GPU Computing with Nvidia CUDA

David Kaeli, **Perhaad Mistry,** Rodrigo Dominguez,
Dana Schaa, Matthew Sellitto,
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA

GPU Computing Course – Lecture 2

Analogic Corp. 4/14/2011

NUCAR

Please make sure you join

https://groups.google.com/group/analogic-gpu-course

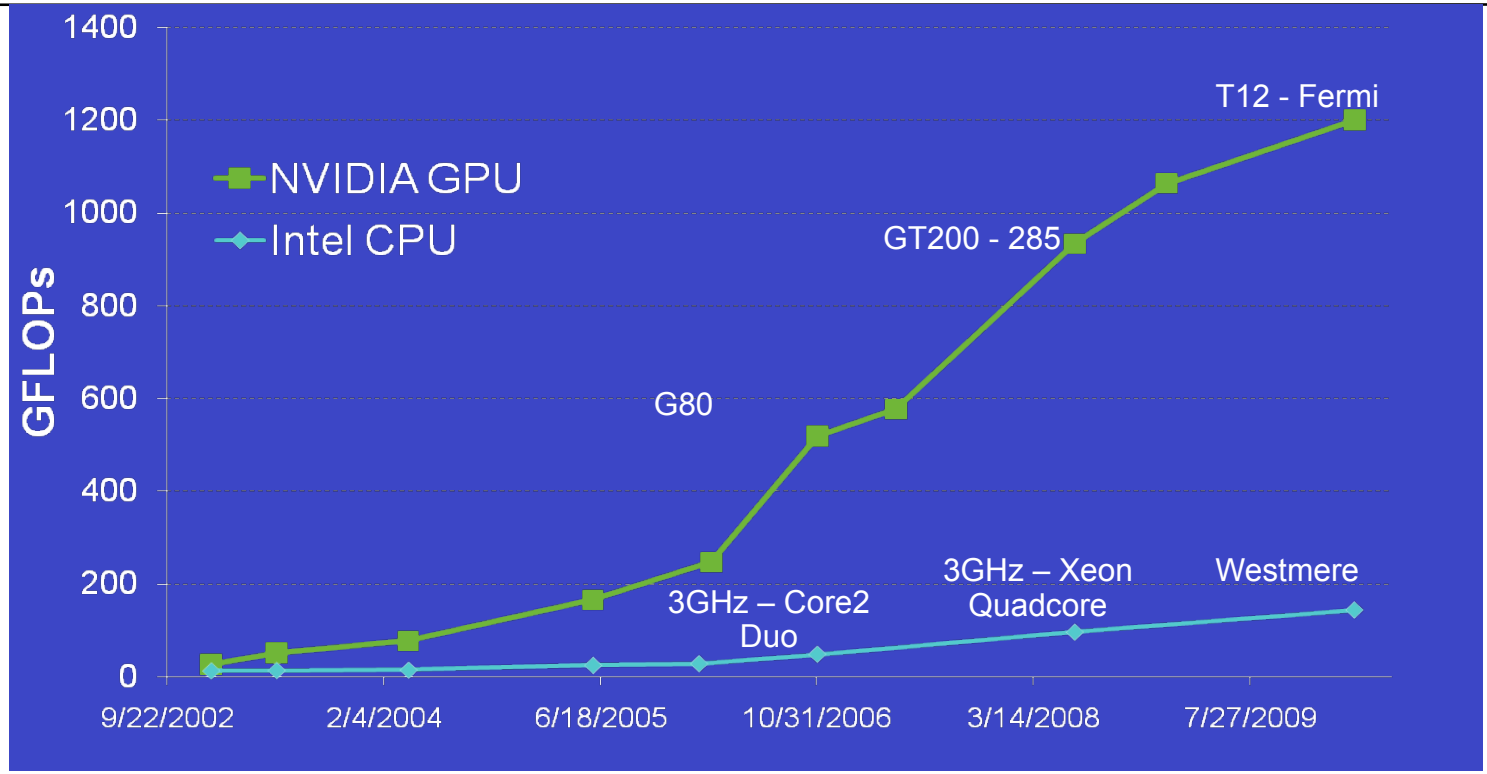
Mail Questions to
analogic-gpu-course@googlegroups.com

Topics – Lecture 2

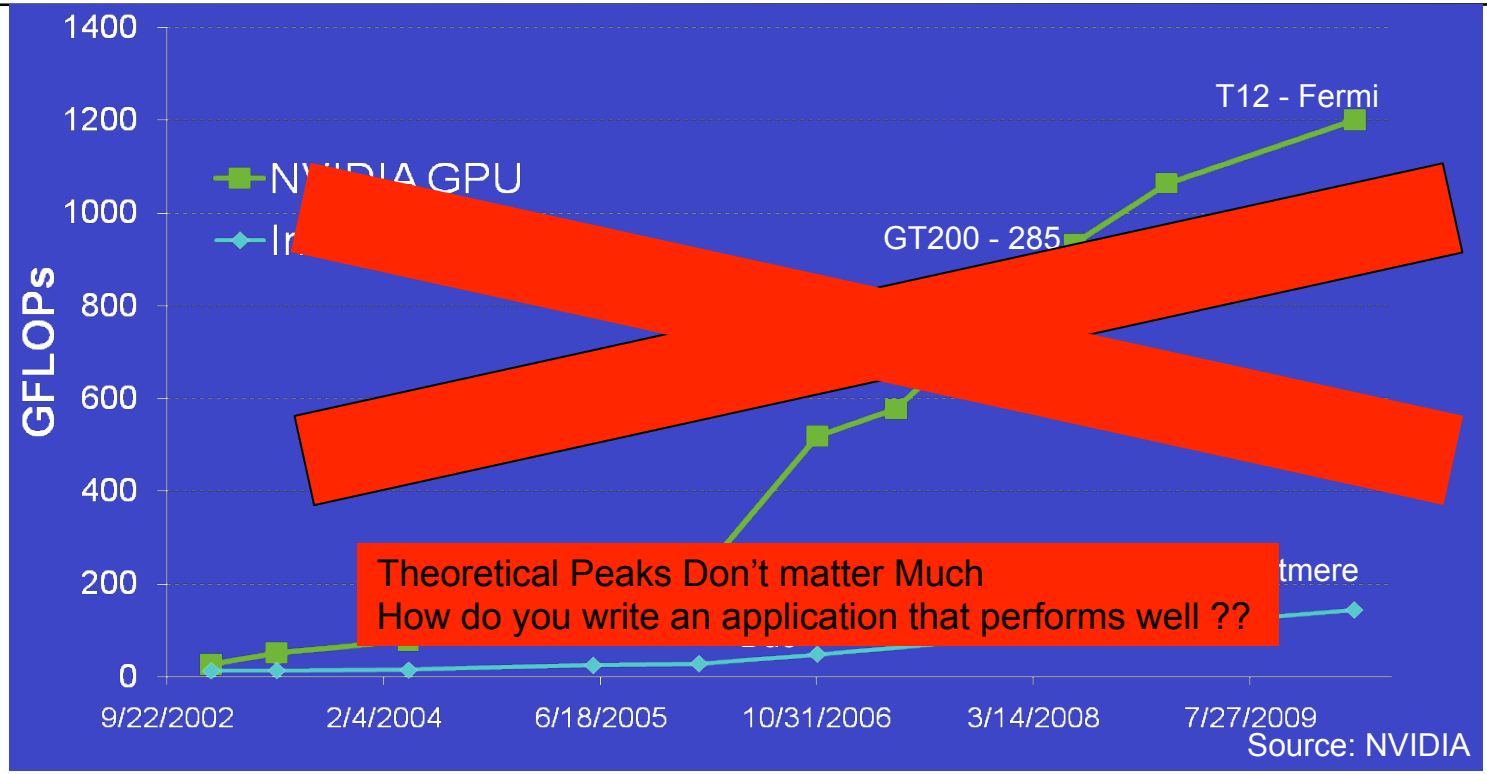
- Review of Lecture 1 and introduction to GPU Computing
- Overview of GPU Architecture
- Nvidia CUDA Syntax
- Basic CUDA optimization steps
- Nvidia Fermi
- Kernel optimizations and host – device IO
- Pointers to useful CUDA tools
- Conclusions and Discussion

NUCAR

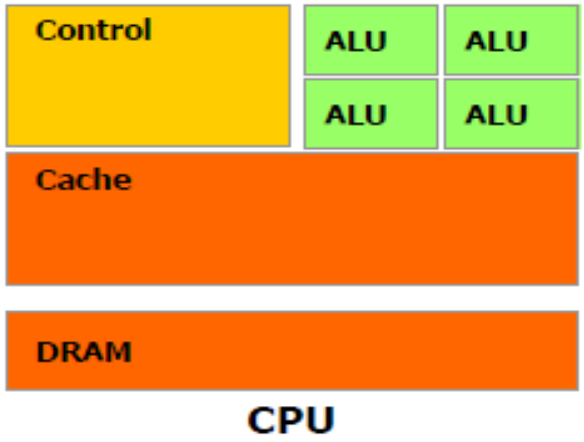
Motivation to study CUDA

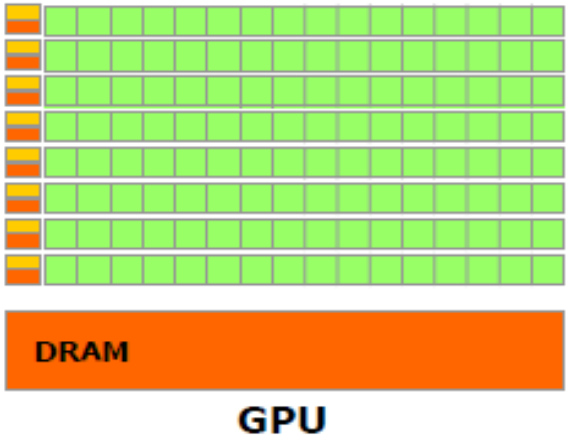


Motivation to study CUDA



CPU vs GPU Architectures





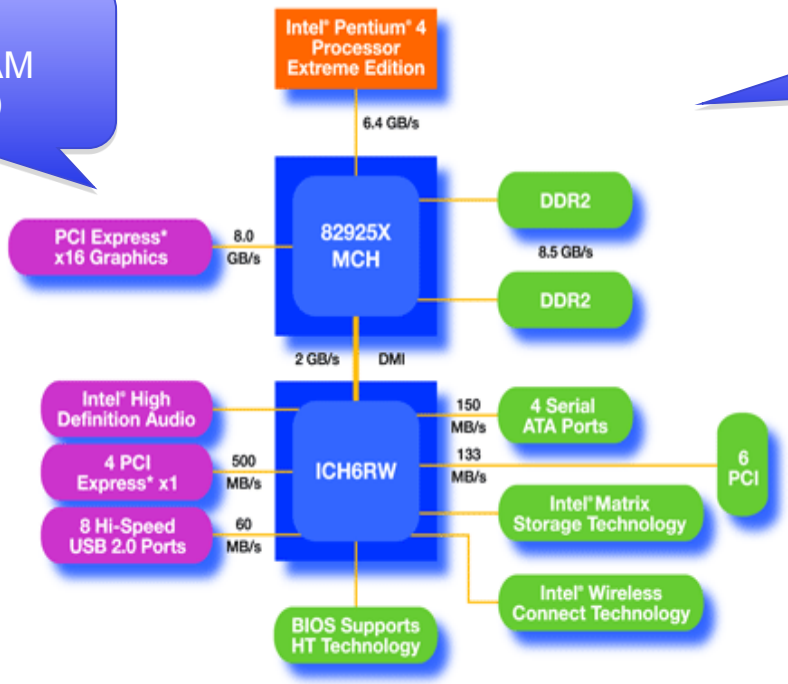
Irregular data accesses
More cache + Control
Focus on per thread performance

Regular data accesses
More ALUs and massively parallel
Throughput oriented

The System

GPU w/
local DRAM
(device)

CPU
(host)



MCH: Memory Controller Hub

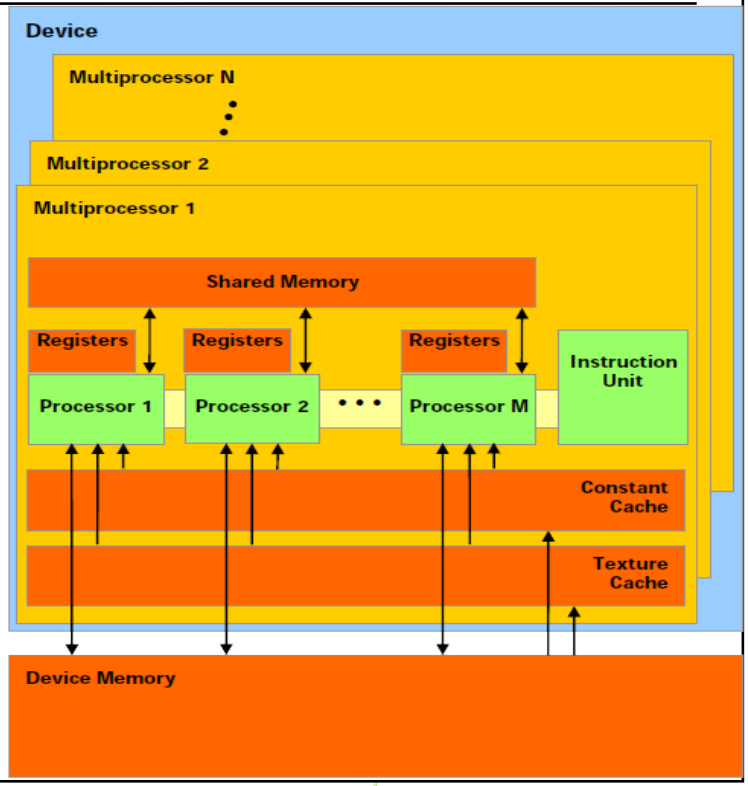
ICH: I/O Controller Hub

DDR: Double Data Rate

NUCAR

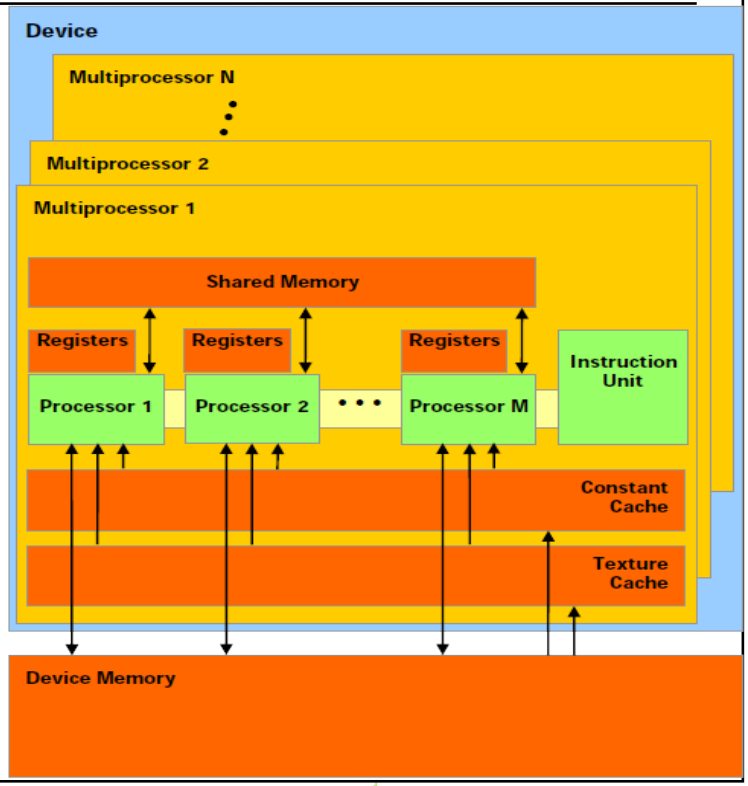
Nvidia GPU Compute Architecture

- Compute Unified Device Architecture
- Hierarchical architecture
  - A device contains many multiprocessors
  - Many scalar “cuda cores” per multiprocessor (32 for Fermi)
  - Single instruction issue unit
- Many memory spaces



GPU Memory Architecture

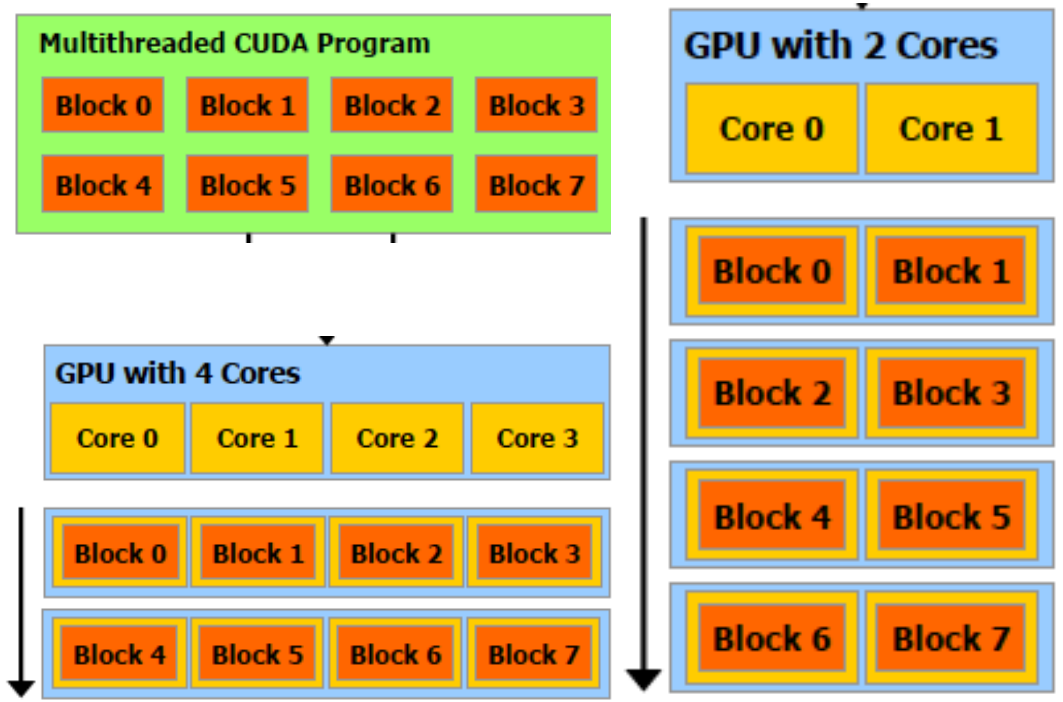
- Device Memory (GDDR):
  - Large memory with a high bandwidth link to multiprocessor
- Registers on chip (~16k)
- Shared memory (on chip)
  - Shared between scalar cores
  - Low latency and banked
- Constant and texture memory
  - Read only and cached



A “Transparently” Scalable Architecture

The CUDA programming model maps easily to underlying architecture

Same program will be scalable across devices



Array Addition (CPU)

```
void arrayAdd(float *A, float *B, float *C, int N) {
   for(int i = 0; i < N; i++)
      C[i] = A[i] + B[i];
}
```

Computational kernel

```
int main() {
   int N = 4096;
   float *A = (float *)malloc(sizeof(float)*N);
   float *B = (float *)malloc(sizeof(float)*N);
   float *C = (float *)malloc(sizeof(float)*N);

   init(A); init(B);

   arrayAdd(A, B, C, N);

   free(A); free(B); free(C);
}
```

Allocate memory

Initialize memory

Deallocate memory

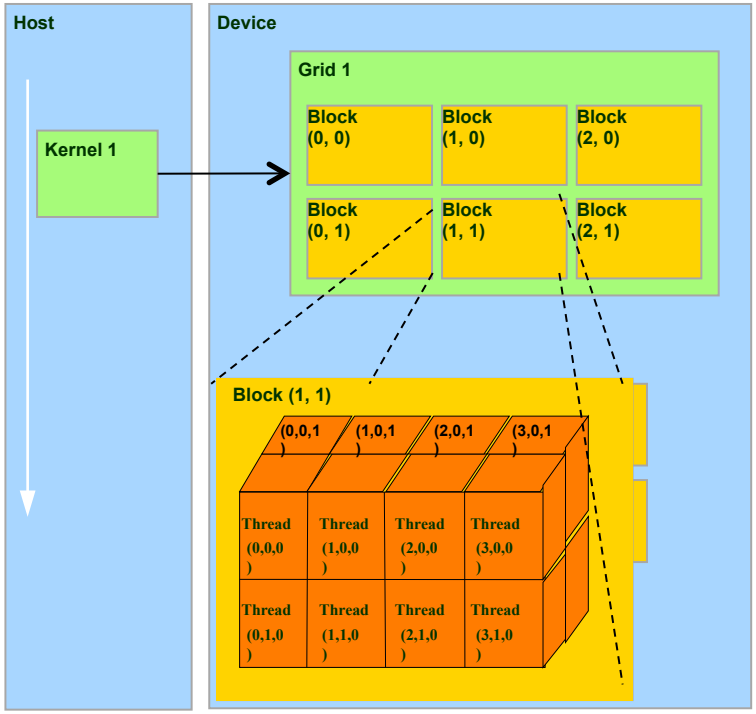
NUCAR

CUDA Programming – High Level View

- Initialize the GPU – done implicitly in CUDA
- Allocate Data on GPU
- Transfer data from CPU to GPU
- Decide how many threads and blocks
- Run the GPU program
- Transfer back the results from GPU to CPU

CUDA terminology

- A Kernel is the computation offloaded to GPUs
- The kernel is executed by a grid of threads
- Threads are grouped into blocks which execute independently
  - Each thread has a unique ID within the block
  - Each block has a unique ID



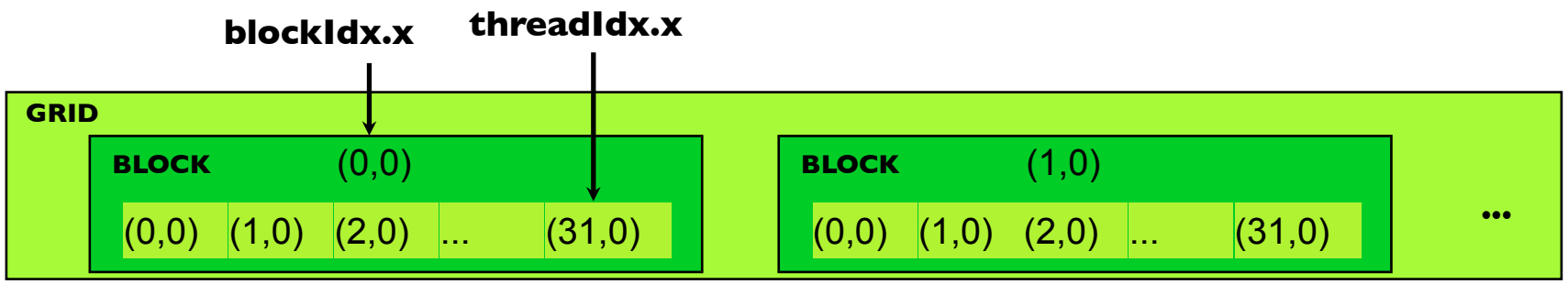
Array Addition (GPU)

```
__global__
```
← Kernel Indentifier

```
void gpuArrayAdd(float *A, float *B, float *C) {

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    C[tid] = A[tid] + B[tid];
}
```

← Index for Thread's Data

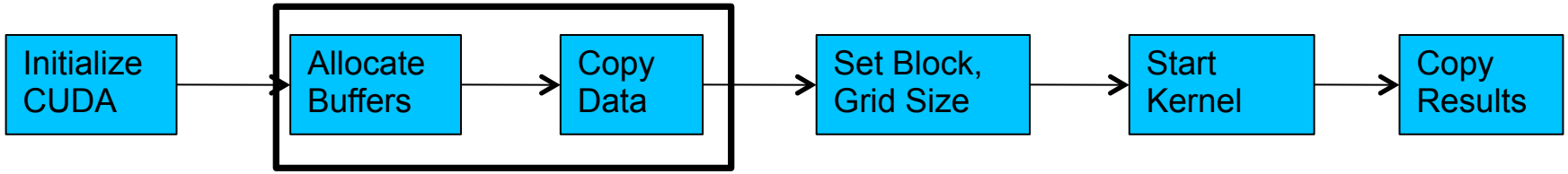
← GPU Computational kernel



blockDim.x = 32

```
tid = blockIdx.x * blockDim.x + threadIdx.x
```

Vector Addition Example



- cudaMalloc allocates space in the global memory

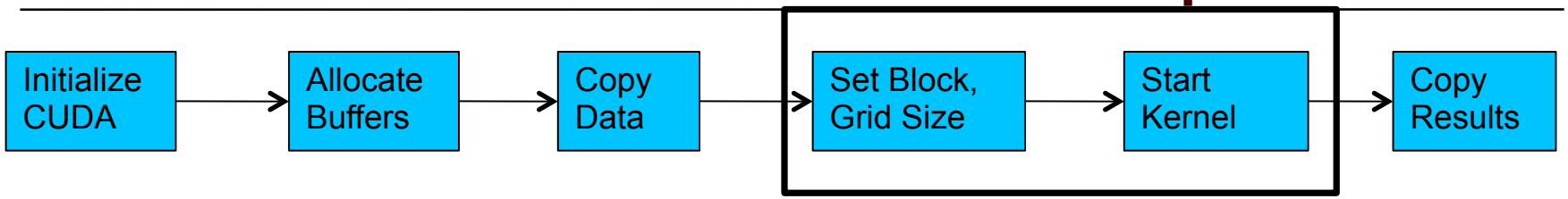
```
float *d_A, *d_B, *d_C;
cudaMalloc(&d_A, sizeof(float)*N);
cudaMalloc(&d_B, sizeof(float)*N);
cudaMalloc(&d_C, sizeof(float)*N);
```

- cudaMemcpy copies from host to global memory over PCI

```
cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
```

NUCAR

Vector Addition Example



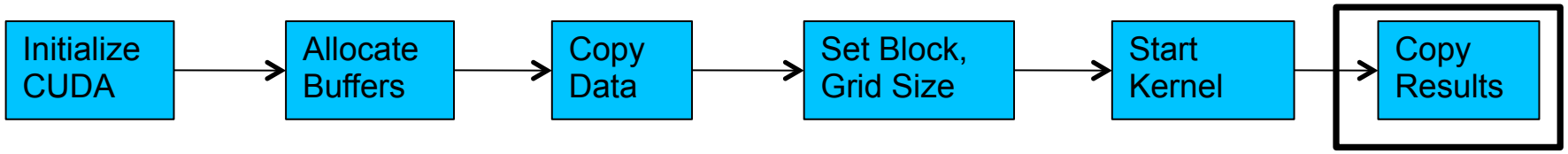
- dim3 – A 3D Vector data type which is used to pass thread and block configuration
  - Natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

```
dim3 dimBlock(32,1);
dim3 dimGrid(N/32,1);
```

- Launch Kernel Call

```
gpuArrayAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
```

Vector Addition Example



- Read results back to host

```
cudaMemcpy(C, d_C, sizeof(float)*N, DtoH);
```

- Cleanup memory and end program
- Our first CUDA program is finished ☺

NUCAR

Summary of Relevant Identifiers

Philosophy: Minimal set of extensions necessary to expose architecture

Function qualifiers:

```
__global__ void MyKernel() { }
__device__ float MyDeviceFunc() { }
```

Variable qualifiers:

```
__constant__ float MyConstantArray[32];
__shared__ float MySharedArray[32];
```

Execution configuration:

```
dim3 dimGrid(100, 50); // 5000 thread blocks
dim3 dimBlock(4, 8, 8); // 256 threads per block
```

Kernel Launch

```
MyKernel <<< dimGrid, dimBlock >>> (...); // Launch kernel
```

NUCAR

Vector Addition (GPU)

```
int main() {
   int N = 4096;
   float *A = (float *)malloc(sizeof(float)*N); init(A);
   float *B = (float *)malloc(sizeof(float)*N); init(B);
   float *C = (float *)malloc(sizeof(float)*N);
   float *d_A, *d_B, *d_C;
   cudaMalloc(&d_A, sizeof(float)*N);
   cudaMalloc(&d_B, sizeof(float)*N);
   cudaMalloc(&d_C, sizeof(float)*N);

   cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
   cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
   dim3 dimBlock(32,1);
   dim3 dimGrid(N/32,1);

   gpuArrayAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);

   cudaMemcpy(C, d_C, sizeof(float)*N, DtoH);

   cudaFree(d_A);
   cudaFree(d_B);
   cudaFree(d_C);
   free(A); free(B); free(C);
```

← Allocate memory on GPU

← Initialize memory on GPU

← Configure threads

← Run kernel (on GPU)

← Copy results back to CPU

← Deallocate memory on GPU

Global Memory Access in GPUs

BAD Access

```
__global__ void
bad_kernel(float *x)
{
 int tid = threadIdx.x + blockDim.x*blockIdx.x;
x[1000*tid] = threadIdx.x;
}
```

GOOD Access

```
__global__ void
good_kernel(float *x)
{
int tid = threadIdx.x + blockDim.x*blockIdx.x;
x[tid] = threadIdx.x;
}
```

- Global memory accessed via 32, 64, or 128-byte transactions
- No of transactions depend on size of data accessed by thread and distribution of the memory addresses across the threads
- **Coalescing:** combining memory requests across threads into a single transaction

NUCAR

Coalescing Data Access

- Memory access requirements between threads depend on compute capability of device
- Memory accesses are handled per 16 or 32 threads
- For devices of capability 2.x, memory transactions are cached
- Data locality is exploited to reduce impact on throughput
  - **Temporal locality:** data accessed is likely to be used in future,
  - **Spatial locality:** neighboring data is also likely to be reused
- Distribution of addresses across threads to get coalescing is very inflexible for older devices (Pg 168 Progg. Guide v4.0)

Application 1: Image Rotation

- Rotate an image by a given angle
- A basic feature in image processing applications

Original Input Image

Rotated Output Image

NUCAR

Example 1 - Image Rotation

- A common image processing routine
  - Applications in matching, alignment, etc.
- New coordinates of (x1,y1) when rotated by an angle Θ around (x0,y0)

$$x_2 = \cos(\theta) * (x_1 - x_0) - \sin(\theta) * (y_1 - y_0) + x_0$$
$$y_2 = \sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0) + x_0$$

- By rotating about the origin (0,0) we get

$$x_2 = \cos(\theta) * (x_1) - \sin(\theta) * (y_1)$$
$$y_2 = \sin(\theta) * (x_1) + \cos(\theta) * (y_1)$$

Original Image

Rotated Image (90°)

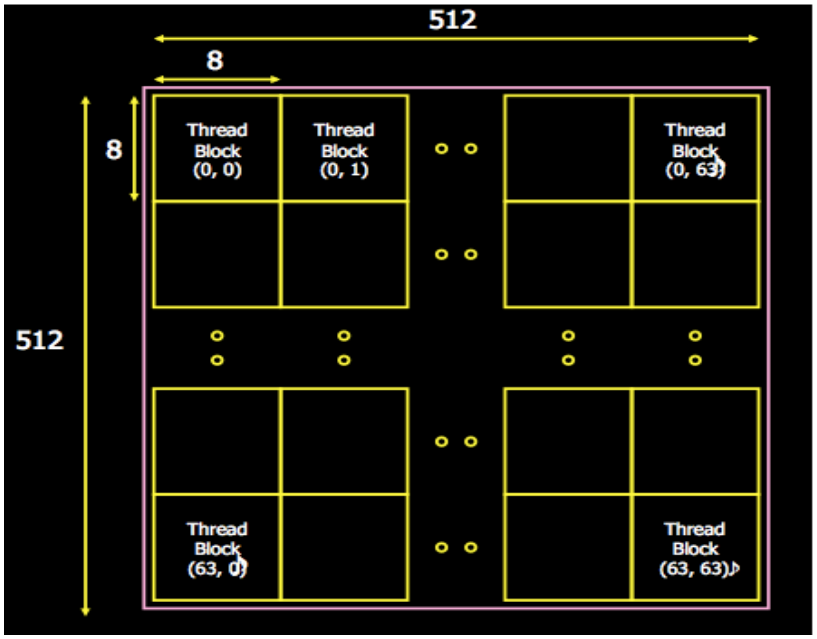
NUCAR

Application 1: Image Rotation

- What the application does:
  - Step 1. Compute a new location according to the rotation angle (trigonometric computation)
  - Step 2. Read the pixel value of original location
  - Step 3. Write the pixel value to the new location computed at Step 1
- Create the same number of threads as the number of pixels
- Each thread takes care of moving one pixel

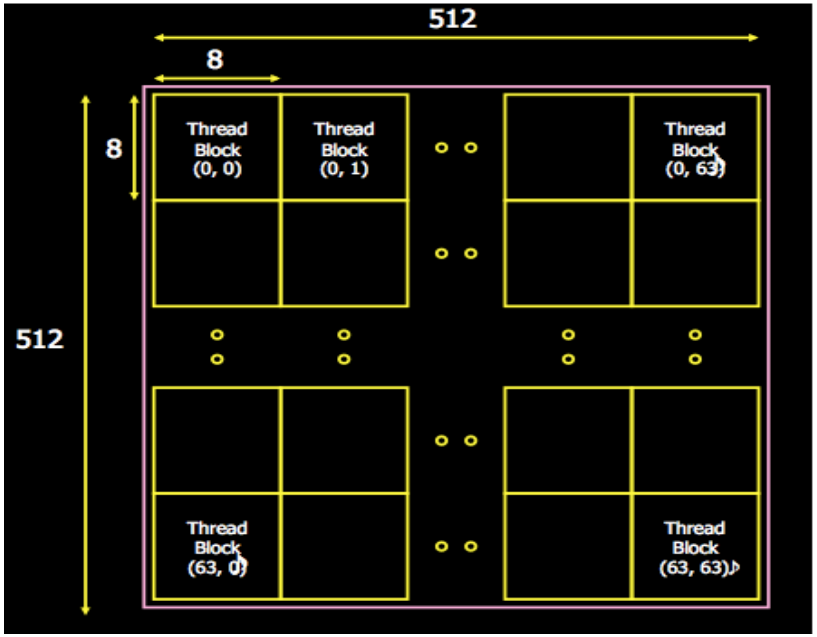
Image Rotation

- Input: To copy to device
  - Image (2D Matrix of floats)
  - Rotation parameters
  - Image dimensions
- Output: From device
  - Rotated Image



Simplified Image Rotation Kernel

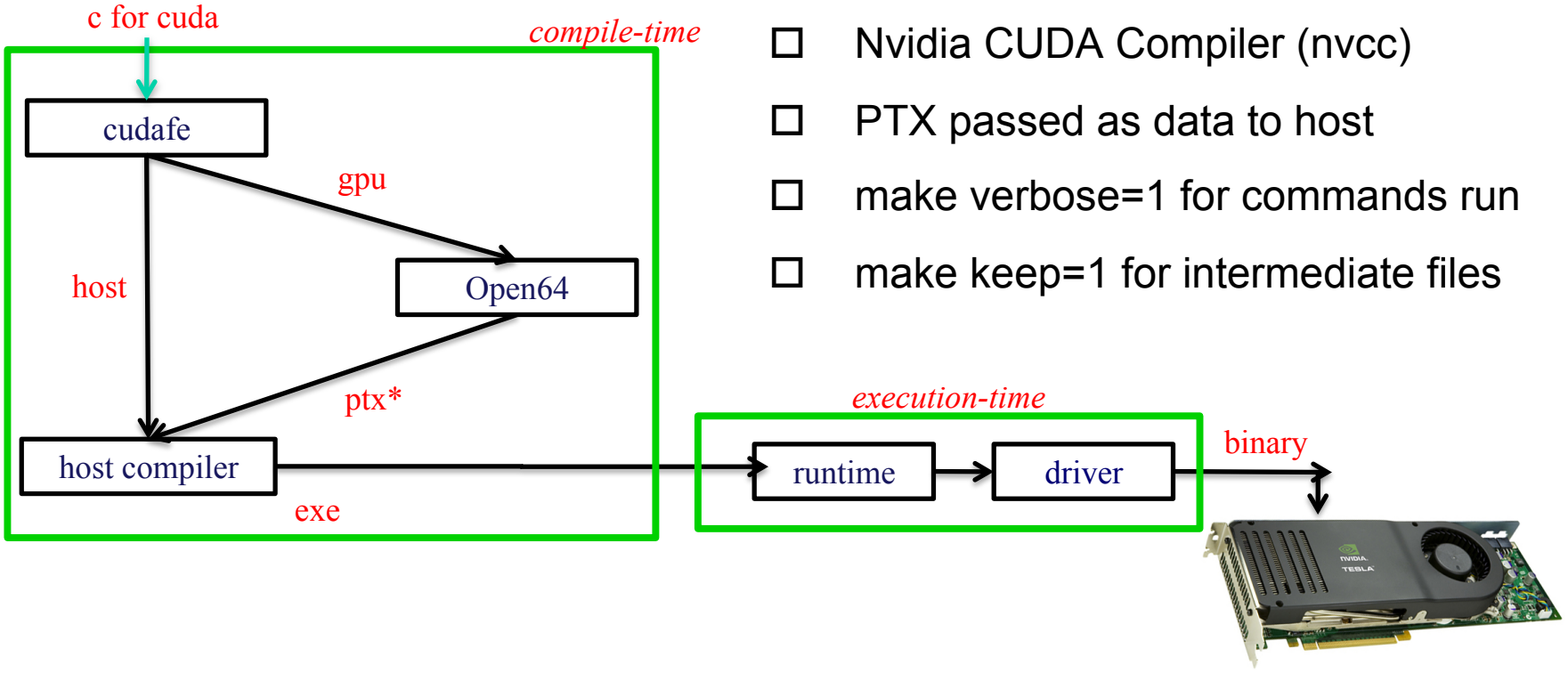
```
__global__ void
transformKernel( float* g_odata, float * d_idata,
          int width, int height)
{

    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

   //! We could use normalized coordinates here if we
   //! were using  textures
   float u = x; float v = y; //Just a 90o flip

   int new_y = int(tv);
   int new_x = int(tu);

    g_odata[ y*width + x] = d_idata[new_y * width +new_x];

}
```



Implementation Steps – Hands on

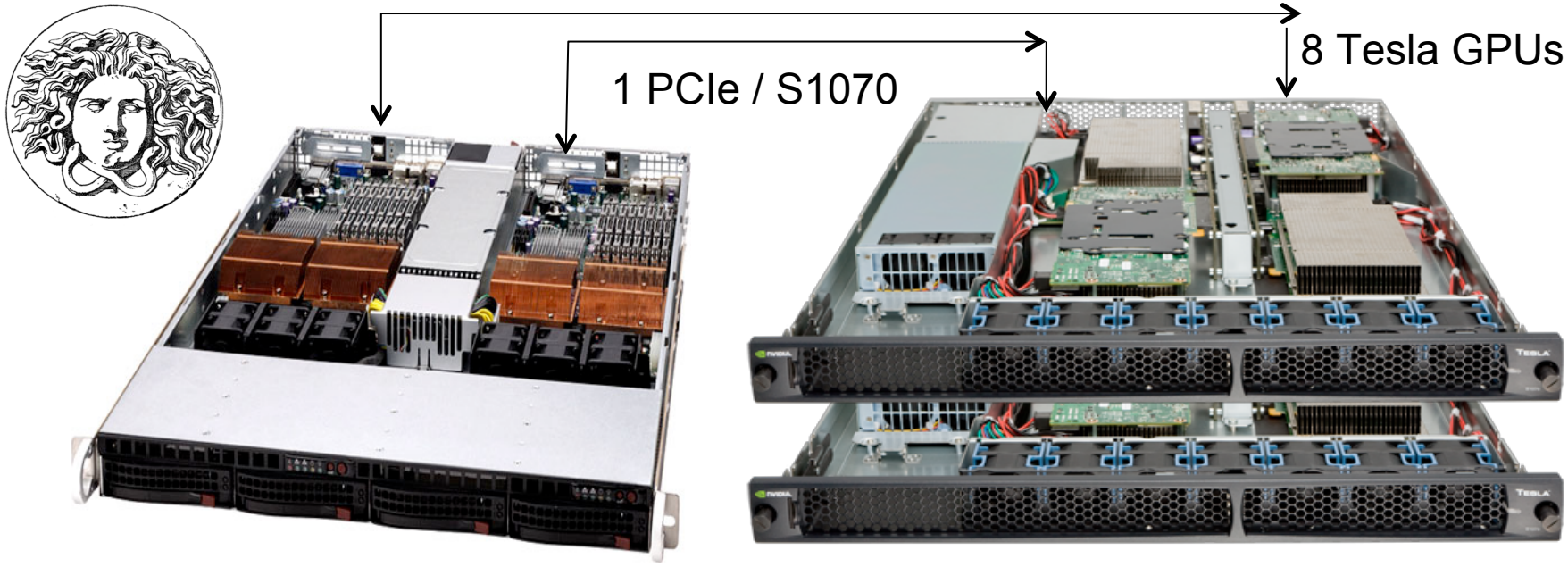
- Copy image to device by enqueueing a write to a buffer on the device from the host
- Decide the work group dimensions
- Run the Image rotation kernel on input image
- We will use the provided Nvidia utilities for image handling
- Copy output image to host by enqueueing a read from a buffer on the device
- Look at Vector add for help and syntax
- cp /sg

Compiling CUDA - C



- Nvidia CUDA Compiler (nvcc)
- PTX passed as data to host
- make verbose=1 for commands run
- make keep=1 for intermediate files

Medusa Cluster – Nvidia Subsystem



compute-0-8

~ 8TFlops in 3 U

NUCAR

Application 1: Image Rotation

- Replace ??? in the skeleton with your own CUDA code
  - Add the cudaMalloc and the cudaMemcpy calls
  - Compile with Makefile and execute
- Goals are
  - Understand how to use GPU for data parallelism
  - To know how to map threads to data

CUDA Abstractions

- Millions of lightweight threads - Simple decomposition
- Hierarchy of concurrent threads - Simple execution model
- Later we will cover :-
  - Lightweight synchronization primitives
    - Simple synchronization model
  - Shared memory model for cooperating threads
    - Simple communication model

Input vs. Output Decomposition

- Identify the data on which computations are performed
- Partition data into sub-units
  - Partition can be as per the input, output or intermediate dimensions for different computations
  - Data partitioning induces one or more decompositions of the computation into tasks e.g., by using the owner computes
- Input decomposition: Cases where we don't know size of output (e.g. finding occurrences in a list)
- Output decomposition: Cases where more than one element of the input is required (e.g. matrix multiplication)

Application 2: Matrix Multiplication

$$A_{HA,WA} \times B_{HB,WB} = C_{HC,WC}$$

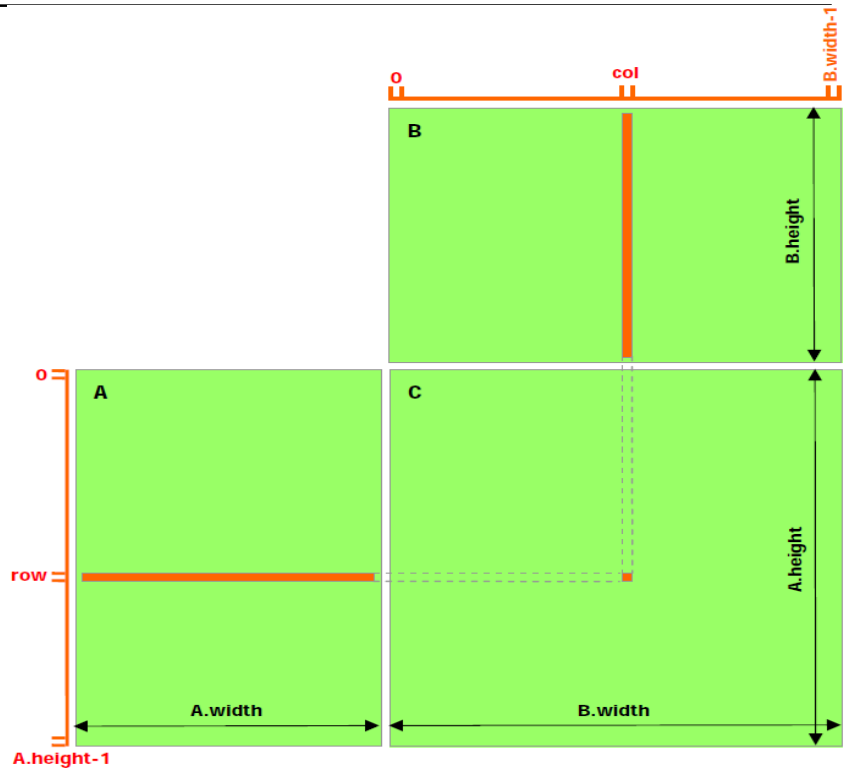
$$\begin{bmatrix} a_{0,0} & & \\ & & \\ a_{HA,0} & & a_{HA,WA} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & & \\ & & \\ b_{HB,0} & & b_{HB,WB} \end{bmatrix} = \begin{bmatrix} c_{0,0} & & \\ & & \\ c_{HC,0} & & a_{HC,WC} \end{bmatrix}$$

```
for (int i=0; i < HC; i++)
        for (int j=0; i < WC; j++)
                for (int k=0; i < WA; k++)
                        C[i][j] += A[i][k] * B[k][j];
```

NUCAR

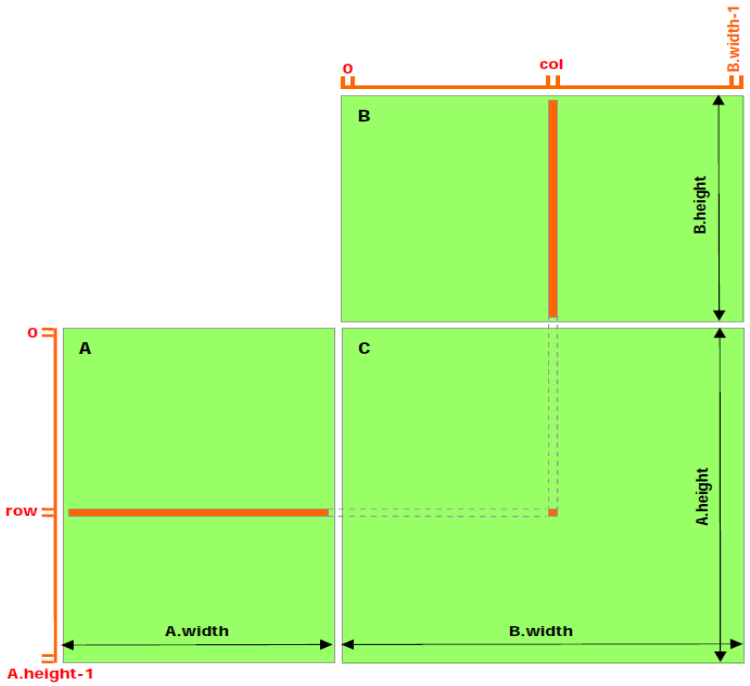
Application 2: Matrix Multiplication

- An $O(n^3)$ computation
- C[i][j] computed in parallel
  - An output decomposition
  - Multiple I/P elements per O/P
  - No of threads = No of elements in C
  - Each thread works independently



Matrix Multiplication Kernel

```
__global__ void
matrixMul ( float * C, float * A, float * B, int wA, int wB)  {

        //! matrixMul( float* C, float* A, float* B, int wA, int wB)
        //! Each thread computes one element of C
        //! by accumulating results into Cvalue
         float Cvalue = 0;
        //! Global index of thread calculated
         int row =blockIdx.y *blockDim.y +threadIdx.y;
         int col =blockIdx.x *blockDim.x +threadIdx.x;
         int wC = wB;

        //!Each thread reads its own data from global memory
         for(int e = 0; e < wA; e++)
                  Cvalue +=   A[row * wA + e] * B[e * wB + col];
           C[row * wC + col] = Cvalue;
}
```



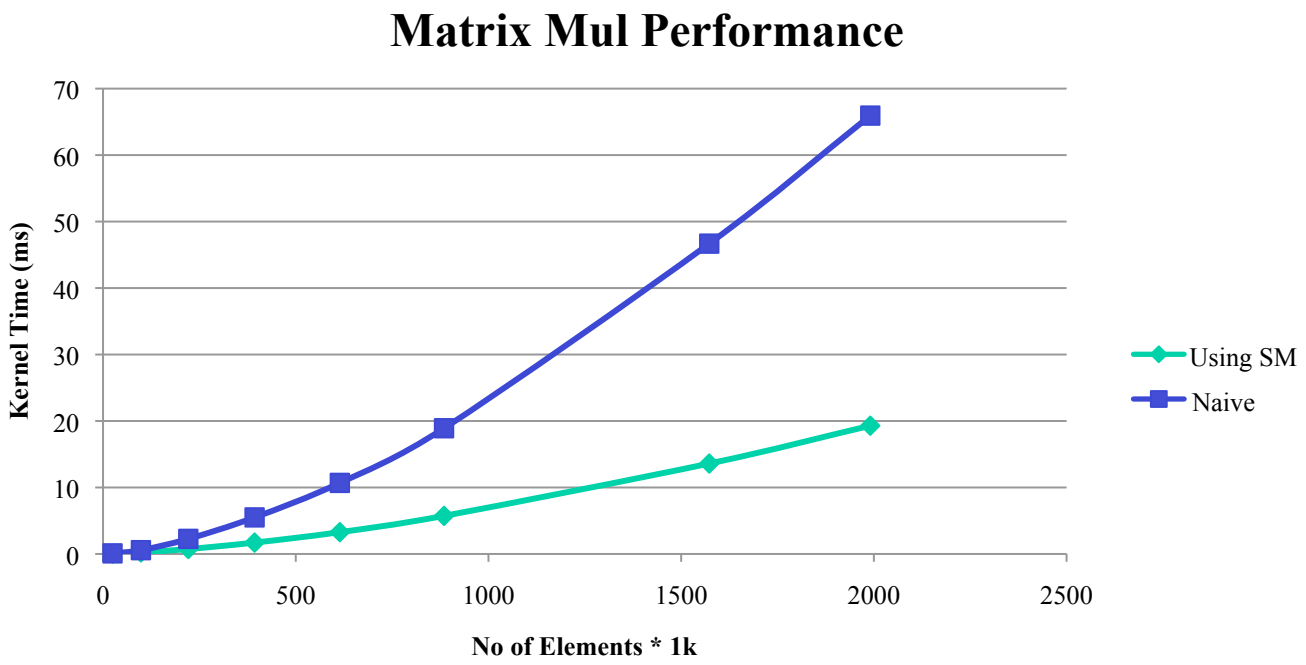
Performance of Matrix Mul

- Previous implementation – Poor Scaling - Why ?
  - No of operations
    - Per thread reads = (Row + Col)
    - Per thread computation = 2(Row + Col)
    - 1 Mul and 1 Add per access
- Redundant memory accesses
  - Each thread reads in **whole** row and **whole** column
  - How do we improve it ? And if its this bad, why discuss it ?

NUCAR

Matrix Multiplication Performance

- Lets compare the shared memory

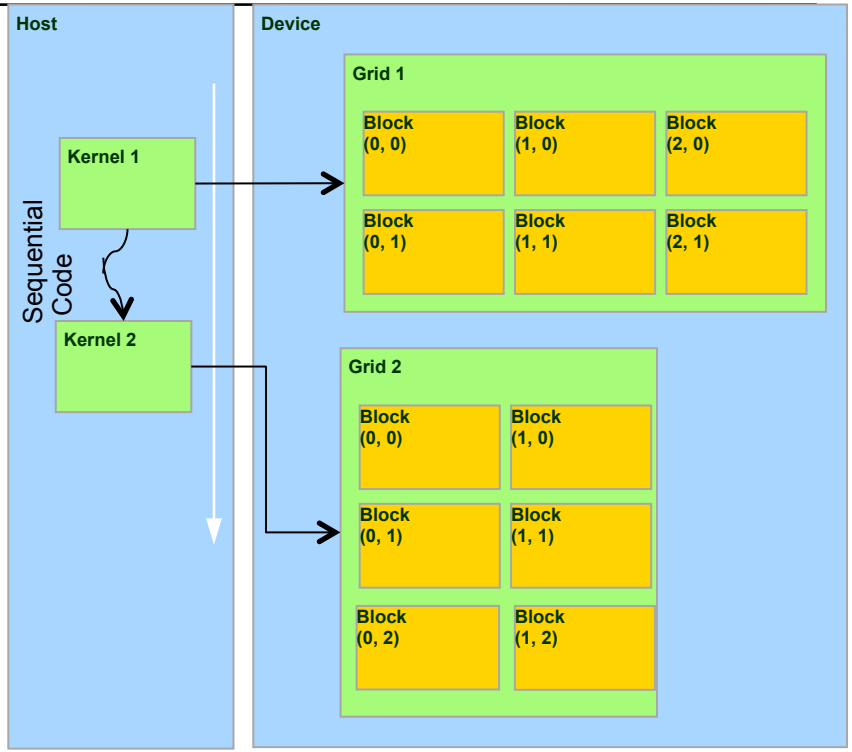


Example Takeaways

- What have we learned through the two projects ?
- Understood a massive parallel computing on GPU
- Experienced what CUDA programming looks like
- Understood how to decompose a simple problem
- Experienced solving problem in massively parallel fashion

Steps Porting to CUDA

- Create standalone C version
- Multi-threaded CPU version (debugging, partitioning)
- Simple CUDA version
- Optimize CUDA version for underlying hardware
- No reason why an application should have only 1 kernel
- Use the right processor for the job



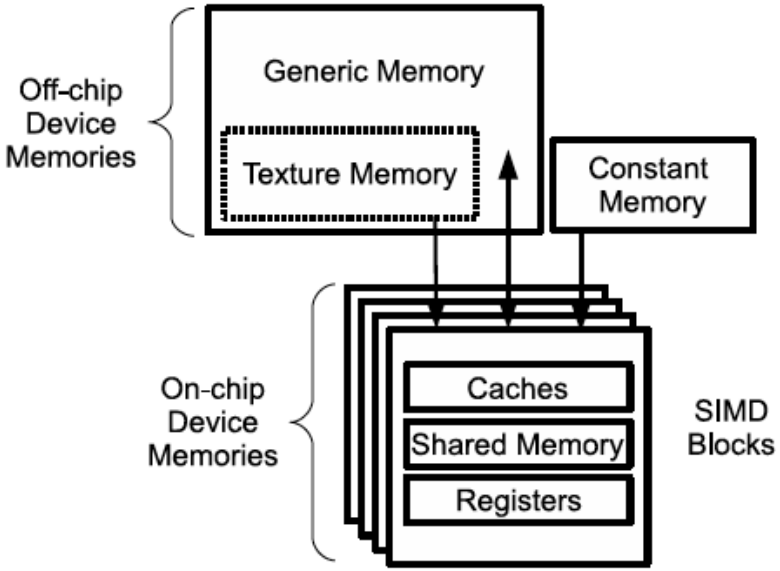
Break

- GPGPU shared memory optimization
- GPGPU Block Synchronization
- Fermi Capabilities
- Page-able and Page-locked memory
- Warps and Occupancy
- Histogram64 Example

NUCAR

GPU Memory Architecture

- Examples have not discussed using shared memory
- Critical for hiding high latency of global memory accesses
- Shared memory provides almost single cycle access to data to each scalar core
  - Shared memory is banked
- Usage rule of thumb: coalesce frequently accessed data



Heterogeneous Apple Picking – Recap…

Trees have a very different number of apples on them?

Different pickers ?

NUCAR

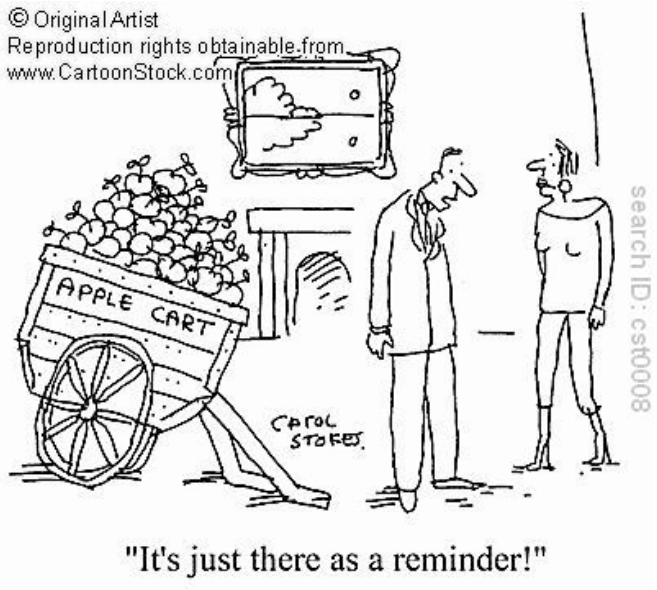
Extending Apple Picking – Again…

- Lets sell the apples in the market
- Pickers cant start pushing cart till **ALL** pickers have loaded their apples
  - Synchronization required within groups

Bulk-Synchronous programming models

Each cart can go to the market independently

cart ~ shared memory/ block



"It's just there as a reminder!"

NUCAR

Synchronization in CUDA

- Threads within block may synchronize with barriers

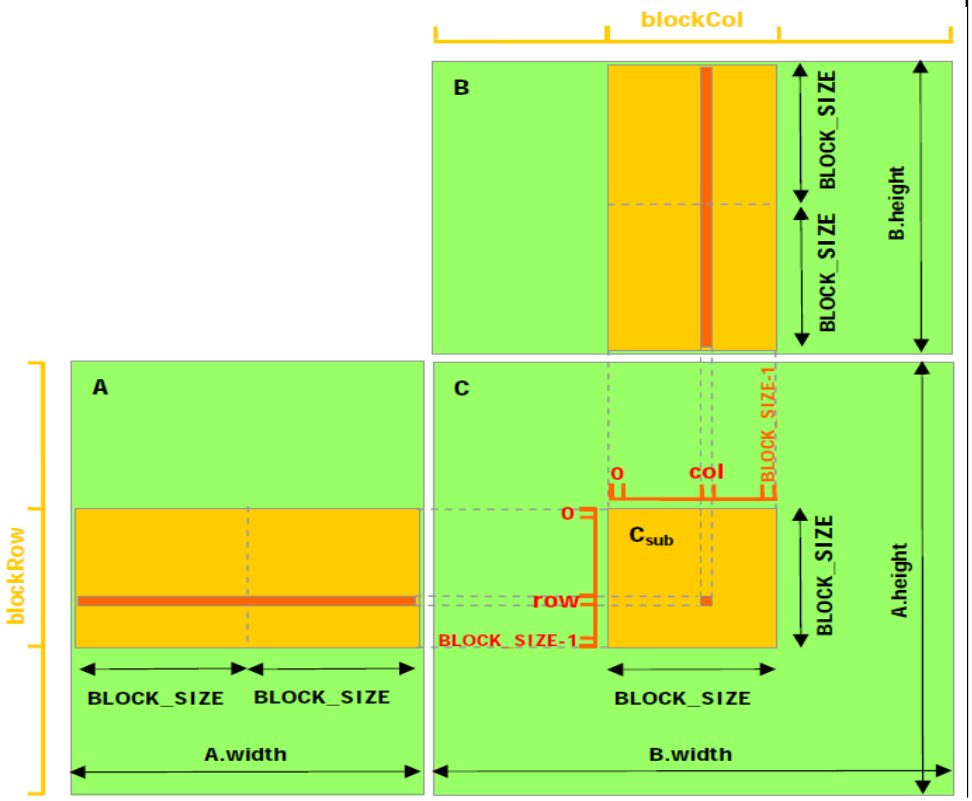
```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks **coordinate** via atomic memory operations
  - e.g., increment shared queue pointer with atomicInc()
- Implicit barrier between dependent kernels (making apple juice)

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```

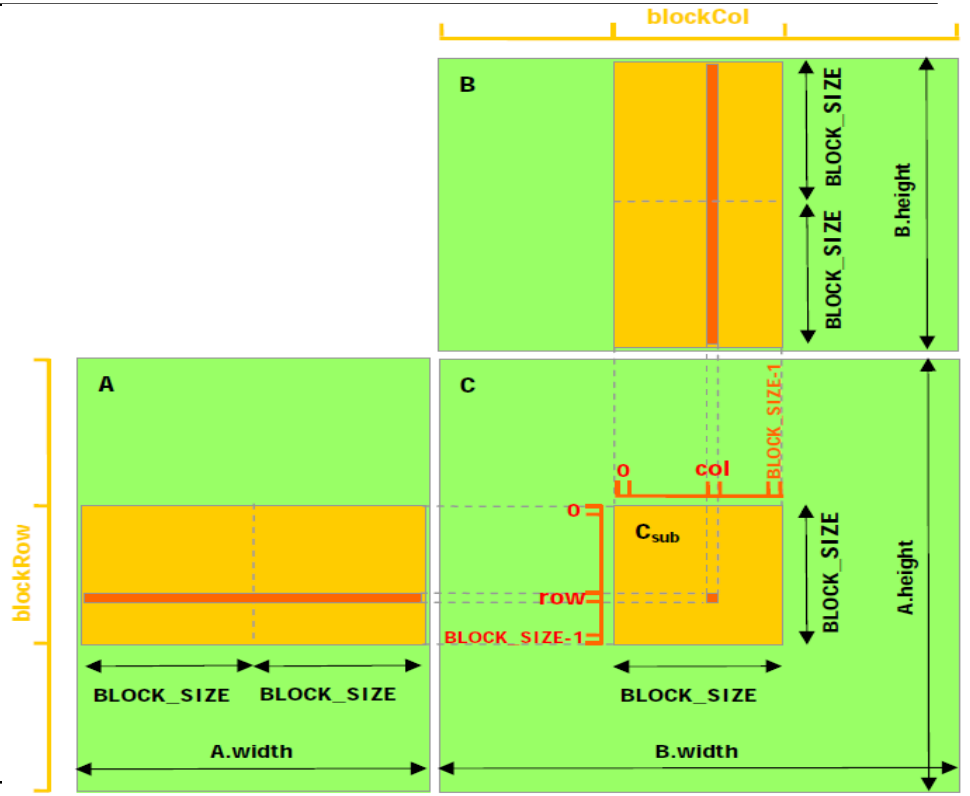
Matrix Multiplication - Blocked

- Why look at matrix mul again ?
  - Gets annoying
- Previous implementation was bad - Repetitive reads
  - Each thread worked independently
- Reuse data read by each thread
  - Inter thread-locality in access of both A and B
- Blocking is known in linear algebra for 20+ years



Matrix Multiplication - Blocked

- Shared memory optimization
- Store per-block matrices (As and Bs)
  - Shared memory is faster
- Synchronization in CUDA - Selling apple analogy
- Each thread reads in a piece of data



Matrix Multiplication - Blocked

```
__global__ void matrixMul( float* C, float* A, float* B,
int wA, int wB)
{
int bx = blockIdx.x;    int by = blockIdx.y;
int tx = threadIdx.x;   int ty = threadIdx.y;
```

Step size used to iterate through the sub-matrices of A →

```
// Index of the first sub-matrix of A processed by the block
int aBegin = wA * BLOCK_SIZE * by;
int aEnd   = aBegin + wA – 1;
int aStep  = BLOCK_SIZE;
```

Step size used to iterate through the sub-matrices of B →

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

Running Sum of result of each thread →

```
float Csub = 0;
```

Matrix Multiplication - Blocked

Loop over sub-matrices of A & B

```
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)  {
```

Declaration of the shared memory array used to store submatrix

```
__shared__ float As [BLOCK_SIZE] [BLOCK_SIZE];
__shared__ float Bs [BLOCK_SIZE] [BLOCK_SIZE];
```

Load matrices from device to shared memory; thread loads **one element**

```
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];
```

Multiply the two matrices together; each thread computes one element of the block sub-matrix

```
for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);

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

NUCAR

Matrix Multiplication - Blocked

Spot the **Race** in the for loop

Make sure the matrices are loaded
Make sure that the preceding computation is done before loading two new sub-matrices of A and B in the next iteration

```
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep)  {

__shared__ float As [BLOCK_SIZE] [BLOCK_SIZE];
__shared__ float Bs [BLOCK_SIZE] [BLOCK_SIZE];

AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];
                                    __syncthreads();

for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);

// Write the block sub-matrix to device memory;
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
                                    __syncthreads();

}
```

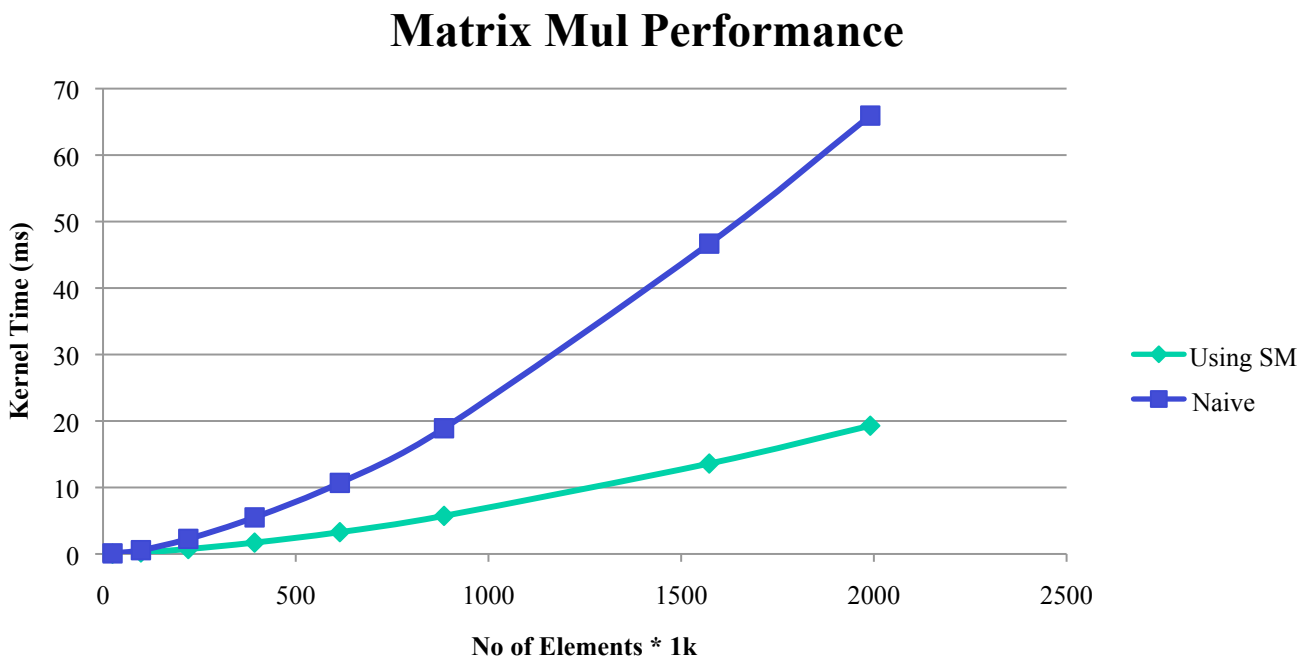
NUCAR

Application 2: Matrix Multiplication

- Hands-on performance comparison
- For a MxN matrix
  - Count no of global reads / thread
  - Count no of global writes / thread
- Compare blocking vs non blocking performance
- You can use the CUDA visual profiler later to count the number of memory accesses.
  - Note: they may not be the same because of coalescing

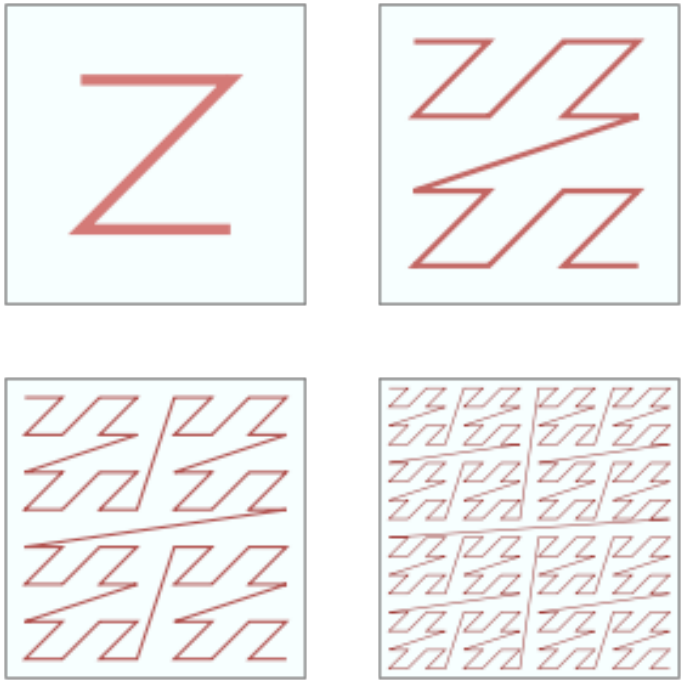
Matrix Multiplication Performance

- Lets compare the shared memory



Textures and Images

- Textures are allocated in global memory and cached.
  - Cache size ~6-8KB per mp,
  - Optimized for 2D locality in accesses
- Constant memory is also cached
- Use to optimize the image rotation example
  - Uncoalesced reads from global memory

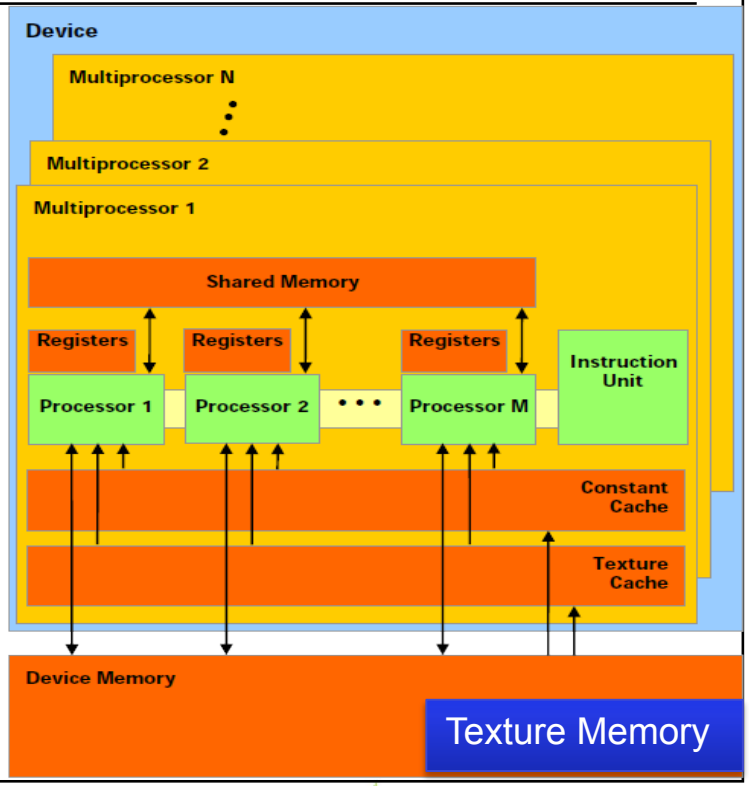


Hands On – Try simpletexture

- Defined at file scope as a type texture:

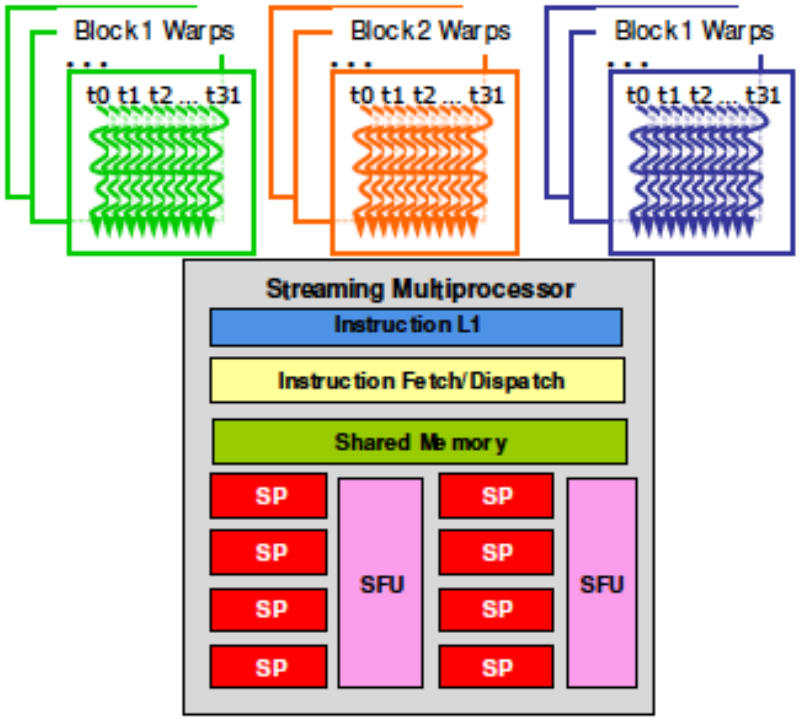
  texture<Type, Dim, ReadMode> mytex;

- Textures are referenced using floating-point coordinates in the range [0, N) or if normalized [0,1.0).
- Addressing mode can be
  - Clamped, 1.25 -> 1.0 in [0,1.0) or
  - Wrapped, eg 1.25 -> 0.25
- Value returned can be a single element or a interpolated value



Warps and Occupancy

- Multiprocessor creates and executes threads in groups of 32 parallel threads called warps.
- Threads in a warp start at the same program address
  - Have individual instruction and register state
  - Free to branch and execute independently
- Enables more applications (See Histogram256)



Using the Occupancy Calculator

- The fact that all instructions in a warp execute together in lock step can be used to our advantage
- **NOTE:** Warps are not part of the CUDA language definition
- Cost of warp divergence = sum of if + sum of else block
- Occupancy is the ratio of active warps to the maximum number supported on a multiprocessor of the GPU
- Determines how efficient the kernel will be on the GPU .
- Get statistics for occupancy calculator with `make keep=1`

Using the Occupancy Calculator

CUDA GPU Occupancy Calculator

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

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:		(Help)
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:		(Help)
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

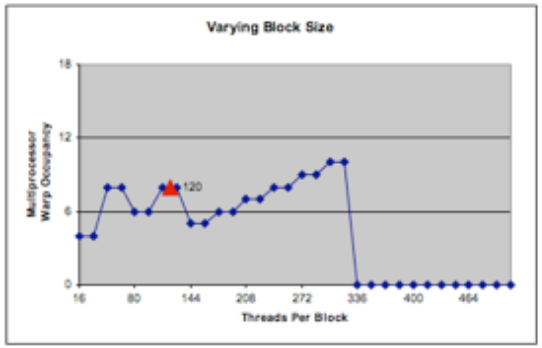
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	6
Limited by Registers / Multiprocessor	2
Limited by Shared Memory / Multiprocessor	4

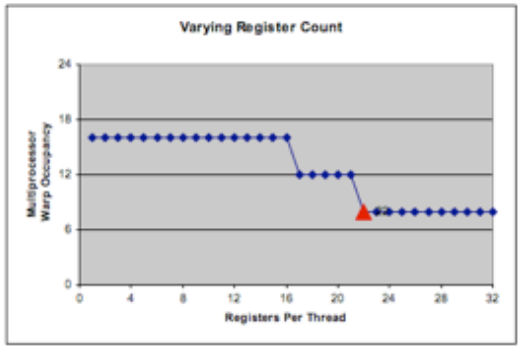
Thread Block Limit Per Multiprocessor is the minimum of these 3

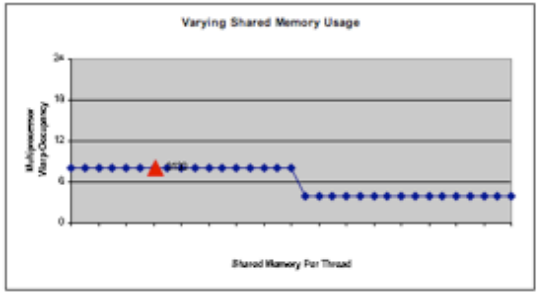
CUDA Occupancy Calculator	
Version:	1.2

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 Tradeoffs

- Occupancy is an empirical measure
  - A last order optimization step and device dependent
- More threads / block
  - Benefits – Helps compute bound workloads (rare for GPUs)
  - Drawbacks – Reduces number of registers per thread and shared memory per block, less blocks to hide latency
- Optimum threads / block
  - IO bound workload has just enough warps to switch with

Experiment with Occupancy

- Download excel file from course web page
  - http://developer.nvidia.com/cuda-downloads
- Occupancy is not a performance counter, it is simply a ratio
- Try with non blocking and blocking matrix multiplication
  - Choose one data set
    - Note: press ‘0’ when verification is not needed
  - Vary number of threads per block

End – Class II

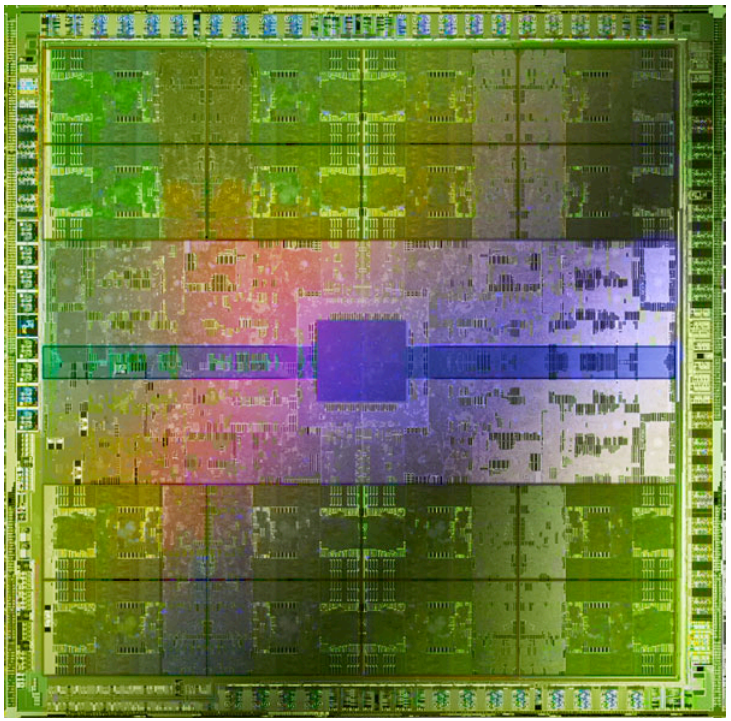
NUCAR

Note: The Next lecture should be covering material below

NUCAR

Nvidia Fermi

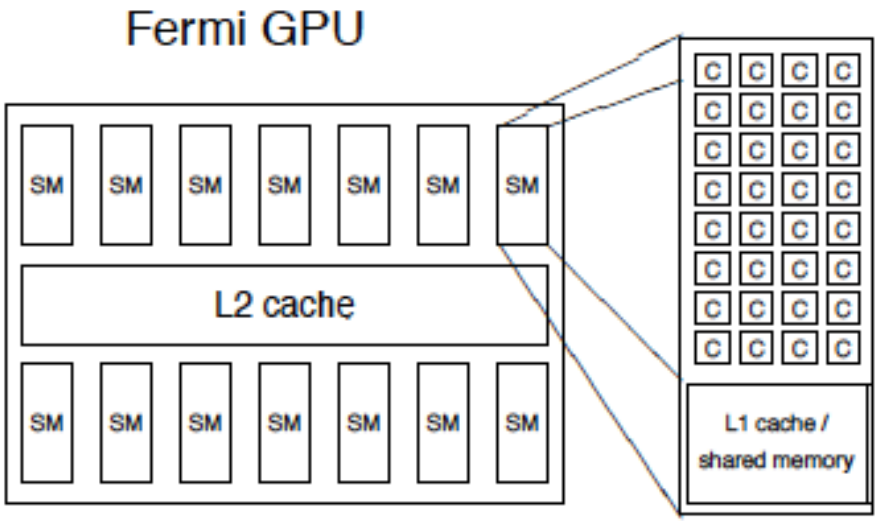
- Compute 2.0 / 2.1 devices
- Better double precision
- ECC support
- Configurable cache hierarchy
- Faster context switching
- Faster atomic operations
- Concurrent kernel execution
- Dual DMA Engines



NUCAR

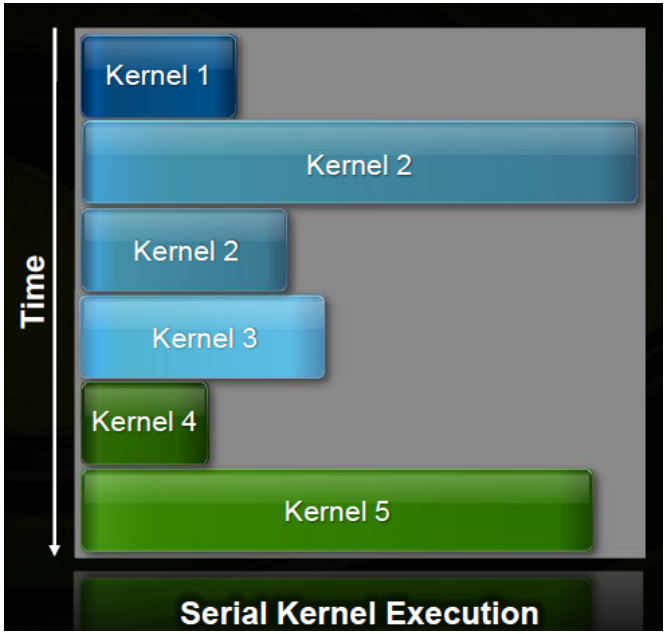
Nvidia Fermi Features

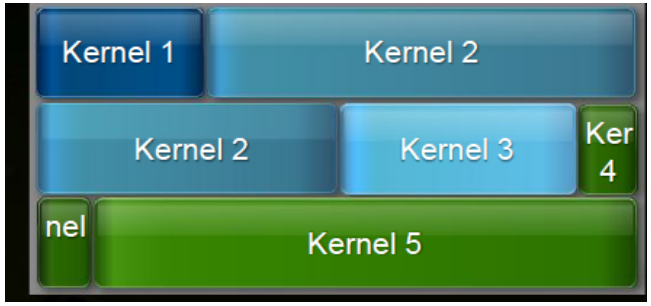
- Everything discussed till now is still relevant ☺
- ECC support - Data-sensitive applications
- Configurable Cache Hierarchy
  - Implementations unable to use shared memory
- Faster Context Switching
  - Application graphics and compute interoperation



Concurrent Kernel Execution

- Concurrent Kernel Operation - Enables smaller data sets



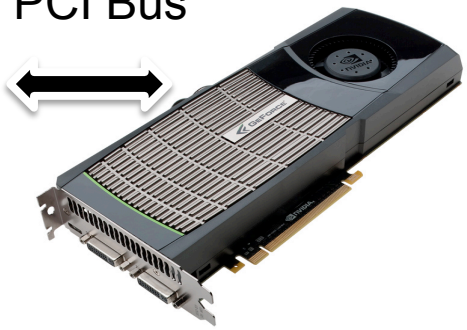


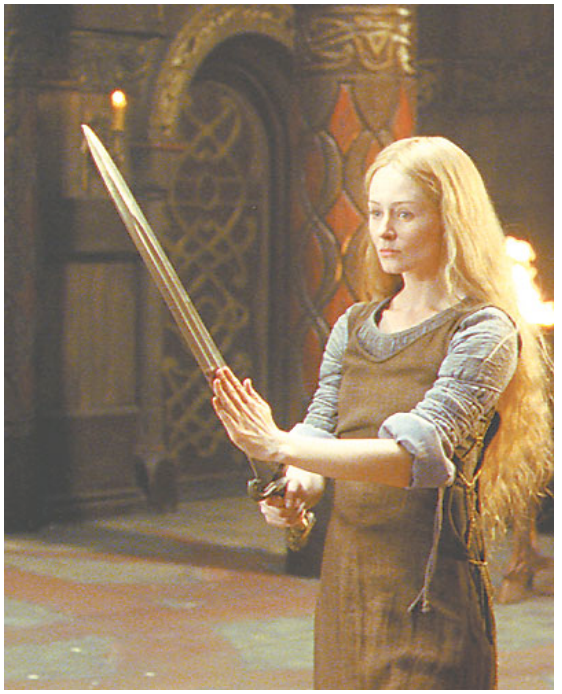
Requires knowledge of CUDA Stream API
More than enough rope provided to hang yourself

Eowyn – Fermi System

- My personal system at NEU
  - Dell XPS Gaming Platform
  - GTX-480

PCI Bus



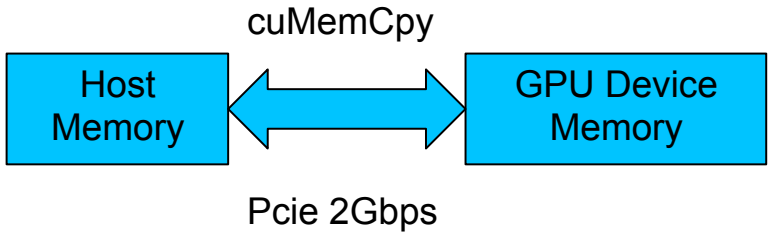


Host – Device Interaction

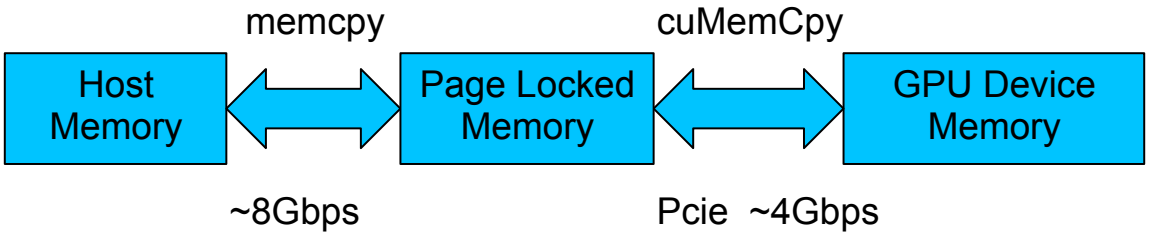
- An application dependent optimization space
- Page-locked Memory
- Asynchronous host – device Application IO
  - Used commonly in medical imaging where data is continuously fed to device
  - Use CUDA stream's asynchronous API
- Divide application into multiple kernels and keep data on device
  - This often means coding non data parallel or inefficient kernels to avoid IO

Pinned Memory Optimization

- Page-able vs. Page-locked memory
- Locked pages will not be swapped out to disk by the OS
- Allocate using cudamallochost
- Fermi + CUDA 4.0 provides non-copy pinning

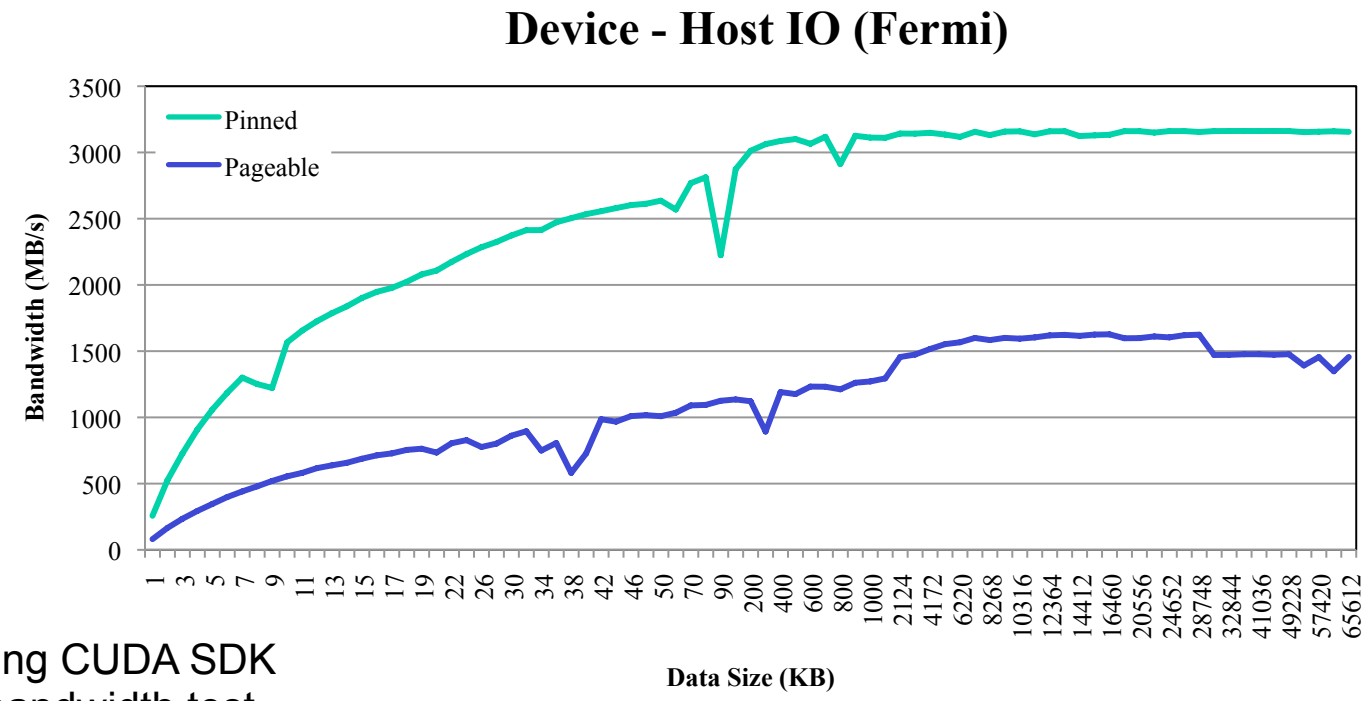


Note: excess page locking affects system performance



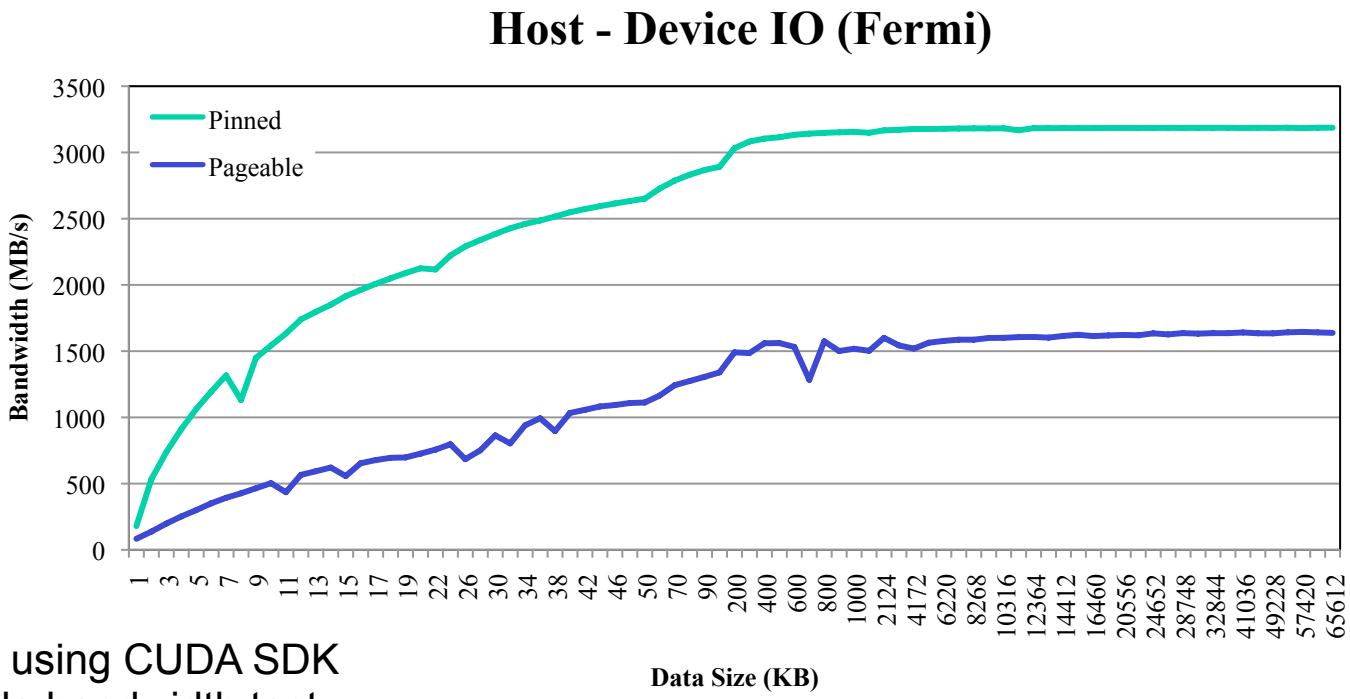
NUCAR

Performance of Page-locked Memory



Tested using CUDA SDK example bandwidth test

Performance of Page-locked Memory



Tested using CUDA SDK example bandwidth test

Device Query & Bandwidth Test

```
[./bandwidthTest] starting...
./bandwidthTest Starting...

Running on...

 Device 0: GeForce GTX 480
 Quick Mode

 Host to Device Bandwidth, 1 Device(s), Paged memory
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     1617.7

 Device to Host Bandwidth, 1 Device(s), Paged memory
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     1465.1

 Device to Device Bandwidth, 1 Device(s)
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     117711.5

[./bandwidthTest] test results...
PASSED
```

Useful tools to check your setup configuration and learn about device

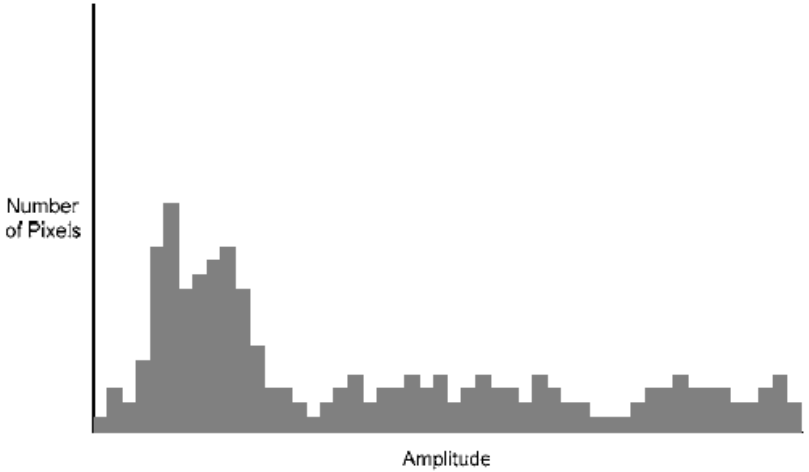
```
dmistry@eowyn:~/NVIDIA_GPU_Computing_SDK_40/C/bin/linux/release$ ./deviceQuery
[./deviceQuery] starting...
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

There is 1 device supporting CUDA

Device 0: "GeForce GTX 480"
  CUDA Driver Version / Runtime Version          4.0 / 4.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 1535 MBytes (1609760768 bytes)
  (15) Multiprocessors x (32) CUDA Cores/MP:     480 CUDA Cores
  GPU Clock Speed:                               1.40 GHz
  Memory Clock rate:                             1848.00 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Application: Histogram64

- 64 bin histogram of data
  - Build per thread subhistogram
  - Build per block sub histogram

```
for (int  i = 0; i < BIN_COUNT; i++)
        result[i] = 0;
for (int  i = 0; i < dataN; i++)
        result[data[i]]++;
```

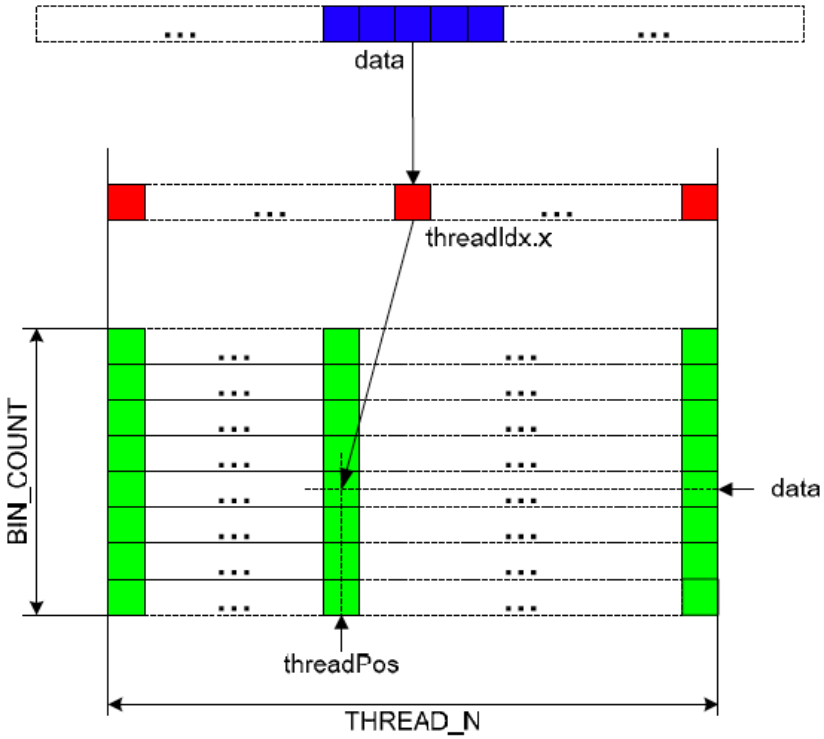
- Homework :- Try Histogram256 using local memory atomics



An example Image Histogram

Implementation of Histogram

- **Kernel 1:** Build per block histogram from per thread histogram
  - Per thread histogram in shared memory
  - Reduce to block histogram
- **Kernel 2:** Combine block histograms into final histogram



NUCAR

Histogram64 Kernel1

- Main Implementation Steps:
  - Initialization of shared memory to 0 is important
  - Make per thread histogram
  - Use 64 threads per block to aggregate per thread into a per-block histogram
- Note: Synchronization after per thread histograms is made
- Also use short data types for the thread histograms
- Later optimization step done in CUDA SDK to remove bank conflicts is left for future discussion

Optimizations in Histogram64

- A simplified version of the Histogram64 kernel is provided
- Optimizations Include
  - Using shared memory
    - Build per block histogram using data gathered by each thread
  - Group 8 bit reads into a 32 bit read
    - As discussed coalescing: needs 32 bit transactions atleast
- Provided implementation includes bank conflicts in shared memory

Summary

- We have studied the architecture of CUDA capable Nvidia GPUs
- We have seen the basics of CUDA and the relationship between the architecture and the programming model
- We have decomposed a data parallel algorithm
- We have used different architectural features of the GPU like shared and texture memory

Summary

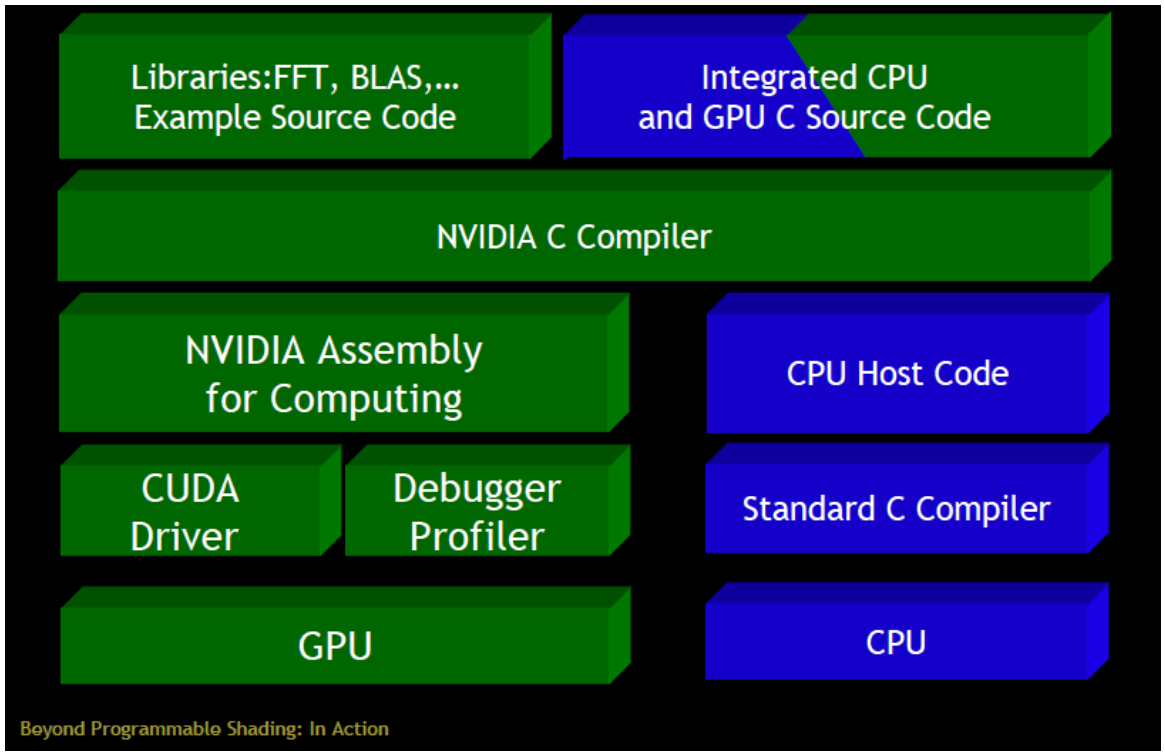
- We have optimized host-device interaction using pinned memory
- CUDA is a powerful parallel programming model
  - Heterogeneous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible - minimal but expressive changes to C
  - Interoperable - simple graphics interop mechanisms

NUCAR

Summarizing Today's Programming

- Array addition, Devicequery and BandwidthTest: Basic CUDA programming, host - device code
- Image Rotation:
  - Flipping: 2D Data Mapping
  - Image rotation extension: using texture memory
- Matrix Multiplication:
  - Naïve: Blocks and threads, coalescing data reads
  - Blocking: Using Shared memory and synchronization in blocks
- Histogram64: Using shared memory to buffer data

Nvidia - CUDA Ecosystem - Today



Productivity Tools Based on CUDA

- **Thrust -** A STL – like library for CUDA
- Linear Algebra and Mathematical Routines
  - CUBLAS and CURAND
  - **MAGMA** and CULA-Tools provide LAPACK
  - CUSP – CUDA Sparse Algebra
  - CUFFT – FFTW for GPUs
  - NPP: Performance Primitives – Video processing
  - Sections of **OpenCV**

green = Nvidia product
bold = open source

NUCAR

Programming Tools for CUDA

Solution	Approach	Availability
CUDA C Runtime	Language Integration	NVIDIA CUDA Toolkit
Fortran	Auto Parallelization	PGI Accelerator
OpenCL	Device-Level API	Khronos standard
DirectCompute	Device-Level API	Microsoft
PyCUDA	API Bindings	Open source
jCUDA	API Bindings	Freely Available
CUDA.NET	API Bindings	Freely Available
OpenCL.NET	API Bindings	Freely Available

Next Class (4/28)

- More advanced CUDA
  - Performance Tools – Using the CUDA Visual Profiler
  - Debugging Techniques – Using cuda-gdb
- Let us know any particular areas of focus you would like
  - Look at the SDK examples for topics you are interested in

More information and References

- NVIDIA GPU Computing Developer Home Page
  - http://developer.nvidia.com/object/gpucomputing.html
- CUDA Download
  - http://developer.nvidia.com/object/cuda_4_0_downloads.html
- Programming Massively Parallel Processors: A Hands-on Approach, David B. Kirk and Wen-mei W. Hwu
- Other resources
  - http://courses.engr.illinois.edu/ece498/al/

More information and References

- Beyond Programmable Shading – David Leubke
- Decomposition Techniques for Parallel Programming – Vivek Sarkar
- CUDA Textures & Image Registration - Richard Ansorge
- Setting up CUDA within Windows Visual Studio
  - http://www.ademiller.com/blogs/tech/2011/03/using-cuda-and-thrust-with-visual-studio-2010/
- SDK examples: Histogram64, Matmul, SimpleTextures

Thank You !
Questions, Comments ?

Perhaad Mistry

pmistry@ece.neu.edu

NUCAR