CUDA Introduction

João Barbosa
GRA – Visualization Software
Texas Advanced Computing Center
jbarbosa@tacc.utexas.edu

Outline

- Brief introduction of some concepts
- Programming model
- Execution model

Brief Notions

- **HOST** : CPU, not a single core but the whole system
- **DEVICE** : On CUDA the GPU it self
- **KERNEL** : The computational payload called from the host to be executed on the device.

CUDA - Compute Unified Device Architecture

- Heterogeneous programming model
  - CPU & GPU are separate devices w/ separate memory spaces
  - CPU code is standard C/C++
    - Driver API: low-level interface
    - Runtime API: high-level interface (one extension to C)
  - GPU code
    - Subset of C with extensions
- CUDA goals
  - Scale GPU code to 100s of cores, 1000s of parallel threads
    Facilitate heterogeneous computing

CUDA - Software Development

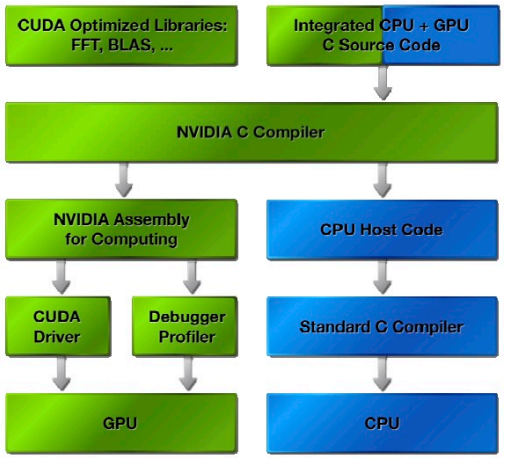


Image from Nvidia"

CPU vs GPU

- Different goals
  - GPU assumes that workload is highly parallel
  - CPU must be good at everything, parallel or not

CPU vs GPU

- CPU: minimize latency of a single thread
  - Large on-chip cache
  - Complex logic

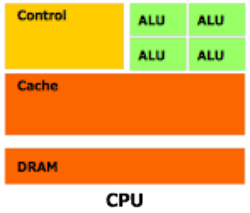


Image from "NVidia CUDA C Programming Guide" NVidia

CPU vs GPU

- GPU: maximize throughput of all threads
  - # Threads limited by resources → lots of resources
  - Thousands of threads → Smaller cache
  - Control shared across multiple threads

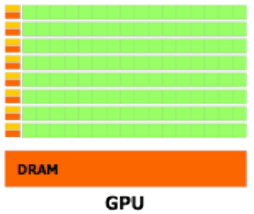


Image from "NVidia CUDA C Programming Guide" NVidia

Outline

- CUDA Programming model
  - Basic concepts
  - Data types
- CUDA Application interface
- SAXPY
  - Basic example

Hierarchy of concurrent threads

- Parallel kernels are composed of many threads
  - STMD - all threads execute the same code
  - Unique ID within the group
- Threads are group into thread blocks
  - Threads within the same block can cooperate
- Threads organized into a grid

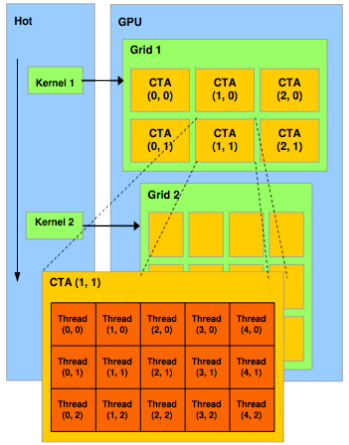
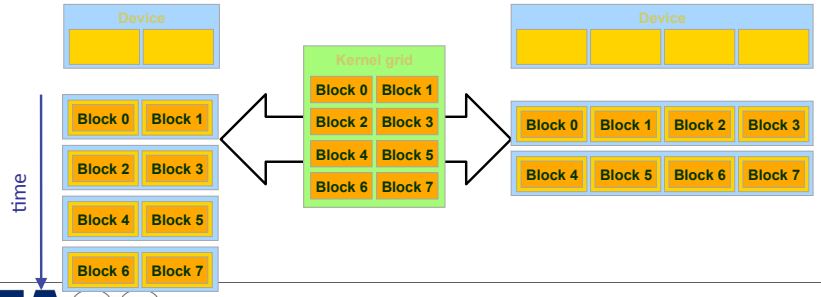


Image from "Parallel Thread Execution ISA 3.0", NVidia

Transparent Scaling

- Blocks can be assigned arbitrarily to any processor
  - Increases scalability to any number of cores
  - blocks must be independent for this reason, to accommodate various GPU architectures



From NVidia

CUDA Function keywords

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

- `__global__` always defines a kernel
- `__device__` and `__host__` may be combined
- Nothing to do with scheduling – only callable from

CUDA Function notes

- Function executed on the device
  - Limited recursion
- C++ Object / Classes are valid
  - Methods must by qualified : __device__ and __host__

Calling kernel functions

```
(...)
__global__ void HelloWorld() {
     printf("Hello World!");
}
(...)
dim3 DimGrid(1,1); // 1 * 1 # Blocks
dim3 DimBlock(1,1,1); // 1 * 1 * 1 # Threads per block
HelloWorld <<< DimGrid , DimBlock >>>();
```

- All kernels calls are asynchronous
  - Explicit sync is required for blocking

Compiling code

```
$ nvcc -O3 helloWorld.c -o helloWorld
$ ./helloWorld
Hello World!
$
```


Memory hierarchy model

- Thread Scope
  - R/W registers (RW)
  - "Local" memory (RW)
- Block scope
  - Shared memory (RW)
- Grid scope
  - Global memory (RW)
  - Constant memory (RO)
  - Texture Memory (RO)

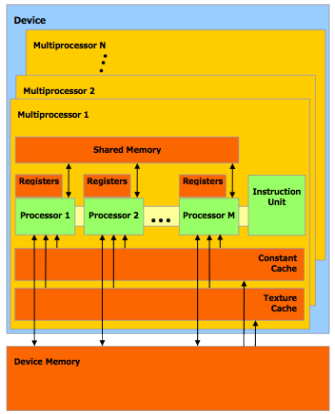


Image from "Parallel Thread Execution ISA 3.0", NVidia

Memory hierarchy model

- Global Memory
  - Communication between host and device
  - Content visible to all threads
  - Long latency access
- Shared Memory
  - Low latency access
  - used for intra-block / inter-thread cooperation

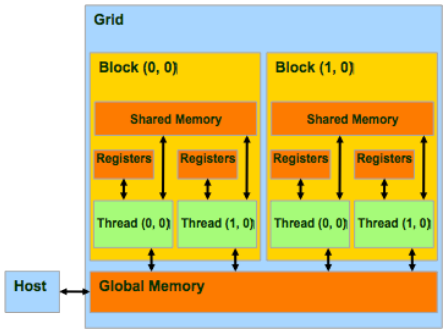


Image from "Parallel Thread Execution ISA 3.0", NVidia

CUDA Variable type qualifiers

Variable declaration	Memory	Scope	Lifetime
`__local__ int LocalVar;`	local	thread	thread
`__shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__constant__ int ConstantVar;`	constant	grid	application

- Automatic variables (without qualifier)
  - Mapped to registers
  - May spill to GPU main memory
- Shared is used for intra-block / inter-thread cooperation

Memory sharing

- "Local" Memory
  - Private per thread
  - Register Spilling
- Shared Memory
  - Threads of the same block
  - Inter-thread comm.
- Global Memory

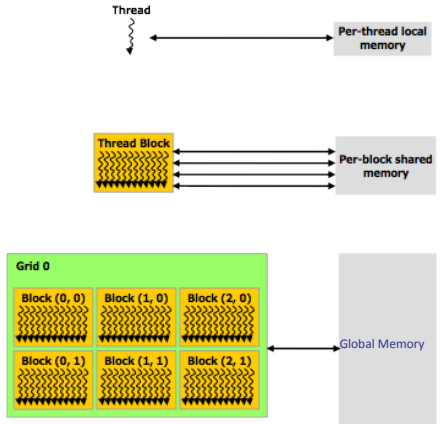


Image from "Parallel Thread Execution ISA 3.0" NVidia

SAXPY example

SAXPY Problem

- Two vectors and a scalar
  - X, Y - Vectors
  - alpha - Scalar
- R = alpha * X + Y

SAXPY Problem - CPU serial

```
void saxpy_cpu(float *X, float *Y, float alpha, float *R, int
  N){

      for(int index=0; index < N; index++)
        R[index] = alpha * X[index] + Y[index];
    }
```


SAXPY Problem - CPU parallel

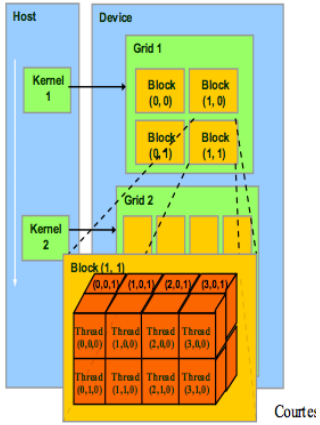
```
void saxpy_cpu(float *X, float *Y, float alpha, float *R,
  int N){

      for(int index=0; index < N; index++)
        R[index] = alpha * X[index] + Y[index];
    }
```


SAXPY Problem - CPU parallel

```
void saxpy_cpu(float *X, float *Y, float alpha, float *R,
  int N){
      #pragma omp parallel for
      for(int index=0; index < N; index++)
        R[index] = alpha * X[index] + Y[index];
    }
```

Hierarchy of concurrent threads

- **blockIdx**.{x,y,z} - stores the block identification
  - Note: Z is always one (for now)
- **threadIdx**.{x,y,z} - stores the thread identification
- **blockDim**.{x,y,z} - stores Block dimension
- **gridDim**.{x,y,z} - stores Grid dimension



SAXPY Problem - CUDA kernel

```
__global__
void saxpy_kernel(float *X, float *Y, float alpha, float
  *R, int N){
      int index = blockIdx.x * blockDim.x + threadIdx.x;
      if(index >= N) return; // Checks the array limit
      R[index] = alpha * X[index] + Y[index];
   }
```


Main function

```
#define N 256
int main(int argc, char* argv[]) {
     float *X_h, *Y_h, *R_h; // Pointer for host objects
     float *X_d, *Y_d, *R_d; // Pointer for device objects
     float alpha = 2.5f;

     /* Allocate Host Memory */
     /* Allocate Device Memory */
     /* Init vectors */
     /* Copy data host to device */
     /* Call Kernel */
     /* Copy data device to host */

     return 0;
}
```


Device Memory Allocation

- cudaMalloc(void **ptr, size_t size)
  - equivalent to “malloc”
  - allocates memory in device global memory
  - ptr - stores pointer to allocated object
  - size - request allocation size
- cudaFree(void* ptr)
  - releases allocated object in device global memory
  - ptr - pointer to device object

Main function

```
/* Allocate Host Memory */
X_h = (float*)malloc(N*sizeof(float));
Y_h = (float*)malloc(N*sizeof(float));
R_h = (float*)malloc(N*sizeof(float));
/* Allocate Device Memory */
cudaMalloc((void**)&X_d, N*sizeof(float));
cudaMalloc((void**)&Y_d, N*sizeof(float));
cudaMalloc((void**)&R_d, N*sizeof(float));
```

Synchronous data transfer

- cudaMemcpy(void* dst, void* src, size_t size, cudaMemcpyKind kind)
  - Synchronous memory data transfer
  - Requires
    - dst - destination pointer
    - src - source pointer
    - size - size
    - kind - Transference type

Synchronous data transfer

- cudaMemcpy(void* dst, void* src, size_t size, cudaMemcpyKind kind)
  - Types of transfer
    - Host to Host
    - Host to Device
    - Device to Host
    - Device to Device
- Asynchronous version are available
- Fermi: new kind cudaMemcpyDefault

Main function

```
/* Init vectors */
initVectors(X_h,Y_h,R_h)
/* Copy data host to device */
cudaMemcpy(X_d,X_h,
  N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(Y_d,Y_h,
  N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(R_d,R_h,
  N*sizeof(float),cudaMemcpyHostToDevice);
```

Main function

```
/* Call Kernel */
dim3 dimGrid(1,1);
dim3 dimBlock(1,N);

saxpy_kernel <<< dimGrid, dimBlock
  >>>(X_d,Y_d,alpha,R_d,N);

/* Copy data device to host */
cudaMemcpy(R_h,R_d,
  N*sizeof(float),cudaMemcpyDeviceToHost);
/* Remember cudaMemcopy is synchronous */
```


1st Lab

- Login to your search account
- Copy the tar file to your home directory
  - cp /tmp/tranning-lab1.tar.bz2 .
  - tar xjf tranning-lab1.tar.bz2
- Move to tranning/saxpy
  - src folder contains the source for the exercise
  - bin will hold the final binary

1st Lab

- Edit the saxpu.cu file and complete the kernel
  - Use your favorite editor (vim, nano, …)
- To compile run "make" on tranning/saxpy

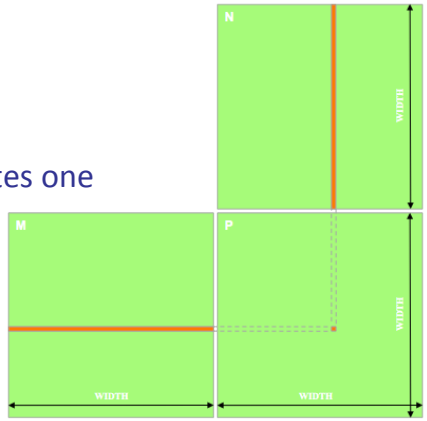
Matrix Multiplication

A more complex example

Squared Matrix Multiplication

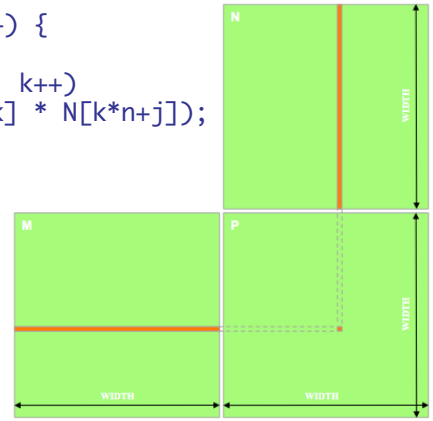
- P = M . N
  - Size (NxN)
- First approach
  - Single threads computes one element of matrix C

M and N are loaded N
Times from global memory



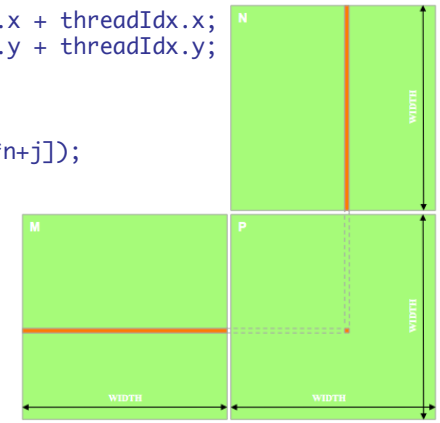
CPU Code

```
for(int i = 0; i < n; i++)
    for(int j=0; j < n; j++) {
        float dot = 0.f;
        for(int k=0; k < n; k++)
            dot += (M[i*n+k] * N[k*n+j]);
        P[i*n+j] = dot;
    }
```



CUDA Code

```
int j = blockIdx.x * blockDim.x + threadIdx.x;
int i = blockIdx.y * blockDim.y + threadIdx.y;

float dot = 0.f;
for(int k=0; k < n; k++)
      dot += (M[i*n+k] * N[k*n+j]);

P[i*n+j] = dot;
```

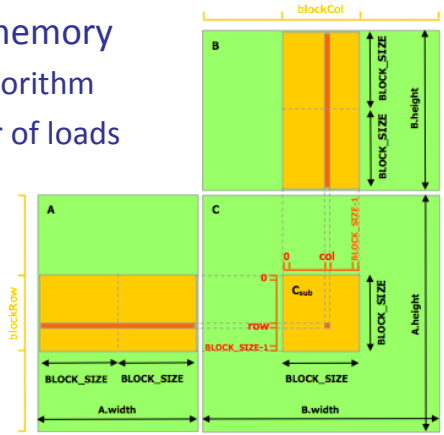


How does it perform?

How many loads per term of dot product?	2 (a & b) = 8 Bytes
How many floating point operations?	2 (multiply & addition)
Global memory access to flop ratio (GMAC)	8 Bytes / 2 ops = 4 B/op
What is the peak fp performance of GeForce GTX 580?	1.581 TFLOPS
Lower bound on bandwidth required to reach peak fp performance	GMAC * Peak FLOPS = 6.33 TB/s
What is the actual memory bandwidth of GeForce GTX 580?	194.2 GB/s
Then what is an upper bound on performance of our implementation?	Actual BW / GMAC = 194.2/ 4 = 48 GFLOPS

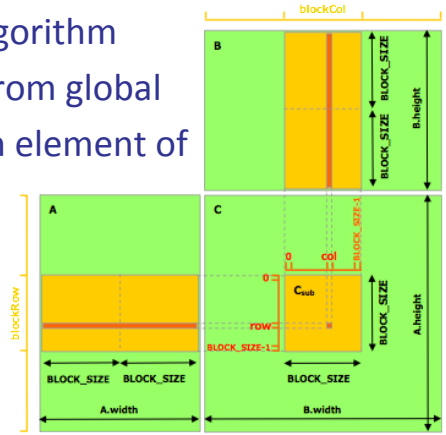
Can we do better ?

- Can we use shared memory
  - Based on CPU tile algorithm
  - Goal: reduce number of loads from global memory



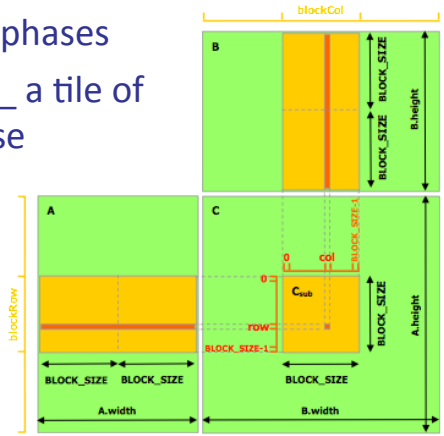
Use Shared Memory

- Based on CPU tile algorithm
- Goal: reduce loads from global
- Each thread loads an element of the tile to memory



Use Shared Memory

- Partition kernel into phases
- Load into __shared__ a tile of each matrix per phase



CUDA Code (Shared memory)

```
// shorthand
int tx = threadIdx.x, ty = threadIdx.y;
int bx = blockIdx.x, by = blockIdx.y;

// allocate tiles in shared memory
__shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
__shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

// calculate the row & col index
int row = by*blockDim.y + ty;
int col = bx*blockDim.x + tx;
float result = 0;
```

CUDA Code (Shared memory)

```
// shorthand
int tx = threadIdx.x, ty = threadIdx.y;
int bx = blockIdx.x, by = blockIdx.y;

// allocate tiles in shared memory
__shared__ float s_a[TILE_WIDTH][TIL
__shared__ float s_b[TILE_WIDTH][TIL

// calculate the row & col index
int row = by*blockDim.y + ty;
int col = bx*blockDim.x + tx;
float result = 0;
```

Identify the thread position within the block and matrix

CUDA Code (Shared memory)

```
// shorthand
int tx = threadIdx.x, ty = threadIdx.y;
int bx = blockIdx.x, by = blockIdx.y;

// allocate tiles in shared memory
__shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
__shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

// calculate the row & col index
int row = by*blockDim.y + ty;
int col = bx*blockDim.x + tx;
float result = 0;
```

Allocate space in shared memory

CUDA Code (Shared memory)

```
for(int p = 0; p < width/TILE_WIDTH; ++p) {

      s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
      s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
      __syncthreads();

      for(int k = 0; k < TILE_WIDTH; ++k)
            result += s_a[ty][k] * s_b[k][tx];

      __syncthreads();
}
ab[row*width+col] = result;
```


CUDA Code (Shared memory)

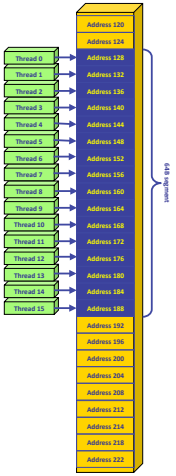
```
for(int p = 0; p < width/TILE_WIDTH; ++p) {

      s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
      s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
      __syncthreads();

      for(int k = 0; k < TILE_WI
            result += s_a[ty][k]

      __syncthreads();
}
ab[row*width+col] = result;
```

Cooperative matrix tile coalesced loading

CUDA Memory Coalescing

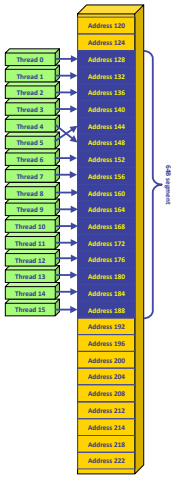
- Organized by half-warp (pre-Fermi) or warp (GF100)
  - Half-warp = 16 int/float = 64 bytes
  - Full-warp = 32 int/float = 128 bytes
- GPU global memory is accessed in 32, 64 or 128 byte blocks
- Each SM detects the memory requests across the active warp and coalesces them into the *fewest* and the *smallest* requests

CUDA Memory Coalescing



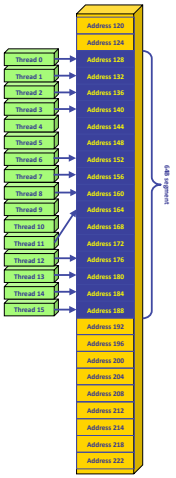
- Simple Examples
  - All threads access a sequential memory address

CUDA Memory Coalescing



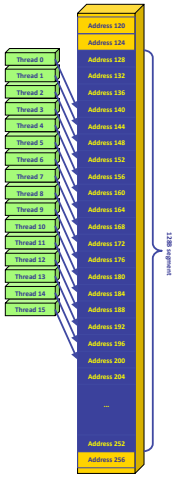
- Simple Examples
  - All threads access a sequential memory address
  - Even if threads access it out of order

CUDA Memory Coalescing



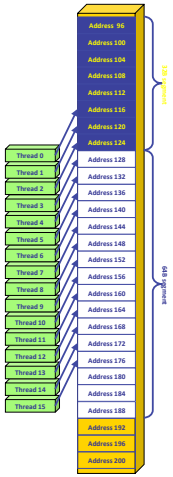
- Simple Examples
  - All threads access a sequential memory address
  - Even if threads access it out of order
  - Or some don't have memory accesses at all

CUDA Memory Coalescing



- Simple Examples
  - All threads access a sequential memory address
  - Even if threads access it out of order
  - Or some don't have memory accesses at all
  - Has long as the HW can coalesced into a single memory access

CUDA Memory Coalescing



- Simple Examples
  - All threads access a sequential memory address
  - Even if threads access it out of order
  - Or some don't have memory accesses at all
  - Has long as the HW can coalesced into a single memory access
- Otherwise memory will be split into several loads
  - Also due to the memory alignment

CUDA Memory Coalescing

- When you have 2D (Dx, Dy) and 3D (Dx, Dy, Dz) blocks, count on this indexing scheme of your threads when considering memory coalescence:
  - 2D: thread ID in the block for thread of index (x,y) is *x + Dx*y*
  - 3D: thread ID in the block for thread of index (x,y,z) is *x + Dx*(y* + Dy*z)*
  - To conclude, the x thread id runs the fastest, followed by the y, and then by the z.

CUDA Memory Coalescing

- Array of Structures vs Structure of Arrays
- Memory allocated by CUDA routines are aligned to at least 256 byte boundaries
- When allocating memory outside of CUDA, user must guarantee alignment for performance

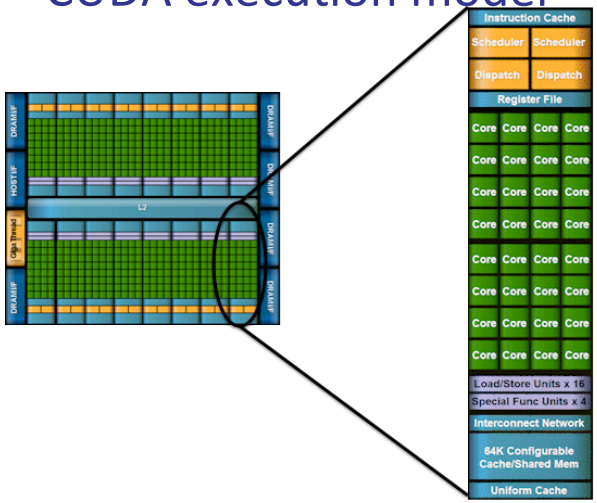
CUDA Code (Shared memory)

```
for(int p = 0; p < width/TILE_WIDTH; ++p) {

        s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
        s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
        __syncthreads();

        for(int k = 0; k < TILE_WIDTH; ++k)
              result += s_a[ty][k] * s_b[k][tx];

        __syncthreads();
}
ab[row*width+col] = result;
```


Block Synchronization barrier

- __syncthreads()
  - All threads wait until all threads reach the barrier
  - Why do we need it?
  - Aren't all threads synchronized with the block?
  - NO

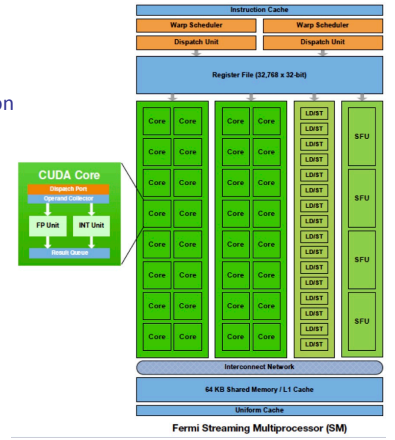
CUDA execution model



From NVidia

CUDA execution model

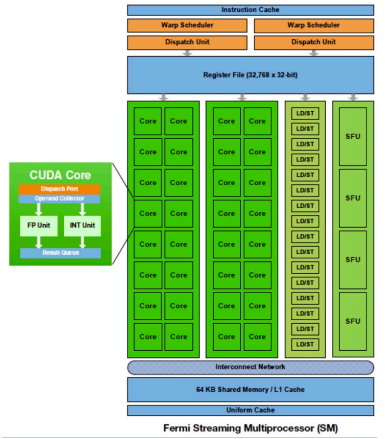
- SIMT (Single Instruction Multiple Thread)
  - Threads run in groups of 32 threads called "warps"
  - Threads in a warp share the same instruction unit
  - Divergence handle automatically by the hardware
- Hardware Multithreading
  - Resource Allocation & thread scheduling
  - Requires a high number of threads to hide latency
- Threads have all resources needed to run
  - Any warp with resource and dependencies meet can run
  - Context switching is almost "free"



From NVidia

CUDA execution model

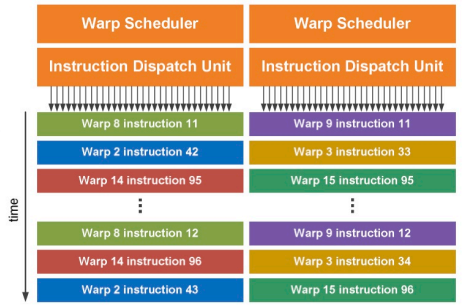
- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks per SM
  - SM in Fermi can take up to 1024 threads
    - Could be 256 (threads/block) * 4 blocks
    - Or 128 (threads/block) * 8 blocks
- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution



Fermi Streaming Multiprocessor (SM)

CUDA execution model

- Thread Blocks divided in 32-thread Warps
- Warps are scheduling units in SM
- Example
  - Block size: 256 Threads
  - 256 / 32 = 8 Warps
  - If SM is execution 3 Blocks
  - 24 Warps in execution



Block Synchronization barrier

- **void __syncthreads();**
- Synchronizes all threads in a block
- 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

CUDA Code (Shared memory)

```
for(int p = 0; p < width/TILE_WIDTH; ++p) {

        s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
        s_b[ty][tx] = b[(m*TILE_WIDTH + ty)*width + col];
        __syncthreads();

        for(int k = 0; k < TILE_WIDTH
                result += s_a[ty][k] *

        __syncthreads();
}
ab[row*width+col] = result;
```

Ensure that all warps reach the barrier
Ensure shared memory tiles are not overwritten.

Use Shared Memory

Implementation	Original	Improved
Global Loads	$2N^3$	$2N^2$ *(N/TILE_WIDTH)
Throughput	10.7 GFLOPS	183.9 GFLOPS
SLOCs	20	44
Relative Improvement	1x	17.2x
Improvement/SLOC	1x	7.8x

Values from a GTX 260

LAB

Implement Matrix Multiplication

LAB – Matrix Multiplication

- Inside the tranning-1 folder
  - GEMM : Implementation of the NAÏVE approach
  - GEMM2 : Project to implement shared memory approach
- Compile & Execution from LAB - 1

Final Notes

The "new" Moore's Law

- Computers no longer get faster, just wider
- We must re-think our algorithms to be parallel
- Data-parallel computing is the most scalable solution

Misconceptions

- ~~CUDA layers normal programs on top of graphics~~
  Compiles and executes directly to hardware
- GPU architectures are:
  - Very ~~wide (1000s)~~ SIMD machines 32 - Wide
  - on which divergence in a warp may be prohibitive...
  - with ~~4-wide vector~~ registers Scalar registers
- GPU's don't do real floating point
  IEEE-745 Compliant (with minor exceptions)