GPU Parallel Computing

Sources and References

- http://en.wikipedia.org/wiki/CUDA
- http://www.computer.org/cms/Computer.org/ComputingNow/homepage/2011/0211/T_MI_GPUComputingEra.pdf
- http://www.computer.org/cms/Computer.org/ComputingNow/homepage/2011/0211/T_SW_JointForces.pdf
- Slides from David Kirk/NVIDIA and Wen-mei W. Hwu, 2007, ECE 498AL, University of Illinois, Urbana-Champaign
- http://www.caam.rice.edu/~timwar/RMMC/CUDA.html

What and Why

- Processors (GPU's) originally intended to accelerate graphics (point transformations, shading, lighting, …)
- Become more and more capable
- Until they become attractive as **General Purpose** compute engines (GPGPU vs. CPU)
- Offering very high performance at low cost

Main Hardware Platforms, 2011

- AMD ATI Radeon series*
- Nvidia GeForce series
- Related: IBM Cell processor

* ATI was acquired by AMD.

Nvidia GPU Timeline

Table 1. NVIDIA GPU technology development.

Date	Product	Transistors	CUDA cores	Technology
1997	RIVA 128	3 million	—	3D graphics accelerator
1999	GeForce 256	25 million	—	First GPU, programmed with DX7 and OpenGL
2001	GeForce 3	60 million	—	First programmable shader GPU, programmed with DX8 and OpenGL
2002	GeForce FX	125 million	—	32-bit floating-point (FP) programmable GPU with Cg programs, DX9, and OpenGL
2004	GeForce 6800	222 million	—	32-bit FP programmable scalable GPU, GPGPU Cg programs, DX9, and OpenGL
2006	GeForce 8800	681 million	128	First unified graphics and computing GPU, programmed in C with CUDA
2007	Tesla T8, C870	681 million	128	First GPU computing system programmed in C with CUDA
2008	GeForce GTX 280	1.4 billion	240	Unified graphics and computing GPU, IEEE FP, CUDA C, OpenCL, and DirectCompute
2008	Tesla T10, S1070	1.4 billion	240	GPU computing clusters, 64-bit IEEE FP, 4-Gbyte memory, CUDA C, and OpenCL
2009	Fermi	3.0 billion	512	GPU computing architecture, IEEE 754-2008 FP, 64-bit unified addressing, caching, ECC memory, CUDA C, C++, OpenCL, and DirectCompute

Performance: GPU vs. CPU

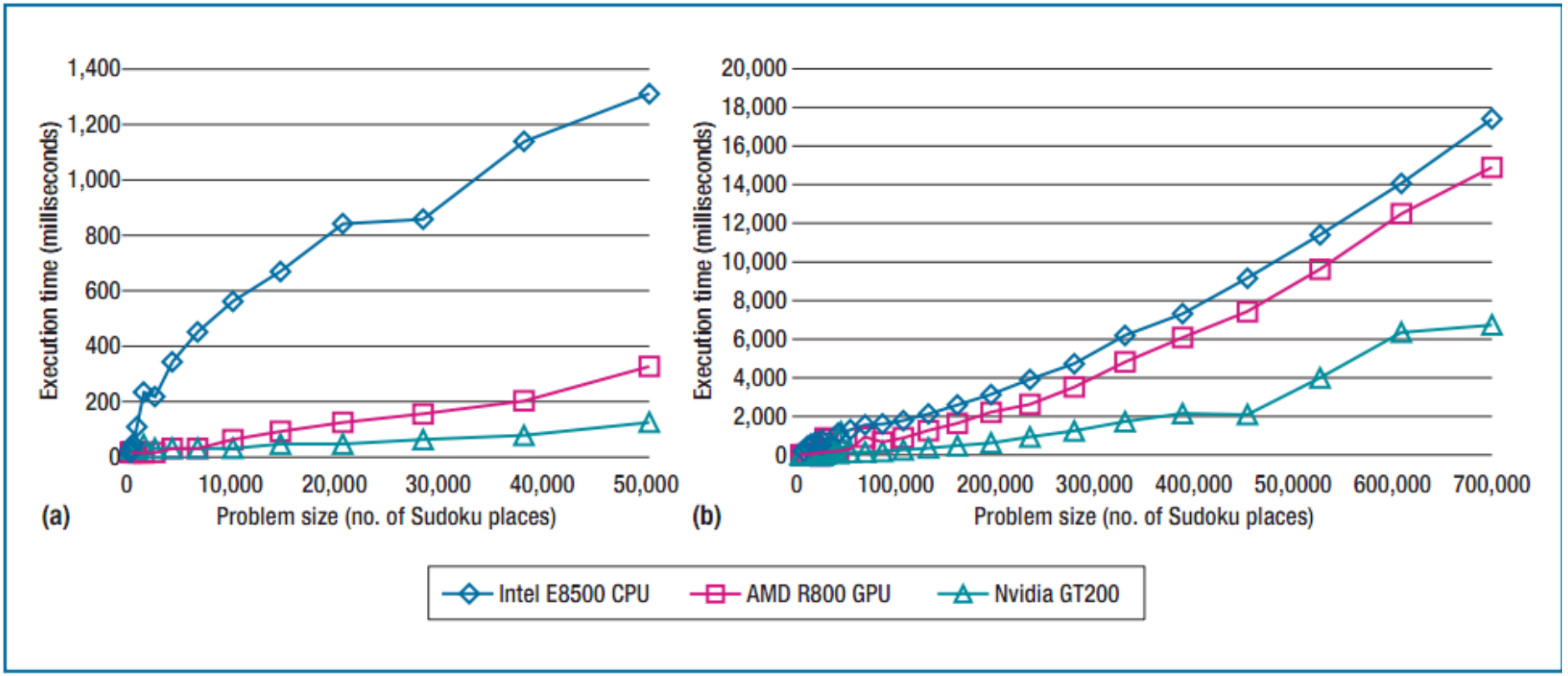


FIGURE 3. Execution time of the Sudoku validation on different compute devices: (a) problem size of 10,000 to 50,000 possible Sudoku places and (b) problem size of 100,000 to 700,000 Sudoku places.

Co-Processor

- Despite being general purpose, GPGPUs are used as a **co-processor** in conjunction with a CPU.
- Communication is asynchronous.
- Terminology:

  Host = the CPU

  Device = the GPU

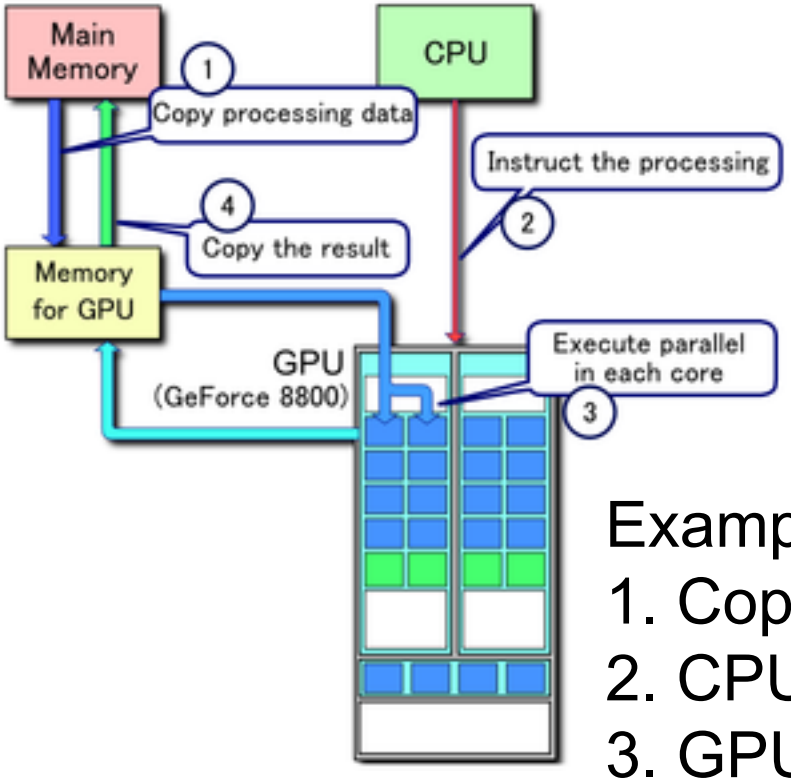
Software Platforms

- CUDA - Limited to Nvidia GPU’s
- OpenCL - Either vendor. Developed by Apple

CUDA

- CUDA =
  Compute Unified Device Architecture
- It is a combined hardware and software architecture.
- It applies to nVidia GeForce GPUs

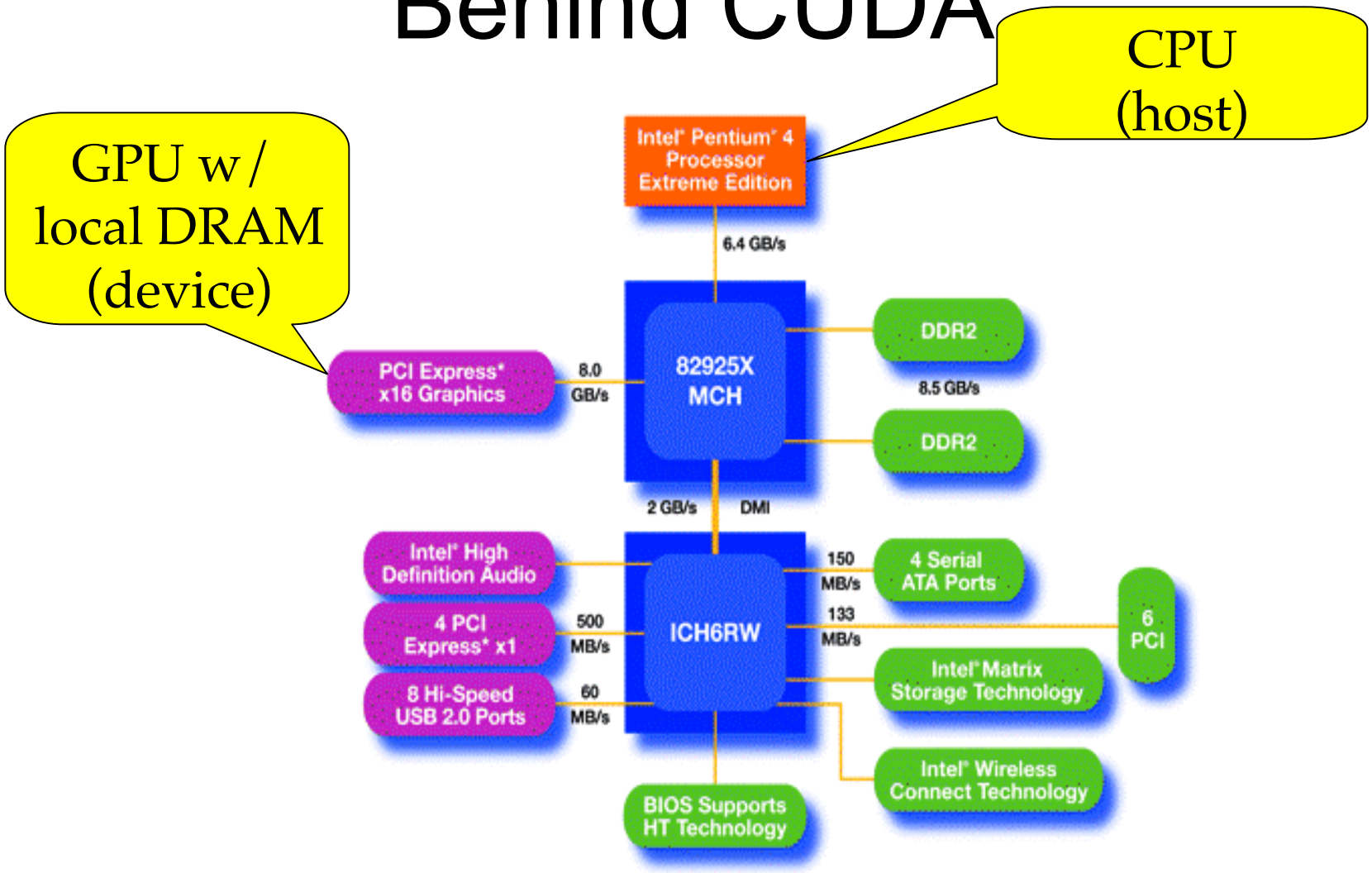
Processing Flow on CUDA



Example of CUDA processing flow

1. Copy data from main mem to GPU mem
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU mem to main mem

An Example of Physical Reality Behind CUDA



CUDA Advantages

- Scattered reads – code can read from arbitrary addresses in memory.
- Shared memory – CUDA exposes a fast shared memory region (up to 48KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and readbacks to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups.

CUDA Limitations

- Only a recursion-free, function-pointer-free subset of the C language, plus some simple extensions.
- A single process must run spread across multiple disjoint memory spaces.
- The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.
- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands.
- Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the **SIMD execution model** becomes a significant limitation for any inherently divergent task.
- Debugging GPU code can be very tedious as no direct form of error checking exists.

Applications and Speedup

Table 3. Representative CUDA application coprocessing speedups.

Application	Field	Speedup
Two-electron repulsion integral[12]	Quantum chemistry	130×
Gromacs[13]	Molecular dynamics	137×
Lattice Boltzmann[14]	3D computational fluid dynamics (CFD)	100×
Euler solver[15]	3D CFD	16×
Lattice quantum chromodynamics[16]	Quantum physics	10×
Multigrid finite element method and partial differential equation solver[17]	Finite element analysis	27×
N-body physics[18]	Astrophysics	100×
Protein multiple sequence alignment[19]	Bioinformatics	36×
Image contour detection[20]	Computer vision	130×
Portable media converter*	Consumer video	20×
Large vocabulary speech recognition[21]	Human interaction	9×
Iterative image reconstruction[22]	Computed tomography	130×
Matlab accelerator**	Computational modeling	100×

* Elemental Technologies, Badaboom media converter, 2009; http://badaboomit.com.
** Accelereyes, Jacket GPU engine for Matlab, 2009; http://www.accelereyes.com.

Nvidia “Fermi” GPU

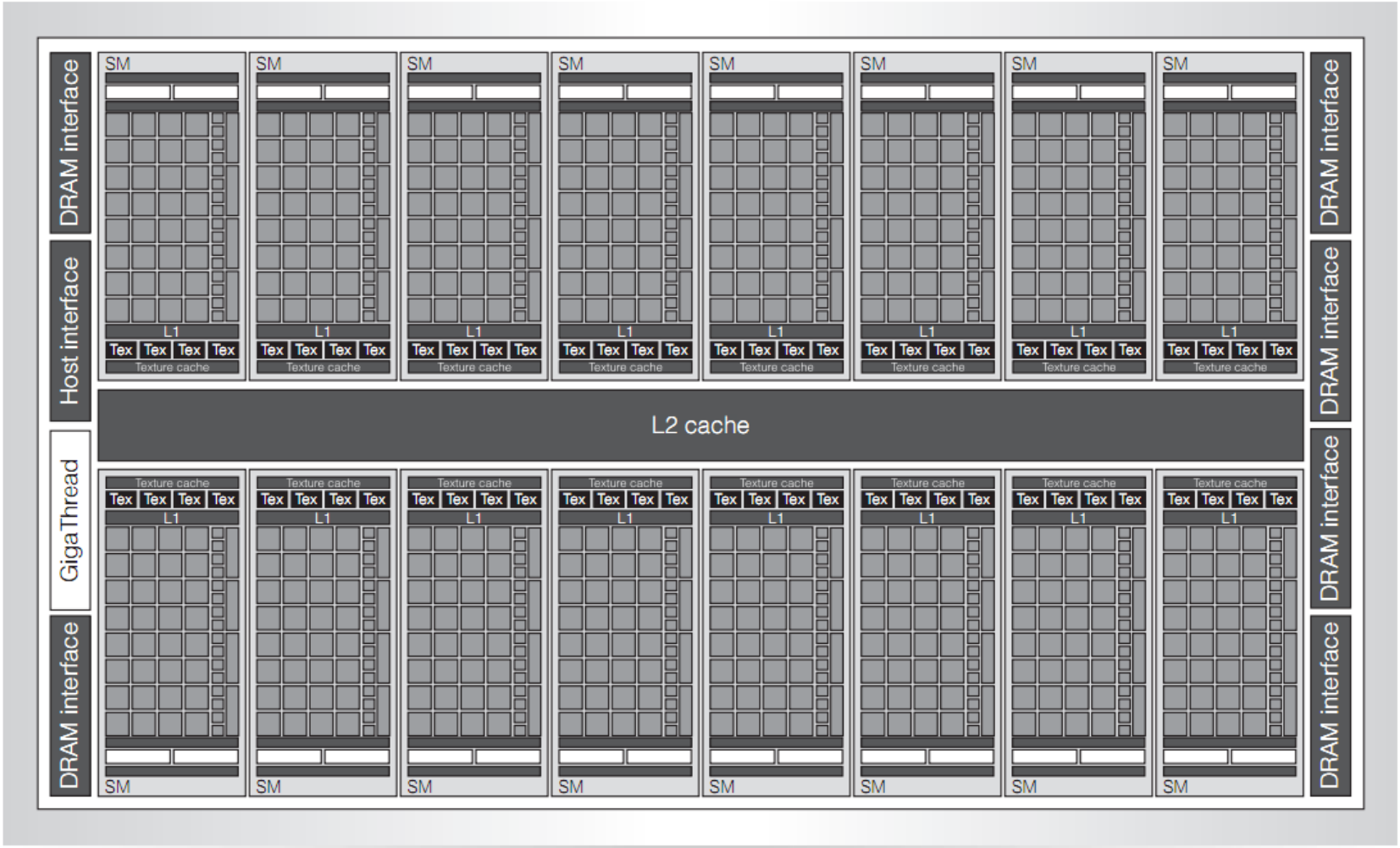
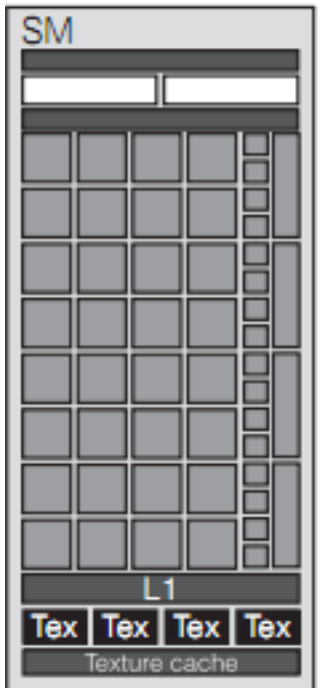


Figure 3. Fermi GPU computing architecture with 512 CUDA processor cores organized as 16 streaming multiprocessors (SMs) sharing a common second-level (L2) cache, six 64-bit DRAM interfaces, and a host interface with the host CPU, system memory, and I/O devices. Each streaming multiprocessor has 32 CUDA cores.

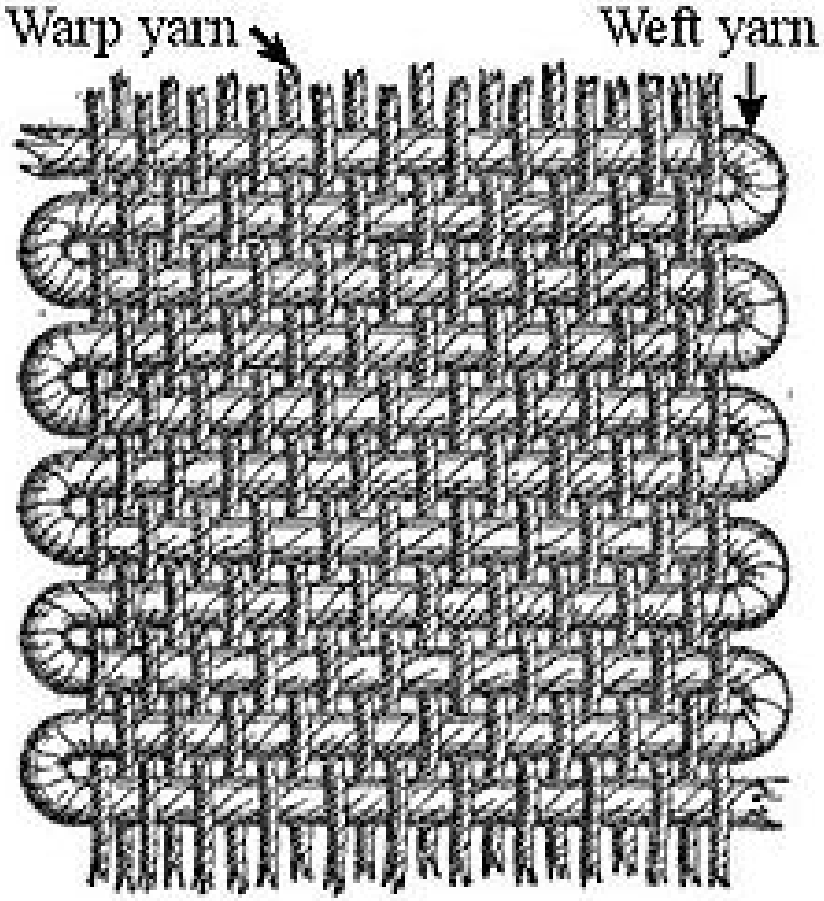
SM = Streaming Multiprocessor

- 1 SM = 32 Cores + L1 cache
- 1 GPU = 16 SM's
- 16x32 = 512 Cores total
- Tex = Texture memory

- 32 Threads = 1 "Warp"



“Warp” terminology is from Weaving



See also CMU iWarp systolic machine from 1980’s.

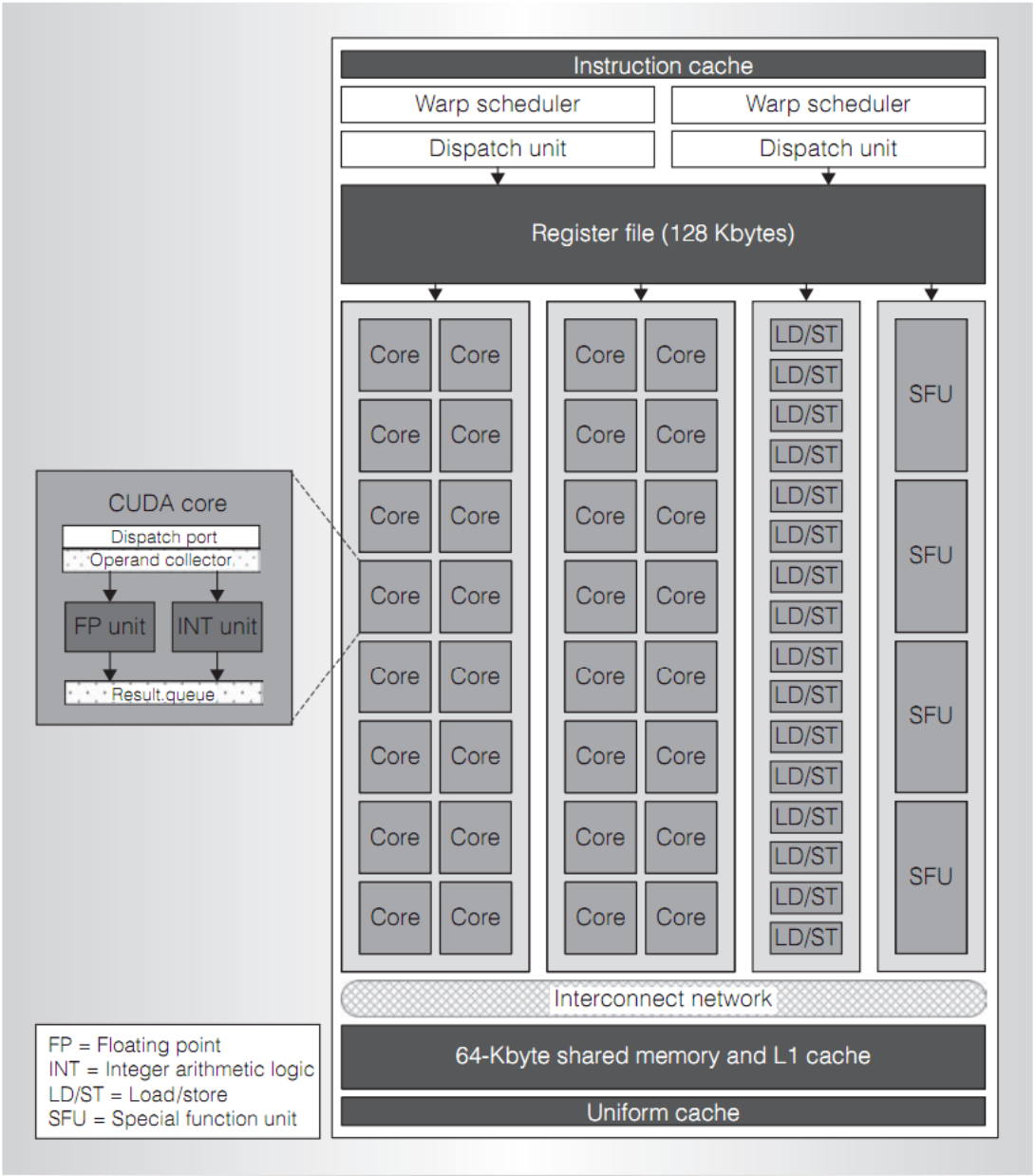


Figure 4. The Fermi streaming multiprocessor has 32 CUDA processor cores, 16 load/store units, four special function units, a 64-Kbyte configurable shared memory/L1 cache, 128-Kbyte register file, instruction cache, and two multithreaded warp schedulers and instruction dispatch units.

1 Core

- 1 core executes 1 scalar floating point or integer instruction per clock for a thread.

- With 32 cores, 32 times as much

CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel

- Data-parallel portions of an application are executed on the device as **kernel functions** which run in parallel on many threads with different data.

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Terminology

- **kernel**: logical device procedure viewed from the host side
- **thread**: a logical unit of execution on the device
- **block**: group of threads assigned to a single processor
- **warp**: a group of threads within a block that execute physically in parallel, but asynchronously
- **grid**: a group of blocks. All threads in a grid perform the same procedure, but with different thread id's.

Further Attributes

- **kernel**:
- **blocks**: used for allocating resources
- **warp**:
- **grid**: all blocks in a grid must complete before beginning execution of the next grid.

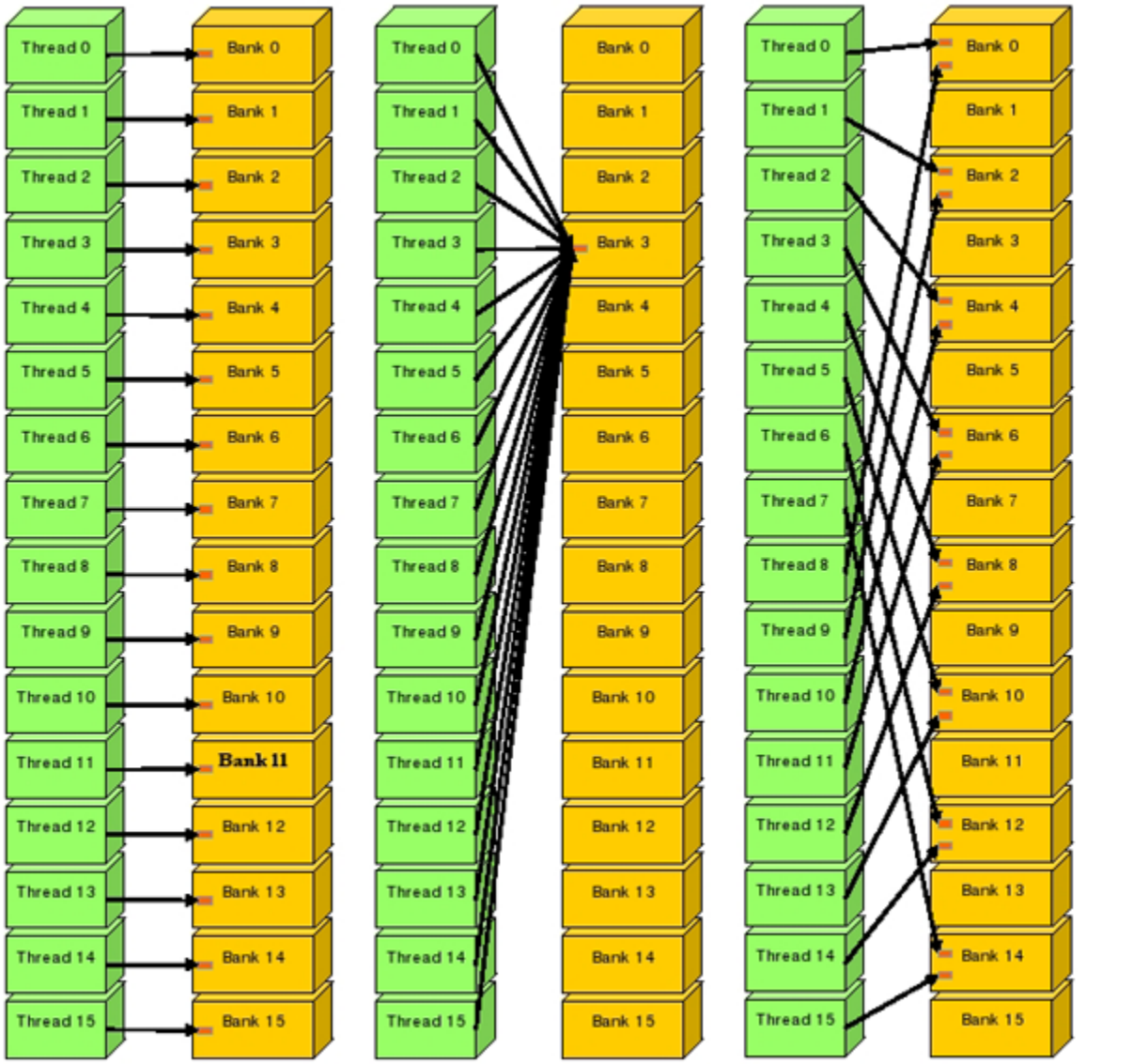
Inter-Thread Cooperation

- Threads within a block can cooperate and share memory.

- Threads in different blocks cannot cooperate.

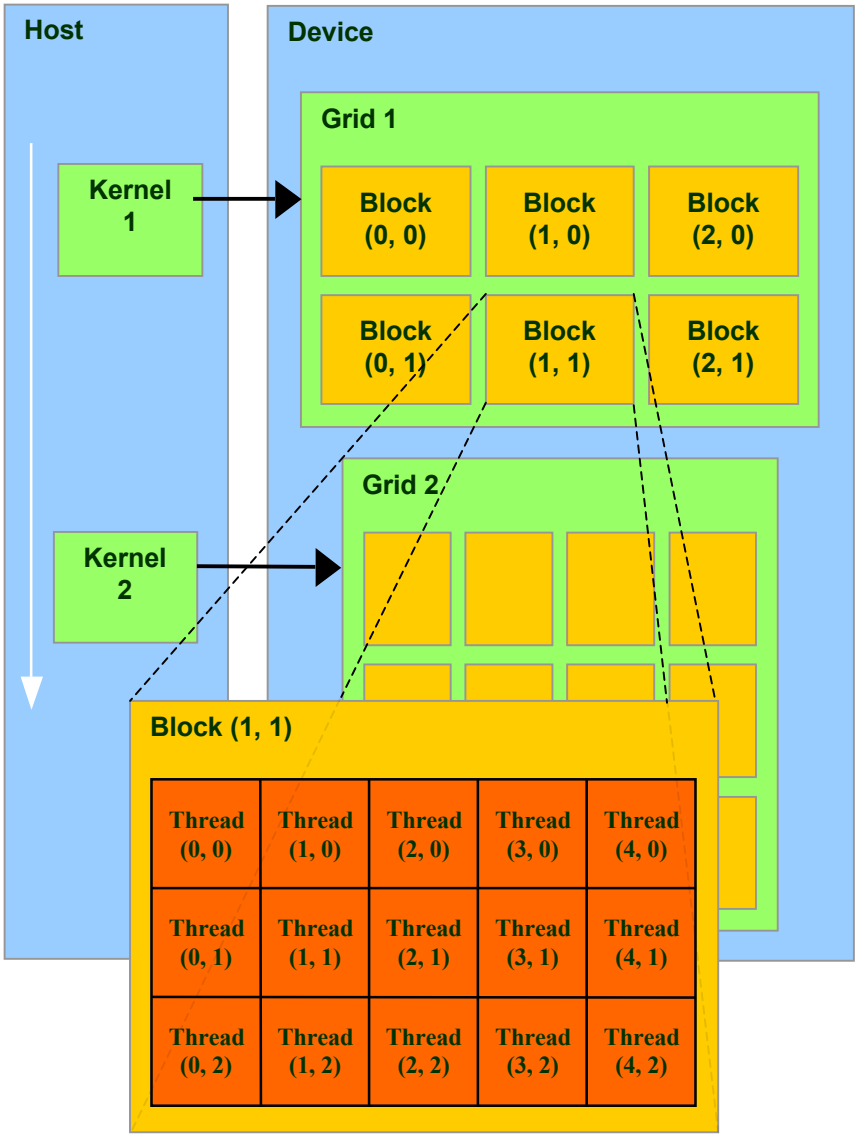
Program Design: On-Chip Memory

- Registers
  - 32 registers per processor
- Shared Memory - per block
  - 16KB per multiprocessor
  - Data should be in 32-bit increments to take advantage of concurrent accesses
  - Access is as fast as register access **if no banks conflict**


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Bank 12
Bank 13
Bank 14
Bank 15
no conflict
no conflict
conflicts
(cause serialization)

Thread Batching: Grids and Blocks

- A kernel function is **executed** as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory



Courtesy: NDVIA

Program Design - Threads

- More threads per block are better for time slicing
  - Minimum: 64, Ideal: 192-256
- More threads per block means fewer registers per thread
  - Kernel invocation may fail if the kernel compiles to more registers than are available
- Threads within a block can be synchronized
  - Important for SIMD efficiency
- The maximum threads allowed per grid is 64K^3

Program Design - Blocks

- There should be at least as many blocks as multiprocessors
  - The number of blocks should be at least 100 to scale to future generations
- Blocks within a grid cannot be synchronized
- Blocks can only be swapped by partitioning registers and shared memory among them

Program Design: Control Flow

Since the hardware is SIMD, control flow instructions can cause thread execution paths to diverge

- Divergent execution paths must be serialized (costly)
- *if, switch,* and *while* statements should be avoided if threads from the same warp will take different paths
- The compiler may remove *if* statements if favor of predicated instructions to prevent divergence

Ideal CUDA Programs

- High intrinsic parallelism
  - per-pixel or per-element operations
    fft, matrix multiply
  - most image processing applications
- Minimal communication (if any) between threads
  - limited synchronization
- Few control flow statements
- High ratio of arithmetic to memory operations

Program Design: On-Chip Memory

- Registers
  - 32 registers per processor
- Shared Memory - per block
  - 16KB per multiprocessor
  - Data should be in 32-bit increments to take advantage of concurrent accesses
  - Access is as fast as register access if no banks conflict

CUDA Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)
{
  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block

convolve<<<100, 10>>> (myimage);
```

Data Type Keyword Meanings

CUDA Function Declarations

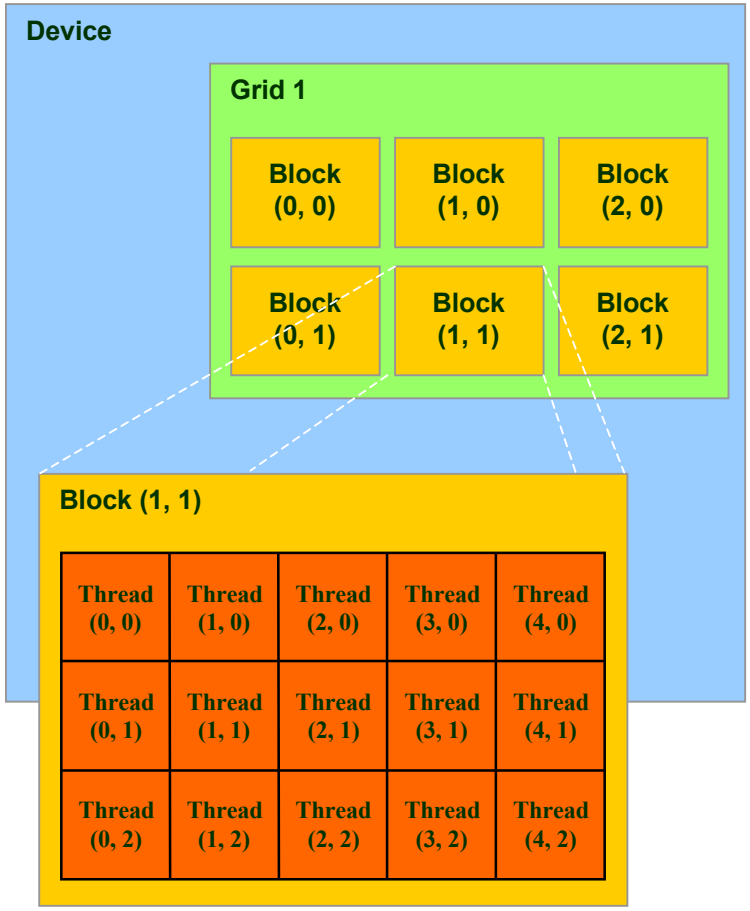
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`

- `__device__` and `__host__` can be used together (compiler generates both types of code)

Block and Thread IDs

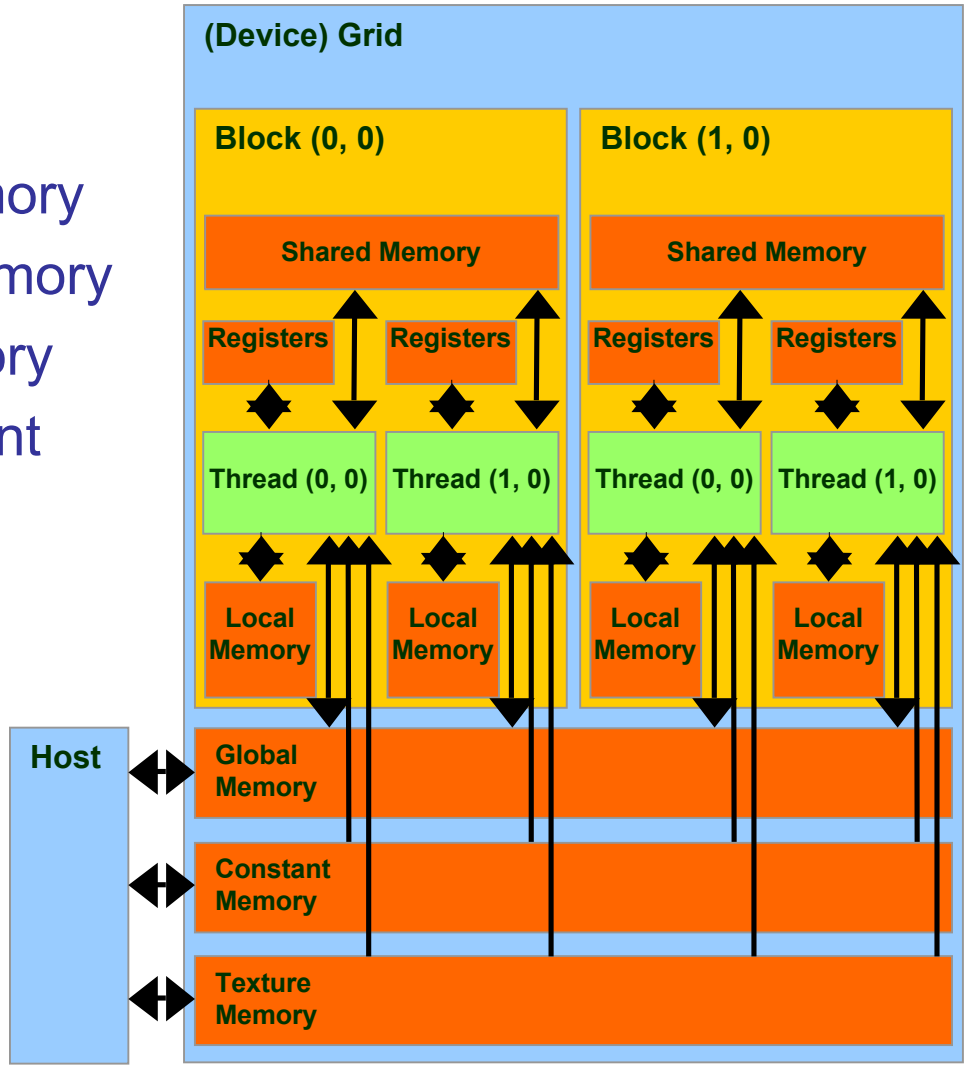
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

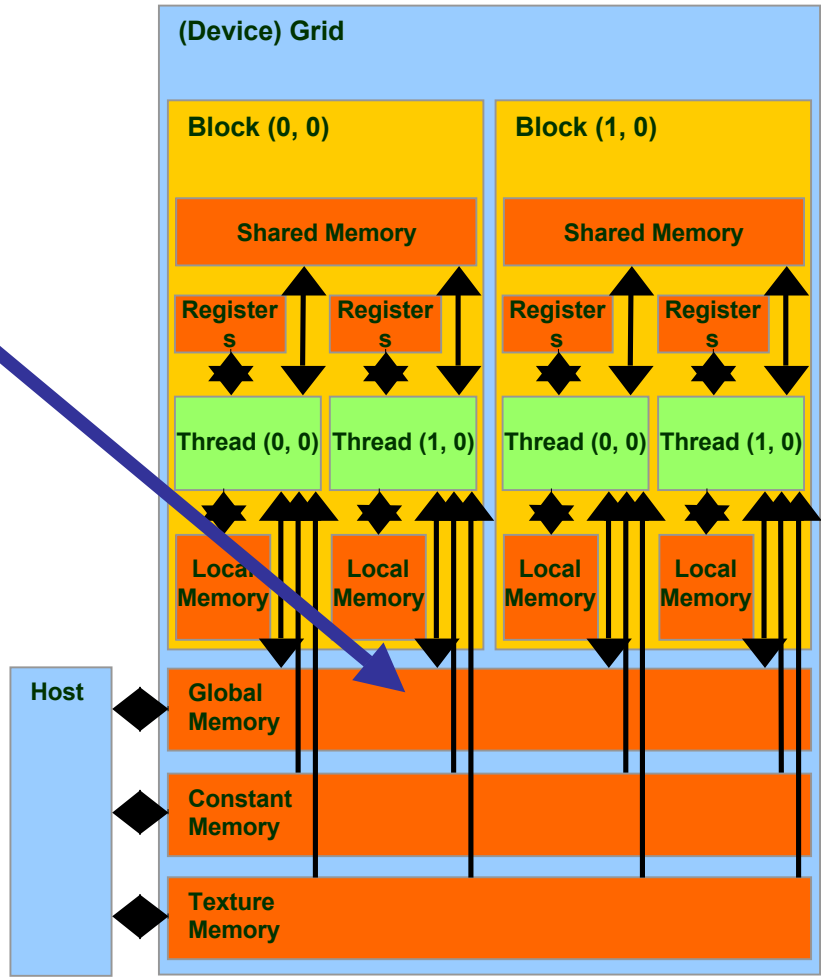
CUDA Device Memory Space Overview

- 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 memories



CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

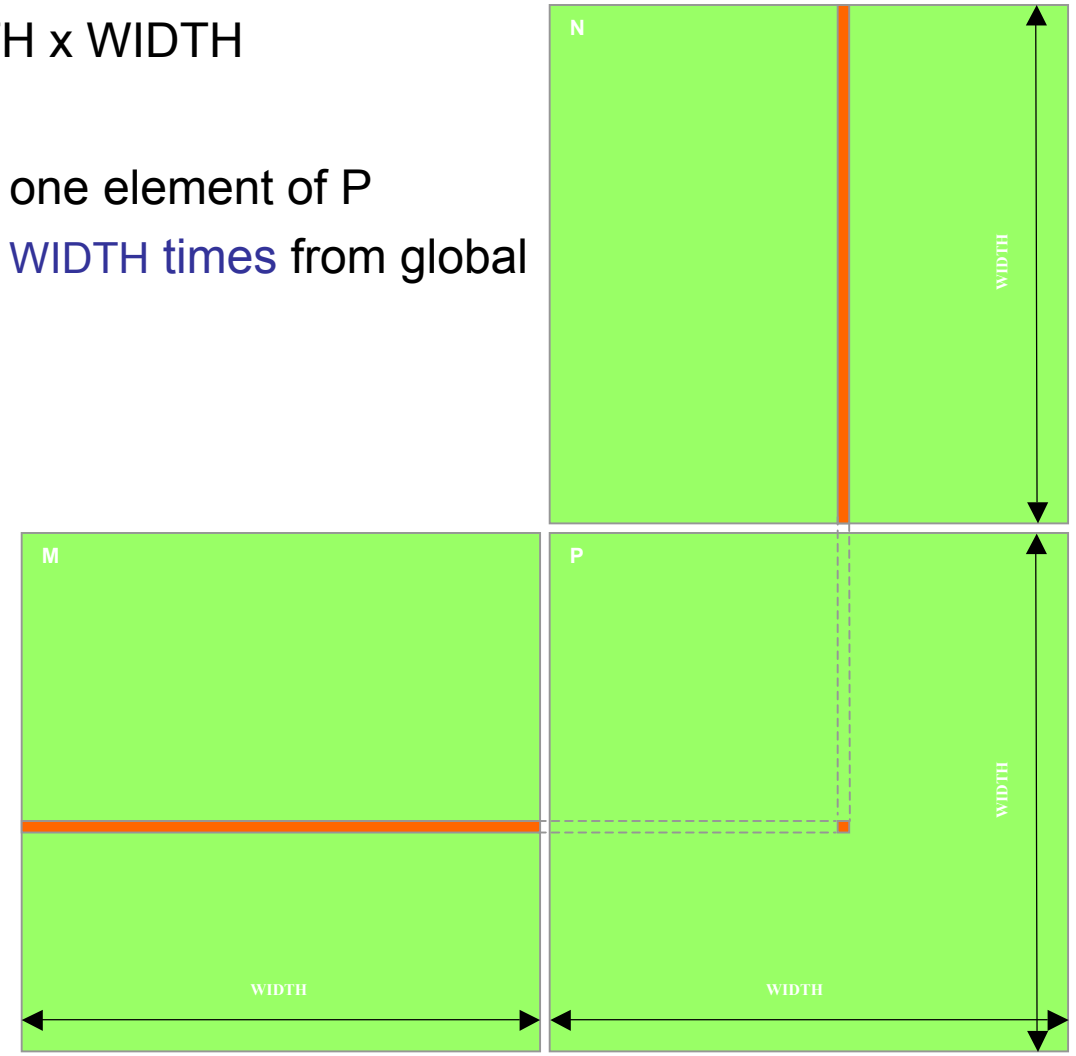
```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
```

```
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory



Matrix Data Transfers

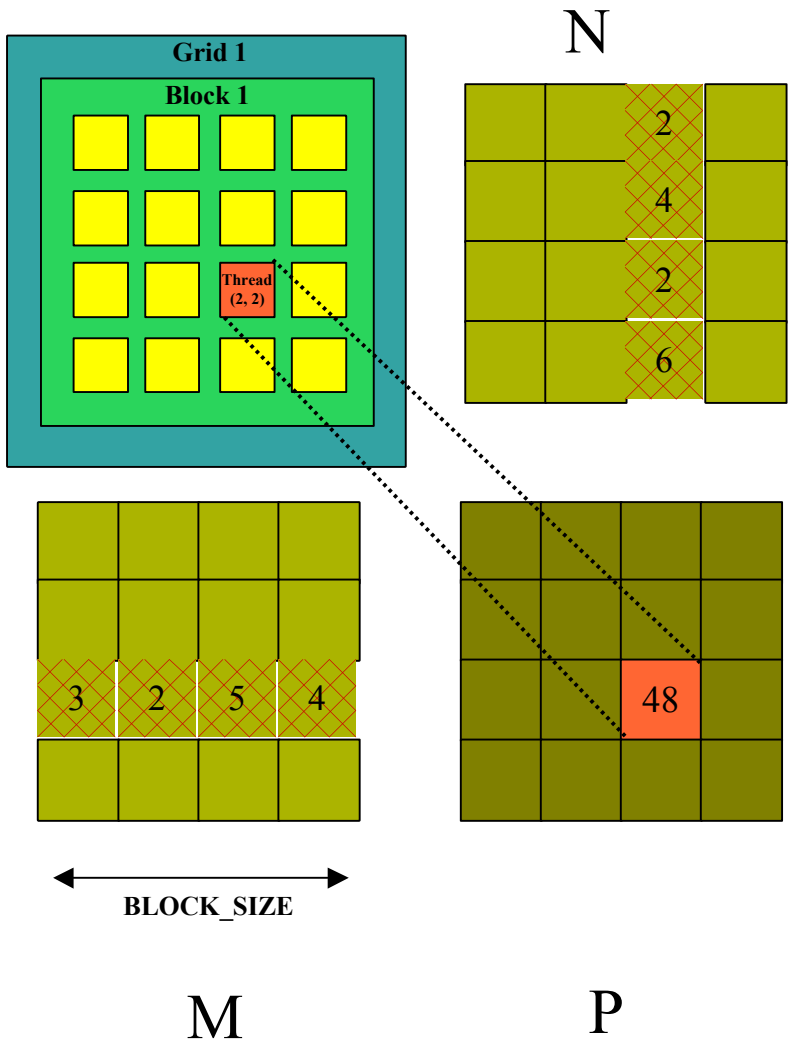
```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size,
    cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size,
    cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

Multiply Using One Thread Block

- One Block of threads compute matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



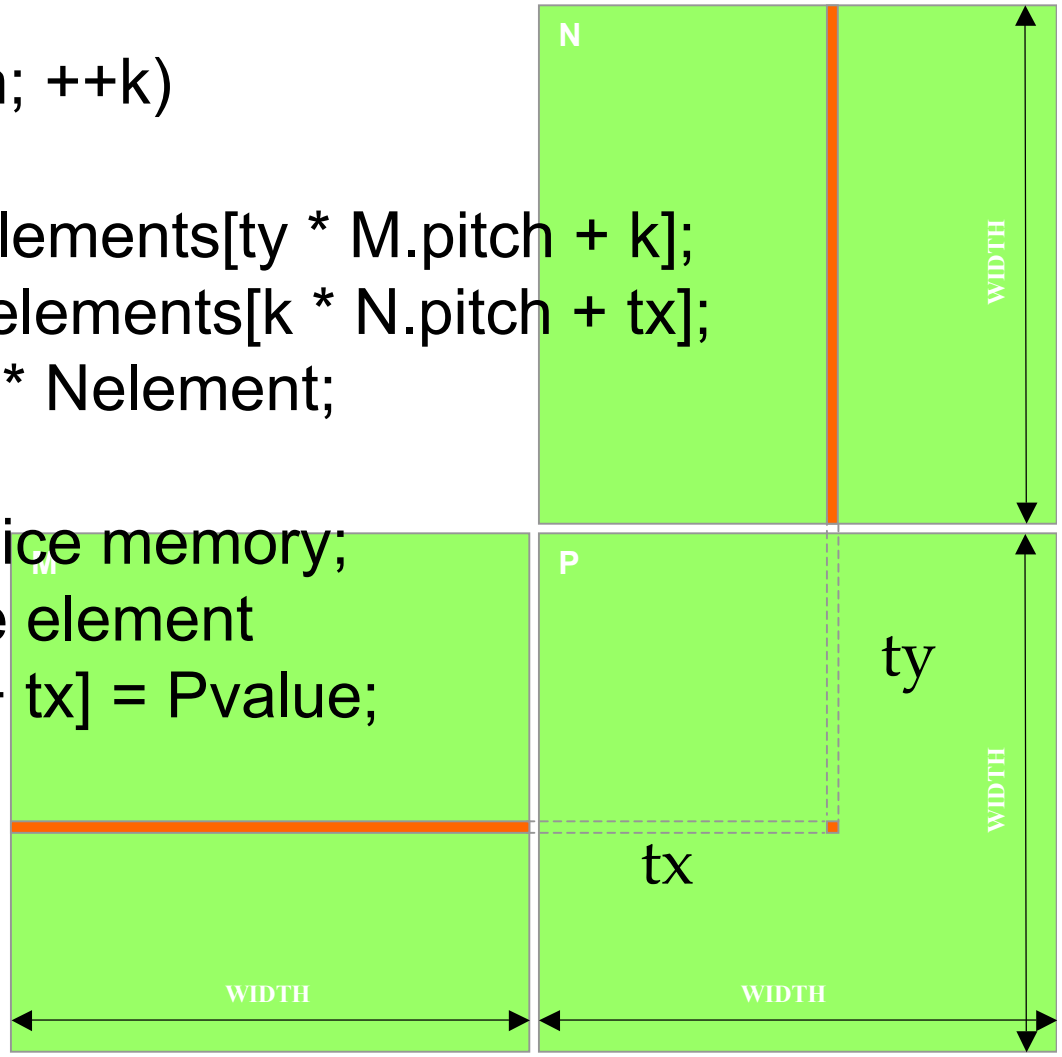
Matrix Multiplication
Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Matrix Multiplication
Device-Side Kernel Function (cont.)

```
    for (int k = 0; k < M.width; ++k)
    {
        float Melement = M.elements[ty * M.pitch + k];
        float Nelement = Nd.elements[k * N.pitch + tx];
        Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P.pitch + tx] = Pvalue;
}
```



Matrix Multiplication Host-side Main Program Code

```
int main(void) {
// Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

// M * N on the device
    MatrixMulOnDevice(M, N, P);

// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```

Matrix Multiplication Host-side code

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
```

Matrix Multiplication Host-side Code (cont.)

```
// Setup the execution configuration
    dim3 dimBlock(WIDTH, WIDTH);
    dim3 dimGrid(1, 1);

    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Utilities

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Utilities

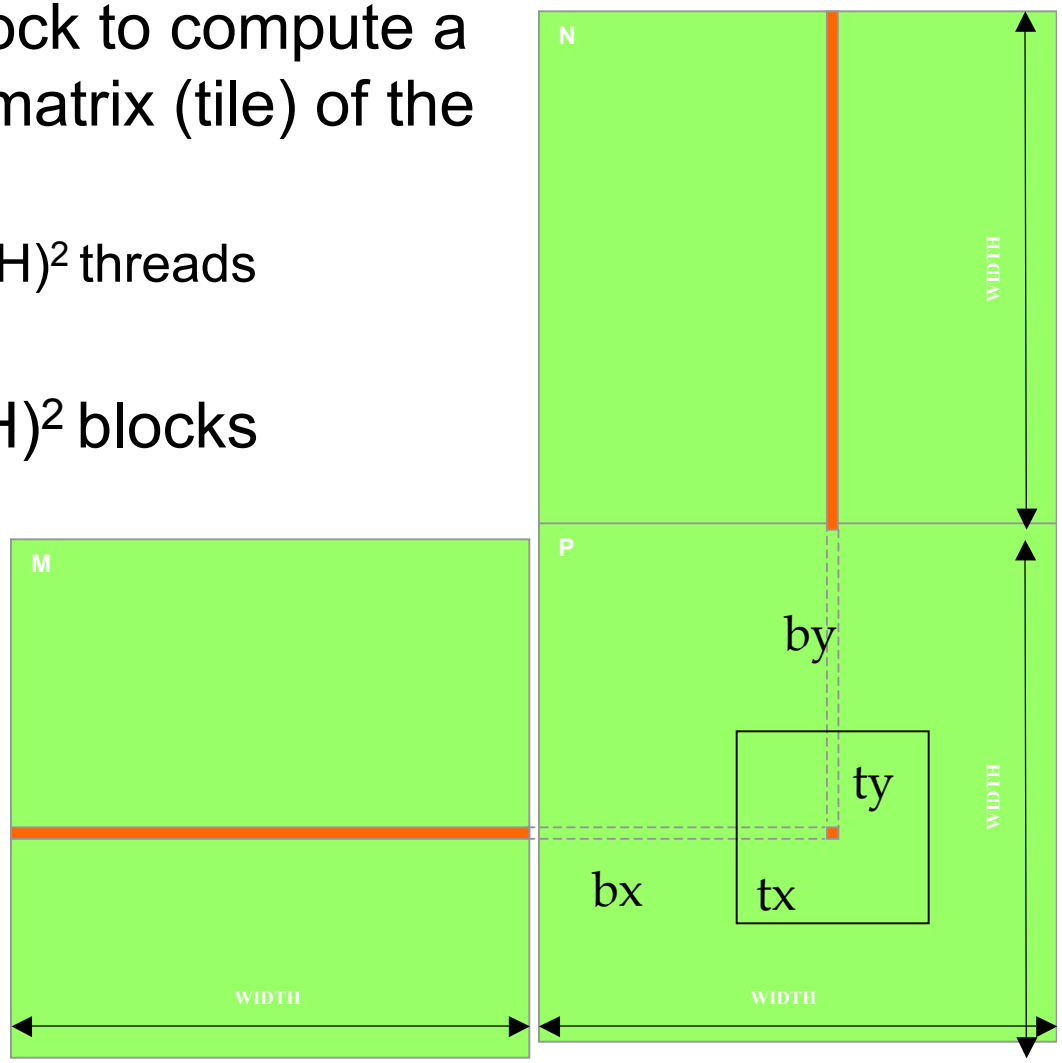
```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
   int size = Mhost.width * Mhost.height * sizeof(float);
   cudaMemcpy(Mdevice.elements, Mhost.elements, size,
            cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
   int size = Mdevice.width * Mdevice.height * sizeof(float);
   cudaMemcpy(Mhost.elements, Mdevice.elements, size,
            cudaMemcpyDeviceToHost);
}
```

Extension to Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a $(BLOCK\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(BLOCK\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/BLOCK\_WIDTH)^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!



Coding Tips

http://www.caam.rice.edu/~timwar/RMMC/CUDA.html

Coding a kernel for the GPU device takes a certain amount of care. It amounts to: taking care of, feeding, and keeping a whole lot of hamsters busy. Here are some rules of thumb:

1.Problem Decomposition: Divide up your problem so that there are a relatively large number (>32, <512) of threads per block.

Example: the 8800 gpus have 128 thread processors, each capable of organizing/executing 96 threads. The thread manager on each thread processor keeps threads in several queues.

When a specific thread requests data from memory it is put into a wait queue, once the data is available it is put into a ready-to-resume queue and is resumed as soon as possible.

This is why it is important to have a good number of threads available for execution in each block, allowing us to hide high memory access times.

Coding Tips

2.Memory types: There are several types of memory available on the device. Unfortunately it is important to know their characteristics to achieve high data throughput:

common: accessible for read & write operations by all thread processors. Memory access to this memory is very slow, taking up to 300 cycles. But keep in mind that while a thread is waiting for data from common memory, other threads can execute instructions.

constant: accessible for read operations by all thread processors. Limited to 64KB (?) and difficult to use efficiently.

local: accessible for read & write, dedicated to individual threads. Unfortunately it has been difficult to use efficiently.

texture: accessible for read operations by all thread processors. Cached. Fast if data is in cache. To use effectively I found it necessary to coalesce reads.

shared: accessible for read & write by all threads in block. Faster than texture, but limited in size. Useful for storing modest amounts of data to be subsequently used by all threads in block. The amount of shared memory required for a block effects the level of occupancy obtainable, i.e. how many threads can be executed simultaneously.

register: accessible for read & write local to each thread. This is likely the **fastest** memory available. But unfortunately it is **very limited** per thread and the occupancy rate is strongly dependent on the amount of register memory each thread requires.

Coding Tips

3.Occupancy calculator: nvidia provides an Excel spreadsheet to compute the level of occupancy that your CUDA kernel can expect. First use nvcc to compile the .cu kernel file, then use ptxas to assemble the CUDA kernel. You can actually view the contents of the cubin.bin output file and find out how much local memory (lmem), register memory (reg), and shared memory (smem) that your compiled/optimized kernel will actually require. Plugging these numbers into the occupancy calculator will yield the estimates of the occupancy rate.

On the next page are two examples of the right hand side function that evaluates the elemental spatial derivative contributions for 3D Maxwells. The treatment of each is somewhat different, yielding different register counts and noticeably different performance.

Example 1: MaxwellsGPU_VOL_Kernel (early version) (N=7):

Parameters: lmem=0, smem=3133, reg=22, ThreadsPerBlock=120

Occupancy Calculator

Example 1: MaxwellsGPU_VOL_Kernel (early version) (N=7):

Parameters: lmem=0, smem=3133, reg=22, ThreadsPerBlock=120

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

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select a GPU from the list (click):	G80

2.) Enter your resource usage:	
Threads Per Block	120
Registers Per Thread	22
Shared Memory Per Block (bytes)	3133

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	240
Active Warps per Multiprocessor	8
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	33%
Maximum Simultaneous Blocks per GPU	32

(Note: This assumes there are at least this many blocks)

Physical Limits for GPU:	G80
Multiprocessors per GPU	16
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384

Allocation Per Thread Block	
Warps	4
Registers	2816
Shared Memory	3584

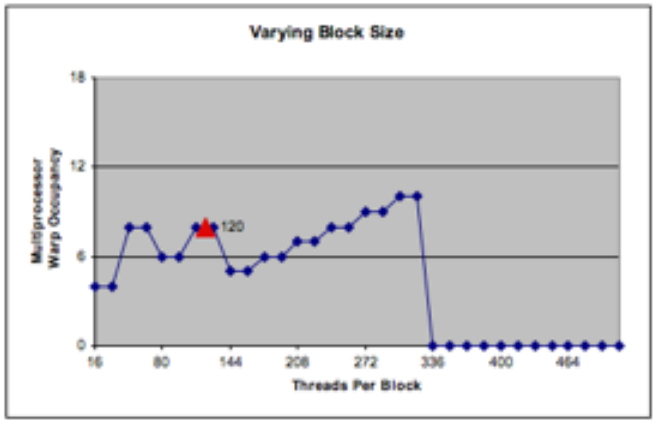
These data are used in computing the occupancy data in blue

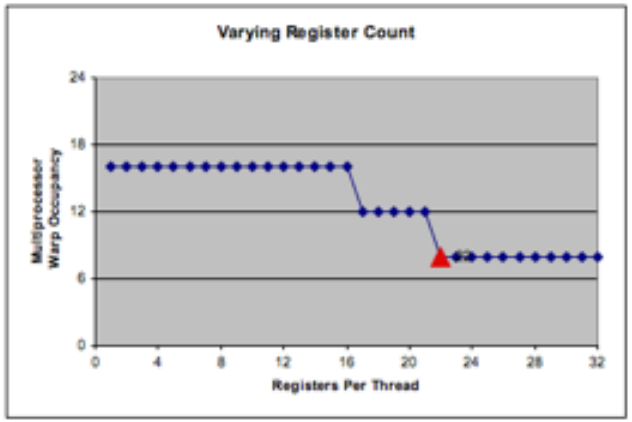
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	6
Limited by Registers / Multiprocessor	2
Limited by Shared Memory / Multiprocessor	4

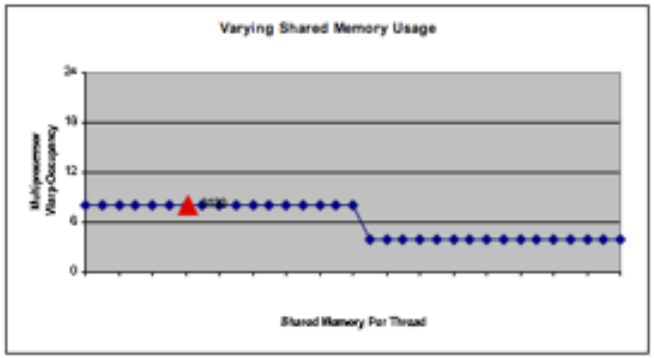
Thread Block Limit Per Multiprocessor is the minimum of these 3

CUDA Occupancy Calculator	
Version:	1.2
Copyright and License	

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.







Occupancy Calculator

Example 2: MaxwellsGPU_VOL_Kernel (N=7):

Parameters: lmem=0, smem=3133, reg=28, ThreadsPerBlock=120

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

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select a GPU from the list (click):	G80	(Help)
2.) Enter your resource usage:		
Threads Per Block	120	(Help)
Registers Per Thread	28	
Shared Memory Per Block (bytes)	3133	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	240	(Help)
Active Warps per Multiprocessor	8	
Active Thread Blocks per Multiprocessor	2	
Occupancy of each Multiprocessor	33%	
Maximum Simultaneous Blocks per GPU	32	

(Note: This assumes there are at least this many blocks)

Physical Limits for GPU:	G80
Multiprocessors per GPU	16
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384

Allocation Per Thread Block	
Warps	4
Registers	3584
Shared Memory	3584

These data are used in computing the occupancy data in blue

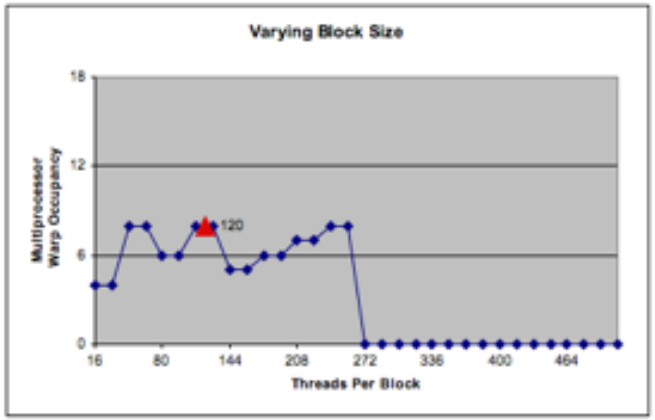
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	6
Limited by Registers / Multiprocessor	2
Limited by Shared Memory / Multiprocessor	4

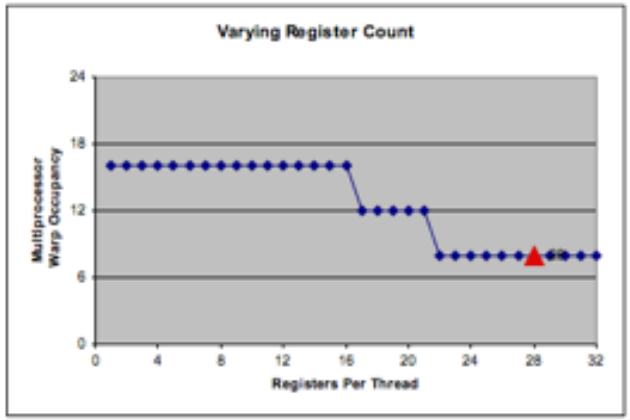
Thread Block Limit Per Multiprocessor is the minimum of these 3

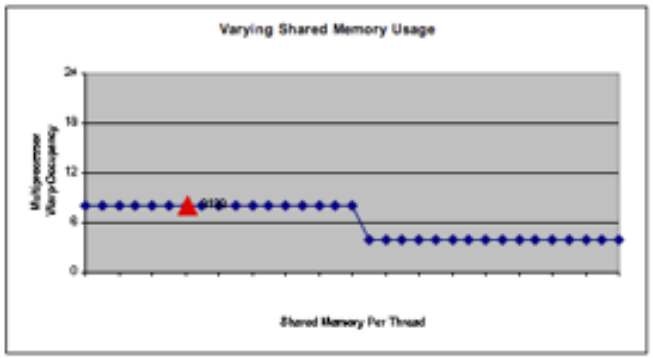
CUDA Occupancy Calculator	
Version:	1.2

Copyright and License

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.







Despite the higher register count, the occupancy level is the same and more importantly the performance is higher. This is because we are using so many registers that a few more does not make much difference to the number of warps that can simultaneously execute on each multiprocessor.

Coding Tips

4. It's an entertaining enterprise trying to maximize the threads per block, keep lmem=0, and minimize rmem and smem in order to achieve a high occupancy rate. My experience is that it is difficult to achieve more than 66% occupancy (this requires 10 or less registers and modest shared memory) and 100% occupancy does not yield much better performance than 66% occupancy for non-trivial kernels. My best code performance has been achieved by a compromise of kernel complexity and register counts.

5. Memory Bank Conflicts: The promise of the fast shared memory, simultaneously accessible by all the threads in a half-warp, is tempered by the possibility that the individual threads may request to read from the shared memory in a random access manner. A simple rules of thumb is that if you intend to use shared memory, try to ensure that each thread is reading an entry that is 32bits shifted from its thread neighbors. Examples:

   i. thread n grabs an entry from a shared memory array as: *float u = s_u[n];*
   ii. all threads read the same entry *float u = s_[3]*; This is referred to as a "broadcast"

Coding Tips

7. **CUDA (2.0 beta) Profiler**: nvidia included a GUI based profiler in the 2.0 version of CUDA. It provides medium grain information at the function level, i.e. it does not give line-by-line profile information. cudaprofile gives information about the following characteristics:

   i. microseconds spent in GPU compute per kernel
   ii. microseconds spent in CPU compute (or busy wait)
   iii. number of coherent load and stores on the GPU (i.e. coalesced read/writes)
   iv. number of incoherent load and stores on the GPU
   v. number of branches (i.e. if statements processed on the GPU)
   vi. number of divergent branches (bad news)
   vii. number of local load and stores (quite bad news)
   viii. number of times the threads being processed in a warp had to be serialized, i.e. processsed sequentially (bad news)