CUDA

Digging further into
the programming manual

Schedule

- Application Programming Interface (API)
  - text only part, sorry
- Image utilities (simple CUDA examples)
- Performace considerations
- Matrix multiplication example
- SDK “browse-through”

API

- A minimal set of extensions to the C language that allow the programmer to target portions of the source code for execution on the device
- Runtime library
  - A host component that provides functions to control and access one or more compute devices from the host, e.g. memory allocation.
  - A device component that provides device-specific functions, e.g. thread synchronization.
  - A common component that provides built-in vector types and a subset of the C standard library that are supported in both host and device code.

Language extensions

- Roughly speaking only four additions to standard C
  - Function type qualifiers to specify whether a function executes on the host or on the device
  - Variable type qualifiers to specify the memory location on the device
  - A new directive to specify how a kernel is executed on the device
  - Four built-in variables that specify the grid and block dimensions and the block and thread indices

nvcc

- CUDA compiler to handle the standard C extensions
  - Each source file containing these extensions must be compiled with the CUDA compiler **nvcc**
  - Easily integrated with Visual Studio in Windows or makefiles in Linux

Function type qualifiers (1)

- **__device__**
  - The **__device__** qualifier declares a function that is
    - Executed on the device
    - Callable from the device only

Function type qualifiers (2)

- **__global__**
  - The **__global__** qualifier declares a function as being a kernel
    - Executed on the device
    - Callable from the host only

Function type qualifiers (3)

- **__host__**
  - The **__host__** qualifier declares a function that is
    - Executed on the host
    - Callable from the host only
    - Default if no function qualifier is specified
    - Can be used in combination with __device__

Function type qualifiers - restrictions

- **__device__** functions are always inlined
- **__device__** and **__global__** do not support recursion
- **__device__** and **__global__** cannot declare static variables inside their bodies
- **__device__** and **__global__** cannot have a variable number of inputs
- **__device__** functions cannot have their address taken but function pointers to **__global__** functions are supported
- The **__global__** and **__host__** qualifiers cannot be used together
- **__global__** functions must have void return type
- Any call to a **__global__** function must specify its execution configuration
- A call to a **__global__** function is asynchronous, meaning it returns before the device has completed its execution
- **__global__** function parameters are currently passed via shared memory to the device and limited to 256 bytes.

Variable type qualifiers (1)

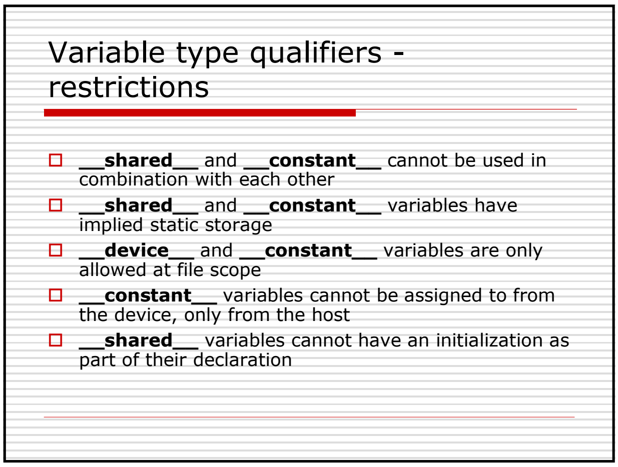
- **__device__**
  - The **__device__** qualifier declares a variable that resides on the device
    - Resides in global memory space
    - Has the lifetime of an application
    - Is accessible from all the threads within the grid and from the host through the runtime library

Variable type qualifiers (2)

- **__constant__**
  - The **__constant__** qualifier declares a variable that resides on the device
    - Resides in constant memory space
    - Has the lifetime of an application
    - Is accessible from all the threads within the grid and from the host through the runtime library

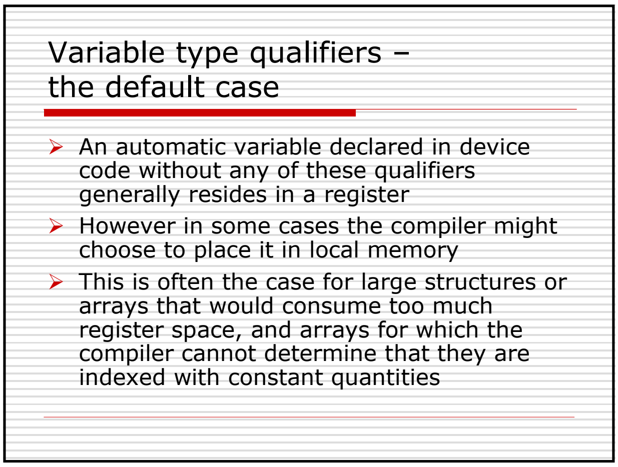
Variable type qualifiers (3)

- **__shared__**
  - The **__shared__** qualifier declares a variable that resides on the device
    - Resides in the shared memory space of a thread block
    - Has the lifetime of the block
    - Is only accessible from threads within the block



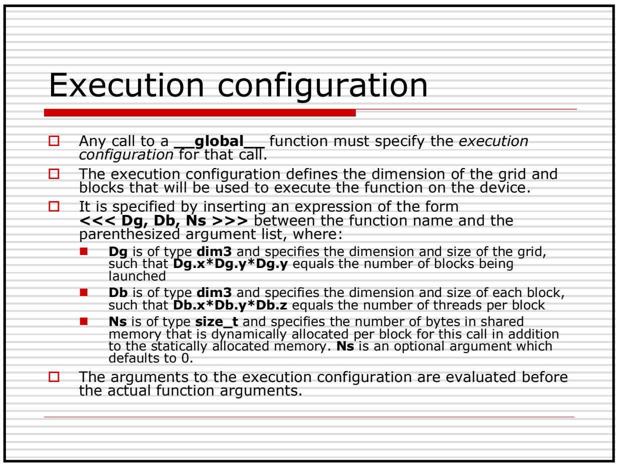
Variable type qualifiers - restrictions

- **__shared__** and **__constant__** cannot be used in combination with each other
- **__shared__** and **__constant__** variables have implied static storage
- **__device__** and **__constant__** variables are only allowed at file scope
- **__constant__** variables cannot be assigned to from the device, only from the host
- **__shared__** variables cannot have an initialization as part of their declaration



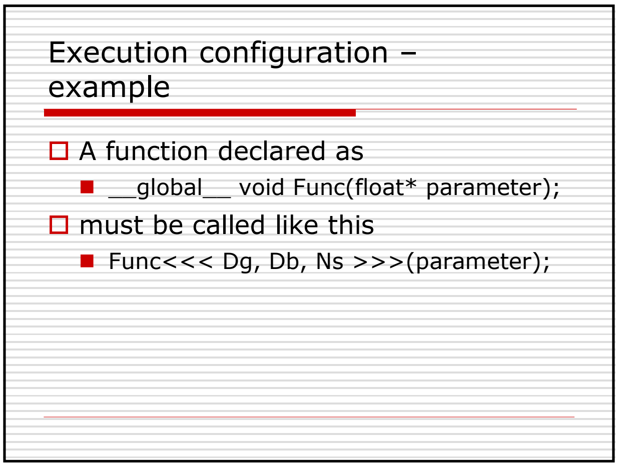
Variable type qualifiers – the default case

- An automatic variable declared in device code without any of these qualifiers generally resides in a register
- However in some cases the compiler might choose to place it in local memory
- This is often the case for large structures or arrays that would consume too much register space, and arrays for which the compiler cannot determine that they are indexed with constant quantities



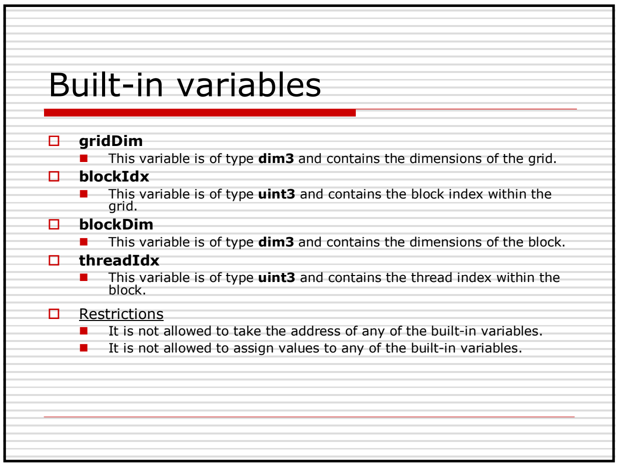
Execution configuration

- Any call to a **__global__** function must specify the *execution configuration* for that call.
- The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device.
- It is specified by inserting an expression of the form **<<< Dg, Db, Ns >>>** between the function name and the parenthesized argument list, where:
  - **Dg** is of type **dim3** and specifies the dimension and size of the grid, such that **Dg.x*Dg.y*Dg.y** equals the number of blocks being launched
  - **Db** is of type **dim3** and specifies the dimension and size of each block, such that **Db.x*Db.y*Db.z** equals the number of threads per block
  - **Ns** is of type **size_t** and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory. **Ns** is an optional argument which defaults to 0.
- The arguments to the execution configuration are evaluated before the actual function arguments.



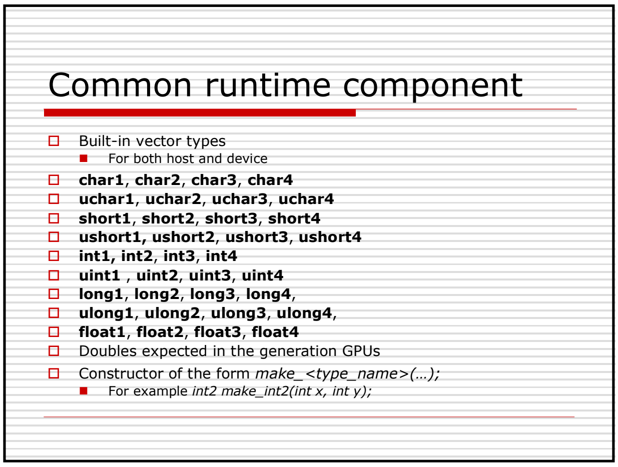
Execution configuration – example

- A function declared as
  - __global__ void Func(float* parameter);
- must be called like this
  - Func<<< Dg, Db, Ns >>>(parameter);



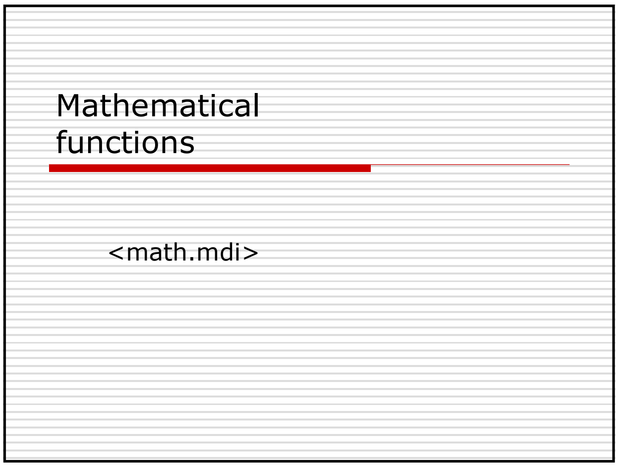
Built-in variables

- **gridDim**
  - This variable is of type **dim3** and contains the dimensions of the grid.
- **blockIdx**
  - This variable is of type **uint3** and contains the block index within the grid.
- **blockDim**
  - This variable is of type **dim3** and contains the dimensions of the block.
- **threadIdx**
  - This variable is of type **uint3** and contains the thread index within the block.
- Restrictions
  - It is not allowed to take the address of any of the built-in variables.
  - It is not allowed to assign values to any of the built-in variables.



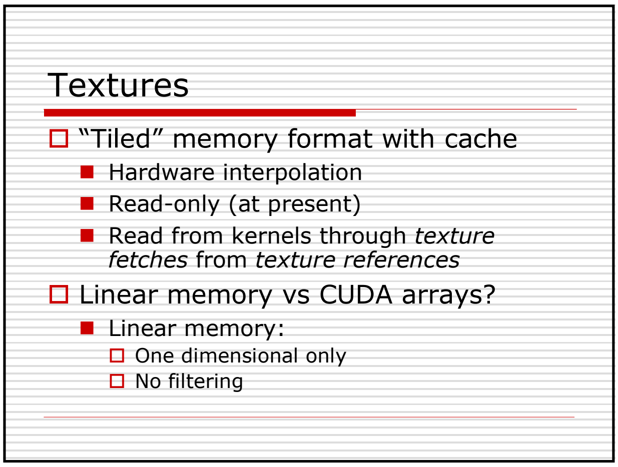
Common runtime component

- Built-in vector types
  - For both host and device
- **char1**, **char2**, **char3**, **char4**
- **uchar1**, **uchar2**, **uchar3**, **uchar4**
- **short1**, **short2**, **short3**, **short4**
- **ushort1**, **ushort2**, **ushort3**, **ushort4**
- **int1**, **int2**, **int3**, **int4**
- **uint1** , **uint2**, **uint3**, **uint4**
- **long1**, **long2**, **long3**, **long4**,
- **ulong1**, **ulong2**, **ulong3**, **ulong4**,
- **float1**, **float2**, **float3**, **float4**
- Doubles expected in the generation GPUs
- Constructor of the form *make_<type_name>(…);*
  - For example *int2 make_int2(int x, int y);*



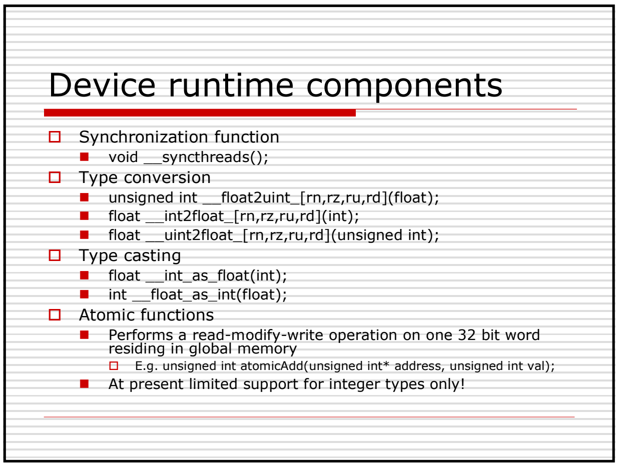
Mathematical functions

<math.mdi>



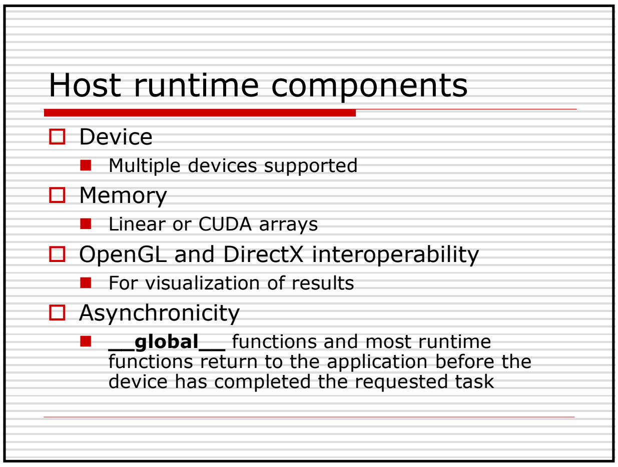
Textures

- □ "Tiled" memory format with cache
  - Hardware interpolation
  - Read-only (at present)
  - Read from kernels through *texture fetches* from *texture references*
- □ Linear memory vs CUDA arrays?
  - Linear memory:
    - □ One dimensional only
    - □ No filtering



Device runtime components

- □ Synchronization function
  - void __syncthreads();
- □ Type conversion
  - unsigned int __float2uint_[rn,rz,ru,rd](float);
  - float __int2float_[rn,rz,ru,rd](int);
  - float __uint2float_[rn,rz,ru,rd](unsigned int);
- □ Type casting
  - float __int_as_float(int);
  - int __float_as_int(float);
- □ Atomic functions
  - Performs a read-modify-write operation on one 32 bit word residing in global memory
    - □ E.g. unsigned int atomicAdd(unsigned int* address, unsigned int val);
  - At present limited support for integer types only!



Host runtime components

- □ Device
  - Multiple devices supported
- □ Memory
  - Linear or CUDA arrays
- □ OpenGL and DirectX interoperability
  - For visualization of results
- □ Asynchronicity
  - **__global__** functions and most runtime functions return to the application before the device has completed the requested task

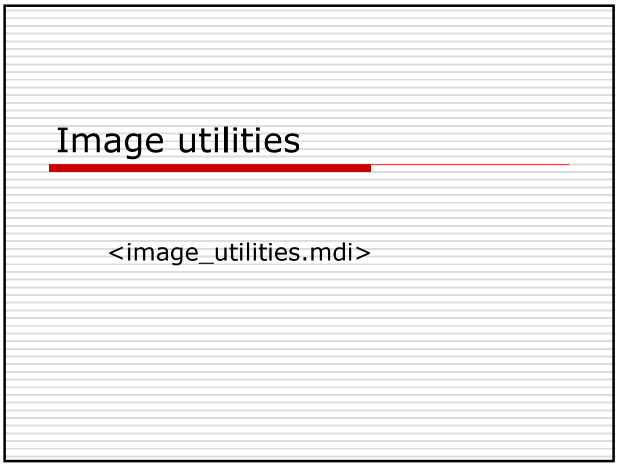
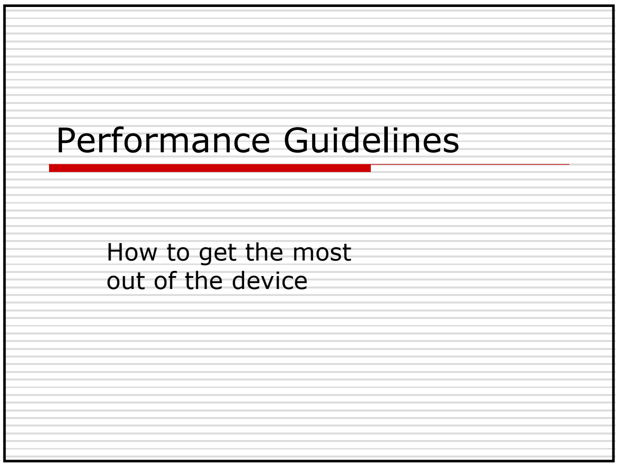


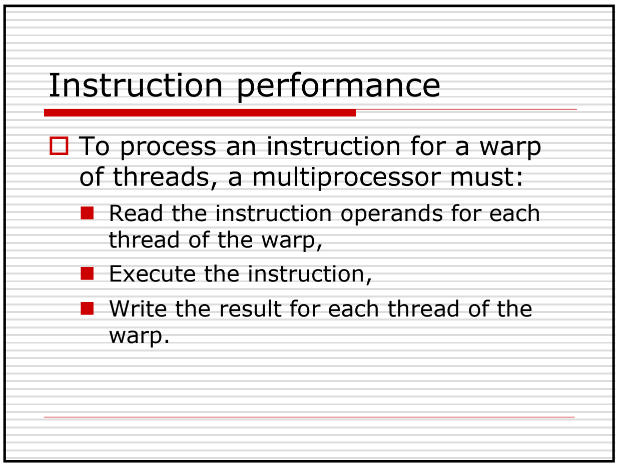
Image utilities

<image_utilities.mdi>



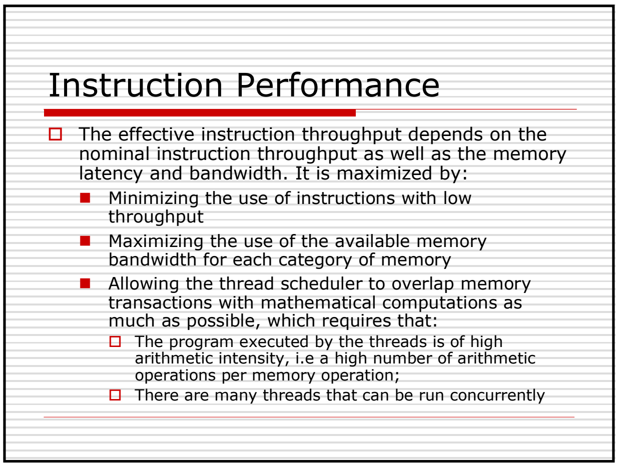
Performance Guidelines

How to get the most
out of the device



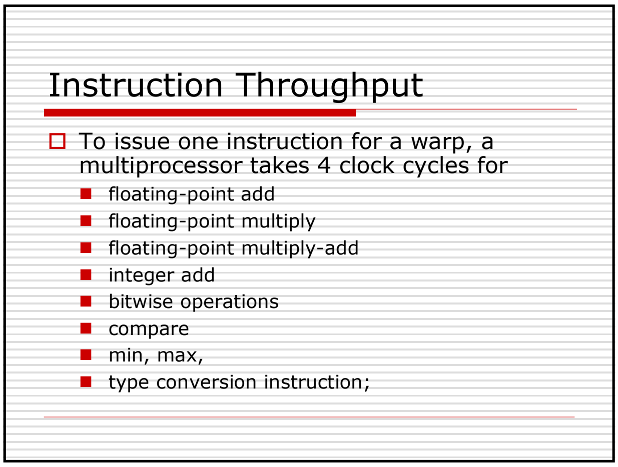
Instruction performance

- To process an instruction for a warp of threads, a multiprocessor must:
  - Read the instruction operands for each thread of the warp,
  - Execute the instruction,
  - Write the result for each thread of the warp.



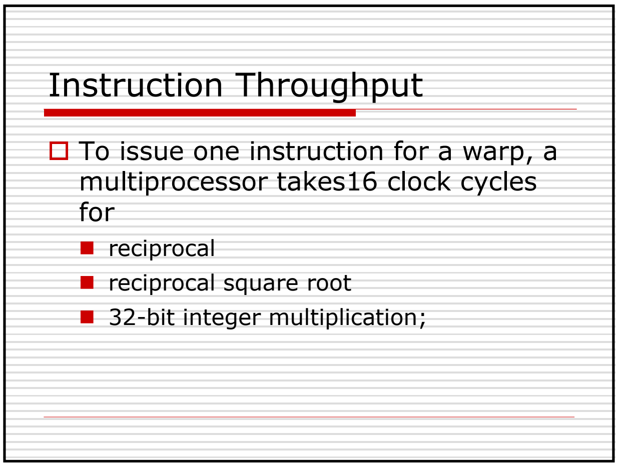
Instruction Performance

- The effective instruction throughput depends on the nominal instruction throughput as well as the memory latency and bandwidth. It is maximized by:
  - Minimizing the use of instructions with low throughput
  - Maximizing the use of the available memory bandwidth for each category of memory
  - Allowing the thread scheduler to overlap memory transactions with mathematical computations as much as possible, which requires that:
    - The program executed by the threads is of high arithmetic intensity, i.e a high number of arithmetic operations per memory operation;
    - There are many threads that can be run concurrently



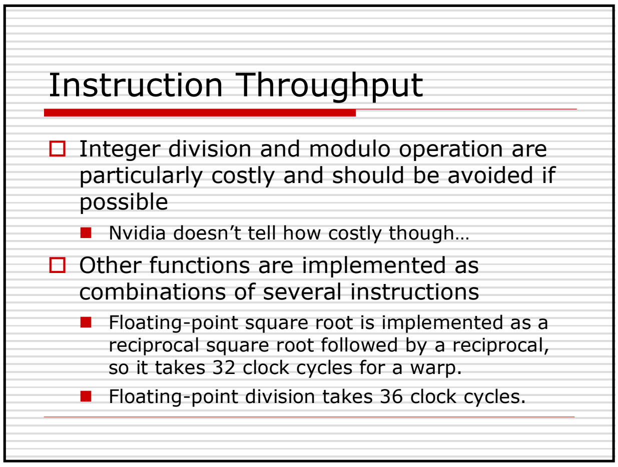
Instruction Throughput

- To issue one instruction for a warp, a multiprocessor takes 4 clock cycles for
  - floating-point add
  - floating-point multiply
  - floating-point multiply-add
  - integer add
  - bitwise operations
  - compare
  - min, max,
  - type conversion instruction;



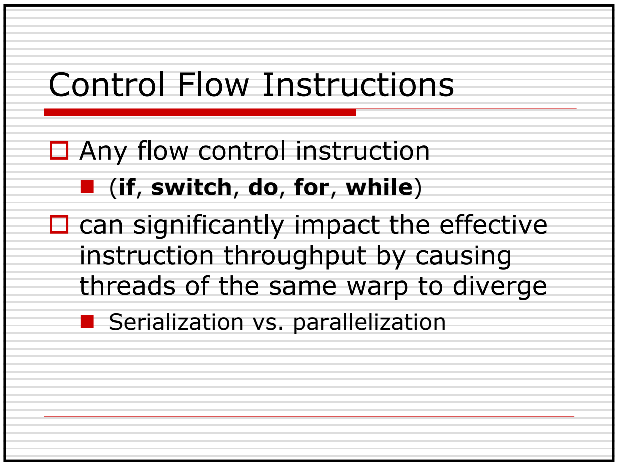
Instruction Throughput

- To issue one instruction for a warp, a multiprocessor takes16 clock cycles for
  - reciprocal
  - reciprocal square root
  - 32-bit integer multiplication;



Instruction Throughput

- Integer division and modulo operation are particularly costly and should be avoided if possible
  - Nvidia doesn’t tell how costly though…
- Other functions are implemented as combinations of several instructions
  - Floating-point square root is implemented as a reciprocal square root followed by a reciprocal, so it takes 32 clock cycles for a warp.
  - Floating-point division takes 36 clock cycles.



Control Flow Instructions

- Any flow control instruction
  - (**if**, **switch**, **do**, **for**, **while**)
- can significantly impact the effective instruction throughput by causing threads of the same warp to diverge
  - Serialization vs. parallelization

Memory Instructions

- □ Memory instructions include any instruction that reads from or writes to shared or global memory.
- □ A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp.
- □ When accessing global memory, there are, in addition, 400 to 600 clock cycles of memory latency!!!

Memory Bandwidth – Global Memory

- □ Uncached
  - ■ One can fetch 32-bit, 64-bit, or 128-bit words into registers in a single instruction
- □ Use coalescence whenever possible

Memory Bandwidth – Constant Memory

- □ Cached
  - ■ Only at cache misses does it cost a read from device memory

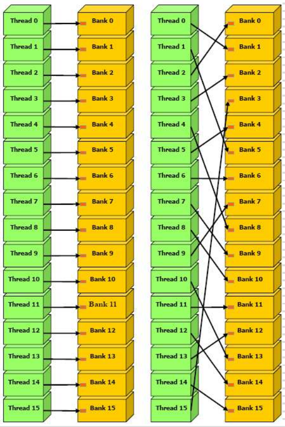
Memory Bandwidth – Texture Memory

- □ Cached
  - ■ Only at cache misses does it cost a read from device memory
  - ■ The texture cache is optimized for 2D spatial locality
  - ■ BUT there is a cost associated to copying into the a CUDA array

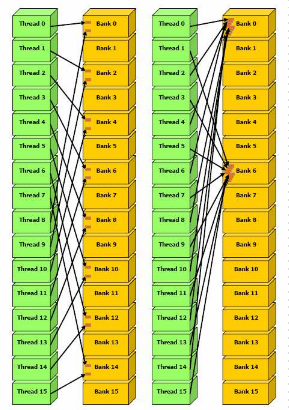
Memory Bandwidth – Shared Memory

- □ On-chip
  - ■ As fast as registers
  - ■ Avoid bank conflicts
    - □ shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously
    - □ any memory read or write request made of *n* addresses that fall in *n* distinct memory banks can be serviced simultaneously

Shared Memory – no bank conflicts



2-way and 8-way bank conflicts



Memory Bandwidth – Registers

- On-chip
  - Generally, accessing a register is zero extra clock cycles per instruction

Number of threads per block

- There should be at least as many blocks as there are multiprocessors in the device.
- Running only one block per multiprocessor can force the multiprocessor to idle during thread synchronization and device memory reads.
- The shared memory available to a block decreases with the number of active blocks
- The number of threads per block should be chosen as a multiple of the warp size!!!

Number of threads per block

- Allocating more threads per block is better for efficient time slicing, but the more threads per block, the fewer registers are available per thread.
- CUDA occupancy calculator

Current generation hardware

<G8x_specs.mdi>

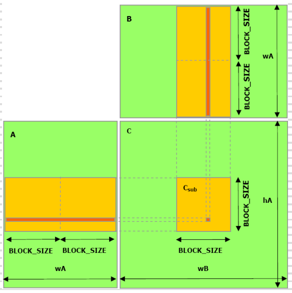
Matrix multiplication

An examle from the CUDA programming guide

Ax=b

- Computing the product *C* of two matrices *A* and *B* of dimensions *(wA, hA)* and *(wB, wA)*
  - Each thread block is responsible for computing one square sub-matrix C_{sub} of *C*;
  - Each thread within the block is responsible for computing one element of C_{sub}.

Split matrix into blocks



CUDA programming guide Fig. 6-1.

Matrix multiplication code

<mat_mult.mdi>

Browse-through SDK

As time permits…