CUDA: Massive Parallelism

- GPU is a massively parallel processor
  - NVIDIA G80: 128 processors
  - Support thousands of active threads (12,288 on G80)

- CUDA provides a programming model that efficiently exposes this massive parallelism

- Simple syntax: minimal extensions to C/C++

- Transparent scalability across varying hardware
C-Code Example to Add 2 Arrays

CPU C program

```c
void addMatrixC(float *a, float *b, float *c, int N) {
    int i, j, index;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            index = i + j * N;
            c[index] = a[index] + b[index];
        }
    }
}

void main() {
    ...
    addMatrixC(a, b, c, N);
}
```

CUDA C program

```cuda
__global__
void addMatrixG(float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j * N;
    if (i < N && j < N)
        c[index] = a[index] + b[index];
}

void main() {
    ...
    dim3 dimBlk (16,16);
    dim3 dimGrd (N/dimBlk.x, N/dimBlk.y);
    addMatrixG<<<dimGrd, dimBlk>>>(a, b, c, N);
}
```

```
gcc addmatrix.c -o addmatrix
nvcc addmatrix.cu -o addmatrix
```
CUDA Kernels and Threads

Parallel portions of an application are executed on the device as **kernels**
- One kernel is executed at a time
- Many threads execute each kernel

Differences between CUDA and CPU threads
- CUDA threads are extremely lightweight
  - Very little creation overhead
  - Instant switching
- CUDA uses 1000s of threads to achieve efficiency
  - Multi-core CPUs can use only a few

Definitions:
Device = GPU; Host = CPU
Kernel = function that runs on the device
Arrays of Parallel Threads

A CUDA kernel is executed by an array of threads
- All threads run the same code
- Each thread has an ID that it uses to compute memory addresses and make control decisions

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```
Thread Cooperation

- Threads in the array need not be completely independent

- Thread cooperation is valuable
  - Share results to save computation
  - Share memory accesses
    - Drastic bandwidth reduction

- Thread cooperation is a powerful feature of CUDA
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via *shared memory*
  - Threads in different blocks cannot cooperate

Enables programs to **transparently scale** to any number of processors!

---

Thread Blocks:

- **Thread Block 0**
  ```
  ... 
  float x = input[threadID];
  float y = func(x);
  output[threadID] = y;
  ... 
  ```

- **Thread Block 0**
  ```
  ... 
  float x = input[threadID];
  float y = func(x);
  output[threadID] = y;
  ... 
  ```

- **Thread Block N - 1**
  ```
  ... 
  float x = input[threadID];
  float y = func(x);
  output[threadID] = y;
  ... 
  ```

---

Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via *shared memory*
  - Threads in different blocks cannot cooperate

Enables programs to **transparently scale** to any number of processors!
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor at any time
- A kernel scales across any number of parallel multiprocessors
CUDA Programming Model

A kernel is executed by a grid of thread blocks

A thread block is a batch of threads that can cooperate with each other by:

- Sharing data through shared memory
- Synchronizing their execution

Threads from different blocks cannot cooperate
Processors execute computing threads
Thread Execution Manager issues threads
128 Thread Processors grouped into 16 multiprocessors (SMs)
Parallel Data Cache enables thread cooperation
Thread and Block 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
Kernel Memory Access

- **Registers**
- **Global Memory**
  - Kernel input and output data reside here
  - Off-chip, large
  - Uncached
- **Shared Memory**
  - Shared among threads in a single block
  - On-chip, small
  - As fast as registers

- The host can read & write global memory but not shared memory
Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate
- Several blocks can reside concurrently on one multiprocessor
  - Control limitations (of G8X/G9X GPUs):
    - At most 8 concurrent blocks per SM
    - At most 768 concurrent threads per SM
  - Number is further limited by SM resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks
CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
  - Thread can access any memory location

- Unlimited access to memory
  - Thread can read/write as many locations as needed

- Shared memory (per block) and thread synchronization
  - Threads can cooperatively load data into shared memory
  - Any thread can then access any shared memory location

- Low learning curve
  - Just a few extensions to C
  - No knowledge of graphics is required

- No graphics API overhead
GPU Memory Allocation / Release

- `cudaMalloc(void ** pointer, size_t nbytes)`
- `cudaMemset(void * pointer, int value, size_t count)`
- `cudaFree(void* pointer)`

```c
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```
Data Copies

`cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
- `direction` specifies locations (host or device) of `src` and `dst`
- Blocks CPU thread: returns after the copy is complete
- Doesn’t start copying until previous CUDA calls complete

`cudaMemcpyAsync(..., cudaStream_t streamId)`
- Host memory must be pinned (allocate with `cudaMallocHost`)
- Returns immediately
- doesn’t start copying until previous CUDA calls in stream `streamId` or 0 complete

`enum cudaMemcpyKind`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`
Executing Code on the GPU

C function with some restrictions
- Can only access GPU memory
- No variable number of arguments ("varargs")
- No static variables

Must be declared with a qualifier
- __global__: invoked from within host (CPU) code, cannot be called from device (GPU) code, must return void
- __device__: called from other GPU functions, cannot be called from host (CPU) code
- __host__: can only be executed by CPU, called from host

__host__ and __device__ qualifiers can be combined
- Sample use: overloading operators
- Compiler will generate both CPU and GPU code
Launching kernels on GPU

**Modified C function call syntax:**

```
kernell<<<dim3 grid, dim3 block, int smem, int stream>>>(...)```

**Execution Configuration ("<<< >>>"):**

- **grid dimensions:** \( x \) and \( y \)
- **thread-block dimensions:** \( x, y, \) and \( z \)
- **shared memory:** number of bytes per block for extern smem variables declared without size
  - optional, 0 by default
- **stream ID**
  - optional, 0 by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernell<<<grid, block, 0, 0>>>(...);
kernell<<<32, 512>>>(...);
```
CUDA Built-in Device Variables

All `__global__` and `__device__` functions have access to these automatically defined variables:

- `dim3 gridDim;`
  - Dimensions of the grid in blocks (`gridDim.z` unused)
- `dim3 blockDim;`
  - Dimensions of the block in threads
- `dim3 blockIdx;`
  - Block index within the grid
- `dim3 threadIdx;`
  - Thread index within the block
Minimal Kernels

```c
__global__ void minimal( int* d_a) {
    *d_a = 13;
}

__global__ void assign( int* d_a, int value) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```
Minimal Kernel for 2D data

```c
__global__ void assign2D(int* d_a, int w, int h, int value) {
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;
    d_a[idx] = value;
}

assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```
Example: Increment Array Elements

CPU program

```c
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}
```

```c
void main()
{
    ..... 
    increment_cpu(a, b, N);
}
```

CUDA program

```c
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}
```

```c
void main()
{
    ..... 
    dim3 dimBlock (blocksize);
    dim3 dimGrid(ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```
Example: Increment Array Elements

Increment N-element vector $a$ by scalar $b$

Let’s assume $N=16$, $\text{blockDim}=4$  -> 4 blocks

\[
\begin{align*}
\text{blockIdx.x} &= 0 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 0,1,2,3
\end{align*}
\]

\[
\begin{align*}
\text{blockIdx.x} &= 1 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 4,5,6,7
\end{align*}
\]

\[
\begin{align*}
\text{blockIdx.x} &= 2 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 8,9,10,11
\end{align*}
\]

\[
\begin{align*}
\text{blockIdx.x} &= 3 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 12,13,14,15
\end{align*}
\]

\[
\text{int idx} = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x};
\]

will map from local index $\text{threadIdx}$ to global index

NB: $\text{blockDim}$ should be $\geq 32$ in real code, this is just an example
Example: Host Code

// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
CUDA Memory Spaces

Each thread can:
- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read only per-grid constant memory
- Read only per-grid texture memory

The host can read/write global, constant, and texture memory (stored in DRAM)
CUDA Memory Spaces

- Global and Shared Memory introduced before
  - Most important, commonly used
- Local, Constant, and Texture for convenience/performance
  - Local: automatic array variables allocated there by compiler
  - Constant: useful for uniformly-accessed read-only data
    - Cached (see programming guide)
  - Texture: useful for spatially coherent random-access read-only data
    - Cached (see programming guide)
    - Provides filtering, address clamping and wrapping

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope (“Who?”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>
Variable Qualifiers (GPU code)

__device__
- stored in device memory (large, high latency, no cache)
- Allocated with `cudaMalloc` (__device__ qualifier implied)
- accessible by all threads
- lifetime: application

__constant__
- same as __device__, but cached and read-only by GPU
- written by CPU via `cudaMemcpyToSymbol(...)` call
- lifetime: application

__shared__
- stored in on-chip shared memory (very low latency)
- accessible by all threads in the same thread block
- lifetime: kernel launch

Unqualified variables:
- scalars and built-in vector types are stored in registers
- arrays of more than 4 elements or run-time indices stored in device memory
Thread Synchronization Function

```c
void __syncthreads();
```

Synchronizes all threads in a block
- Generates barrier synchronization instruction
- No thread can pass this barrier until all threads in the block reach it
- Used to avoid RAW / WAR / WAW hazards when accessing shared memory

**Allowed in conditional code only if the conditional is uniform across the entire thread block**
Atomic operations on integers in global memory
- Resolve simultaneous operations on a single address by multiple threads

```cpp
atomicAdd(d_a, myVal); // all active threads add to d_a
```

Associative operations on signed/unsigned ints
- add, sub, min, max, ...
- and, or, xor
- Increment, decrement
- Exchange, compare and swap

Requires hardware with compute capability 1.1
- Compute capability 1.2 adds shared mem atomics
Device Management

- **CPU can query and select GPU devices**
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *current_device)`
  - `cudaGetDeviceProperties(cudaDeviceProp* prop, int device)`
  - `cudaChooseDevice(int *device, cudaDeviceProp* prop)`

- **Multi-GPU setup:**
  - device 0 is used by default
  - one CPU thread can control only one GPU
    - multiple CPU threads can control the same GPU
      - calls are serialized by the driver
CUDA / Graphics Interoperability

CUDA enables buffers from graphics APIs to be mapped to device pointers for kernel access.

CUDA 1.1 has basic interoperability
- OpenGL: Buffer Objects (PBOs and VBOs)
- DirectX 9: Vertex Buffers (VBs)

CUDA 2.0 will improve DX9 interop
- Index Buffers (IBs) and Textures/Surfaces

CUDA 2.0 will add Vista and DX10 support
Graphics Interop: OpenGL

- Register buffer object (once)
  
  ```c
  GLuint bufferObj;
  cudaGLRegisterBufferObject(bufferObj);
  ```

- Map bufferObj to device pointer:
  
  ```c
  float* devPtr;
  cudaGLMapBufferObject((void**)&devPtr, bufferObj);
  ```

- Unmap: `cudaGLUnmapBufferObject()`
- Unregister: `cudaGLUnregisterBufferObject()`
Graphics Interop: DX9

- Initialize/terminate with `cudaD3D9Begin() /End()`
  - Below must fall between begin/end pair

- Register VB:
  
  ```c
  LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
  cudaD3D9RegisterVertexBuffer(vertexBuffer);
  ```

- Map VB to `__device__` pointer:
  
  ```c
  float* devPtr;
  cudaD3D9MapVertexBuffer((void**) &devPtr, vertexBuffer);
  ```

- Unmap: `cudaD3D9UnmapVertexBuffer()`
- Unregister: `cudaD3D9UnregisterVertexBuffer()`
Compiling CUDA

C/C++ CUDA Application

NVCC

PTX Code

CPU Code

Virtual

PTX to Target Compiler

PTX Code

Physical

G80

... GPU

Target code
NVCC & PTX Virtual Machine

- **C/C++ CUDA Application**
  - **EDG**
    - Separate GPU vs. CPU code
  - **Open64**
    - Generates GPU PTX assembly
  - **Parallel Thread eXecution (PTX)**
    - Virtual Machine and ISA
    - Programming model
    - Execution resources and state

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

```
ld.global.v4.f32 {f1, f3, f5, f7}, [r9+0];
mad.f32 f1, f5, f3, f1;
```
Warps

- Instructions are executed one SIMT warp at a time
  - Warp = 32 threads on current CUDA-capable GPUs
  - Launching thread blocks whose size is not a multiple of warp size results in inefficient processor utilization
  - SIMT = single instruction multiple thread

- Divergent branches within a warp cause serialization
  - If all threads in a warp take the same branch, no extra cost
  - If threads each take one of two different branches, entire warp pays cost of both branches of code
  - If threads take $n$ different branches, entire warp pays cost of $n$ branches of code
CUDA Performance Strategies
Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host
Optimize Memory Coherence

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory

- Optimize for spatial locality in cached texture memory

- In shared memory, avoid high-degree bank conflicts
Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory

- Use one / a few threads to load / compute data shared by all threads

- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later
Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks

- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory
Memory optimizations

- Optimizing memory transfers
- Coalescing global memory accesses
- Using shared memory effectively
Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI-e x16) vs. 80 GB/s peak (Quadro FX 5600)
  - 8GB/s for PCI-e 2.0

- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory

- Group transfers
  - One large transfer much better than many small ones
Page-Locked Memory Transfers

- cudaMallocHost() allows allocation of page-locked host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s+ common on PCI-express (x16)
  - ~4 GB/s measured on nForce 680i motherboards (overclocked PCI-e)

- See the “bandwidthTest” CUDA SDK sample

- Use with caution
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits
Global Memory Reads/Writes

- Highest latency instructions: 400-600 clock cycles
- Likely to be performance bottleneck
- Optimizations can greatly increase performance
  - Coalescing: up to 10x speedup
  - Latency hiding: up to 2.5x speedup
Coalescing

A coordinated read by a half-warp (16 threads)
A contiguous region of global memory:
- 64 bytes - each thread reads a word: int, float, ...
- 128 bytes - each thread reads a double-word: int2, float2, ...
- 256 bytes – each thread reads a quad-word: int4, float4, ...

Additional restrictions on G8X/G9X architecture:
- Starting address for a region must be a multiple of region size
- The $k^{th}$ thread in a half-warp must access the $k^{th}$ element in a block being read

Exception: not all threads must be participating
- Predicated access, divergence within a halfwarp
Coalesced Access: Reading floats

All threads participate

Some Threads Do Not Participate
Uncoalesced Access: Reading floats

Permuted Access by Threads

Misaligned Starting Address (not a multiple of 64)
Coalescing: Timing Results

- **Experiment on G80:**
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs

- **12K blocks x 256 threads:**
  - 356µs – coalesced
  - 357µs – coalesced, some threads don’t participate
  - 3,494µs – permuted/misaligned thread access
Uncoalesced float3 Code

```c
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float3 a = d_in[index];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    d_out[index] = a;
}
```
Uncoalesced Access: float3 Case

- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) ≠ 4, 8, or 12
  - Half-warp reads three 64B non-contiguous regions

First read
Coalescing float3 Access

Similarly, Step3 starting at offset 512
Coalesced Access: float3 Case

- Use shared memory to allow coalescing
  - Need `sizeof(float3)*(threads/block)` bytes of SMEM
  - Each thread reads 3 scalar floats:
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync

Processing

- Each thread retrieves its float3 from SMEM array
  - Cast the SMEM pointer to (float3*)
  - Use thread ID as index

- Rest of the compute code does not change!
Coalesced float3 Code

```c
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x]         = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];
    a.x += 2;
    a.y += 2;
    a.z += 2;
    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index]         = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

Compute code is not changed

Write the result through SMEM
Coalescing: Timing Results

Experiment:
- Kernel: read a float, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

12K blocks x 256 threads:
- 356µs – coalesced
- 357µs – coalesced, some threads don’t participate
- 3,494µs – permuted/misaligned thread access

4K blocks x 256 threads:
- 3,302µs – float3 uncoalesced
- 359µs – float3 coalesced through shared memory
Coalescing:
Structures of size $\neq 4, 8, 16$ Bytes

- Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)

- If SoA is not viable:
  - Force structure alignment: `__align(X)`, where $X = 4, 8, \text{ or } 16$
  - Use SMEM to achieve coalescing

---

Point structure

```
x y z
```

AoS

```
x y z  x y z  x y z
```

SoA

```
x x x   y y y   z z z
```
Coalescing: Summary

- Coalescing greatly improves throughput
- Critical to memory-bound kernels
- Reading structures of size other than 4, 8, or 16 bytes will break coalescing:
  - Prefer Structures of Arrays over AoS
  - If SoA is not viable, read/write through SMEM

- Additional resources:
  - Aligned Types SDK Sample
The CUDA Visual Profiler

- Helps measure and find potential performance problem
  - GPU and CPU timing for all kernel invocations and memcpy
  - Time stamps

- Access to hardware performance counters
Signals

Events are tracked with hardware counters on signals in the chip:

- timestamp
- gld_incoherent
- gld_coherent
- gst_incoherent
- gst_coherent
- local_load
- local_store
- branch
- divergent_branch
- instructions – instruction count
- warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
- cta_launched – executed thread blocks

Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent)

Local loads/stores

Total branches and divergent branches taken by threads
Interpreting profiler counters

- Values represent events within a thread warp

- Only targets one multiprocessor
  - Values will not correspond to the total number of warps launched for a particular kernel.
  - Launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work.

- Values are best used to identify relative performance differences between unoptimized and optimized code
  - In other words, try to reduce the magnitudes of gld/gst_incoherent, divergent_branch, and warp_serialize
In a parallel machine, many threads access memory. Therefore, memory is divided into banks. Essential to achieve high bandwidth.

Each bank can service one address per cycle. A memory can service as many simultaneous accesses as it has banks.

Multiple simultaneous accesses to a bank result in a bank conflict. Conflicting accesses are serialized.
Bank Addressing Examples

- **No Bank Conflicts**
  - Linear addressing
  - stride == 1

- **No Bank Conflicts**
  - Random 1:1 Permutation
Bank Addressing Examples

2-way Bank Conflicts
- Linear addressing
- stride == 2

8-way Bank Conflicts
- Linear addressing
- stride == 8
How addresses map to banks on G80

- Bandwidth of each bank is 32 bits per 2 clock cycles
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So \( \text{bank} = \text{address} \mod 16 \)
  - Same as the size of a half-warp
  - No bank conflicts between different half-warps, only within a single half-warp
Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts

The fast case:
- If all threads of a half-warp access different banks, there is no bank conflict
- If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)

The slow case:
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank
Optimization Example: Matrix Transpose
Matrix Transpose

SDK Sample (“transpose”)
Illustrates:
- Coalescing
- Avoiding SMEM bank conflicts
- Speedups for even small matrices
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in   = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
Uncoalesced Transpose

Reads input from GMEM

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,15</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,15</td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
<td>15,15</td>
</tr>
</tbody>
</table>

Write output to GMEM

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td>15,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td>15,1</td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
<td>15,15</td>
</tr>
</tbody>
</table>

Stride = 16, uncoalesced

Stride = 1, coalesced
Coalesced Transpose

- **Assumption:** matrix is partitioned into square tiles

- **Threadblock** \((bx, by)\):
  - Read the \((bx,by)\) input tile, store into SMEM
  - Write the SMEM data to \((by,bx)\) output tile
  - Transpose the indexing into SMEM

- **Thread** \((tx,ty)\):
  - Reads element \((tx,ty)\) from input tile
  - Writes element \((tx,ty)\) into output tile

- **Coalescing is achieved if:**
  - Block/tile dimensions are multiples of 16
# Coalesced Transpose

<table>
<thead>
<tr>
<th>Reads from GMEM</th>
<th>Writes to SMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0 0,1 0,2 0,15</td>
<td>0,0 0,1 0,2 0,15</td>
</tr>
<tr>
<td>1,0 1,1 1,2 1,15</td>
<td>1,0 1,1 1,2 1,15</td>
</tr>
<tr>
<td>15,0 15,1 15,2 15,15</td>
<td>15,0 15,1 15,2 15,15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reads from SMEM</th>
<th>Writes to GMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0 1,0 2,0 15,0</td>
<td>0,0 0,1 0,2 0,15</td>
</tr>
<tr>
<td>0,1 1,1 2,1 15,1</td>
<td>1,0 1,1 1,2 1,15</td>
</tr>
<tr>
<td>0,15 1,15 2,15 15,15</td>
<td>15,0 15,1 15,2 15,15</td>
</tr>
</tbody>
</table>
SMEM Optimization

Reads from SMEM

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>1,0</th>
<th>2,0</th>
<th>15,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td>15,1</td>
<td></td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
<td>15,15</td>
<td></td>
</tr>
</tbody>
</table>
__global__ void transpose(float *odata, float *idata, int width, int height) {
    __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;
    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
    unsigned int index_out, index_transpose;

    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * (BLOCK_DIM+1) + threadIdx.x;
        block[index_block] = idata[index_in];
        index_transpose = threadIdx.x * (BLOCK_DIM+1) + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if (xIndex < width && yIndex < height)
    {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * (BLOCK_DIM+1) + threadIdx.x;
        block[index_block] = idata[index_in];
        index_transpose = threadIdx.x * (BLOCK_DIM+1) + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;

        odata[index_out] = block[index_transpose];
    }
}
Transpose Timings

Speedups with coalescing and SMEM optimization:
- 128x128: 0.011ms vs. 0.022ms (2.0X speedup)
- 512x512: 0.07ms vs. 0.33ms (4.5X speedup)
- 1024x1024: 0.30ms vs. 1.92ms (6.4X speedup)
- 1024x2048: 0.79ms vs. 6.6ms (8.4X speedup)

Coalescing without SMEM optimization:
- 128x128: 0.014ms
- 512x512: 0.101ms
- 1024x1024: 0.412ms
- 1024x2048: 0.869ms
Execution Configuration Optimizations
Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy.

**Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently.

Limited by resource usage:
- Registers
- Shared memory
Grid/Block Size Heuristics

- **# of blocks > # of multiprocessors**
  - So all multiprocessors have at least one block to execute

- **# of blocks / # of multiprocessors > 2**
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren’t waiting at a `__syncthreads()` keep the hardware busy
  - Subject to resource availability – registers, shared memory

- **# of blocks > 100 to scale to future devices**
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
Register Dependency

Read-after-write register dependency
- Instruction’s result can be read ~22 cycles later
- Scenarios: CUDA: PTX:

To completely hide the latency:
- Run at least 192 threads (6 warps) per multiprocessor
- At least 25% occupancy
- Threads do not have to belong to the same thread block
Register Pressure

- Hide latency by using more threads per SM
  - Limiting Factors:
    - Number of registers per kernel
      - 8192 per SM, partitioned among concurrent threads
    - Amount of shared memory
      - 16KB per SM, partitioned among concurrent threadblocks
  - Check .cubin file for # registers / kernel
  - Use `--maxrregcount=N` flag to NVCC
    - N = desired maximum registers / kernel
    - At some point “spilling” into LMEM may occur
      - Reduces performance – LMEM is slow
      - Check .cubin file for LMEM usage
Determining resource usage

- Use “–ptxoptions=-v” option to nvcc
- Or, compile the kernel code with the -cubin flag to determine register usage.
- Open the .cubin file with a text editor and look for the “code” section.

```
architecture {sm_10}
abiversion {0}
modname {cubin}

code {
    name = BlackScholesGPU
    lmem = 0
    smem = 68
    reg = 20
    bar = 0
    bincode {
        0xa0004205 0x04200780 0x40024c09 0x00200780
        ...
    }
}
```

- per thread local memory
- per thread block shared memory
- per thread registers
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

Your chosen resource usage is indicated by the red triangle on the graph.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

<table>
<thead>
<tr>
<th>Resource Usage</th>
<th>Varying Block Size</th>
<th>Varying Register Count</th>
<th>Varying Shared Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads Per Block</td>
<td><img src="image1.png" alt="Graph" /></td>
<td><img src="image2.png" alt="Graph" /></td>
<td><img src="image3.png" alt="Graph" /></td>
</tr>
<tr>
<td>Active Threads per Multiprocessor</td>
<td>384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Warps per Multiprocessor</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Thread Blocks per Multiprocessor</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occupancy of each Multiprocessor</td>
<td>58%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Simultaneous Blocks per GPU</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical Limits for GPU:
- Multiprocessors per GPU: 16
- Threads per Warps: 32
- Warps per Multiprocessor: 24
- Threads per Multiprocessor: 768
- Thread Blocks per Multiprocessor: 32
- Total # of 32-bit registers/Multiprocessor: 6192
- Shared Memory/ Multiprocessor (Bytes): 16384

Allocation Per Thread Block:
- Warps: 256
- Registers: 32
- Shared Memory: 64

These limits are used in computing the occupancy data in blue.

CUDA Occupancy Calculator

![Graph](image4.png)
Optimizing threads per block

Choose threads per block as a multiple of warp size
- Avoid wasting computation on under-populated warps

More threads per block == better memory latency hiding

But, more threads per block == fewer registers per thread
- Kernel invocations can fail if too many registers are used

Heuristics
- Minimum: 64 threads per block
  - Only if multiple concurrent blocks
- 192 or 256 threads a better choice
  - Usually still enough regs to compile and invoke successfully
- This all depends on your computation, so experiment!
Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

  \textit{BUT}...

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - \textit{(It all comes down to arithmetic intensity and available parallelism)}
Parameterize Your Application

- Parameterization helps adaptation to different GPUs

- GPUs vary in many ways
  - # of multiprocessors
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Threads per block

- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration
Conclusion

- Understand CUDA performance characteristics
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding

- Use peak performance metrics to guide optimization

- Understand parallel algorithm complexity theory

- Know how to identify type of bottleneck
  - e.g. memory, core computation, or instruction overhead

- Optimize your algorithm, then unroll loops

- Use template parameters to generate optimal code
Questions?

http://developer.nvidia.com/
Built-in Vector Types

Can be used in GPU and CPU code

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

- dim3
  - Based on uint3
  - Used to specify dimensions
  - Default value (1,1,1)
Multiple CPU Threads and CUDA

CUDA resources allocated by a CPU thread can be consumed only by CUDA calls from the same CPU thread

Violation Example:

- CPU thread 2 allocates GPU memory, stores address in \( p \)
- thread 3 issues a CUDA call that accesses memory via \( p \)
CUDA Error Reporting to CPU

All CUDA calls return error code:
- except for kernel launches
- cudaError_t type

cudaError_t cudaGetLastError(void)
- returns the code for the last error (no error has a code)

char* cudaGetErrorString(cudaError_t code)
- returns a null-terminated character string describing the error

printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed

- cudaMemcpy is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed

- cudaMemcpyAsync() blocks until all previous CUDA calls complete

Async API provides:
- GPU CUDA-call streams
- non-blocking cudaMemcpyAsync
CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
- asyncAPI sample in CUDA SDK

```c
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernelfile<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start); cudaEventDestroy(stop);
```
Compilation

Any source file containing CUDA language extensions must be compiled with **nvcc**

**NVCC is a compiler driver**

- Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...

**NVCC can output:**

- Either C code (CPU Code)
  - That must then be compiled with the rest of the application using another tool
- Or PTX object code directly

**An executable with CUDA code requires:**

- The CUDA core library (**cuda**)  
- The CUDA runtime library (**cudart**)  
  - (only if runtime API is used)
  - loads **cuda** library
Asynchronous memory copy

- Asynchronous host ↔ device memory copy for page-locked memory frees up CPU on all CUDA capable devices

- Overlap implemented by using a CUDA stream

- CUDA Stream = Sequence of CUDA operations that execute in order

- Stream API:
  - Each stream has an ID: 0 = default stream
  - `cudaMemcpyAsync(dst, src, size, 0);`
Overlap kernel and memory copy

- Concurrent execution of a kernel and a host ↔
  device memory copy for page-locked memory
  - Compute capability >= 1.1 (G84 and up)
  - Available as a preview feature in CUDA 1.1
  - Overlaps kernel execution in one stream with a memory copy from another stream

Stream API:

```c
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, stream1);
kernal<<<grid, block, 0, stream2>>>(...);
cudaStreamQuery(stream2);
```

overlapped
Global and Shared Memory

- Global memory not cached on G8x GPUs
  - High latency, but running many threads hides latency
  - Important to minimize accesses
  - Coalesce global memory accesses (more later)

- Shared memory is on-chip, very high bandwidth
  - Low latency
  - Like a user-managed per-multiprocessor cache
  - Try to minimize or avoid bank conflicts (more later)
Texture and Constant Memory

Texture partition is cached
- Uses the texture cache also used for graphics
- Optimized for 2D spatial locality
- Best performance when threads of a warp read locations that are close together in 2D

Constant memory is cached
- 4 cycles per address read within a single warp
  - Total cost 4 cycles if all threads in a warp read same address
  - Total cost 64 cycles if all threads read different addresses