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

Darmstadt University
6-10 October
2014
Before we get started

- Yesterday
  - Discussed parallel computing issues
    - Intel Haswell, 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
  - Started discussion of CUDA API
Before we get started

● Today
  ● Wrap up CUDA API discussion
  ● Discuss
    ● The NVIDIA GPU memory ecosystem
    ● Scheduling issues
    ● Atomic operations
    ● Optimization issues

● Follow up on the hands-on component
Why Learn More about GPUs?

- Hone our “Computational Thinking” skills

- “Computational Thinking” cannot be built without
  - Working on our 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 the correct results fast
The CUDA API
What is an API?

- Application Programming Interface (API)
  - “A set of functions, procedures or classes that an operating system, library, or service provides to support requests made by computer programs” (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA’s GPUs
  - This is what replaced old GPGPU way of programming the hardware
  - CUDA API exposed to you through a collection of header files that you include in your program
On the CUDA API

- Reading the CUDA Programming Guide you’ll run into numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about “CUDA runtime” and “CUDA driver”. What they mean is CUDA Runtime API and CUDA Driver API

- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with “C CUDA”
  - Needs `nvcc` compiler to generate an executable

- CUDA Driver API – low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly more clunky way to dialogue with the GPU, typically only needs a C compiler

- I don’t anticipate any reason to use the CUDA Driver API
Talking about the API: The C CUDA Software Stack

- Image at right indicates where the API fits in the picture

An API layer is indicated by a thick red line:

- NOTE: any CUDA runtime function has a name that starts with “cuda”
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.

- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.
Application Programming Interface (API)
~Taking a Step Back~

- CUDA runtime API: exposes a set of extensions to the C language
  - Spelled out in an appendix of “NVIDIA CUDA C Programming Guide”
  - There is many of them → Keep in mind the 20/80 rule

- CUDA runtime API:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library, which is split into:
    - A common component providing built-in vector types and a subset of the C runtime library available in both host and device codes
      - Callable both from device and host
    - A host component to control and access devices from the host
      - Callable from the host only
    - A device component providing device-specific functions
      - Callable from the device only
Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable Type Qualifiers</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong></td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong></td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong></td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__
- **Automatic variables** without any qualifier reside in a **register**
  - **Except arrays**, which reside in local memory (unless they are small and of known constant size)
Common Runtime Component

- “Common” above refers to functionality that is provided by the CUDA API and is common both to the device and host.

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes.
Common Runtime Component: Built-in Vector Types

- `[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4], double[1..2]`
  - Structures accessed with `x, y, z, w` fields:
    ```c
    uint4 param;
    int dummy = param.y;
    ```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - You see a lot of it when defining the execution configuration of a kernel (any component left uninitialized assumes default value 1)

See Appendix B in “NVIDIA CUDA C Programming Guide”
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- etc.

  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions only supported for scalar types, not vector types
Host Runtime Component

- Provides functions available only to the host to deal with:
  - **Device** management (including multi-device systems)
  - **Memory** management
  - **Error** handling

- **Examples**
  - **Device memory allocation**
    - cudaMalloc(), cudaFree()
  - **Memory copy from host to device, device to host, device to device**
    - cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()
  - **Memory addressing** – returns the address of a device variable
    - cudaMemcpySymbolAddress()
CUDA Device Memory Space Overview
[Quick Overview of GPU Memory Ecosystem]

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

- The host can R/W global, constant, and texture memory
CUDA API: Device Memory Allocation
[Note: picture assumes two blocks, each with two threads]

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- cudaFree()
  - Frees object from device Global Memory
  - Pointer to freed object
Example
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to \texttt{Md.elements}
  - “d” in “Md” is often used to indicate a device data structure

```c
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
...
//use it for what you need, then free the device memory
cudaFree(Md.elements);
```

**Question**: why is the type of the first argument \texttt{(void **)}?
CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a $64 \times 64$ single precision float array
  - $M$ is in host memory and $Md$ is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` are symbolic constants

```c
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
```
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. \( \sin(x) \)) have a less accurate, but faster device-only version (e.g. \( \_\_\sin(x) \))
  - \( \_\_\text{pow} \)
  - \( \_\_\log, \_\_\log2, \_\_\log10 \)
  - \( \_\_\exp \)
  - \( \_\_\sin, \_\_\cos, \_\_\tan \)
- Some of these have hardware implementations
- By using the “-use_fast_math” flag, \( \sin(x) \) is substituted at compile time by \( \_\_\sin(x) \)

\[
\text{>> nvcc } -\text{arch=sm}_20 -\text{use_fast_math} \text{ foo.cu}
\]
CPU vs. GPU – Flop Rate (GFlops)
End API discussion
…… transitioning into…
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.
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 physically exist – it’s an abstraction to the local scope of a thread.
- 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 – resident</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A – resident</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 if not 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

- Thread
- Grid 0
- Grid 1
- Sequential Grids in Time
- Global Memory
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
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
  - No sharing of data but why not use it? Don’t let it go wasted…
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:** \( A \) and \( B \) are *square matrices* and their dimensions of 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 1:** Similar example provided in the CUDA Programming Guide 3.2
- Available on the 2011 class website

**NOTE 2:** A similar technique is used on CPUs to improve cache hits. See slide “Blocking Example” at
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);
}
First entry of the tile

(number of tiles along the width of B)

(number of tiles down the height of A)
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Mul(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
    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;
}
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:
    - 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:

\[
400/(6 \times 4) = 16.6667 \rightarrow 17 \text{ Warps}
\]
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?
- $15 \text{ SMs} \times 32 \text{ SPs} \times 1215 \times 2 \text{ (Fused Multiplied Add)} = 1166400 \text{ Mflops}$
- 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).

```
<table>
<thead>
<tr>
<th>Warp Scheduler</th>
<th>Warp Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Dispatch Unit</td>
<td>Instruction Dispatch Unit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUDA Cores (x16)</th>
<th>CUDA Cores (x16)</th>
<th>SFUs (x4)</th>
<th>LD/ST Units (x16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>FADD</td>
<td>RCP</td>
<td>LD</td>
</tr>
<tr>
<td>FFMA</td>
<td>FFMA</td>
<td>SIN</td>
<td>LD</td>
</tr>
<tr>
<td>IADD</td>
<td>IADD</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>FFMA</td>
<td>ICMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFMA</td>
<td>IADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFMA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Past

Present

Time

Past

Present
## NVIDIA Architecture Specifications

<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]}</td>
</tr>
<tr>
<td>Number of special function units for single-precision floating-point transcendental functions</td>
<td>2</td>
</tr>
<tr>
<td>Number of texture filtering units for every texture address unit or render output unit (ROP)</td>
<td>2</td>
</tr>
<tr>
<td>Number of warp schedulers</td>
<td>1</td>
</tr>
<tr>
<td>Number of instructions issued at once by scheduler</td>
<td>1</td>
</tr>
</tbody>
</table>
Threads are Executed in Warps

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

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

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

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

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

- Prefer thread block sizes that result in mostly full warps
  
  **Bad:**  kernel<<<N, 1>>>( ... )
  **Okay:** kernel<<<(N+31) / 32, 32>>>( ... )
  **Better:** 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 __shared__ may constrain number of threads per block

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

- When 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, 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, 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></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></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></td>
</tr>
</tbody>
</table>

[Wikipedia]→
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
Thread Divergence

Consider the following code:

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

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

[2/4]

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

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

[2/2]

- 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:

  ```
  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

NVIDIA [J. Balfour]→
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
Two aspects of global memory access are relevant when fetching data into shared memory and/or registers:

- The layout of the access to global memory (the pattern of the access)
- The size/alignment of the data you try to fetch from global memory
“Memory Access Layout”

What is it?

- The basic idea:
  - Suppose each thread in a warp accesses a global memory address for a load operation at some point in the execution of the kernel
  - These threads can access global memory data that is either (a) neatly grouped, or (b) scattered all over the place
  - Case (a) is called a “coalesced memory access”
    - If you end up with (b) this will adversely impact the overall program performance

- Analogy
  - Can send one truck on six different trips to bring back each time a bundle of wood
  - Alternatively, can send truck to one place and get it back fully loaded with wood
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-warp, if the size is 16 bytes.

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

- NOTE: a cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise
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**

  ![Aligned and non-sequential diagram]

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>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>
</tr>
<tr>
<td></td>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Scenario C: Misaligned and sequential**

  ![Misaligned and sequential diagram]

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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></td>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></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.
Don’t Do This at Home

- Assume thread T1 reads a value defined by thread T0

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

// main.cpp
update_race<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Program needs to ensure that thread T1 reads location after thread T0 has written location
Synchronization within Block

- Threads in same block: can use `__syncthreads()` to specify synchronization point that orders accesses.

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

// main.cpp
update<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Here's a fun question: would this work if the kernel is launched with an execution configuration that has two blocks?
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.
Atoms
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

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 = *x+i;
}

// 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

[2/3]

- Atomic functions perform read-modify-write operations on data that can reside in global or shared memory.
- Synopsis of atomic function $\text{atomicOP}(a,b)$ is typically

```
t1 = *a;       // read
t2 = (*a) OP (*b); // modify
*a = t2;       // write
return t1;
```

- 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
A Histogram Example

// 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

- 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>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>3</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>64</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</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>16 KB</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
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.
**CUDA GPU Occupancy Calculator**

Just follow steps 1, 2, and 3 below (for click for help):

1. Select Compute Capability (click): 2.0

2. Enter your resource usage:
   - Threads Per Block: 128
   - Registers Per Thread: 64
   - Shared Memory Per Block (bytes): 128

3. GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multithread: 512
   - Active Warps per Multithread: 16
   - Occupancy of each Multithread: 33%

Physical limits for GPU Compute Capability: 2.0

- Threads per Warp: 32
- Warps per Multithread: 16
- Threads per Multithread: 1636
- Blocks per Multithread: 0
- Total # of 32-bit registers per Multithread: 32768
- Warp allocation unit size: 64
- Warp allocation granularity: warp
- Shared Memory per Multithread (bytes): 49152
- Shared Memory Allocation Unit size: 128
- Warp allocation granularity (for register allocation): 0

These data are used in computing the occupancy data in blue:

- Maximum Thread Blocks Per Multithread: 8
- Limited by Warps / Blocks per Multithread: 8
- Limited by Registers per Multithread: 0
- Limited by Shared Memory per Multithread: 0
- Thread Block Limit Per Multithread highlighted: RED

CUDA Occupancy Calculator

Version: 2.1

Click here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit 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.
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