NVIDIA®

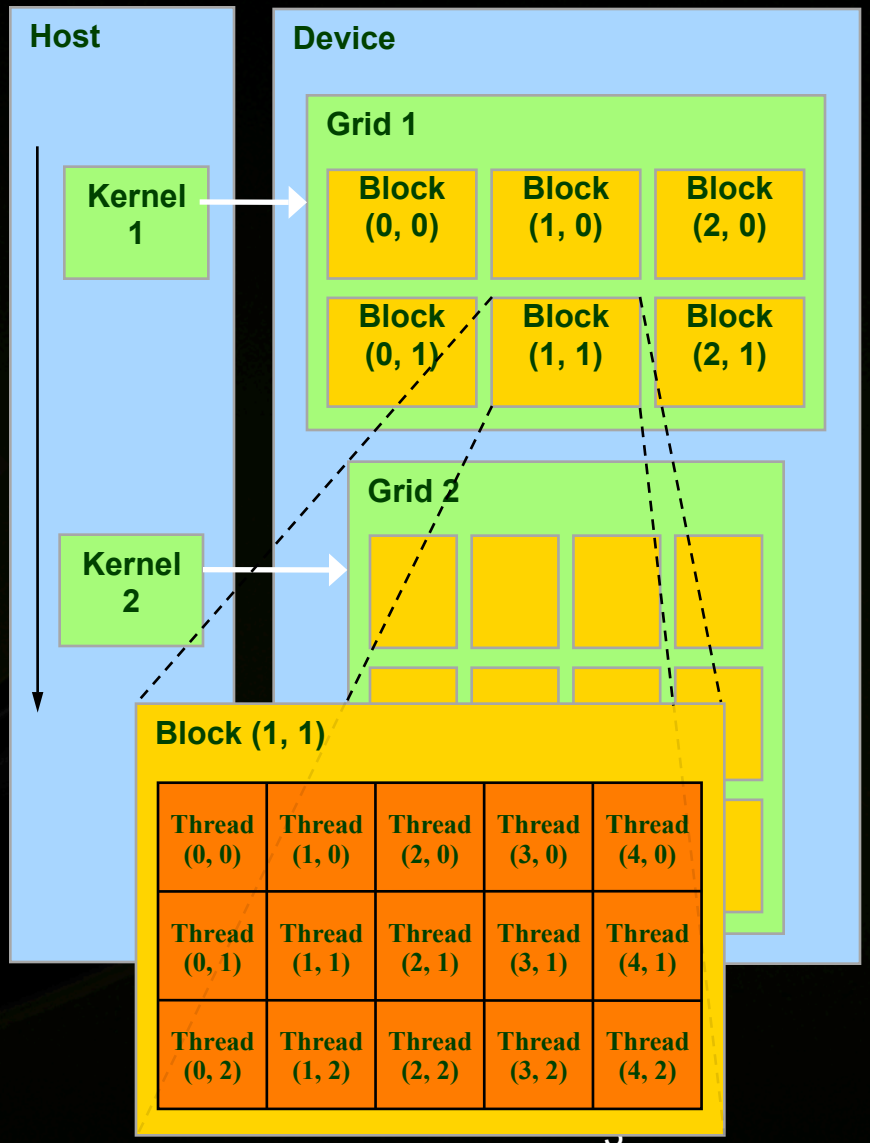
CUDA Programming Model Overview

CUDA Programming Model

- 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

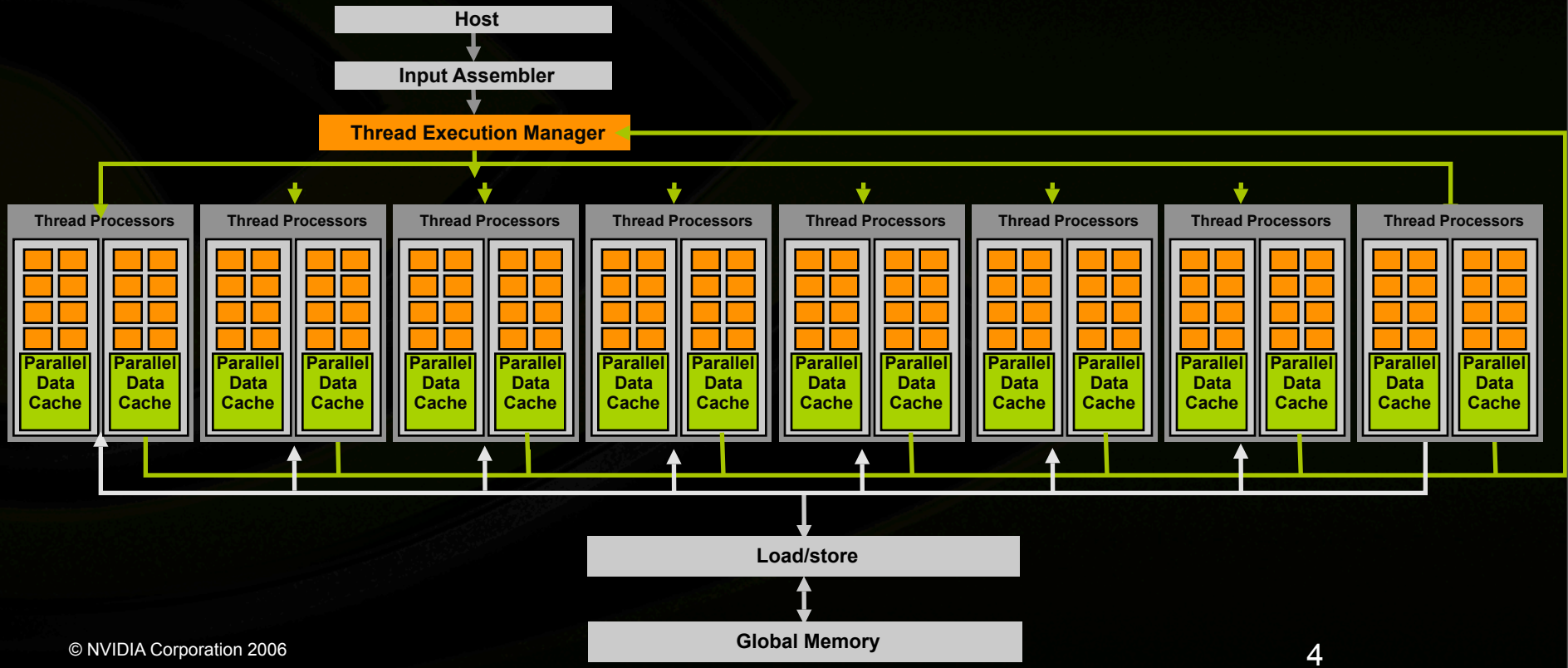
Programming Model

- A kernel is executed as 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



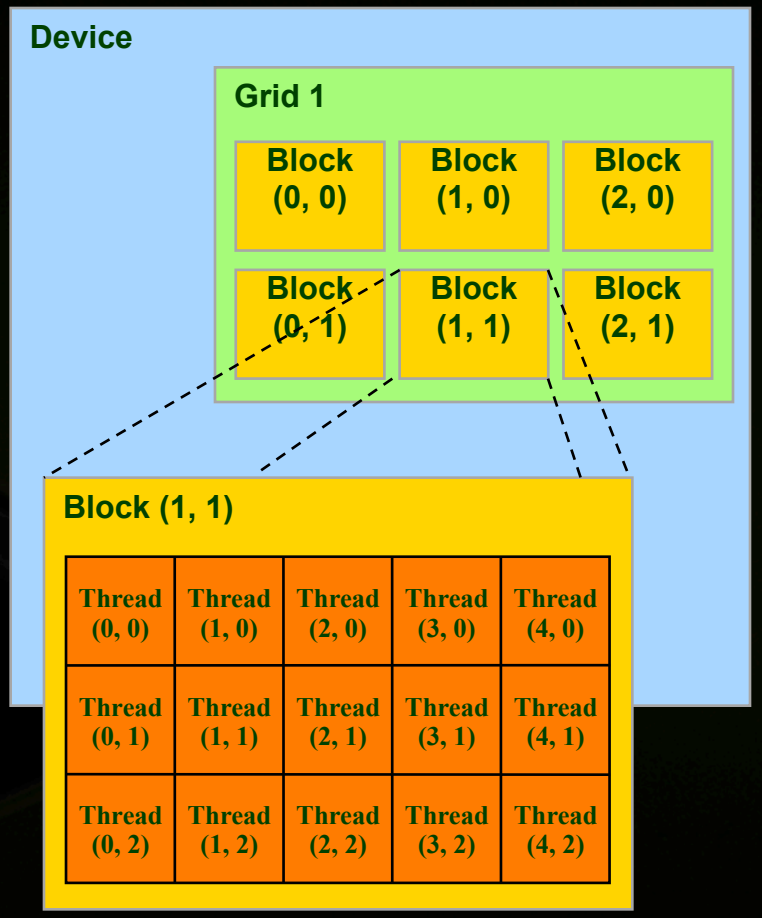
G80 Device

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors
- Parallel Data Cache accelerates processing



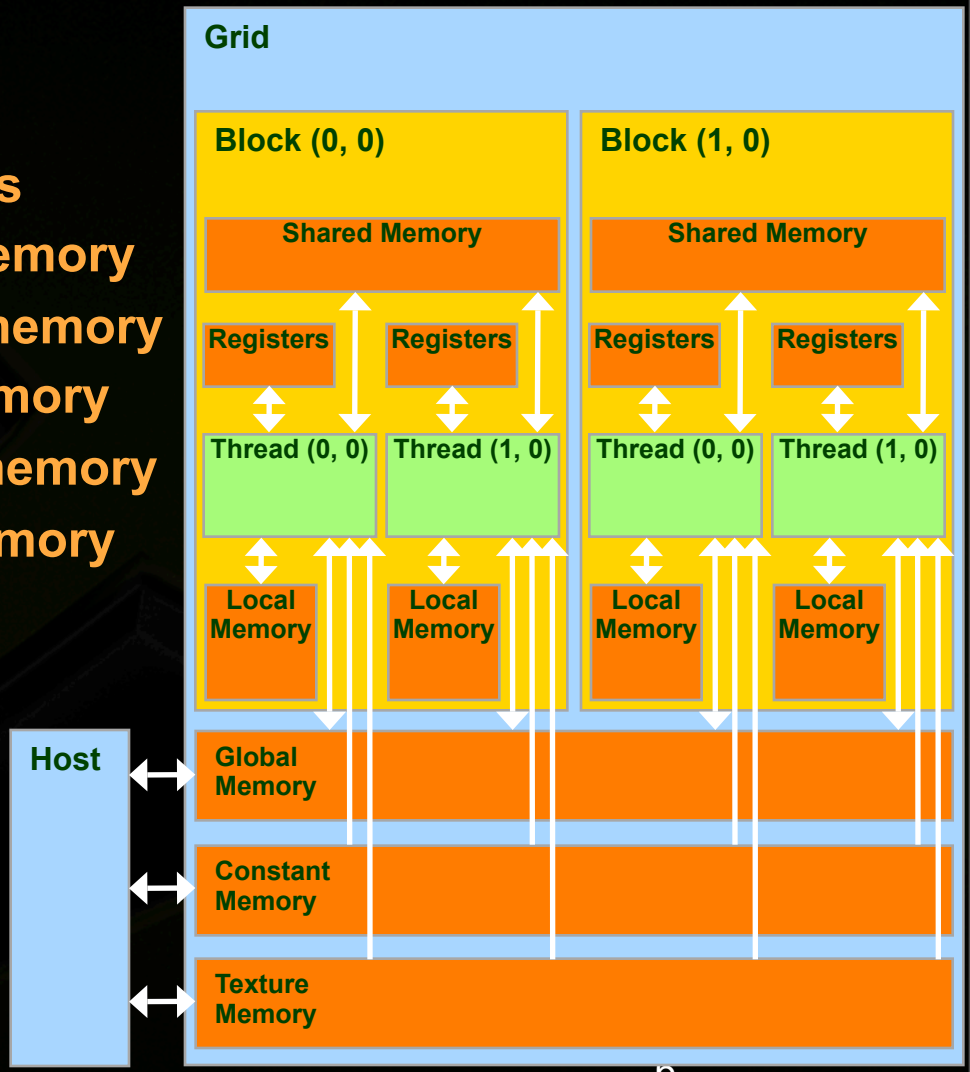
Programming Model

- 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



Programming Model: 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)



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 execute concurrently on one multiprocessor
  - Control limitations:
    - At most 8 concurrent blocks per SM
    - At most 768 concurrent threads per SM
  - Number is limited further by SM resources
    - Register file is partitioned among the threads
    - Shared memory is partitioned among the blocks

Example

- Resource requirements:
  - 5KB of SMEM per block
  - 30 registers used by the program
  - 128 threads per block
- Max concurrent blocks per SM during execution:
  - 3 due to SMEM partitioning
  - (8192/30) / 128 -> 2 due to RF partitioning
  - Therefore: 2 concurrent blocks per SM
    - 2*128 = 256 < 768
- If 512 threads per block:
  - Only 1 concurrent block per SM

CUDA Advantages over Legacy GPGPU

- Random access to memory
  - Thread can access any memory location
- Unlimited access to memory
  - Thread can read/write as many locations as needed
- User-managed cache (per block)
  - Threads can cooperatively load data into SMEM
  - Any thread can then access any SMEM location
- Low learning curve
  - Just a few extensions to C
  - No knowledge of graphics is required
- No graphics API overhead

CUDA Model Summary

- Thousands of lightweight concurrent threads
  - No switching overhead
  - Hide instruction latency
- Shared memory
  - User-managed L1 cache
  - Thread communication within blocks
- Random access to global memory
  - Any thread can read/write any location(s)
- Current generation hardware:
  - Up to 128 streaming processors

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

NVIDIA®

CUDA Programming Basics

CUDA: C on the GPU

- A simple, explicit programming language solution
- Extend only where necessary

```
__global__ void KernelFunc(...);

__shared__ int SharedVar;
```

- Kernel launch
  - `KernelFunc<<< 500, 128 >>>(...);`
- Explicit GPU memory allocation
  - `cudaMalloc(), cudaFree()`
- Memory copy from host to device, etc.
  - `cudaMemcpy(), cudaMemcpy2D(), ...`

Example: Increment Array Elements

CPU program

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


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

CUDA program

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


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (N/blocksize);
    increment_gpu<<<dimGrid, dimBlock>>>(a,b);
}
```

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

will map from local index threadIdx to global index

NB: blockDim should be bigger than 4 in real code, this is just an example

Example: Increment Array Elements

CPU program

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


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

CUDA program

```
__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;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (ceil(N / (float)blocksize));
    increment_gpu<<<dimGrid, dimBlock>>>(a,b,N);
}
```

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);
```

Application Programming Interface

- Extension to the C programming language
- CUDA API:
  - Language extensions
    - Target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library supported in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

Language Extensions: Function Type Qualifiers

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

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together
- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
`__shared__ int SharedVar;`	shared	thread block	thread block
`__device__ int GlobalVar;`	global	grid	application
`__constant__ int ConstantVar;`	constant	grid	application

- Automatic variables without any qualifier reside in registers
  - Except for large structures or arrays that reside in local memory
- Pointers can point to memory allocated or declared in either global or shared memory:
  - Global memory:
    - Memory allocated in the host and passed to the kernel:
    - Obtained as the address of a global variable
  - Shared memory: statically allocated during the call

Language Extensions: Execution Configuration

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

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

  - The optional `SharedMemBytes` bytes are:
    - Allocated in addition to the compiler allocated shared memory
    - Mapped to any variable declared as:

```
extern __shared__ float DynamicSharedMem[];
```

- A call to a kernel function is asynchronous

Language Extensions: Built-in 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

Common Runtime Component

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes

Common Runtime Component: Built-in Vector Types

- `[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)

Common Runtime Component: Mathematical Functions

- `powf, sqrtf, cbrtf, hypotf`
- `expf, exp2f, expm1f`
- `logf, log2f, log10f, log1pf`
- `sinf, cosf, tanf`
- `asinf, acosf, atanf, atan2f`
- `sinhf, coshf, tanhf`
- `asinhf, acoshf, atanhf`
- `ceil, floor, trunc, round`
- etc.

- When executed in host code, a given function uses the C runtime implementation if available
- These functions are only supported for scalar types, not vector types

Host Runtime Component

- Provides functions to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Texture management
  - Interoperability with OpenGL and Direct3D
  - Error handling
- Initializes the first time a runtime function is called
- A host thread can execute device code on only one device
  - Multiple host threads required to run on multiple devices
  - CUDA resources can be used from host thread that allocated them

Host Runtime Component: Device Management

- Device enumeration
  - `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`
- Device selection
  - `cudaChooseDevice()`, `cudaSetDevice()`

```
>  ~/NVIDIA_CUDA_SDK/bin/linux/release/deviceQuery
There is 1 device supporting CUDA

Device 0: "Quadro FX 5600"
  Major revision number:                         1
  Minor revision number:                         0
  Total amount of global memory:                 1609891840 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1350000 kilohertz
```

Host Runtime Component: Memory Management

- Two kinds of memory:
  - Linear memory: accessed through 32-bit pointers
  - CUDA arrays:
    - opaque layouts with dimensionality
    - readable only through texture objects
- Memory allocation
  - `cudaMalloc()`, `cudaFree()`, `cudaMallocPitch()`, `cudaMallocArray()`, `cudaFreeArray()`
- Memory copy
  - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToArray()`, `cudaMemcpyFromArray()`, etc. `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
- Memory addressing
  - `cudaGetSymbolAddress()`

Host Runtime Component: Interoperability with Graphics APIs

- OpenGL buffer objects and Direct3D vertex buffers can be mapped into the address space of CUDA:
  - Covered later

Device Runtime Component: Synchronization Function

- `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
- Allowed in conditional code only if the conditional is uniform across the entire thread block

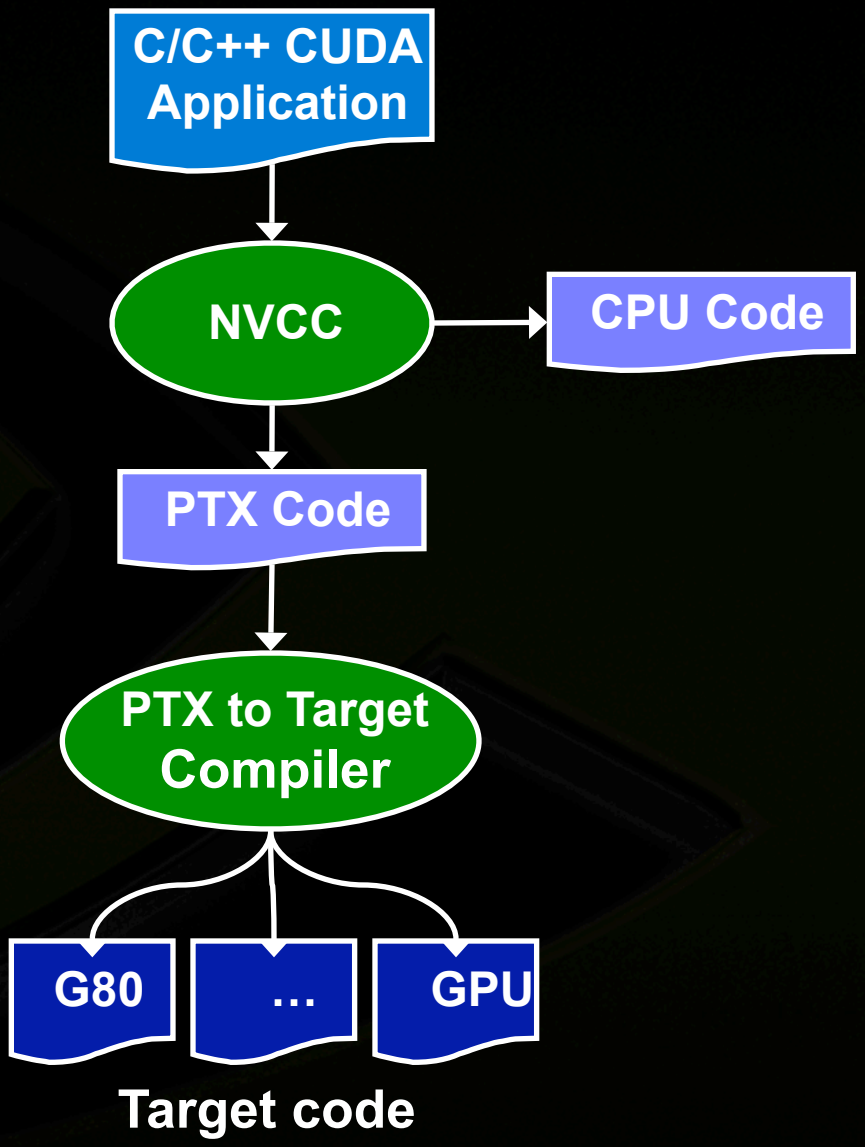
Device Runtime Component: Atomics

- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
- Require hardware with 1.1 compute capability

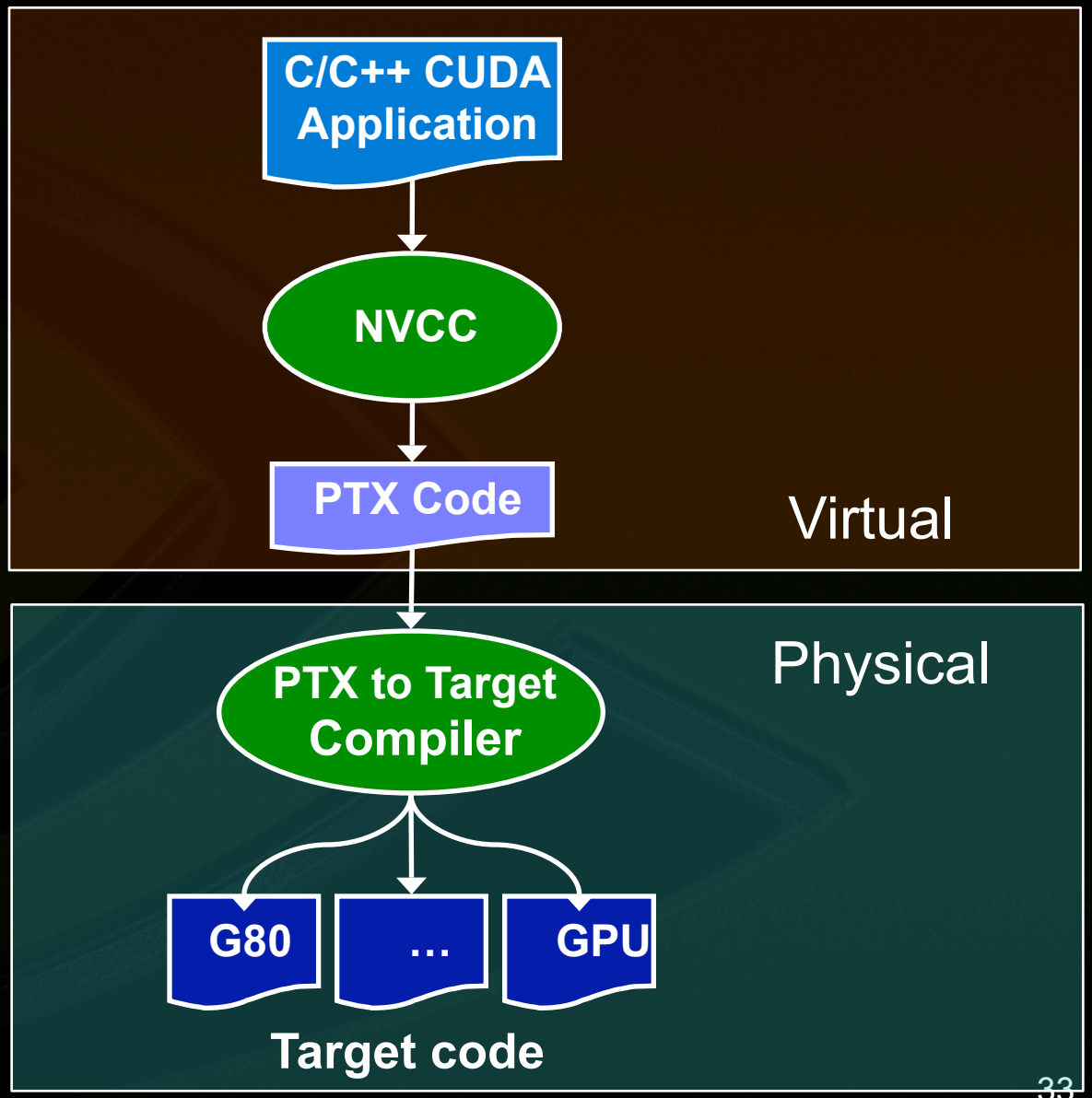
Device Runtime Component: Intrinsics

- Some mathematical functions have a less accurate, but faster device-only version
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`
  - `__umul24`

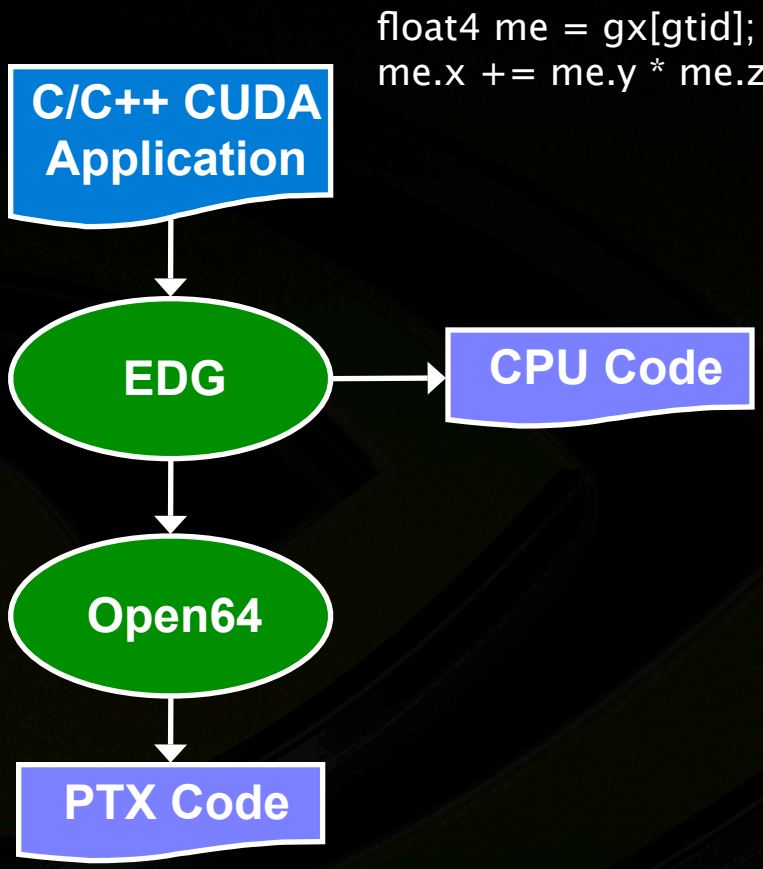
Compiling CUDA



Compiling CUDA



NVCC & PTX Virtual Machine



- 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

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

Compilation

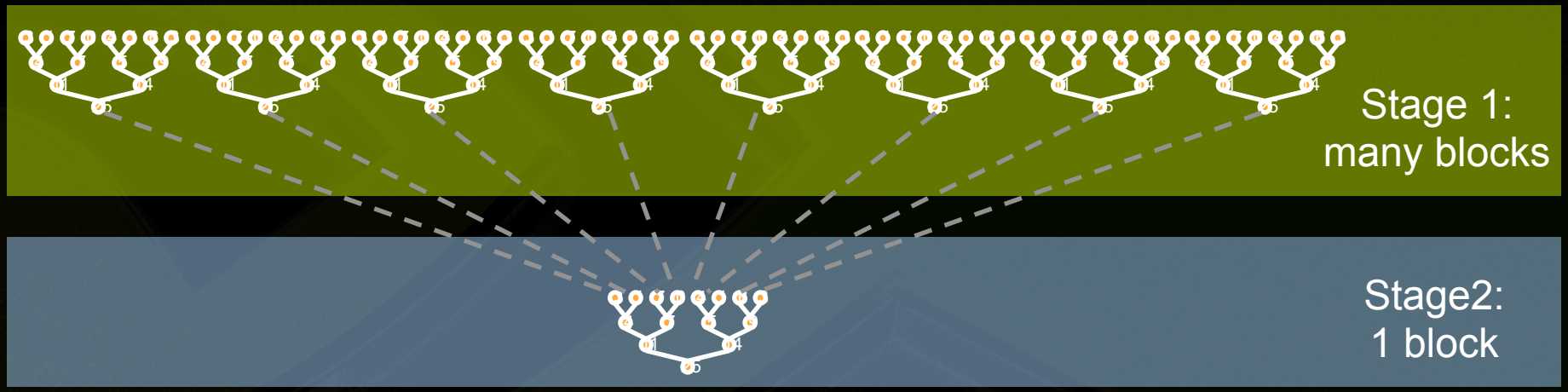
- Any source file containing CUDA language extensions must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application using another tool
  - Or PTX object code directly
- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

NVIDIA®

Code Walkthrough 2: Parallel Reduction

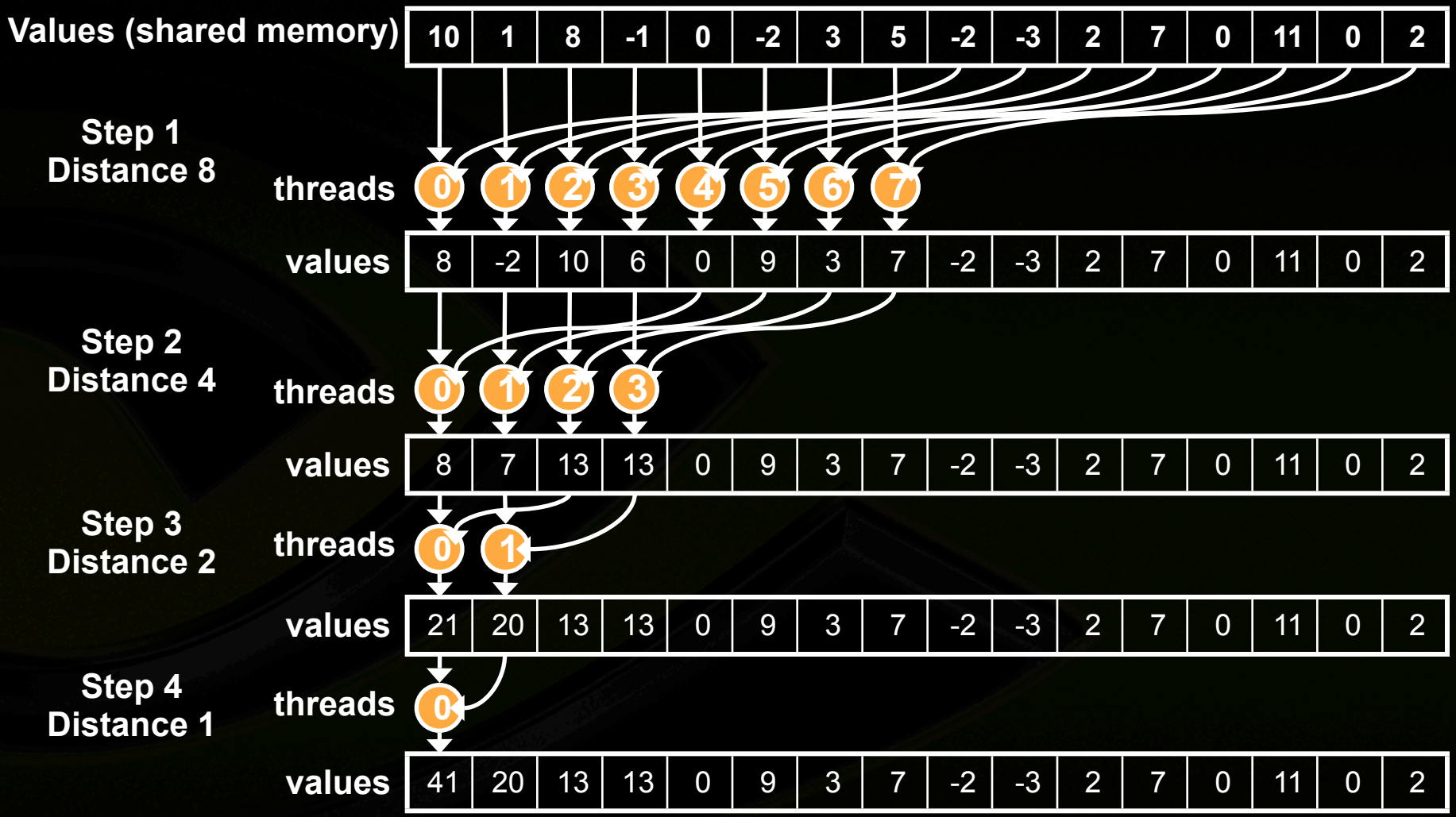
Execution Decomposition

- Two stages of computation:
  - Sum within each block
  - Sum partial results from the blocks



- For reductions, code for all levels is the same

Kernel execution



Kernel Source Code

```
__global__ void sum_kernel(int *g_input, int *g_output)
{
    extern __shared__ int s_data[];  // allocated during kernel launch

    // read input into shared memory
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    s_data[threadIdx.x] = g_input[idx];
    __syncthreads();

    // compute sum for the threadblock
    for(int dist = blockDim.x/2; dist>0; dist/=2)
    {
        if(threadIdx.x<dist)
            s_data[threadIdx.x] += s_data[threadIdx.x+dist];
        __syncthreads();
    }

    // write the block's sum to global memory
    if(threadIdx.x==0)
        g_output[blockIdx.x] = s_data[0];
}
```

Host Source Code (1)

```
int main()
{
    // data set size in elements and bytes
    unsigned int n = 4096;
    unsigned int num_bytes = n*sizeof(int);

    // launch configuration parameters
    unsigned int block_dim = 256;
    unsigned int num_blocks = n / block_dim;
    unsigned int num_smem_bytes = block_dim*sizeof(int);

    // allocate and initialize the data on the CPU
    int *h_a=(int*)malloc(num_bytes);
    for(int i=0;i<n;i++)
        h_a[i]=1;

    // allocate memory on the GPU device
    int *d_a=0, *d_output=0;
    cudaMalloc((void**)&d_a, num_bytes);
    cudaMalloc((void**)&d_output, num_blocks*sizeof(int));

                                        ...
```

Host Source Code (2)

```
                    ...

    // copy the input data from CPU to the GPU device
    cudaMemcpy(d_a, h_a, num_bytes, cudaMemcpyHostToDevice);

    // two stages of kernel execution
    sum_kernel<<<num_blocks, block_dim, num_smem_bytes>>>(d_a, d_output);
    sum_kernel<<<1, num_blocks, num_blocks*sizeof(int)>>>(d_output, d_output);

    // copy the output from GPU device to CPU and print
    cudaMemcpy(h_a, d_output, sizeof(int), cudaMemcpyDeviceToHost);
    printf("%d\n", h_a[0]);

    // release resources
    cudaFree(d_a);
    cudaFree(d_output);
    free(h_a);

    return 0;
}
```

NVIDIA®

CUDA Libraries

Outline

- CUDA includes 2 widely used libraries:
  - CUBLAS: BLAS implementation
  - CUFFT: FFT implementation

CUBLAS

CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA driver. It allows access to the computational resources of NVIDIA GPUs.

The library is self-contained at the API level, that is, no direct interaction with the CUDA driver is necessary.

The basic model by which applications use the CUBLAS library is to:

- create matrix and vector objects in GPU memory space,
- fill them with data,
- call a sequence of CUBLAS functions,
- upload the results from GPU memory space back to the host.

CUBLAS provides helper functions for creating and destroying objects in GPU space, and for writing data to and retrieving data from these objects.

Supported features

- **BLAS functions implemented (single precision only):**
  - **Real data: level 1, 2 and 3**
  - **Complex data: level1 and CGEMM**

  (Level 1=vector vector O(N), Level 2= matrix vector O(N^2), Level 3=matrix matrix O(N^3))

- **For maximum compatibility with existing Fortran environments, CUBLAS uses column-major storage, and 1-based indexing:**

  Since C and C++ use row-major storage, this means applications cannot use the native C array semantics for two-dimensional arrays. Instead, macros or inline functions should be defined to implement matrices on top of one-dimensional arrays.

Using CUBLAS

- The interface to the CUBLAS library is the header file cublas.h

- Function names: cublas(Original name).
  cublasSgemm

- Because the CUBLAS core functions (as opposed to the helper functions) do not return error status directly, CUBLAS provides a separate function to retrieve the last error that was recorded, to aid in debugging

- CUBLAS is implemented using the C-based CUDA tool chain, and thus provides a C-style API. This makes interfacing to applications written in C or C++ trivial.

cublasInit, cublasShutdown

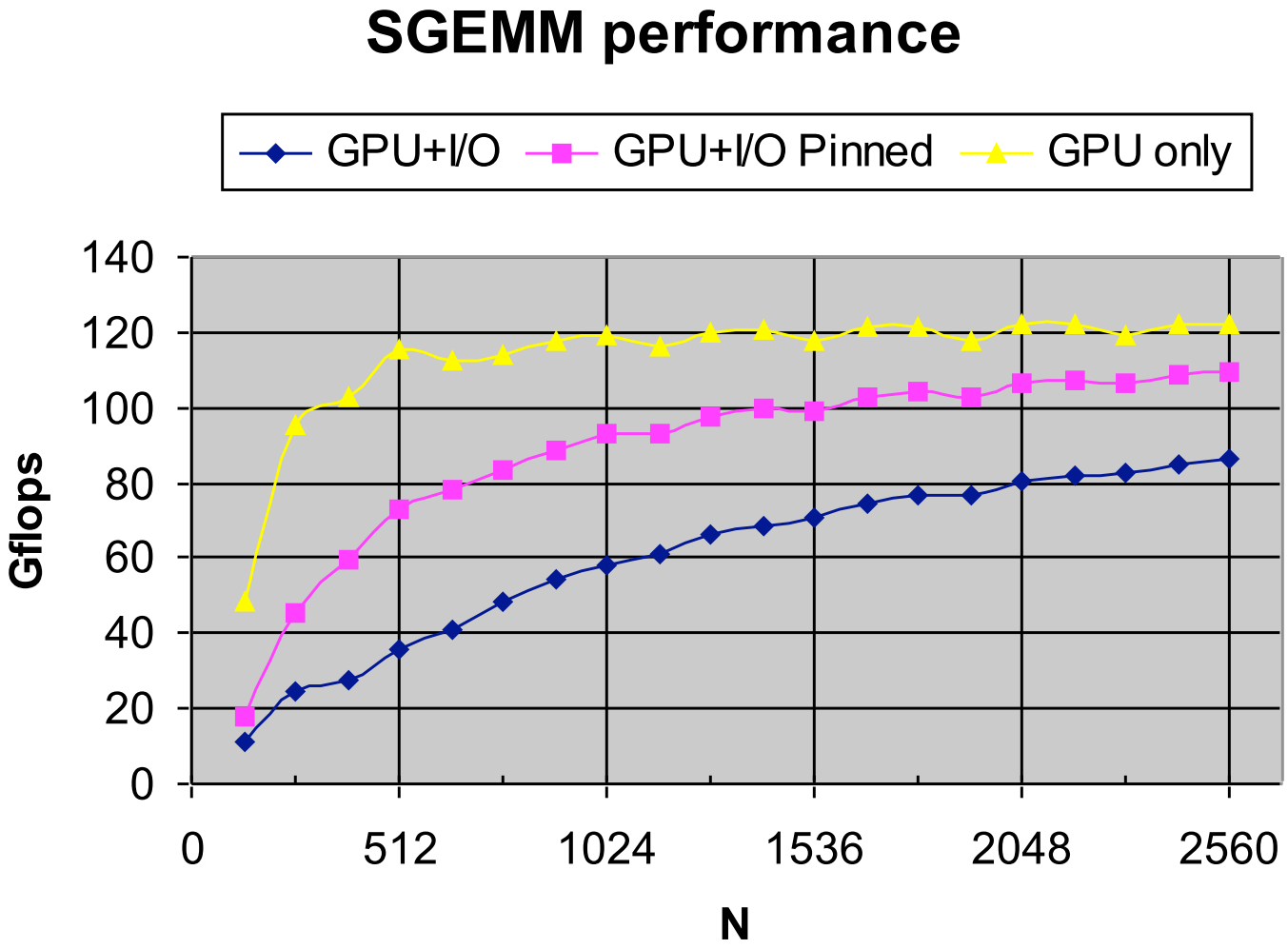
cublasStatus cublasInit()

initializes the CUBLAS library and must be called before any other CUBLAS API function is invoked. It allocates hardware resources necessary for accessing the GPU.

cublasStatus cublasShutdown()

releases CPU-side resources used by the CUBLAS library. The release of GPU-side resources may be deferred until the application shuts down.

CUBLAS performance



cublasGetError, cublasAlloc, cublasFree

cublasStatus cublasGetError()

returns the last error that occurred on invocation of any of the CUBLAS core functions. While the CUBLAS helper functions return status directly, the CUBLAS core functions do not, improving compatibility with those existing environments that do not expect BLAS functions to return status. Reading the error status via cublasGetError() resets the internal error state to CUBLAS_STATUS_SUCCESS..

cublasStatus cublasAlloc (**int n, int elemSize, void **devicePtr**)

creates an object in GPU memory space capable of holding an array of n elements, where each element requires elemSize bytes of storage.

Note that this is a device pointer that cannot be dereferenced in host code.

cublasAlloc() is a wrapper around cudaMalloc().

Device pointers returned by cublasAlloc() can therefore be passed to any CUDA device kernels, not just CUBLAS functions.

cublasSetVector, cublasGetVector

cublasStatus cublasSetVector(int n, int elemSize, const void *x, int incx, void *y, int incy)

copies n elements from a vector x in CPU memory space to a vector y in GPU memory space. Elements in both vectors are assumed to have a size of elemSize bytes. Storage spacing between consecutive elements is incx for the source vector x and incy for the destination vector y

cublasStatus cublasGetVector(int n, int elemSize, const void *x, int incx, void *y, int incy)

copies n elements from a vector x in GPU memory space to a vector y in CPU memory space. Elements in both vectors are assumed to have a size of elemSize bytes. Storage spacing between consecutive elements is incx for the source vector x and incy for the destination vector y

cublasSetMatrix, cublasGetMatrix

cublasStatus cublasSetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb)

copies a tile of rows x cols elements from a matrix A in CPU memory space to a matrix B in GPU memory space. Each element requires storage of elemSize bytes. Both matrices are assumed to be stored in column-major format, with the leading dimension (that is, the number of rows) of source matrix A provided in lda, and the leading dimension of destination matrix B provided in ldb.

cublasStatus cublasGetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb)

copies a tile of rows x cols elements from a matrix A in GPU memory space to a matrix B in CPU memory space. Each element requires storage of elemSize bytes. Both matrices are assumed to be stored in column-major format, with the leading dimension (that is, the number of rows) of source matrix A provided in lda, and the leading dimension of

Calling CUBLAS from FORTRAN

Fortran-to-C calling conventions are not standardized and differ by platform and toolchain.

In particular, differences may exist in the following areas:

- symbol names (capitalization, name decoration)
- argument passing (by value or reference)
- passing of string arguments (length information)
- passing of pointer arguments (size of the pointer)
- returning floating-point or compound data types (for example, single-precision or complex data type)

- CUBLAS provides wrapper functions (in the file fortran.c) that need to be compiled with the user preferred toolchain. Providing source code allows users
to make any changes necessary for a particular platform and toolchain.

Calling CUBLAS from FORTRAN

Two different interfaces:

•**Thunking** (define CUBLAS_USE_THUNKING when compiling fortran.c): allow interfacing to existing Fortran applications without any changes to the application. During each call, the wrappers allocate GPU memory, copy source data from CPU memory space to GPU memory space, call CUBLAS, and finally copy back the results to CPU memory space and deallocate the GPGPU memory. As this process causes significant call overhead, these wrappers are intended for light testing,not for production code.

•**Non-Thunking** (default):
intended for production code, substitute device pointers for vector and matrix arguments in all BLAS functions. To use these interfaces, existing applications need to be modified slightly to allocate and deallocate data structures in GPGPU memory space (using CUBLAS_ALLOC and CUBLAS_FREE) and to copy data between GPU and CPU memory spaces (using CUBLAS_SET_VECTOR, CUBLAS_GET_VECTOR, CUBLAS_SET_MATRIX, and CUBLAS_GET_MATRIX).

FORTRAN 77 Code example:

```
program matrixmod
implicit none
integer M, N
parameter (M=6, N=5)
real*4 a(M,N)
integer i, j

do j = 1, N
  do i = 1, M
     a(i,j) = (i-1) * M + j
  enddo
enddo

call modify (a, M, N, 2, 3, 16.0, 12.0)

do j = 1, N
  do i = 1, M
     write(*,"(F7.0$)") a(i,j)
  enddo
  write (*,*) ""
enddo

stop
end
```

```
subroutine modify (m, ldm, n, p, q, alpha, beta)
implicit none
integer ldm, n, p, q
real*4 m(ldm,*), alpha, beta

external sscal

call sscal (n-p+1, alpha, m(p,q), ldm)

call sscal (ldm-p+1, beta, m(p,q), 1)

return
end
```

FORTRAN 77 Code example: Non-thunking interface

```
program matrixmod
implicit none
integer M, N, sizeof_real, devPtrA
parameter (M=6, N=5, sizeof_real=4)
real*4 a(M,N)
integer i, j, stat
external cublas_init, cublas_set_matrix,cublas_get_matrix
external cublas_shutdown, cublas_alloc
integer cublas_alloc

do j = 1, N
  do i = 1, M
     a(i,j) = (i-1) * M + j
  enddo
enddo

call cublas_init
stat = cublas_alloc(M*N, sizeof_real, devPtrA)
if (stat .NE. 0) then
   write(*,*) "device memory allocation failed"
   stop
endif

call cublas_set_matrix (M, N, sizeof_real, a, M, devPtrA, M)
call modify (devPtrA, M, N, 2, 3, 16.0, 12.0)
call cublas_get_matrix (M, N, sizeof_real, devPtrA, M, a, M)
call cublas_free(devPtrA)
call cublas_shutdown
```

```
do j = 1, N
  do i = 1, M
     write(*,"(F7.0$)") a(i,j)
  enddo
  write (*,*) ""
enddo

stop
end

#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1)

subroutine modify (devPtrM, ldm, n, p, q, alpha, beta)
implicit none
integer ldm, n, p, q
integer sizeof_real, devPtrM
parameter (sizeof_real=4)
real*4  alpha, beta
call cublas_sscal (n-p+1, alpha,
                   devPtrM+IDX2F(p,q,ldm)*sizeof_real,
                   ldm)
call cublas_sscal (ldm-p+1, beta,
                   devPtrM+IDX2F(p,q,ldm)*sizeof_real,
                   1)
return
end
```

If using fixed format check that the line length is below the 72 column limit !!!

CUFFT

The Fast Fourier Transform (FFT) is a divide-and-conquer algorithm for efficiently computing discrete Fourier transform of complex or real-valued data sets.

The FFT is one of the most important and widely used numerical algorithms.

CUFFT, the “CUDA” FFT library, provides a simple interface for computing parallel FFT on an NVIDIA GPU. This allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation.

Supported features

- 1D, 2D and 3D transforms of complex and real-valued data
- Batched execution for doing multiple 1D transforms in parallel
- 1D transform size up to 8M elements
- 2D and 3D transform sizes in the range [2,16384]
- In-place and out-of-place transforms for real and complex data.

CUFFT Types and Definitions

type cufftHandle:
is a handle type used to store and access CUFFT plans

type cufftResults:
is an enumeration of values used as API function values return values.

CUFFT_SUCCESS	Any CUFFT operation is successful.
CUFFT_INVALID_PLAN	CUFFT is passed an invalid plan handle.
CUFFT_ALLOC_FAILED	CUFFT failed to allocate GPU memory.
CUFFT_INVALID_TYPE	The user requests an unsupported type.
CUFFT_INVALID_VALUE	The user specifies a bad memory pointer.
CUFFT_INTERNAL_ERROR	Used for all internal driver errors.
CUFFT_EXEC_FAILED	CUFFT failed to execute an FFT on the GPU.
CUFFT_SETUP_FAILED	The CUFFT library failed to initialize.
CUFFT_SHUTDOWN_FAILED	The CUFFT library failed to shut down.
CUFFT_INVALID_SIZE	The user specifies an unsupported FFT size.

Transform types

- **The library supports complex and real data transforms:**
  CUFFT_C2C, CUFFT_C2R ,CUFFT_R2C
  with directions:
  CUFFT_FORWARD (-1) and CUFFT_BACKWARD (1)
  according to the sign of the complex exponential term

- **For complex FFTs, the input and output arrays must interleaved the real and imaginary part (cufftComplex type is defined for this purpose)**

- **For real-to-complex FFTs, the output array holds only the non-redundant complex coefficients:**
  N -> N/2+1
  N0 x N1 x …. x Nn -> N0 x N1 x …. X (Nn/2+1)
  To perform in-place transform the input/output needs to be padded

More on transforms

- For 2D and 3D transforms, CUFFT performs transforms in row-major (C-order).
- If calling from FORTRAN or MATLAB, remember to change the order of size parameters during plan creation.
- CUFFT performs un-normalized transforms:

  IFFT(FFT(A))= length(A)*A
- CUFFT API is modeled after FFTW. Based on plans, that completely specify the optimal configuration to execute a particular size of FFT.
- Once a plan is created, the library stores whatever state is needed to execute the plan multiple times without recomputing the configuration: it works very well for CUFFT, because different kinds of FFTs require different thread configurations and GPU resources.

cufftPlan1d()

cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int batch);

creates a 1D FFT plan configuration for a specified signal size and data type.

The batch input parameter tells CUFFT how many 1D transforms to configure.

Input:

- plan — Pointer to a cufftHandle object
- nx — The transform size (e.g., 256 for a 256-point FFT)
- type — The transform data type (e.g., CUFFT_C2C for complex-to-complex)
- batch — Number of transforms of size nx

Output:

- plan — Contains a CUFFT 1D plan handle value

cufftPlan2d()

cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type);

creates a 2D FFT plan configuration for a specified signal size and data type.

Input:

- plan Pointer to a cufftHandle object
- nx The transform size in X dimension
- ny The transform size in Y dimension
- type The transform data type (e.g., CUFFT_C2C for complex-to-complex)

Output:

- plan Contains a CUFFT 2D plan handle value

cufftPlan3d()

cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type);

creates a 3D FFT plan configuration for a specified signal size and data type.

Input:

- plan — Pointer to a cufftHandle object
- nx — The transform size in X dimension
- ny — The transform size in Y dimension
- nz — The transform size in Z dimension
- type — The transform data type (e.g., CUFFT_C2C for complex-to-complex)

Output:

- plan — Contains a CUFFT 3D plan handle value

cufftDestroy(),

cufftResult cufftDestroy(cufftHandle plan);

frees all GPU resources associated with a CUFFT plan and destroys the internal plan data structure. This function should be called once a plan is no longer needed to avoid wasting GPU memory**.**

Input:

plan cufftHandle object

cufftExecC2C()

cufftResult cufftExecC2C(cufftHandle plan,
cufftComplex *idata, cufftComplex *odata,
int direction);

executes a CUFFT complex to complex transform plan.CUFFT uses as input data the GPU memory pointed to by the idata parameter. This function stores the Fourier coefficients in the odata array. If idata and odata are the same, this method does an in-place transform.

Input:

plan	cufftHandle object for the plane to update
idata	Pointer to the input data (in GPU memory) to transform
odata	Pointer to the output data (in GPU memory)
direction	The transform direction (CUFFT_FORWARD or CUFFT_BACKWARD)

Output:

odata	Contains the complex Fourier coefficients)

cufftExecR2C()

cufftResult cufftExecR2C(cufftHandle plan,
cufftReal *idata, cufftComplex *odata);

executes a CUFFT real to complex transform plan.CUFFT uses as input data the GPU memory pointed to by the idata parameter. This function stores the Fourier coefficients in the odata array. If idata and odata are the same, this method does an in-place transform.
The output hold only the non-redundant complex Fourier coefficients.

Input:

plan	Pointer to a cufftHandle object
idata	Pointer to the input data (in GPU memory) to transform
odata	Pointer to the output data (in GPU memory)

Output:

odata	Contains the complex Fourier coefficients

cufftExecC2R()

cufftResult cufftExecC2R(cufftHandle plan,
cufftComplex *idata, cufftReal *odata);

executes a CUFFT complex to real transform plan. CUFFT uses as input
data the GPU memory pointed to by the idata parameter. This function stores the Fourier coefficients in the odata array. If idata and odata are the same, this method does an in-place transform.
The input hold only the non-redundant complex Fourier coefficients.

Input:

plan	Pointer to a cufftHandle object
idata	Pointer to the complex input data (in GPU memory) to transform
odata	Pointer to the real output data (in GPU memory)

Output:

odata	Contains the real-valued Fourier coefficients

Accuracy and performance

The CUFFT library implements several FFT algorithms, each with different performances and accuracy.

The best performance paths correspond to transform sizes that:

1. Fit in CUDA'a shared memory
2. Are powers of a single factor (e.g. power-of-two)

If only condition 1 is satisfied, CUFFT uses a more general mixed-radix factor algorithm that is slower and less accurate numerically.

If none of the above conditions is satisfied, CUFFT uses an out-of-place, mixed-radix algorithm that stores all intermediate results in global GPU memory.

One notable exception is for long 1D transforms, where CUFFT uses a distributed algorithm that perform 1D FFT using 2D FFT, where the dimensions of the 2D transform are factors of

CUFFT does not implement any specialized algorithms for real data, and so there is no direct performance benefit to using real to complex (or complex to real) plans instead of complex to complex. For this release, the real data API exists primarily for convenience

Code example:
1D complex to complex transforms

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);

/* Create a 1D FFT plan. */
 cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);


 /* Use the CUFFT plan to transform the signal in place. */
 cufftExecC2C(plan, data, data, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
 cufftExecC2C(plan, data, data, CUFFT_INVERSE);

/* Note:
   (1) Divide by number of elements in data-set to get back original data
   (2) Identical pointers to input and output arrays implies in-place transformation
 */


 /* Destroy the CUFFT plan. */
   cufftDestroy(plan);

   cudaFree(data);
```

Code example:
2D complex to complex transform

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);

/* Create a 1D FFT plan. */
 cufftPlan2d(&plan, NX,NY, CUFFT_C2C);


 /* Use the CUFFT plan to transform the signal out of place. */
 cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Note:
    Different pointers to input and output arrays implies out of place transformation
 */


 /* Destroy the CUFFT plan. */
   cufftDestroy(plan);

   cudaFree(idata), cudaFree(odata);
```

NVIDIA®

Hands on exercises

Copying between host and device

- Start from the “handson1” template.
- **Part1**: Allocate memory for pointers *a_d* and *b_d* on the device.
- **Part2**: Copy *a* on the host to *a_d* on the device.
- **Part3**: Do a device to device copy from *a_d* to *b_d*.
- **Part4**: Copy *b_d* on the device back to *a* on the host.
- **Bonus**: Experiment with *cudaMallocHost* in place of *malloc* for allocating *a* and *b*.

Launching kernels

- Start from the “handson2” template.
- Part1: Allocate device memory for the result of the kernel using pointer *a_d*.
- Part2: Configure and launch the kernel using a 1-D grid and 1-D blocks.
- Part3: Have each thread set an element of *a_d* as follows:

  idx = blockIdx.x*blockDim.x + threadIdx.x

  a_d[idx] = 1000*blockIdx.x + threadIdx.x
- Part4: Copy the result in *a_d* back to the host.
- Part5: Verify that the result is correct.

Circular shift

- Shift all of the elements in an array. The shift is circular, i.e. elements shifted off one end are inserted again at the other end.
- The absolute value of SHIFT determines the amount of shift.
- The sign of SHIFT determines the direction:
  - Positive SHIFT moves each element toward the beginning
  - Negative SHIFT moves each element toward the end
  - Zero SHIFT does no shifting

NVIDIA®

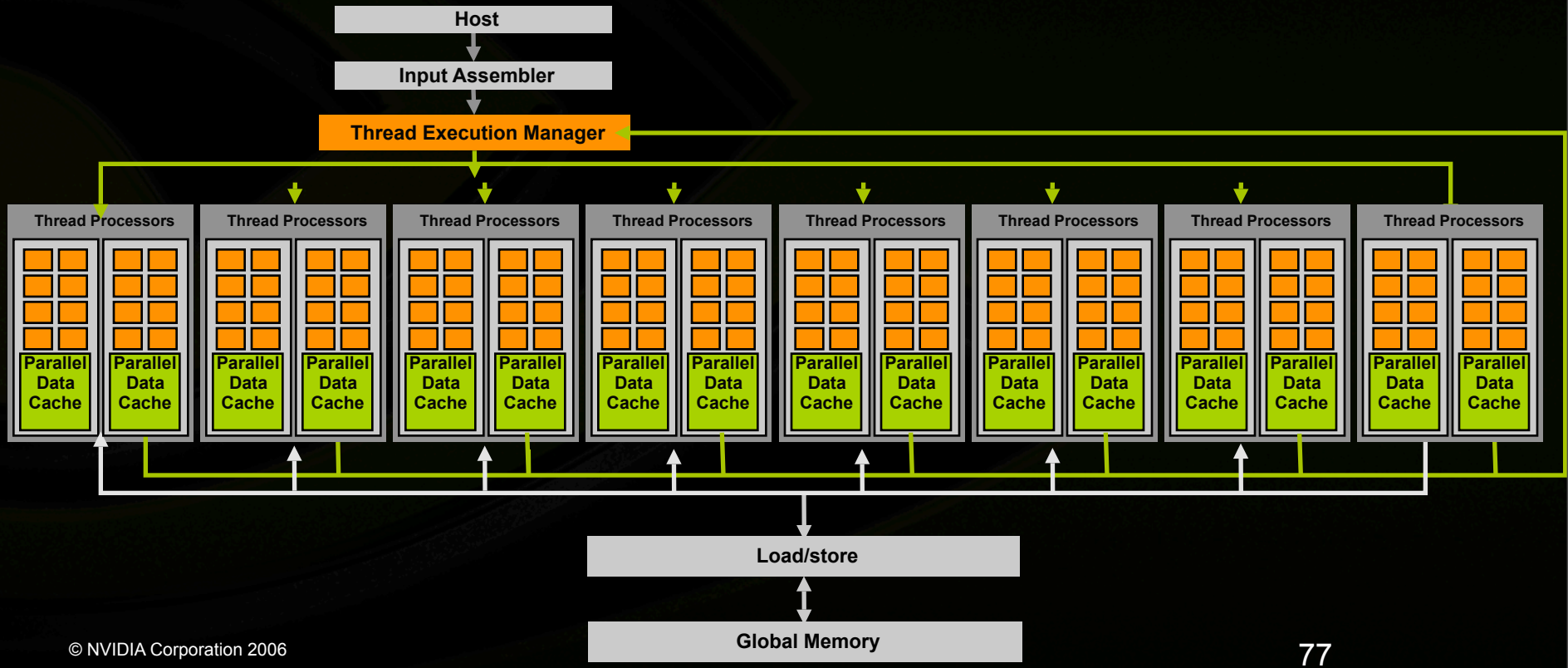
G8x Hardware Overview

Outline

- Hardware Overview
- CUDA Programming Model Overview
- Putting Hardware and Software Models Together
- CUDA Advantages over Legacy GPGPU

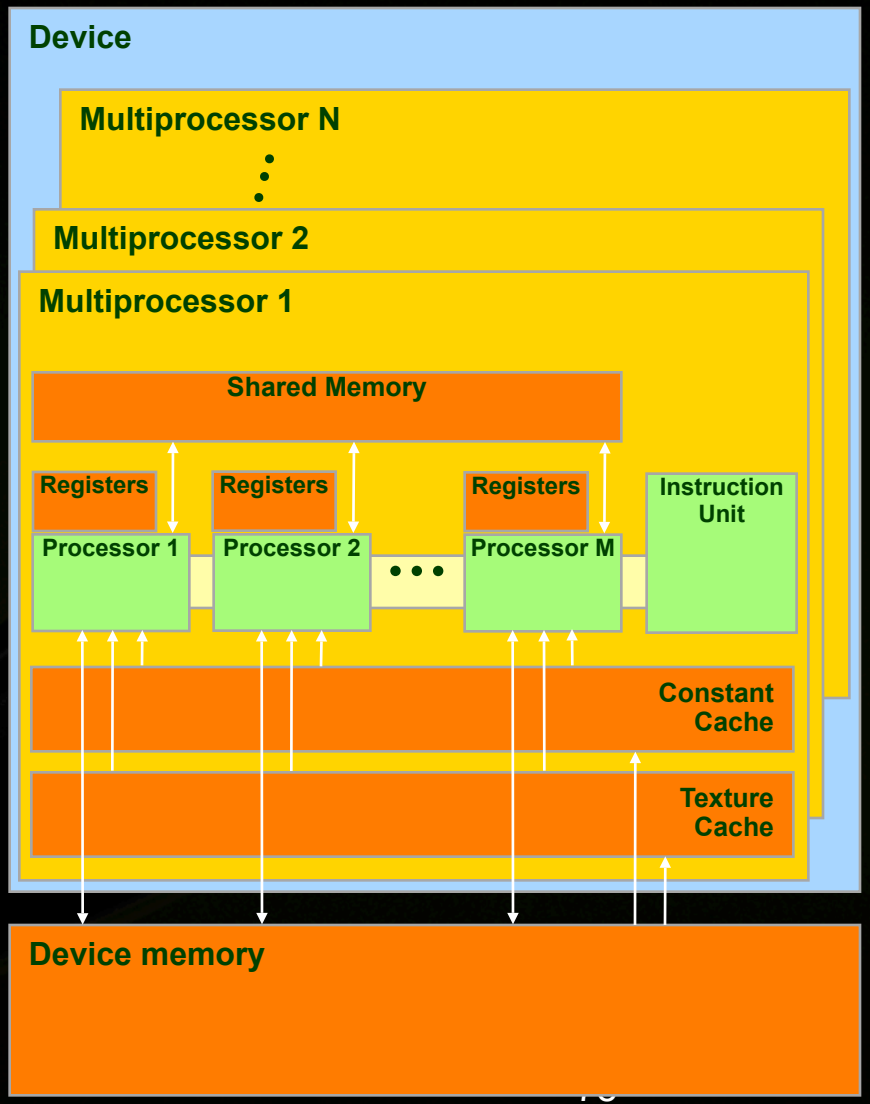
G80 Device

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors
- Parallel Data Cache accelerates processing



Hardware Implementation: Memory Architecture

- The local, global, constant, and texture spaces are regions of device memory
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space



NVIDIA®

Performance Optimization

CUDA is fast and efficient

- CUDA enables efficient use of the massive parallelism of NVIDIA GPUs
  - Direct execution of data-parallel programs
  - Without the overhead of a graphics API
- Even better speedups are achievable by understanding and tuning for GPU architecture
  - This presentation covers general performance, common pitfalls, and useful strategies

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Quick terminology review

- **Thread**: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA
  - Note difference from CPU threads: creation cost, resource usage, and switching cost of GPU threads is much smaller
- **Warp**: a group of threads executed *physically* in parallel (SIMD)
  - *Half-warp*: the first or second half of a warp of threads
- **Thread Block**: a group of threads that are executed together and can share memory on a single multiprocessor
- **Grid**: a group of thread blocks that execute a single CUDA kernel *logically* in parallel on a single GPU

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

CUDA Optimization Strategies

- Optimize Algorithms for the GPU
- Optimize Memory Accesses
- Take Advantage of On-Chip Shared Memory
- Use Parallelism Efficiently

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 Coalescing

- 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 SDK example

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

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Global and Shared Memory

- Global memory not cached on G8x GPUs
  - High latency, but launching more 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

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Memory Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI-e x16 Gen 1) vs. 80 GB/s peak (Quadro FX 5600)
- 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 ("pinned") host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s common on PCI-e x16
  - ~4 GB/s measured on nForce 680i motherboards
- 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

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Global Memory Reads/Writes

- Global memory is not cached on G8x
- Highest latency instructions: 400-600 clock cycles
- Likely to be performance bottleneck
- Optimizations can greatly increase performance

Loading and storing global memory

- Use -ptx flag to nvcc to inspect instructions:

4 byte load and store:

```
ld.global.f32       $f1, [$rd4+0];       //  id:74
…
st.global.f32       [$rd4+0], $f2;       //  id:75
```

…

8 byte load and store:

```
ld.global.v2.f32    {$f3,$f5}, [$rd7+0];              //
…
st.global.v2.f32    [$rd7+0], {$f4,$f6};              //
```

…

16 byte load and store:

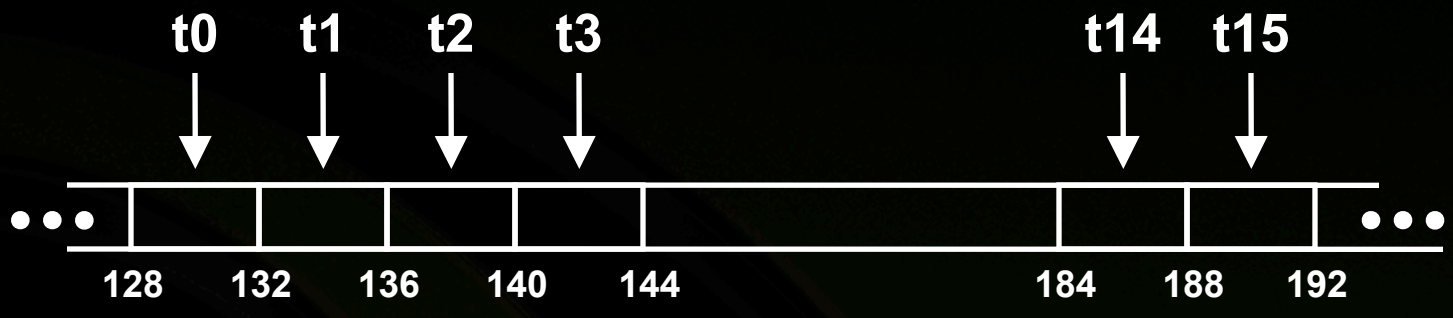
```
ld.global.v4.f32    {$f7,$f9,$f11,$f13}, [$rd10+0];          //
…
st.global.v4.f32    [$rd10+0], {$f8,$f10,$f12,$f14};         //
```

- If per-thread memory accesses for a single half-warp form a contiguous range of addresses, accesses will be *coalesced* into a single access
  - Coalesced accesses are much faster than non-coalesced

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:
  - 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

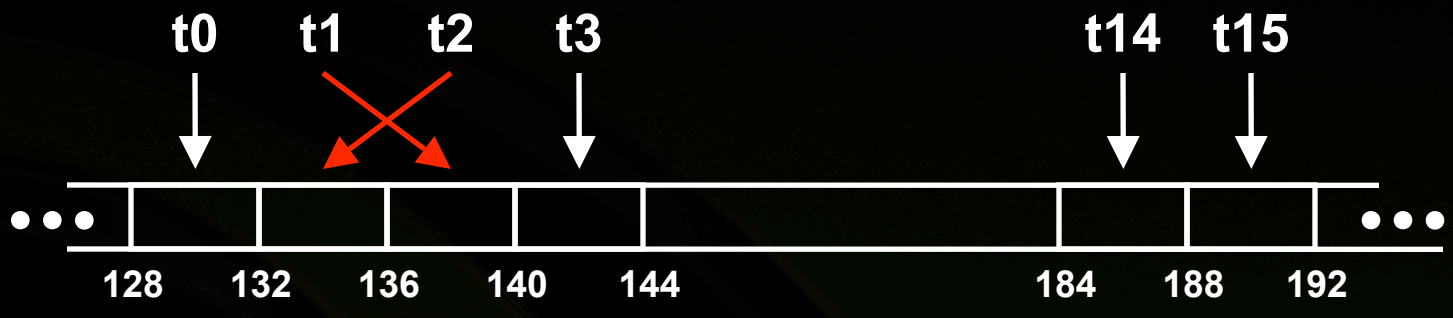
Coalesced Access: Reading floats



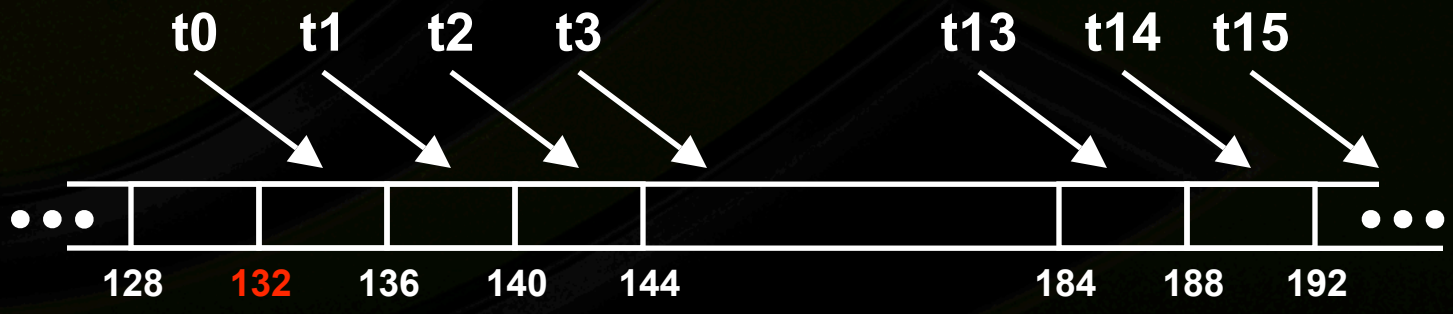
All threads participate

Some Threads Do Not Participate

Uncoalesced Access: Reading floats



Permuted Access by Threads



Misaligned Starting Address (not a multiple of 64)

Coalescing: Timing Results

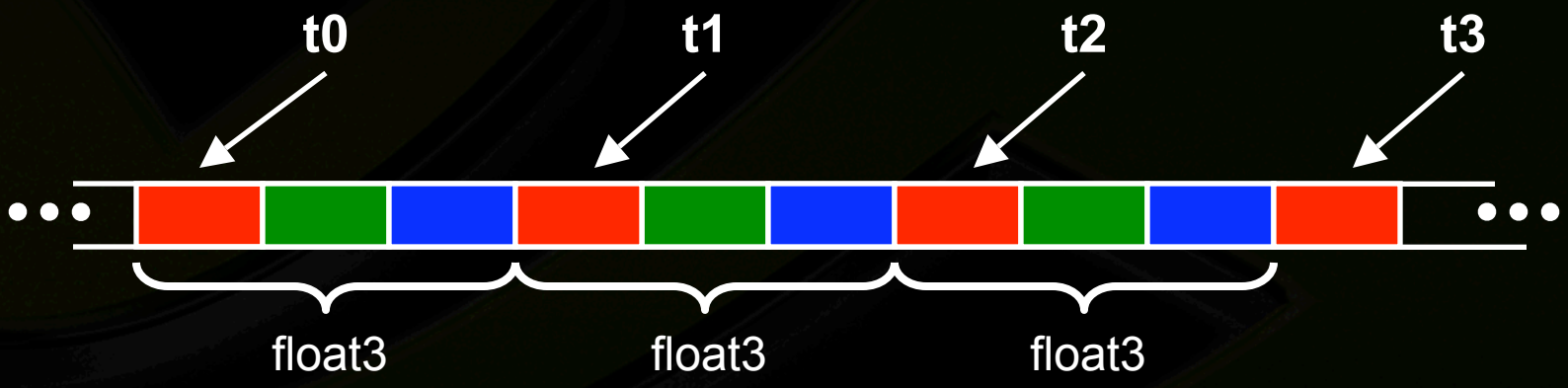
- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads:
  - 356µs – coalesced
  - 357µs – coalesced, some threads don't participate
  - 3,494µs – permuted/misaligned thread access

Uncoalesced float3 Code

```
__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;
}
```

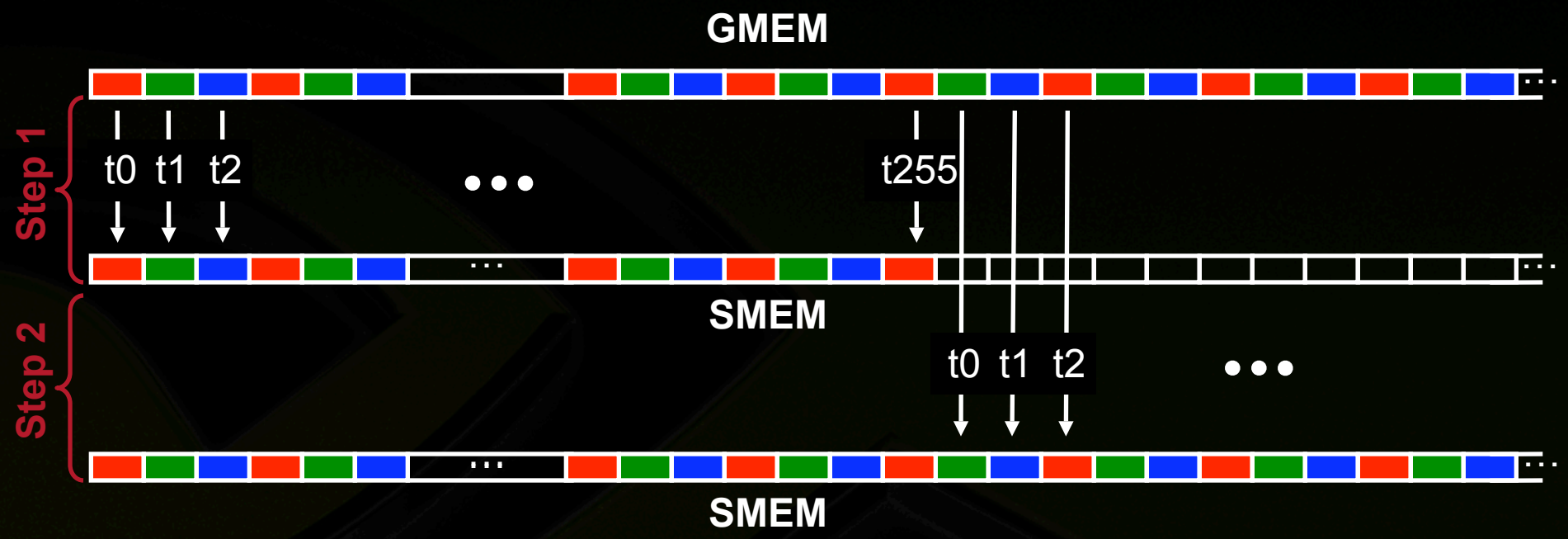
Uncoalesced Access: float3 Case

- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) ≠ 4, 8, or 16
  - 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

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = 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 ≠ 4, 8, or 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, or 16
  - Use SMEM to achieve coalescing

Point structure

AoS

SoA

Coalescing: Summary

- Coalescing greatly improves throughput
- Critical to small or 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

Outline

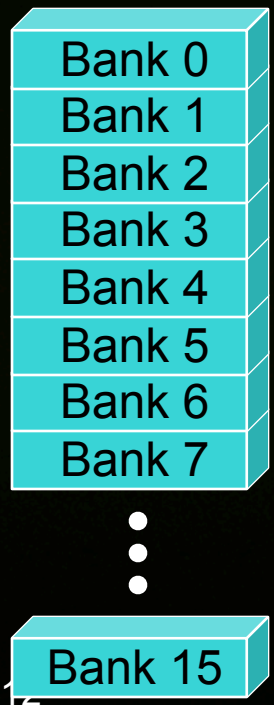
- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Shared Memory

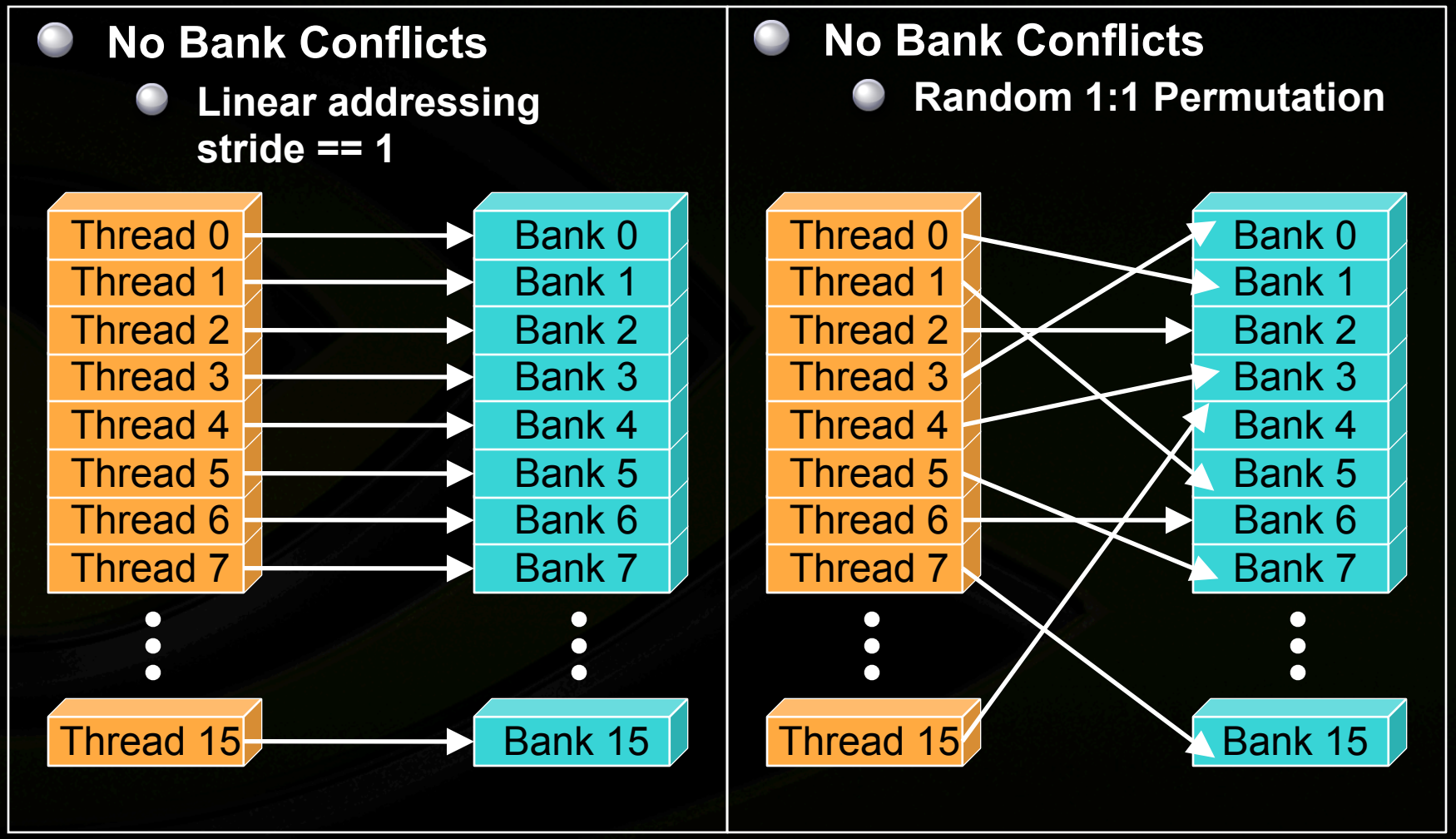
- Hundred times faster than global memory
- Cache data to prevent global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - See Matrix transpose SDK example

Parallel Memory Architecture

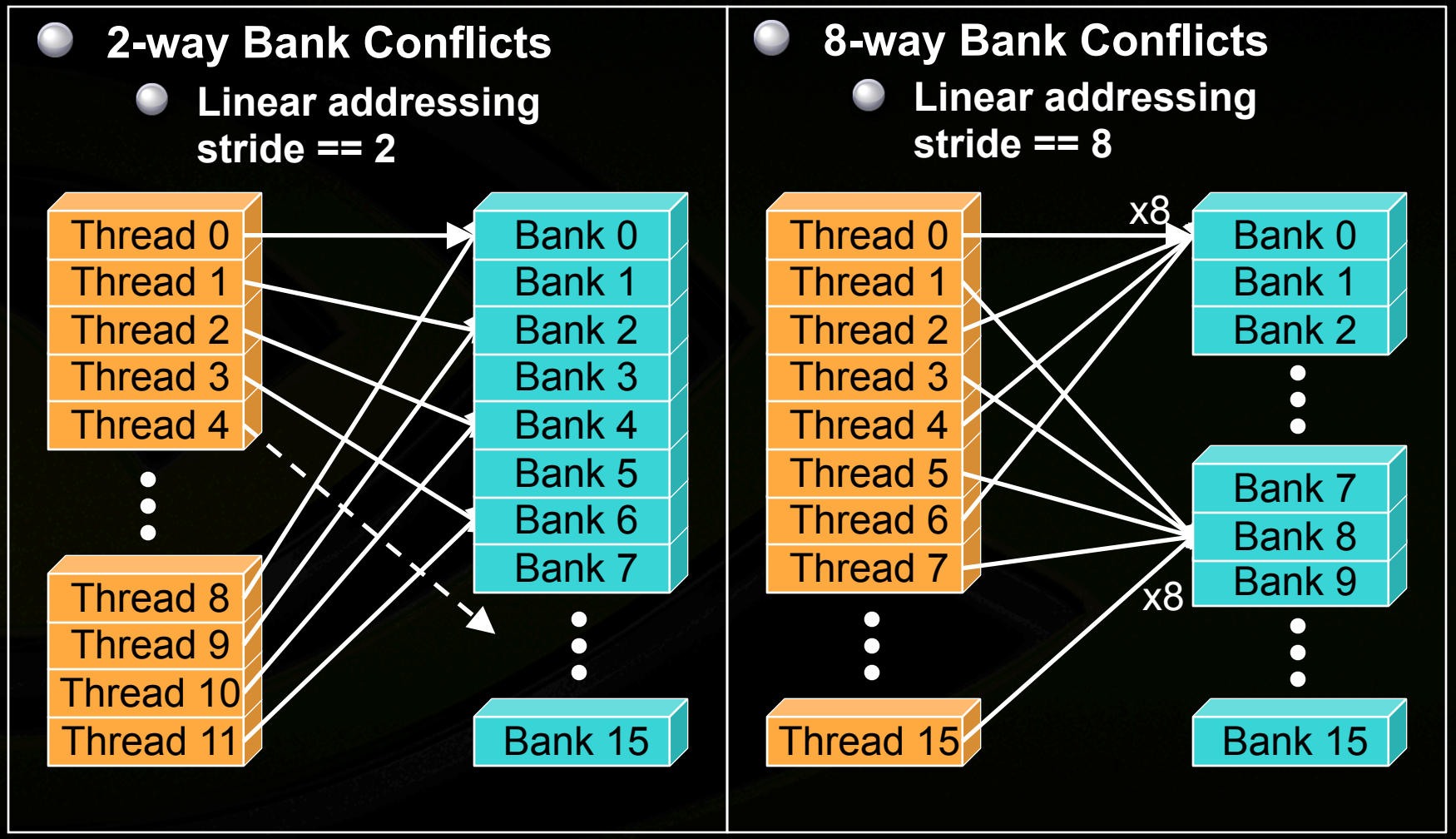
- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



Bank Addressing Examples



Bank Addressing Examples



Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- Use the bank checker macro in the SDK to check for 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

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Occupancy

- Thread instructions executed sequentially, 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
- Minimize occupancy requirements by minimizing latency
- Maximize occupancy optimizing threads per multiprocessor

Minimize Occupancy Requirements

- Optimize global memory access:
  - 400-600 cycle latency
- Maximize arithmetic intensity (math/bandwidth)
- Follow all the global memory optimizations described before!

Register Dependency

- Read-after-write register dependency
  - Instruction's result can be read ~11 cycles later
  - Scenarios:

CUDA:

```
x = y + 5;
z = x + 3;
```

PTX:

```
add.f32  $f3, $f1, $f2
add.f32  $f5, $f3, $f4
```

CUDA:

```
s_data[0] += 3;
```

PTX:

```
ld.shared.f32  $f3, [$r31+0]
add.f32        $f3, $f3, $f4
```

- 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

Grid/Block Size Heuristics

- # of blocks / # of multiprocessors > 1
  - So all multiprocessors have at least one block to execute
- Per-block resources at most half of total available
  - Shared memory and registers
  - Multiple blocks can run concurrently in a multiprocessor
  - If multiple blocks coexist that aren't all waiting at a __syncthreads(), machine can stay busy
- # of blocks / # of multiprocessors > 2
  - So multiple blocks run concurrently in a multiprocessor
- # of blocks > 100 to scale to future devices
  - Blocks stream through machine in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations

Register Pressure

- Solution to latency issues = 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

- 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.

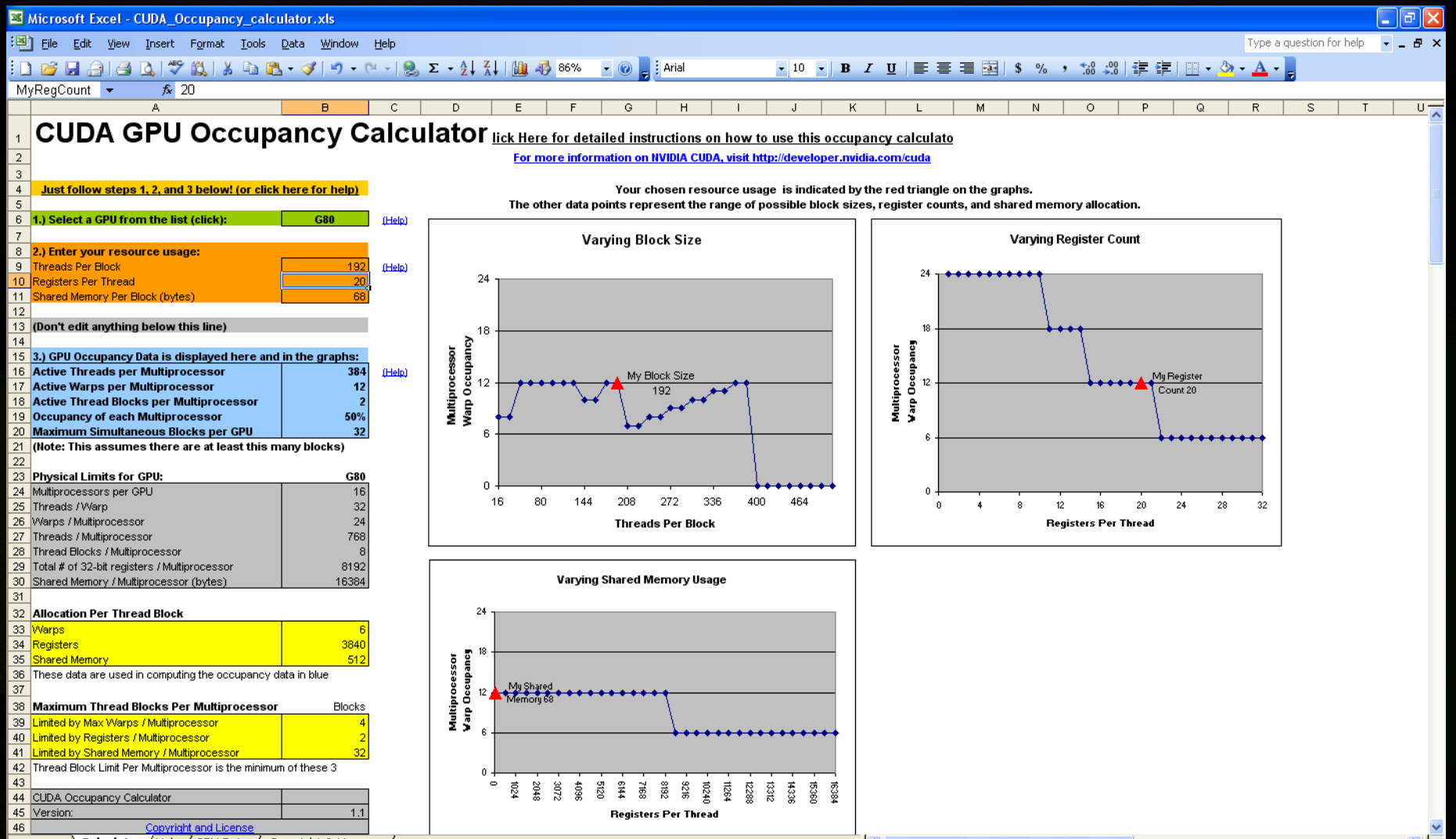
```
architecture {sm_10}
abiversion {0}
modname {cubin}
code  {
        name = BlackScholesGPU
        lmem = 0
        smem = 68
        reg = 20
        bar = 0
        bincode  {
                0xa0004205 0x04200780 0x40024c09 0x00200780
                …
```

per thread local memory

per thread block shared memory

per thread registers

CUDA Occupancy Calculator



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!
    - Experiment!

Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

BUT…

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - (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

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

CUDA Instruction Performance

- Instruction cycles (per warp) = sum of
  - Operand read cycles
  - Instruction execution cycles
  - Result update cycles
- Therefore instruction throughput depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth
- “Cycle” refers to the multiprocessor clock rate
  - 1.35 GHz on the GeForce 8800 GTX GPU, for example

Maximizing Instruction Throughput

- Maximize use of high-bandwidth memory
  - Maximize use of shared memory
  - Minimize accesses to global memory
  - Maximize coalescing of global memory accesses
- Optimize performance by overlapping memory accesses with HW computation
  - High arithmetic intensity programs
    - i.e. high ratio of math to memory transactions
  - Many concurrent threads

Arithmetic Instruction Throughput

- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply
- Integer divide and modulo are more expensive
  - Compiler will convert literal power-of-2 divides to shifts
    - But we have seen it miss some cases
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2

Arithmetic Instruction Throughput

- The intrinsics reciprocal, reciprocal square root, sin/cos, log, exp prefixed with "__" are 16 cycles per warp
  - Examples: __rcp(), __sin(), __exp()
- Other functions are combinations of the above
  - y / x == rcp(x) * y takes 20 cycles per warp
  - sqrt(x) == x * rsqrt(x) takes 20 cycles per warp

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Runtime Math Library

- There are two types of runtime math operations
  - __func(): direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x,y)
- The -use_fast_math compiler option forces every func() to compile to __func()

Make your program float-safe!

- Future hardware will have double precision support
  - G8x is single-precision only
  - Double precision will have additional cost
- Important to be float-safe to avoid using double precision where it is not needed
  - Add ‘f’ specifier on float literals:

```
foo = bar * 0.123;   // double assumed
foo = bar * 0.123f;  // float explicit
```

  - Use float version of standard library functions

```
foo = sin(bar);   // double assumed
foo = sinf(bar);  // float explicit
```

G8x Deviations from IEEE-754

- Addition and Multiplication are IEEE compliant
  - Maximum 0.5 ulp error
- However, often combined into multiply-add (FMAD)
  - Intermediate result is truncated

- Division is non-compliant (2 ulp)
- Not all rounding modes are supported
- Denormalized numbers are not supported
- No mechanism to detect floating-point exceptions

GPU Floating Point Features

	G8x	SSE	IBM Altivec	Cell SPE
Format	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes, only clamps to max norm	Yes	Yes	No, infinity
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No

GPU results may not match CPU

- Many variables: hardware, compiler, optimization settings
- CPU operations aren't strictly limited to 0.5 ulp
  - Sequences of operations can be more accurate due to 80-bit extended precision ALUs
- Floating-point arithmetic is not associative!

FP Math is Not Associative!

- In symbolic math, (x+y)+z == x+(y+z)
- This is not necessarily true for floating-point addition
  - Try $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ in the above equation
- When you parallelize computations, you potentially change the order of operations
- Parallel results may not exactly match sequential results
  - This is not a GPU or CUDA bug!

Outline

- CUDA optimization strategies
- Memory optimizations
  - Optimizing memory transfers
  - Coalescing global memory accesses
  - Using shared memory effectively
  - Hiding latency and balancing resource usage
- Code optimizations
  - Instruction performance & latency
  - Instruction accuracy & precision
  - Control flow

Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths must be serialized
- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `if (threadIdx.x > 2) { }`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) { }`
    - Branch granularity is a whole multiple of warp size

Conclusion

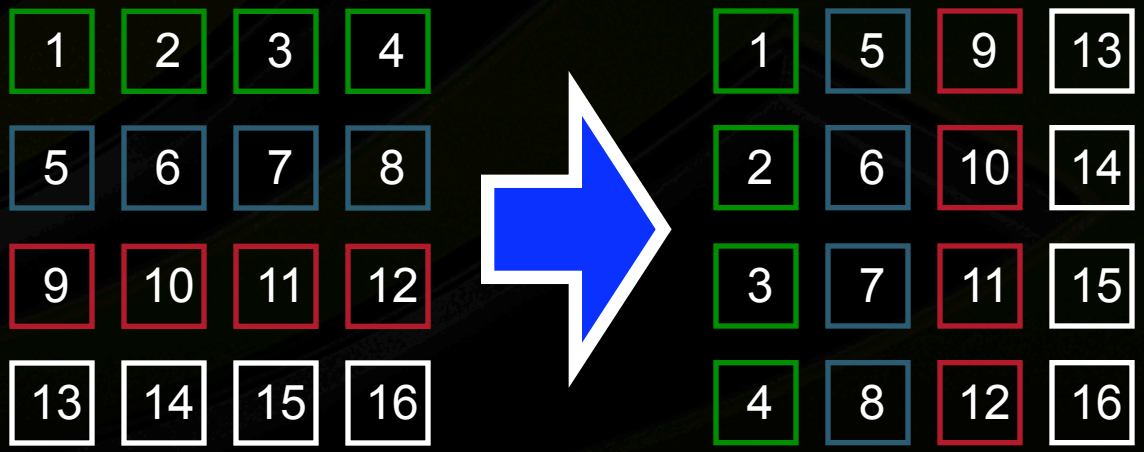
- G8x hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
  - Coalesce memory accesses
  - Take advantage of shared memory
  - Use parallelism efficiently
  - Avoid bank conflicts

NVIDIA®

Optimization Example 1: Matrix Transpose

Matrix Transpose

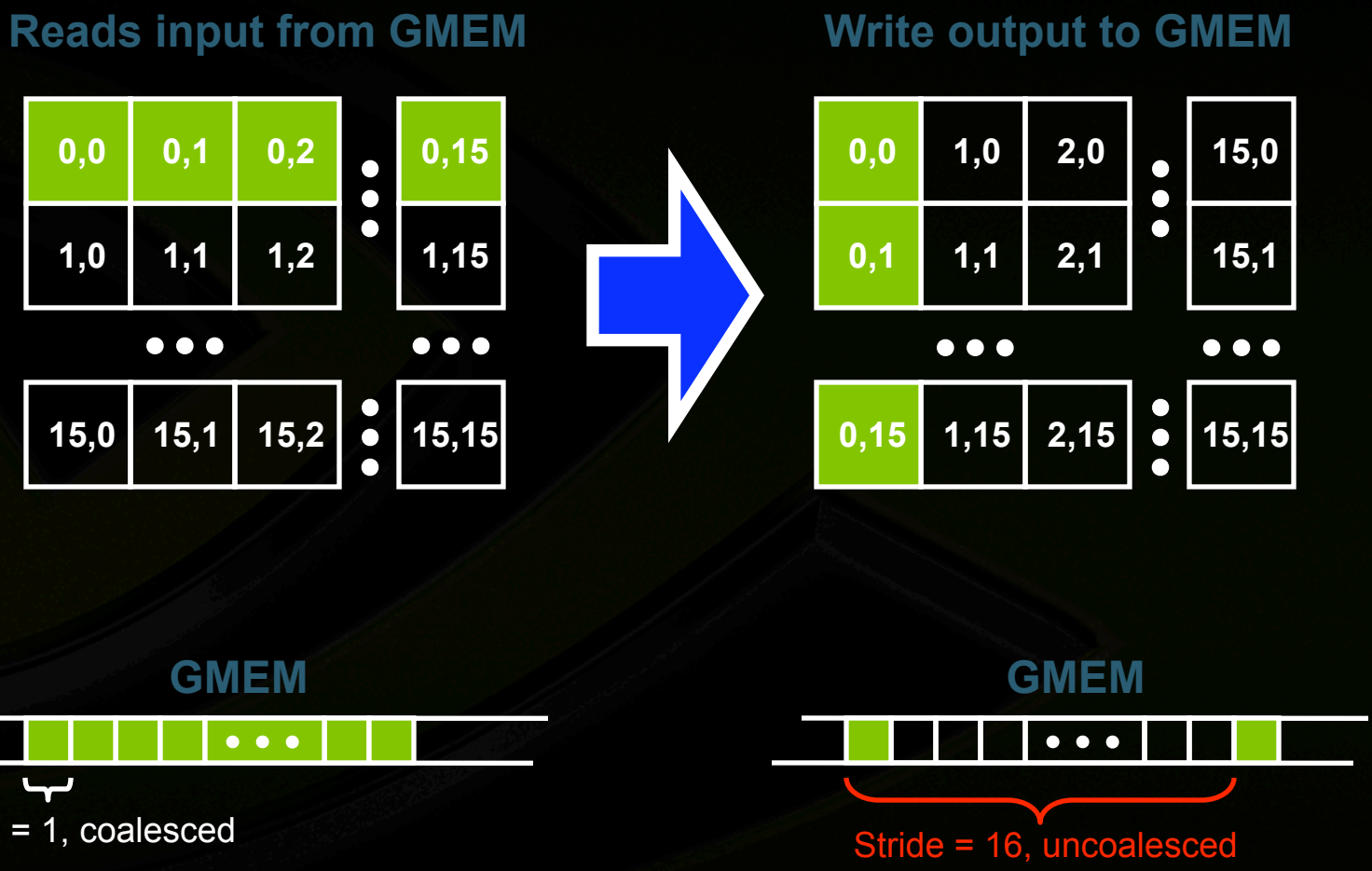
- SDK Sample (“transpose”)
- Illustrates:
  - Coalescing
  - Avoiding SMEM bank conflicts
  - Speedups for even small matrices



Uncoalesced Transpose

```
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
1.  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
2.  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

3.  if (xIndex < width && yIndex < height)
    {
4.      unsigned int index_in  = xIndex + width * yIndex;
5.      unsigned int index_out = yIndex + height * xIndex;
6.      odata[index_out] = idata[index_in];
    }
}
```

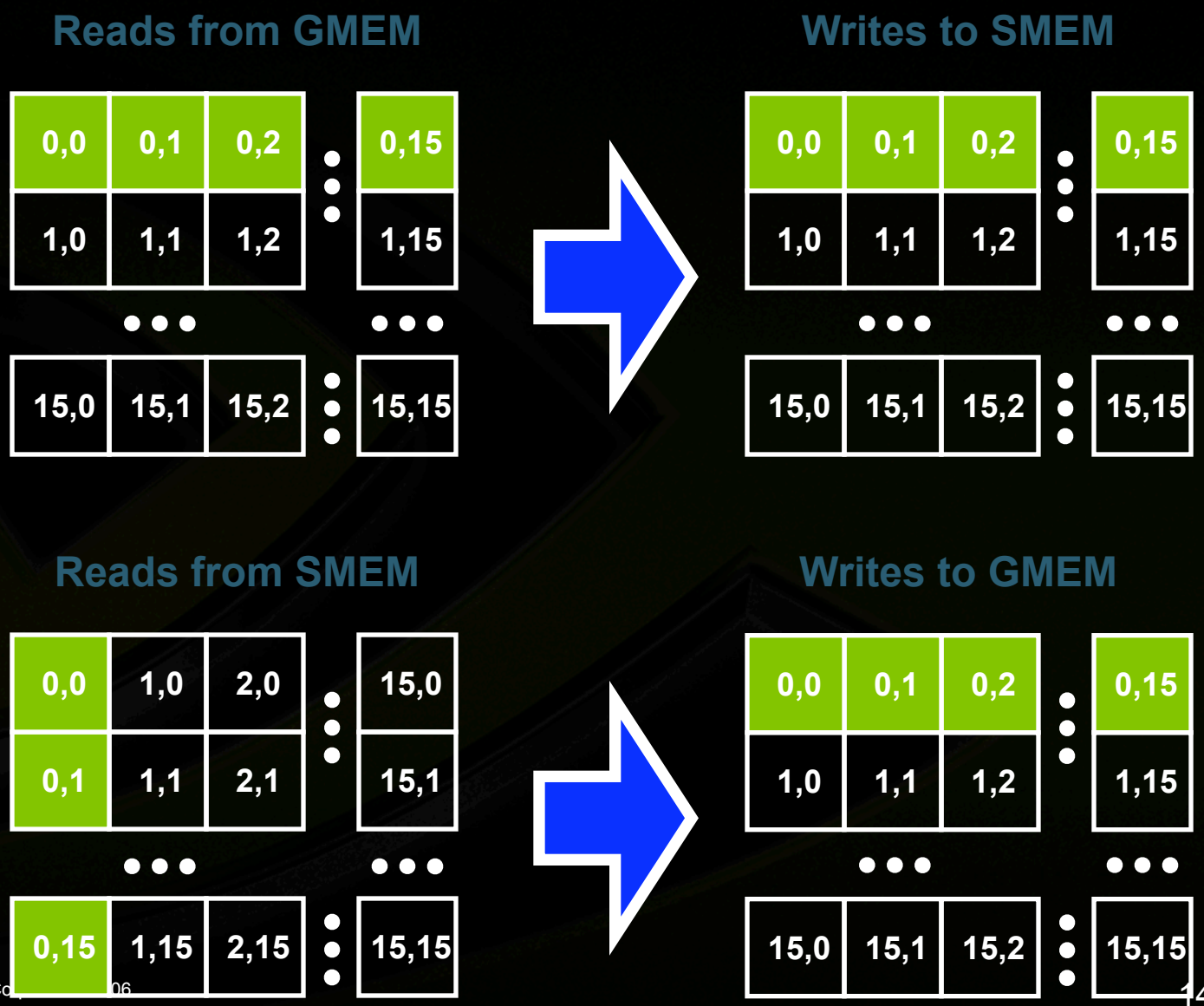
Uncoalesced Transpose



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



SMEM Optimization

Reads from SMEM

- Threads read SMEM with stride = 16
  - Bank conflicts

- Solution
  - Allocate an “extra” column
  - Read stride = 17
  - Threads read from consecutive banks

Coalesced Transpose

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
1.  __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];

2.  unsigned int xBlock = __mul24(blockDim.x, blockIdx.x);
3.  unsigned int yBlock = __mul24(blockDim.y, blockIdx.y);
4.  unsigned int xIndex = xBlock + threadIdx.x;
5.  unsigned int yIndex = yBlock + threadIdx.y;
6.  unsigned int index_out, index_transpose;

7.  if (xIndex < width && yIndex < height)
    {
8.      unsigned int index_in = __mul24(width, yIndex) + xIndex;
9.      unsigned int index_block = __mul24(threadIdx.y, BLOCK_DIM+1) + threadIdx.x;
10.     block[index_block] = idata[index_in];
11.     index_transpose = __mul24(threadIdx.x, BLOCK_DIM+1) + threadIdx.y;
12.     index_out = __mul24(height, xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
13. __syncthreads();

14. if (xIndex < width && yIndex < height)
15.     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

NVIDIA®

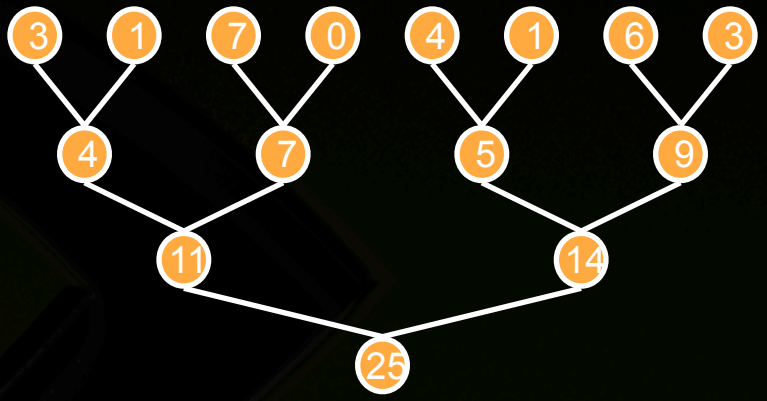
Optimization Example 2: Parallel Reduction

Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - We'll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies

Parallel Reduction

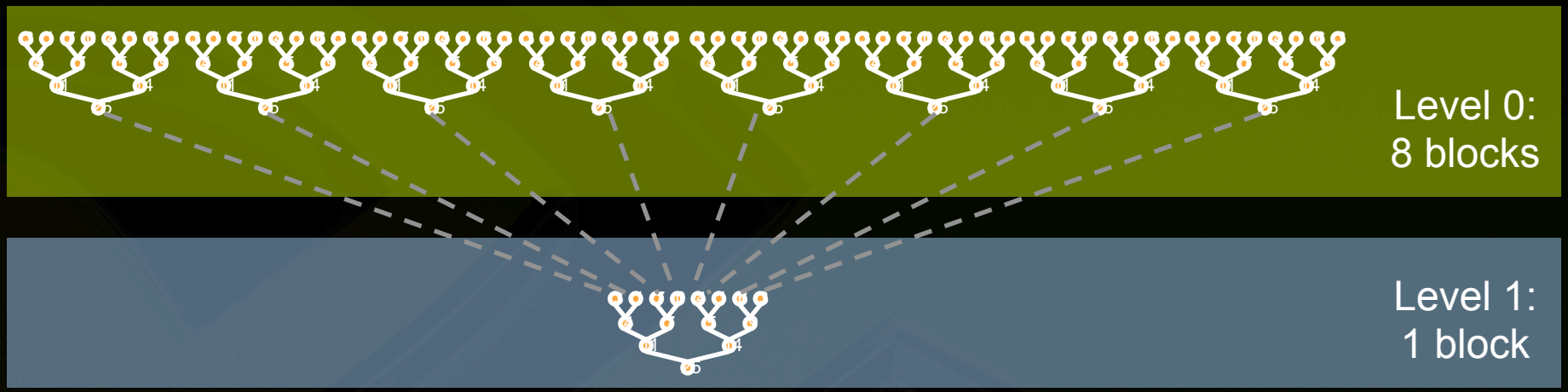
- Tree-based approach used within each thread block



- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations



- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

What is Our Optimization Goal?

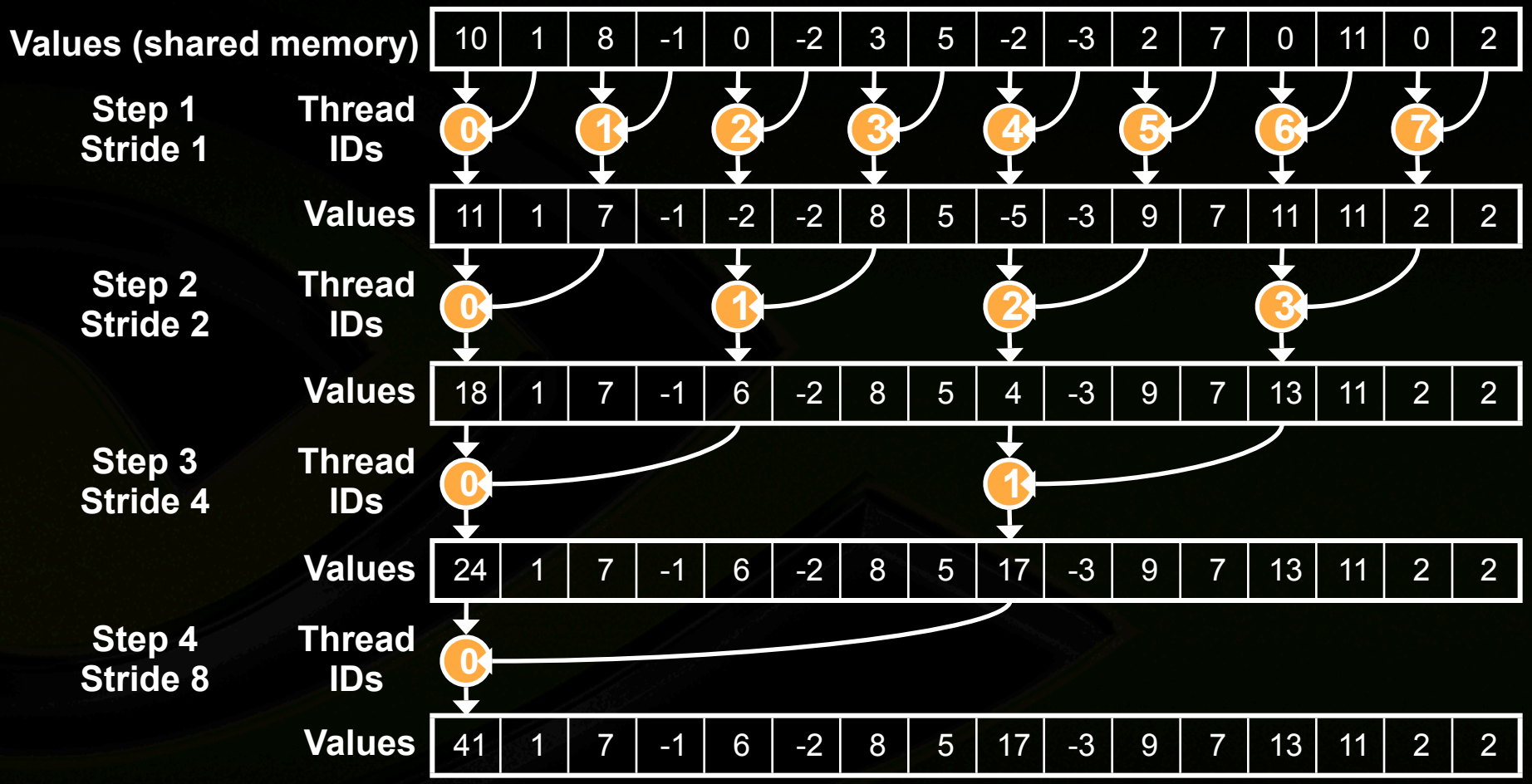
- We should strive to reach GPU peak performance
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth

- Will use G80 GPU for this example
  - 384-bit memory interface, 900 MHz DDR
  - 384 * 1800 / 8 = 86.4 GB/s

Reduction #1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Parallel Reduction: Interleaved Addressing



Reduction #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent branching results in very poor performance!

Performance for 4M element reduction

Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

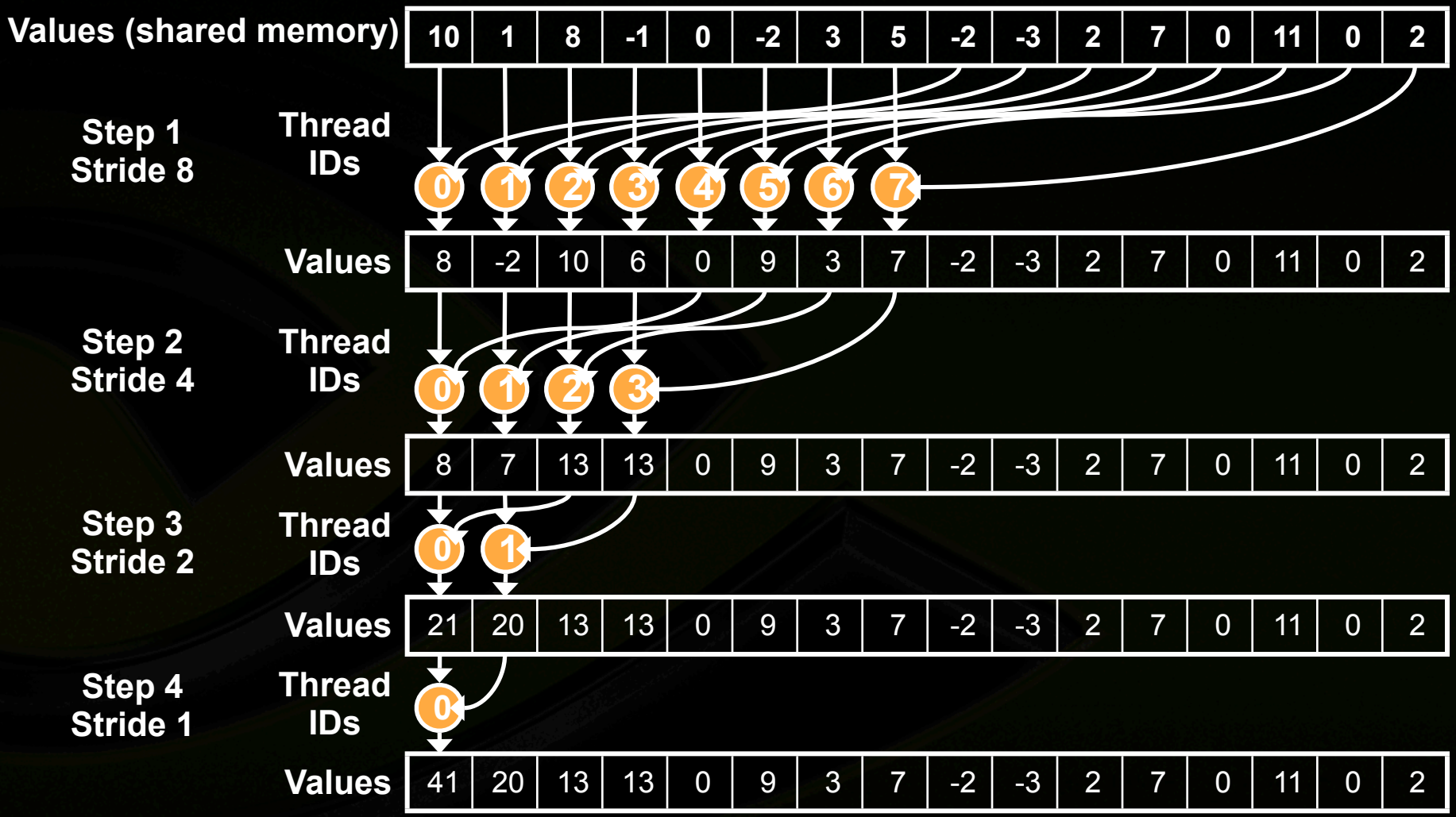
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

New Problem: Shared Memory Bank Conflicts

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x

Parallel Reduction: Sequential Addressing



Sequential addressing is conflict free

Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x

Idle Threads

Problem:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Half of the threads are idle on first loop iteration!

This is wasteful…

Reduction #4: First Add During Load

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x

Instruction Bottleneck

- At 17 GB/s, we're far from bandwidth bound
  - And we know reduction has low arithmetic intensity
- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead
- Strategy: unroll loops

Unrolling the Last Warp

- As reduction proceeds, # “active” threads decreases
  - When s <= 32, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when s <= 32:
  - We don’t need to __syncthreads()
  - We don’t need “if (tid < s)” because it doesn’t save any work
- Let’s unroll the last 6 iterations of the inner loop

Reduction #5: Unroll the Last Warp

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}

if (tid < 32)
{
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid +  8];
    sdata[tid] += sdata[tid +  4];
    sdata[tid] += sdata[tid +  2];
    sdata[tid] += sdata[tid +  1];
}
```

Note: This saves useless work in *all* warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x

Complete Unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Luckily, the block size is limited by the GPU to 512 threads
  - Also, we are sticking to power-of-2 block sizes
- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?
- Templates to the rescue!
  - CUDA supports C++ template parameters on device and host functions

Unrolling with Templates

- Specify block size as a function template parameter:

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

Reduction #6: Completely Unrolled


```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads();
}

if (tid < 32) {
    if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8)  sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4)  sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2)  sdata[tid] += sdata[tid +  1];
}
```

Note: all code in RED will be evaluated at compile time.

Results in a very efficient inner loop!

Invoking Template Kernels

- Don't we still need block size at compile time?
  - Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
      {
      case 512:
         reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case 256:
         reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case 128:
         reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case 64:
         reduce5<  64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case 32:
         reduce5<  32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case 16:
         reduce5<  16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case  8:
         reduce5<   8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case  4:
         reduce5<   4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case  2:
         reduce5<   2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      case  1:
         reduce5<   1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
      }
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x

Parallel Reduction Complexity

- Log(*N*) parallel steps, each step *S* does $N/2^S$ independent ops
  - Step Complexity is O(log *N*)
- For $N=2^D$, performs $\sum_{S\in[1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is O(*N*) – It is work-efficient
  - i.e. does not perform more operations than a sequential algorithm
- With *P* threads physically in parallel (*P* processors), time complexity is O(*N*/*P* + log *N*)
  - Compare to O(*N*) for sequential reduction
  - In a thread block, N=P, so O(log N)

What About *Cost?*

- *Cost* of a parallel algorithm is processors × time complexity
  - Allocate threads instead of processors: O(N) threads
  - Time complexity is O(log N), so *cost* is O(N log N) : not cost efficient!
- Brent's theorem suggests O(N/log N) threads
  - Each thread does O(log N) sequential work
  - Then all O(N/log N) threads cooperate for O(log N) steps
  - Cost = O((N/log N) * log N) = O(N)
- Known as *algorithm cascading*
  - Can lead to significant speedups in practice

Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Brent's theorem says each thread should sum O(log n) elements
  - i.e. 1024 or 2048 elements per block vs. 256
- In my experience, beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks
- On G80, best perf with 64-256 blocks of 128 threads
  - 1024-4096 elements per thread

Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

do {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
} while (i < n);
__syncthreads();
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x

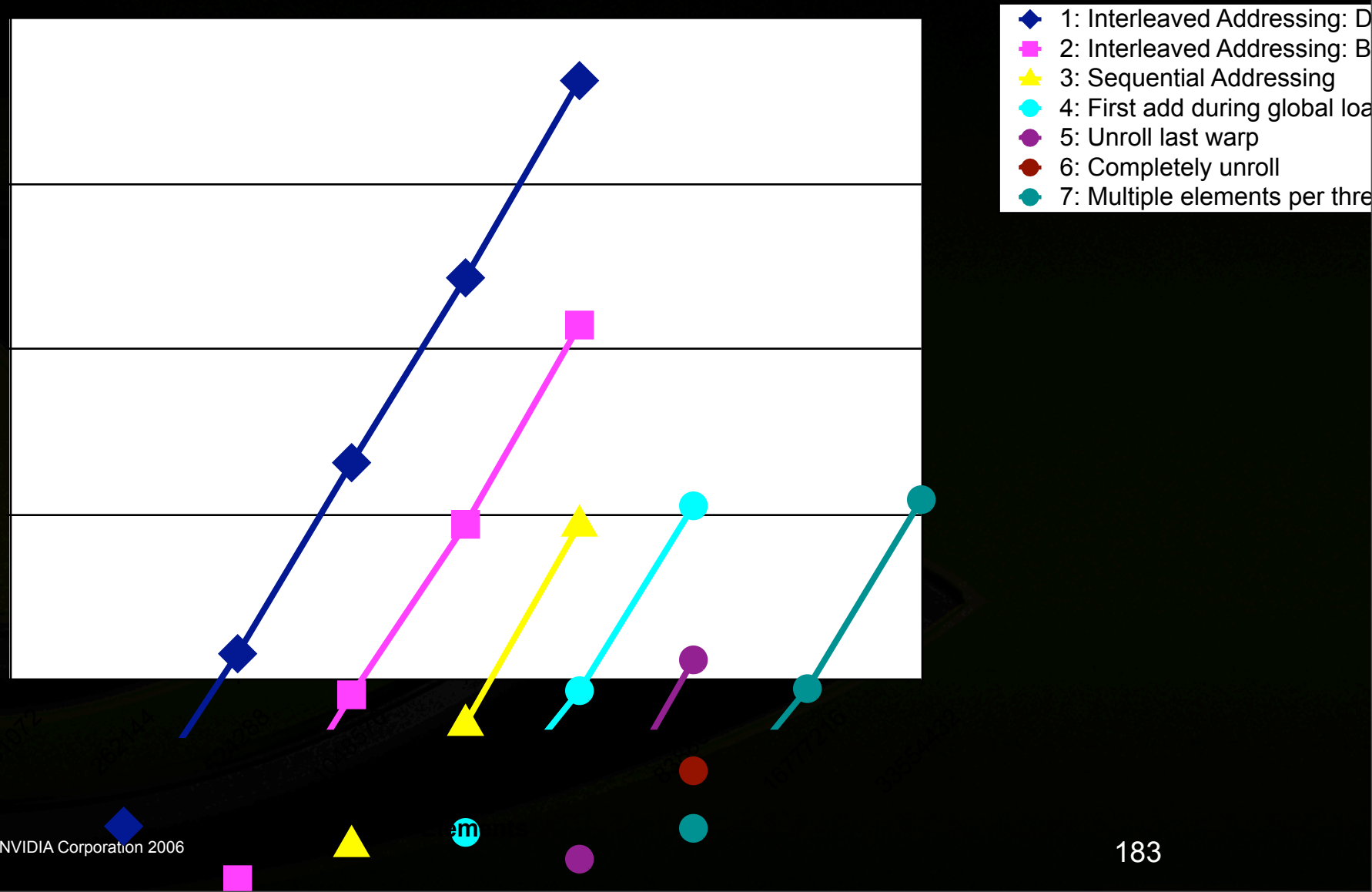
Kernel 7 on 32M elements: 72 GB/s!

Total Speedup: 30x!

Final Optimized Kernel

```
template <unsigned int [illegible]
__global__ void [illegible] int * [illegible] int * [illegible] unsigned int [illegible]

    extern __shared__ int [illegible]

    unsigned int [illegible] threadIdx.[illegible]
    unsigned int [illegible] blockIdx.[illegible] tid [illegible]
    unsigned int [illegible] blockSize [illegible] gridDim.[illegible]
    sdata[illegible]

    do { sdata[tid] += g[illegible] g_idata[i+blockSize]; i += gridSize; } while (i < n);
    [illegible]threads();

    if ([illegible] >= 512) { if (tid [illegible] sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if ([illegible] 256) { if (tid < 1[illegible] [illegible]] += sdata[tid + 128]; } __syncthreads(); }
    if (blo[illegible] 128) { if (tid < [illegible] [illegible]id] += sdata[tid +  64]; } __syncthreads(); }

    if (tid < 3[illegible] {
        if (blockSize >=  64) [illegible] sdata[tid + 32];
        if (blockSize >= [illegible] sd[illegible] += sdata[tid + [illegible]
        if (blockSize >= 16) [illegible]tid] += sdata[ti[illegible]
        if (blockSize >= [illegible] [illegible]ata[tid] += sda[illegible]
        if (blockSiz[illegible] sdata[tid] += sdata[ti[illegible]
        if (bl[illegible] 2) sdata[tid] += sd[illegible]
    [illegible]

    if (tid == 0) g_odata[blockIdx.[illegible]
```

Performance Comparison



Conclusion

- Understand CUDA performance characteristcs
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding
- Understand parallel algorithm complexity theory
- Use peak performance metrics to guide optimization
- Know how to identify type of bottleneck
  - e.g. memory, core computation, or instruction overhead
- Unroll loops
- Use template parameters to generate optimal code

NVIDIA®

CUDA 1.1 Preview

New Features

- Language improvements
- Asynchronous API
- Multi GPU interoperability support
- Windows XP 64 support
- CUDA driver integrated with display driver
- Preview of visual profiler

Language Improvements

- Subset of C++ supported
- `volatile` is now honored
- Loop unrolling with: `#pragma unroll`
- Inlining control with: `__noinline__`
- New intrinsics: `__[u]sad(a,b,s), __ffs[ll](x)`

Asynchronous API

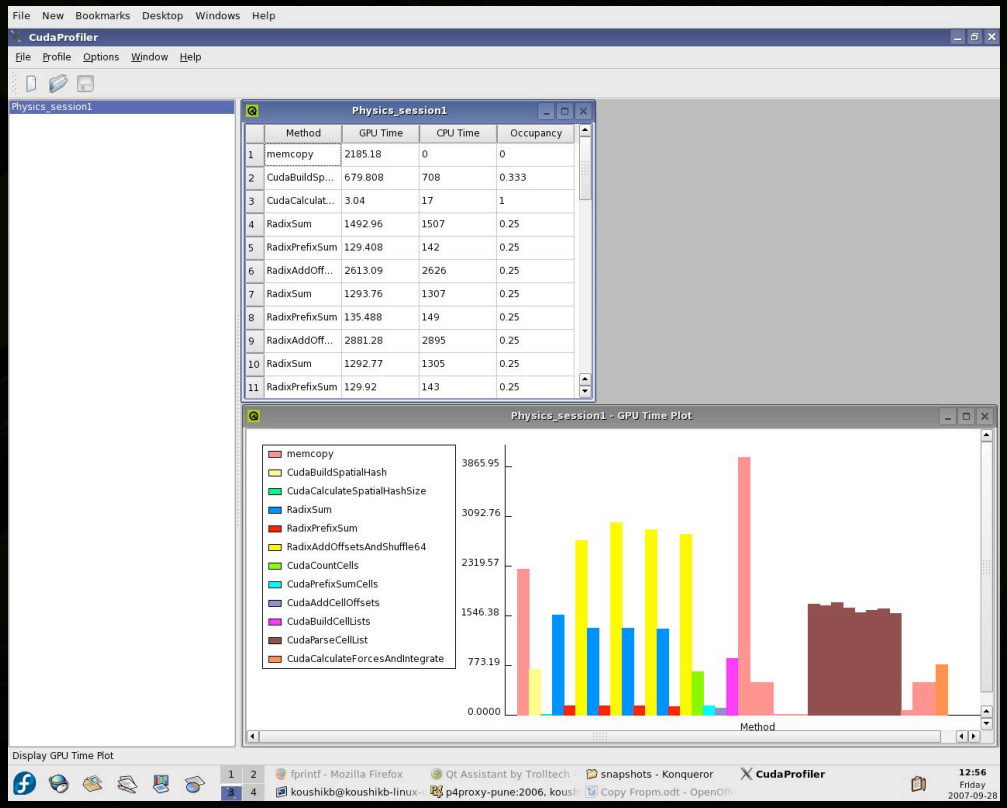
- Asynchronous host ←→ device memory copies for pinned memory
- Concurrent execution of kernels and memory copies (on compute capability >= 1.1)
- Only possible through stream abstraction
- Stream = Sequence of operations that execute in order
- Stream API:
  - `cudaStreamCreate(&stream)`
  - `cudaMemcpyAsync(src, dst, size, stream)`
  - `kernel<<<grid, block, shared, stream>>>(…)`
  - `cudaStreamQuery(stream)`
  - `cudaStreamSynchronize(stream)`

Asynchronous API

- Insert events in the stream to query whether preceding operations have finished
- Event API:
  - `cudaEventCreate(&event, flags)`
  - `cudaEventRecord(event, stream)`
  - `cudaEventQuery(event)`
  - `cudaEventSynchronize()`

Visual Profiler

- Alpha version for Linux and Windows
- Launch application with profiling enabled
- Gather timings of kernels and memory copies



Debugger

- GDB extended to switch between blocks and threads
- Can also access through a GUI (DDD – Data Display Debugger)
- Demo at Supercomputing next month

NVIDIA®

SDK

CUDA SDK

- Utilities
  - Bank checker
  - Timer
- Syntax highlighting for Visual Studio
- Samples
  - 37 sample projects in CUDA 1.0

SDK Samples

- Data alignment and copy bandwidth
  - alignedTypes
  - bandwidthTest
- Financial
  - binomialOptions
  - BlackScholes
  - MonteCarlo
- Image processing
  - boxFilter
  - convolutionFFT2D
  - convolutionSeparable
  - convolutionTexture
  - imageDenoising
  - postProcessGL
  - SobelFilter

SDK Samples

- Linear Algebra
  - matrixMul
  - matrixMulDrv
  - scalarprod
  - transpose
  - simpleCUBLAS
- Classic Parallel Algorithms
  - bitonic
  - scan
  - scanLargeArray
- CUDA libraries
  - simpleCUBLAS
  - simpleCUFFT

SDK Samples

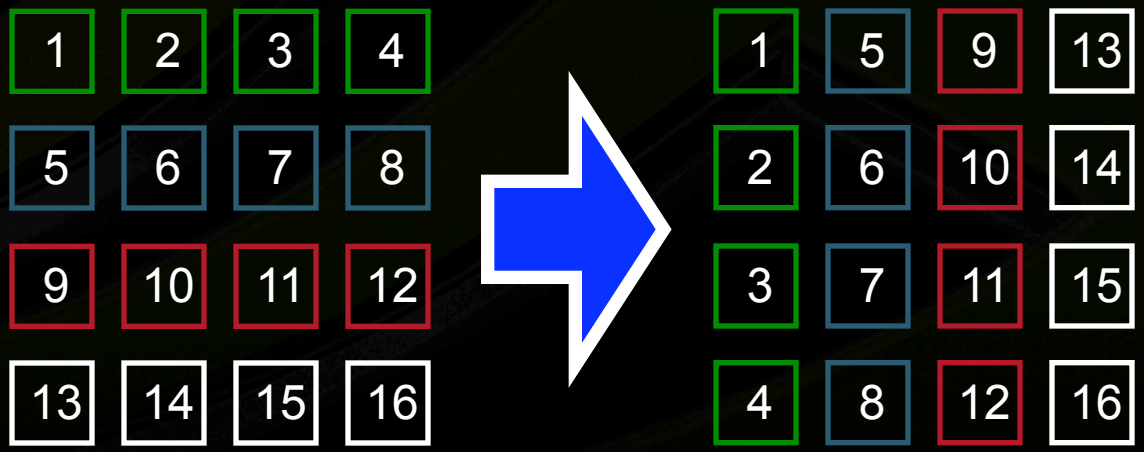
- Graphics Interop
  - fluidsD3D
  - fluidsGL
  - simpleD3D
  - simpleGL
  - simpleTexture/sipmleTextureDrv
- Other
  - dwtHaar1D
  - histogram64
  - multigpu
  - simpleTexture
  - simpleTextureDrv

NVIDIA®

Matrix Transpose

Matrix Transpose

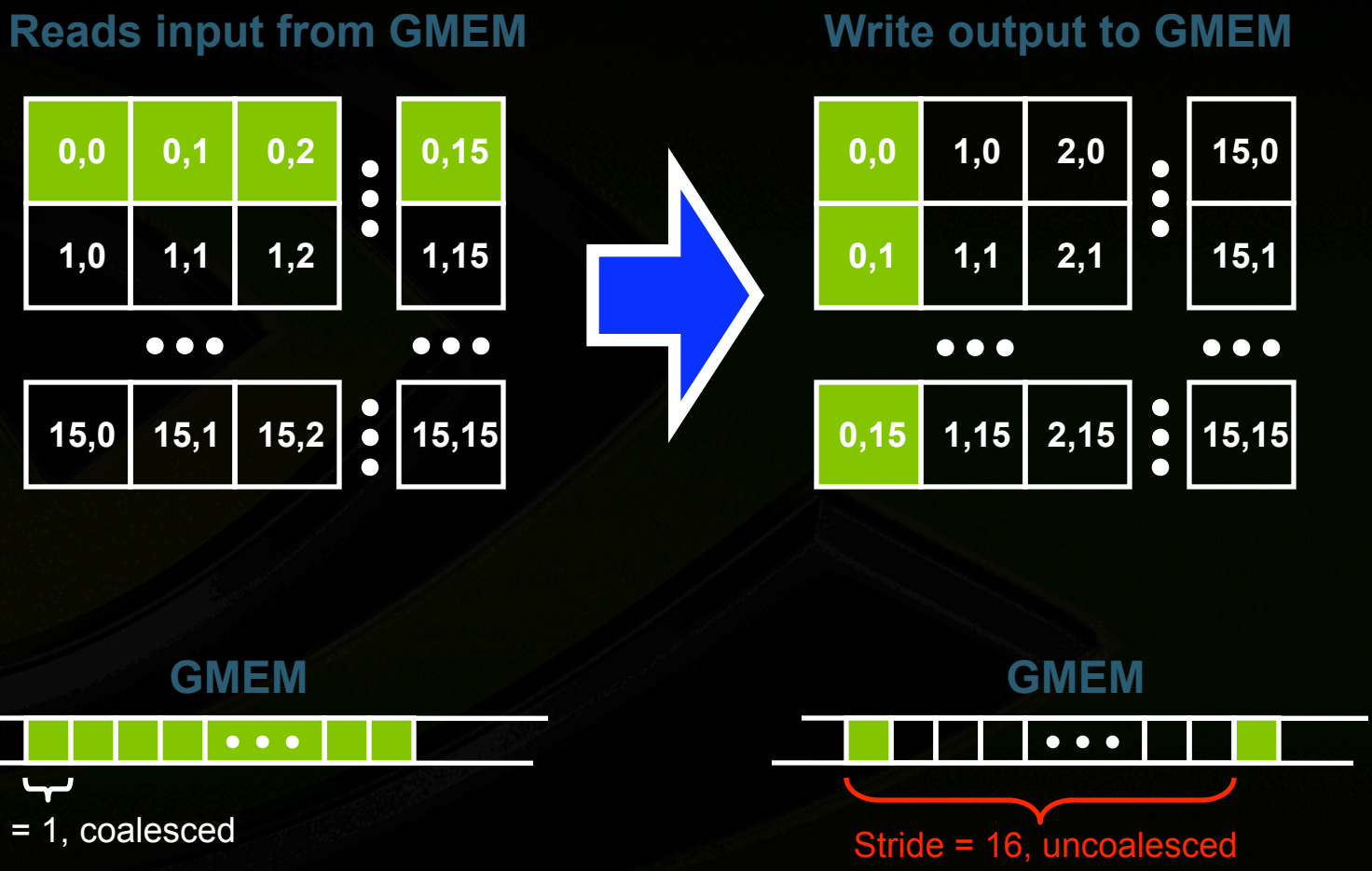
- SDK Sample (“transpose”)
- Illustrates:
  - Coalescing
  - Avoiding SMEM bank conflicts
  - Speedups for even small matrices



Uncoalesced Transpose

```
__global__ void transpose_naive(float *odata, float *idata, int width, int height)
{
1.  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
2.  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

3.  if (xIndex < width && yIndex < height)
    {
4.      unsigned int index_in  = xIndex + width * yIndex;
5.      unsigned int index_out = yIndex + height * xIndex;
6.      odata[index_out] = idata[index_in];
    }
}
```

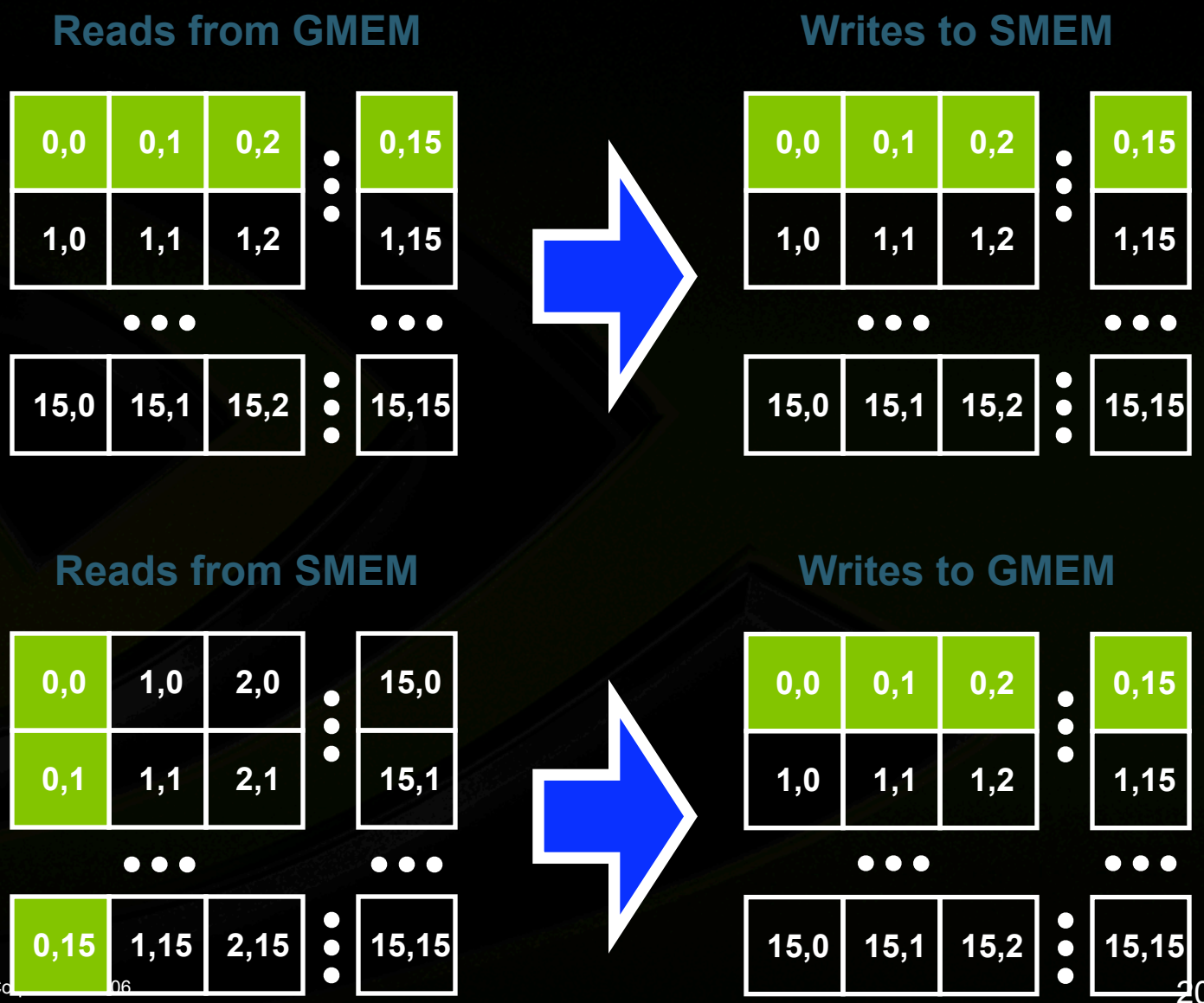
Uncoalesced Transpose



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



SMEM Optimization

Reads from SMEM

- Threads read SMEM with stride = 16
  - Bank conflicts

- Solution
  - Allocate an “extra” column
  - Read stride = 17
  - Threads read from consecutive banks

Coalesced Transpose

```
__global__ void transpose_exp(float *odata, float *idata, int width, int height)
{
1.  __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

2.  unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
3.  unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
4.  if( (xIndex < width)&&(yIndex < height) )
    {
5.      unsigned int index_in = xIndex + yIndex * width;
6.      block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

7.  __syncthreads();

8.  xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
9.  yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
10. if( (xIndex < height)&&(yIndex < width) )
    {
11.     unsigned int index_out = yIndex * height + xIndex;
12.     odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```