDevicePointer()`: given a pointer to pinned host memory produces a new pointer that can be invoked within the kernel to access data stored on the host

- The need for the `cudaHostGetDevicePointer()` call eliminated in CUDA 4.0 with the introduction of the Unified Virtual Addressing (UVA) mechanism
Unified Virtual Addressing: CUDA 4.0

- CUDA runtime can identify where the data is stored based on the value of the pointer
  - Possible since one address space was used for all CPU and GPU memory

- In a unified virtual address space setup, the runtime manipulates the pointer and allocation mappings used in device code (through cudaMalloc) as well as pointers and allocation mappings used in host code (through cudaHostAlloc()) inside a single unified space
UVA - Consequences

- There is no need to deal with cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, and cudaMemcpyDeviceToDevice scenarios
  - Simply use the generic cudaMemcpyDefault flag

- Technicalities regarding the need to call cudaMemcpyDefault flag
  - cudaGetDeviceProperties() for all participating devices (to check cudaDeviceProp::unifiedAddressing flag) to figure out whether they’re game for UVA are skipped

- What this buys us: ability to do, for instance, inter-device copy that does not rely on the host for staging data movement:
  - cudaMemcpy(gpuDst_memPntr, gpuSrc_memPntr, byteSize, cudaMemcpyDefault)
UVA – Showcasing Its Versatility…

- Set of commands below can be issued by one host thread to multiple devices
  - No need to use anything beyond cudaMemcpyDefault
    cudaMemcpy(gpu1Dst_memPntr, host_memPntr, byteSize1, cudaMemcpyDefault)
    cudaMemcpy(gpu2Dst_memPntr, host_memPntr, byteSize2, cudaMemcpyDefault)
    cudaMemcpy(host_memPntr, gpu1Dst_memPntr, byteSize1, cudaMemcpyDefault)
    cudaMemcpy(host_memPntr, gpu2Dst_memPntr, byteSize2, cudaMemcpyDefault)

- UVA support is the enabler for the peer-to-peer (P2P), inter-GPU, data transfer
  - P2P not topic of discussion here
  - UVA is the underpinning technology for P2P
UVA is a Step Forward Relative to Z-C

- Z-C Key Accomplishment: use pointer within device function access host data
  - Z-C focused on a data access issue relevant in the context of functions executed on the device

- UVA had a data access component but also a data transfer component:
  - Data access: A GPU could access data on a different GPU, a novelty back in CUDA 4.0
  - Data transfer: copy data in between GPUs
    - cudaMemcpy is the main character in this play, data transfer initiated on the host side
Zero-Copy, UVA, and How UM Fits In [1/2]

- Both for Z-C and UVA the memory was allocated on the device with `cudaMalloc` and on the host with `cudaHostAlloc`

- The magic ensued upon the `cudaHostAlloc/cudaMalloc` duo

- Examples of things that can be done:
  - Data on host accessed on the device
  - Data transferred effectively in between devices without intermediate staging on the host
  - Data stored by one GPU accessed directly by a different GPU
  - Etc.
Zero-Copy, UVA, and How UM Fits In

[1/2]

- Unified Memory (UM) eliminates the need to call the cudaMalloc/cudaHostAlloc duo
  - It takes a different perspective on handling memory in the GPU/CPU interplay

- Note that in theory one could get by using Z-C and only calling cudaHostAlloc once. This is not recommended when having repeated accesses by device to host-side memory
  - Each device request that ends up accessing the host-side memory incurs high latency and low bandwidth (relative to the latency and bandwidth of an access to device global memory)

- This is the backdrop against which the role of UM is justified
  - Data is stored and migrated in a user-transparent fashion
    - To the extent possible, the data is right where it’s needed thus enabling fast access
Unified Memory (UM)

- One memory allocation call takes care of memory setup at both ends; i.e., device and host
  - The main actor: the CUDA runtime function cudaMallocManaged()
- New way of perceiving the memory interplay in GPGPU computing
  - No distinction is made between memory on the host and memory on the device
  - It’s just memory, albeit with different access times when accessed by different processors
Unified Memory (UM) – Semantics Issues
[clarifications of terms used on previous slide]

- “processor” (from NVIDIA documentation): any independent execution unit with a dedicated memory management unit (MMU)
  - Includes both CPUs and GPUs of any type and architecture

- “different access time”: time is higher when, for instance, the host accesses for the first time data stored on the device.
  - Subsequent accesses to the same data take place at the bandwidth and latency of accessing host memory
    - This is why access time is different and lower
    - Original access time higher due to migration of data from device to host
  - NOTE: same remarks apply to accesses from the device
```cpp
#include <iostream>
#include "math.h"

const int ARRAY_SIZE = 1000;
using namespace std;

__global__ void increment(double* aArray, double val, unsigned int sz)
{
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main(int argc, char **argv) {
    double* mA;
    cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));
    for (int i = 0; i < ARRAY_SIZE; i++)
        mA[i] = 1.*i;
    double inc_val = 2.0;
    increment <<<2, 512 >>>(mA, inc_val, ARRAY_SIZE);
    cudaMemcpyDeviceToHost(hA);
    double error = 0.;
    for (int i = 0; i < ARRAY_SIZE; i++)
        error += fabs(mA[i] - (1 + inc_val));
    cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;
    cudaFree(mA);
    return 0;
}
```
UM vs. Z-C

- Recall that with Z-C, data is always on the host in pinned CPU system memory
  - The device reaches out to it

- UM: data stored on the device but made available where needed
  - Data access and locality managed by underlying system, handling transparent to the user
  - UM provides “single-pointer-to-data” model

- Support for UM called for only *three* additions to CUDA:
  - cudaMallocManaged, __managed__, cudaStreamAttachMemAsync()
Technicalities...

- `cudaError_t cudaMallocManaged ( void** devPtr, size_t size, unsigned int flag)
  - Returns pointer accessible from both Host and Device
  - Drop-in replacement for `cudaMalloc()` – they are semantically similar
  - Allocates managed memory on the device
    - First two arguments have the expected meaning
  - “flag” controls the default stream association for this allocation
    - `cudaMemAttachGlobal` - memory is accessible from any stream on any device
    - `cudaMemAttachHost` – memory on this device accessible by host only
  - Free memory with the same `cudaFree()`

- **managed**
  - Global/file-scope variable annotation combines with **device**
  - Declares global-scope migrateable device variable
  - Symbol accessible from both GPU and CPU code

- `cudaStreamAttachMemAsync()
  - Manages concurrency in multi-threaded CPU applications`
UM, Quick Points

- In the current implementation, managed memory is allocated on the device that happens to be active at the time of the allocation.

- Managed memory is interoperable and interchangeable with device-specific allocations, such as those created using the cudaMalloc() routine.

- All CUDA operations that are valid on device memory are also valid on managed memory.
Example: UM and thrust

```cpp
#include <iostream>
#include <cmath>
#include <thrust/reduce.h>
#include <thrust/system/cuda/execution_policy.h>
#include <thrust/system/omp/execution_policy.h>

const int ARRAY_SIZE = 1000;

int main(int argc, char **argv) {
    double* mA;
    cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));

    thrust::sequence(mA, mA + ARRAY_SIZE, 1);

    double maximumGPU = thrust::reduce(thrust::cuda::par, mA, mA + ARRAY_SIZE, 0.0, thrust::maximum<double>());
    cudaDeviceSynchronize();
    double maximumCPU = thrust::reduce(thrust::omp::par, mA, mA + ARRAY_SIZE, 0.0, thrust::maximum<double>());

    std::cout << "GPU reduce: " << (std::fabs(maximumGPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed") << std::endl;
    std::cout << "CPU reduce: " << (std::fabs(maximumCPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed") << std::endl;

    cudaFree(mA);
    return 0;
}
```
Advanced Features: UM

- Managed memory migration is at the page level
  - The default page size is currently the same as the OS page size today (typically 4 KB)

- The runtime intercepts CPU dirty pages and detects page faults
  - Moves from device over PCI-E only the dirty pages
  - Transparently, pages touched by the CPU (GPU) are moved back to the device (host) when needed

- Coherence points are kernel launch and device/stream sync.
  - Important: the same memory cannot be operated upon, at the same time, by the device and host
Advanced Features: UM

- Issues related to “managed memory size”:
  - For now, there is no oversubscription of the device memory
    - In fact, if there are several devices available, the max amount of managed memory that can be allocated is the smallest of the memories available on the devices

- Issues related to “transfer/execution overlap”:
  - Pages from managed allocations touched by CPU migrated back to GPU before any kernel launch
    - Consequence: there is no kernel execution/data transfer overlap in that stream
    - Overlap possible with UM but just like before it requires multiple kernels in separate streams
      - Enabled by the fact that a managed allocation can be specific to a stream
      - Allows one to control which allocations are synchronized on specific kernel launches, enables concurrency
The GPU has *exclusive* access to this memory when any kernel is executed on the device:
- Holds even if the kernel doesn’t touch the managed memory

The CPU cannot access *any* managed memory allocation or variable as long as GPU is executing.

A `cudaDeviceSynchronize()` call required for the host to be allowed to access managed memory:
- To this end, any function that logically guarantees the GPU finished execution is acceptable:
  - Examples: `cudaStreamSynchronize()`, `cudaMemcpy()`, `cudaMemset()`, etc.
Left: Seg fault
Right: Runs ok

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
  x = 10;
}

int main() {
  kernel << < 1, 1 >> >();
  y = 20; // ERROR: CPU access concurrent with GPU
  cudaDeviceSynchronize();
  return 0;
}
```

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
  x = 10;
}

int main() {
  kernel << < 1, 1 >> >();
  cudaDeviceSynchronize();
  y = 20; // GPU is idle so access is OK
  return 0;
}
```
UM – Current Limitations in CUDA 6.0

- Ability to allocate more memory than the physically available on the GPU
- Prefetching
- Finer Grain Migration
UM – Why Bother?

1. A matter of convenience
   - Much simpler to write code using this memory model
   - For the casual programmer, the code will run faster due to data locality
     - The runtime will take care of moving the data where it ought to be

2. Looking ahead, physical CPU/GPU integration around the corner – memory will be shared
   - Already the case for integrated GPUs that are part of the system chipset
   - The trend in which the industry is moving (AMD’s APU, Intel’s Haswell, NVIDIA Denver Project)
   - The functionality provided by the current software backend that supports the cudaMallocManaged() paradigm will be eventually implemented in hardware
Sparse Matrix Reordering for LU Factorization on the GPU

CUDA 6.0 AT WORK
Sparse Linear Algebra on the GPU

- **Problem Statement:**
  - Solve $Ax=b$, where $A$ is large and sparse

- **Solutions for parallel execution already exist...**
  - In cusparse if $A$ is symmetric and positive definite
  - Using MPI or OpenMP

- **Ongoing work on general purpose preconditioner using a partitioning of the problem into subproblems that are solved in parallel**
  - Approach known in the literature as SPIKE
  - GitHub repo: https://github.com/spikegpu/SpikeLibrary
  - Project webpage: http://spikegpu.sbel.org/
  - Focus herein on the performance of CUDA 6.0 in the context of sparse matrix reordering
Introduction, Matrix Reordering For Diagonal Dominance

- Non-symmetric reordering approach – moves around entries in a matrix by shifting the order of the rows and columns in the matrix

- What’s desired: migrate to the diagonal the matrix elements whose absolute value is large

- Why? Helps achieving a high degree of diagonal dominance
  - Decreases probability of encountering a zero pivot during the LU factorization of the sparse matrix
  - Improve the quality of the LU factorization

- Core part of algorithm: finding minimum perfect bipartite matching
Matrix Reordering Algorithm: Four Stages

- **First stage: forming bipartite graph**
  - View each row as a row node and each column as a column node. There is an edge between each pair of row node and column node.
  - An edge has finite weight if and only if the matrix has a non-zero value in the corresponding entry.
  - Given a matrix, the structure of matrix is unchanged, only values need to be processed to get the weights of the bipartite graph.
  - Can be done in parallel on the GPU.

- **Second stage: finding initial partial match (optional)**
  - For each column node j, find a row node i so that edge (i, j) is the minimum over all edges connected to column node j.
  - If row node i is not yet matched to another column node, match i to j.
  - Hard to do on the GPU.
Stages of the Algorithm (Cont’d)

- Third stage: finding a perfect match
  - If in the second stage all column nodes are matched, then this stage is not necessary
  - Apply Hungarian algorithm (i.e. finding augmenting path) to match each unmatched column node
  - Report error if Hungarian algorithm fails to find a match with finite value for a certain node
  - Totally sequential in nature, no good parallel algorithm suitable for GPU as of now

- Fourth stage: extracting the permutation and the scaling factors
  - Permutation can be obtained directly form the resulting perfect match
  - Stage is highly parallelizable and suitable for GPU computing
Sparse Matrix Reordering Strategy

- Stages 1 and 4 done on the GPU
- Stages 2 and 3 done on the CPU
- Use of UM:
  - Store all data required to perform matrix reordering in managed memory
    - The GPU accesses data stored in managed memory during stage 1 and 4
    - The CPU accesses data stored in managed memory during stage 2 and 3
- Goals:
  - Simplify code
  - Don’t hurt performance
Matrix Reordering Example [Part 1/3]: Sparsity Pattern of Original Matrix

- Garon2: sparse matrix from Florida collection
- Dimension of the sparse matrix
  - \( N = 13535 \)
- Number of nonzero entries in the matrix
  - \( \text{NNZ} = 373235 \)
- Half bandwidth of the matrix
  - \( K_{\text{ori}} = 13531 \)
Matrix Reordering Example [Part 2/3]: Sparsity Pattern after the four-stage reordering

- This is the algorithm of interest, gauge overhead associated with use of UM
  - Timing results are presented shortly

- Half bandwidth after reordering:
  - $K_{4s} = 13531$

- Bandwidth still large, but all diagonal entries are nonzero and large in magnitude
Matrix Reordering Example [Part 3/3]: Sparsity Pattern after reverse Cuthill-Mckee (RCM)

- All non-zeros of the matrix now close to the diagonal
  - Small half bandwidth:
    - $K_{RCM} = 585$

- This matrix is passed to the SPIKE::GPU preconditioner to produce an LU factorization
  - GPU LU factorization details not discussed here
The Key Slide: Performance Comparison

- Ran more than 120 sparse matrix reordering tests
  - Ranked from the best to the worst CUDA 6 performance
  - Took every tenth matrix in this ranking, see table
- Times reported were measured in milliseconds
  - Time that it takes to get the matrix with K_4s
- Bottom line: using UM and letting the runtime take care of business never resulted in more than a 25% slowdown over hand-tuned code
- Almost half of the tests ran faster
  - See “Speedup” column, which reports nominal time divided by time using UM in CUDA 6

<table>
<thead>
<tr>
<th>Name</th>
<th>Dim.</th>
<th>NNZ</th>
<th>Time Basic</th>
<th>Time UM</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>poisson3Db</td>
<td>85623</td>
<td>2374949</td>
<td>120.293</td>
<td>86.068</td>
<td>1.397651</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>36417</td>
<td>4344765</td>
<td>153.276</td>
<td>128.269</td>
<td>1.194957</td>
</tr>
<tr>
<td>inline_1</td>
<td>503712</td>
<td>36816342</td>
<td>1299.3</td>
<td>1129.12</td>
<td>1.150719</td>
</tr>
<tr>
<td>qa8fk</td>
<td>66127</td>
<td>1660579</td>
<td>62.088</td>
<td>55.216</td>
<td>1.124457</td>
</tr>
<tr>
<td>finan512</td>
<td>74752</td>
<td>596992</td>
<td>29.526</td>
<td>29.155</td>
<td>1.012725</td>
</tr>
<tr>
<td>lhr71</td>
<td>70304</td>
<td>1528092</td>
<td>726.646</td>
<td>762.697</td>
<td>0.952732</td>
</tr>
<tr>
<td>g7jac140</td>
<td>41490</td>
<td>565956</td>
<td>701.929</td>
<td>761.265</td>
<td>0.922056</td>
</tr>
<tr>
<td>ASIC_100k</td>
<td>99340</td>
<td>954163</td>
<td>51.512</td>
<td>56.924</td>
<td>0.904926</td>
</tr>
<tr>
<td>gearbox</td>
<td>153746</td>
<td>9080404</td>
<td>1257.32</td>
<td>1424.52</td>
<td>0.882627</td>
</tr>
<tr>
<td>rma10</td>
<td>46835</td>
<td>2374001</td>
<td>211.592</td>
<td>252.221</td>
<td>0.838915</td>
</tr>
<tr>
<td>bmw3_2</td>
<td>227362</td>
<td>11288630</td>
<td>741.092</td>
<td>911.724</td>
<td>0.812847</td>
</tr>
<tr>
<td>stomach</td>
<td>213360</td>
<td>3021648</td>
<td>174.105</td>
<td>226.585</td>
<td>0.768387</td>
</tr>
</tbody>
</table>
Where Else Are We Using CUDA and GPU Computing?

- Fluid-solid interaction problems
- Many-body dynamics, for rigid or flexible bodies
  - Example: dynamics granular material
Interacting rigid and flexible objects in channel flow

Fluid:
\[ \rho = 1000 \text{ kg/m}^3 \]
\[ \mu = 1 \text{ N s/m}^2 \]
\[ (l_x, l_y, l_z) = (1.4, 1, 1) \text{ m} \]
\[ Re = 45 \]

Ellipsoids:
\[ \rho_s = 1000 \text{ kg/m}^3 \]
\[ (a_1, a_2, a_3) = (2.25, 2.25, 3) \text{ cm} \]
\[ N_r = 2000 \]
\[ Re_p = 2 \]

Beams:
\[ \rho_s = 1000 \text{ kg/m}^3 \]
\[ E = 0.2 \text{ MPa} \]
\[ a = 1.5 \text{ cm} \]
\[ l = 64 \text{ cm} \]
\[ N_f = 40 \]
\[ n_e = 4 \]
Scaling analysis (all together, table)

<table>
<thead>
<tr>
<th>$N_m \times 10^6$</th>
<th>0.08</th>
<th>0.16</th>
<th>0.29</th>
<th>0.63</th>
<th>0.95</th>
<th>1.54</th>
<th>2.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_r \times 10^3$</td>
<td>0.17</td>
<td>0.52</td>
<td>1.12</td>
<td>4.48</td>
<td>7.84</td>
<td>14.56</td>
<td>24.64</td>
</tr>
<tr>
<td>$N_f \times 10^3$</td>
<td>0.16</td>
<td>0.42</td>
<td>0.84</td>
<td>2.10</td>
<td>3.36</td>
<td>5.88</td>
<td>9.66</td>
</tr>
<tr>
<td>$t$ (ms)</td>
<td>45</td>
<td>74</td>
<td>120</td>
<td>230</td>
<td>343</td>
<td>522</td>
<td>820</td>
</tr>
</tbody>
</table>

$N_m \approx 3.0 \times 10^6$, $N_f = 0$

<table>
<thead>
<tr>
<th>$N_r$</th>
<th>0</th>
<th>36</th>
<th>120</th>
<th>480</th>
<th>1800</th>
<th>8400</th>
<th>33600</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$ (ms)</td>
<td>906</td>
<td>919</td>
<td>923</td>
<td>925</td>
<td>926</td>
<td>926</td>
<td>921</td>
</tr>
</tbody>
</table>

$N_m \approx 3.0 \times 10^6$, $N_r = 0$

<table>
<thead>
<tr>
<th>$N_f$</th>
<th>0</th>
<th>45</th>
<th>140</th>
<th>440</th>
<th>1152</th>
<th>2100</th>
<th>4704</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = 10$</td>
<td>906</td>
<td>923</td>
<td>928</td>
<td>916</td>
<td>960</td>
<td>950</td>
<td>921</td>
</tr>
<tr>
<td>$t = 50$</td>
<td>906</td>
<td>973</td>
<td>978</td>
<td>965</td>
<td>1066</td>
<td>1060</td>
<td>1060</td>
</tr>
</tbody>
</table>

Simulation time per step (s)

Number of SPH markers ($\times 10^6$)

Number of solid objects ($\times 10^3$)