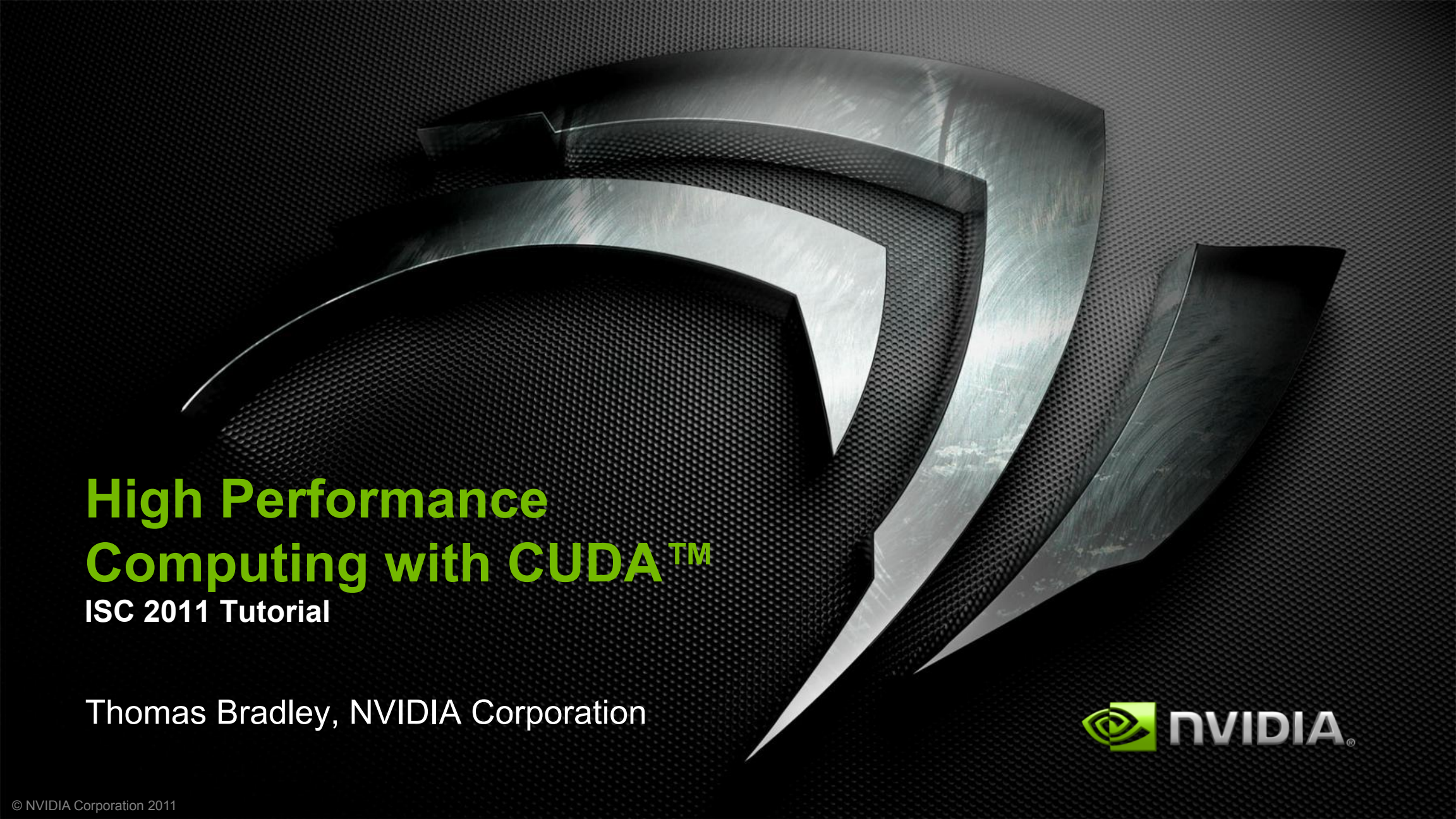


© NVIDIA Corporation 2011

Welcome

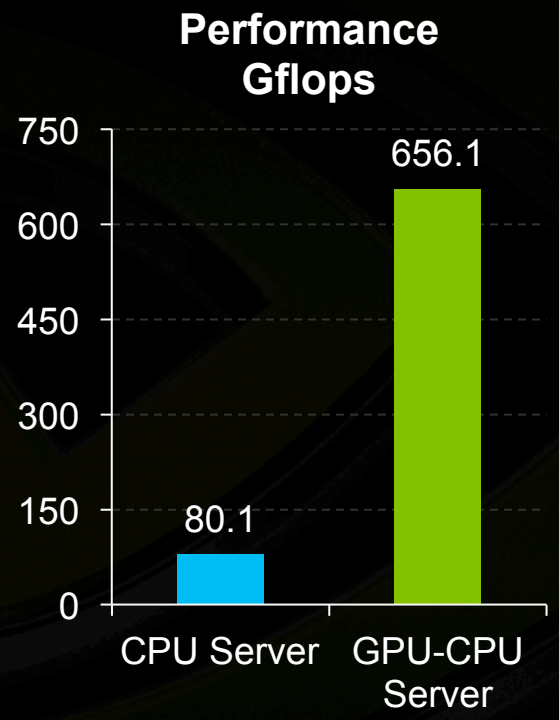
- Goals:
  - An introduction to High Performance Computing with CUDA

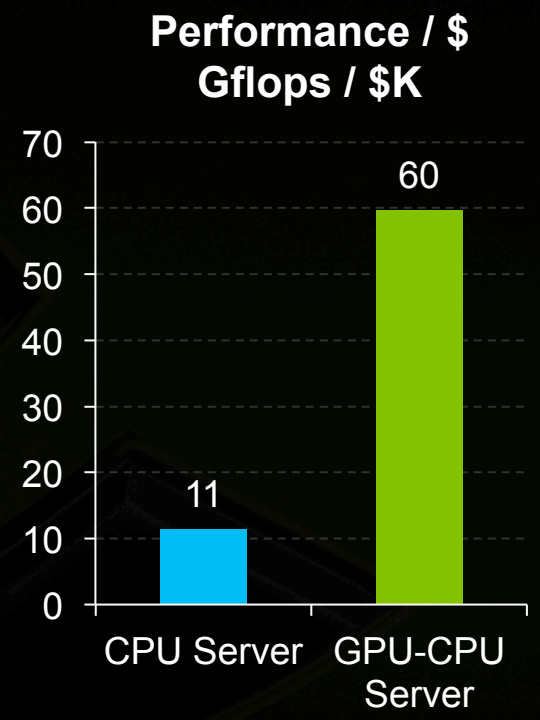
- Outline
  - Motivation and introduction
  - CUDA C/C++ basics
  - Analysis and optimization
  - CUDA Fortran and CUDA libraries
  - Lessons learned in production codes

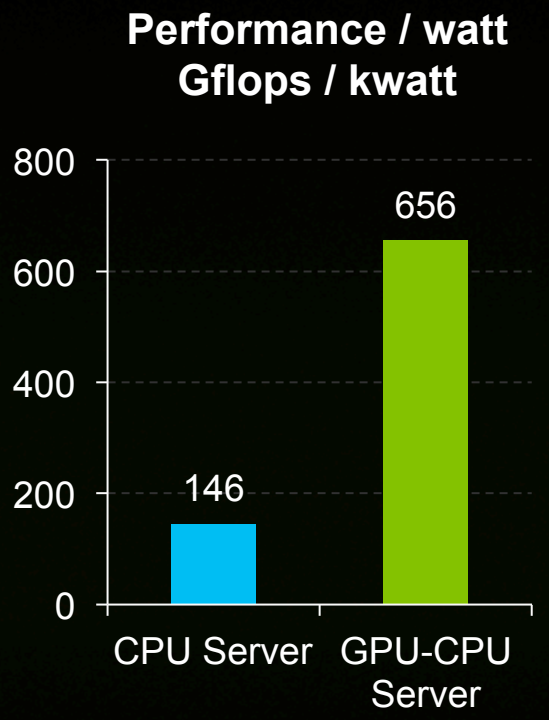
© NVIDIA Corporation 2011

GPUs are Fast!

8x Higher Linpack







CPU 1U Server: 2x Intel Xeon X5550 (Nehalem) 2.66 GHz, 48 GB memory, $7K, 0.55 kw

GPU-CPU 1U Server: 2x Tesla C2050 + 2x Intel Xeon X5550, 48 GB memory, $11K, 1.0 kw

© NVIDIA Corporation 2011

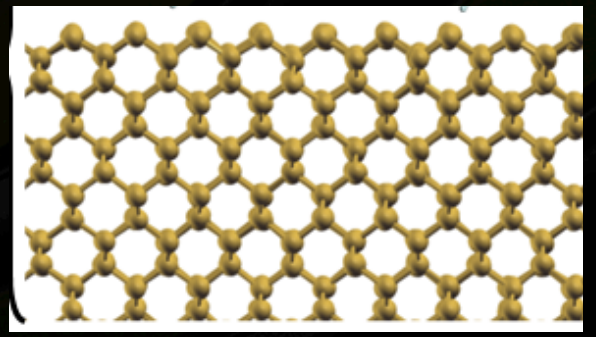
World's Fastest MD Simulation

Sustained Performance of 1.87 Petaflops/s

Institute of Process Engineering (IPE)

Chinese Academy of Sciences (CAS)

MD Simulation for Crystalline Silicon



Used all 7168 Tesla GPUs on
Tianhe-1A GPU Supercomputer

© NVIDIA Corporation 2011

Tesla GPUs Power 3 of Top 5 Supercomputers

#1 : Tianhe-1A
7168 Tesla GPU's 2.5 PFLOPS

#3 : Nebulae
4650 Tesla GPU's 1.2 PFLOPS

#4 : Tsubame 2.0
4224 Tesla GPU's 1.194 PFLOPS

"We not only created the world's fastest computer, but also implemented a heterogeneous computing architecture incorporating CPU and GPU, this is a new innovation."

Premier Wen Jiabao
Public comments acknowledging Tianhe-1A

World's Greenest Petaflop Supercomputer

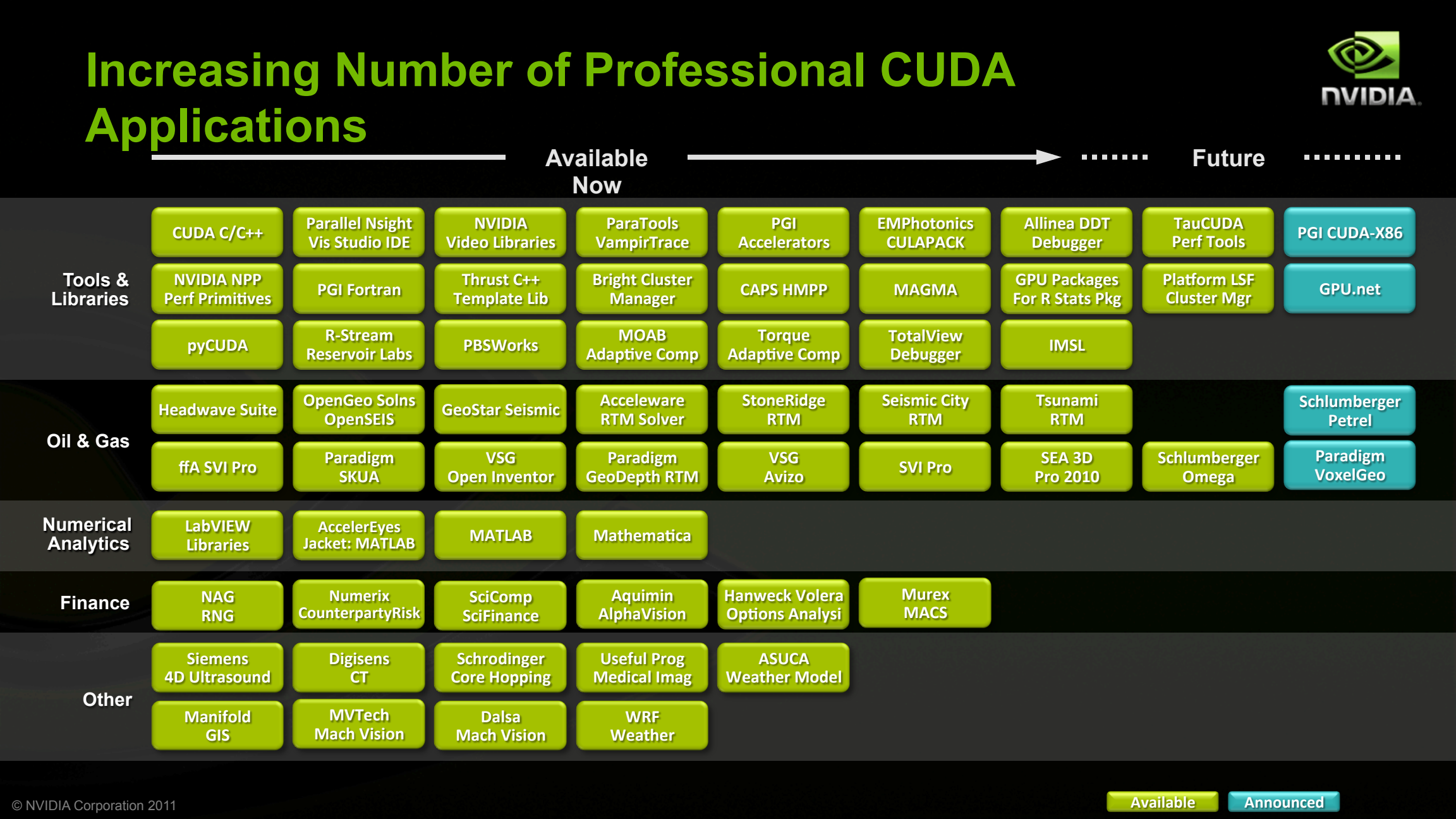
Tsubame 2.0

Tokyo Institute of Technology

- 1.19 Petaflops
- 4,224 Tesla M2050 GPUs

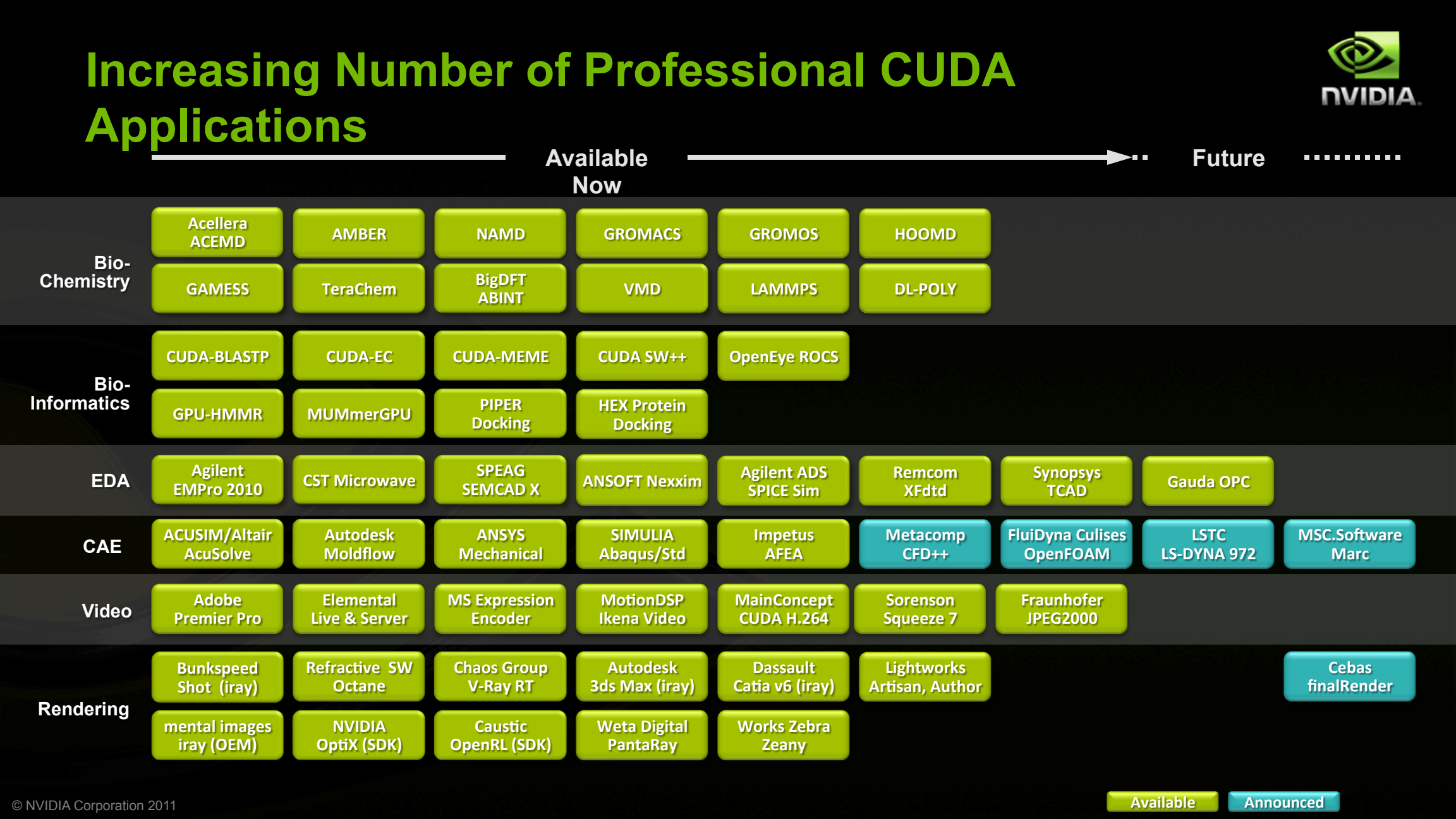
© NVIDIA Corporation 2011

Increasing Number of Professional CUDA Applications



© NVIDIA Corporation 2011

Increasing Number of Professional CUDA Applications



© NVIDIA Corporation 2011

CUDA by the Numbers

300,000,000	CUDA Capable GPUs
500,000	CUDA Toolkit Downloads
100,000	Active CUDA Developers
400	Universities Teaching CUDA
100	% OEMs offer CUDA GPU PCs

© NVIDIA Corporation 2011

GPU Computing Applications

Libraries & Middleware

CUBLAS	CUFFT	CULAPACK	NPP & CUDPP	Video	PhysX Physics	OptiX Ray tracing	mental ray iray Rendering	Reality Server 3D web services

C	C++	OpenCL™	Direct Compute	Fortran	Java & Python

NVIDIA GPU with CUDA Parallel Computing Architecture

Fermi architecture (compute capability 2.x)	GeForce 500 series GeForce 400 series	Quadro Fermi series	Tesla 20 series
Tesla architecture (compute capability 1.x)	GeForce 200 series GeForce 9 series GeForce 8 series	Quadro FX series QuadroPlex series Quadro NVS series	Tesla 10 series

© NVIDIA Corporation 2011

OpenCL is trademark of Apple Inc. used under license to the Khronos Group Inc.

Tesla Data Center & Workstation GPU Solutions

Tesla M-series GPUs
M2090 | M2070 | M2050

Servers & Blades

Tesla C-series GPUs
C2070 | C2050

Workstations

		M2090	M2070	M2050	C2070	C2050
Cores		512	448	448	448	448
Memory		6 GB	6 GB	3 GB	6 GB	3 GB
Memory bandwidth (ECC off)		177.6 GB/s	150 GB/s	148.8 GB/s	148.8 GB/s	148.8 GB/s
Peak Perf Gflops	Single Precision	1331	1030	1030	1030	1030
	Double Precision	665	515	515	515	515

© NVIDIA Corporation 2011

NVIDIA Developer Ecosystem

NVIDIA

Numerical Packages

MATLAB
Mathematica
NI LabView
pyCUDA

Debuggers & Profilers

cuda-gdb
NV Visual Profiler
Parallel Nsight
Visual Studio
Allinea
TotalView

GPU Compilers

C
C++
Fortran
OpenCL
DirectCompute
Java
Python

Parallelizing Compilers

PGI Accelerator
CAPS HMPP
mCUDA
OpenMP

Libraries

BLAS
FFT
LAPACK
NPP
Video
Imaging
GPULib

GPGPU Consultants & Training

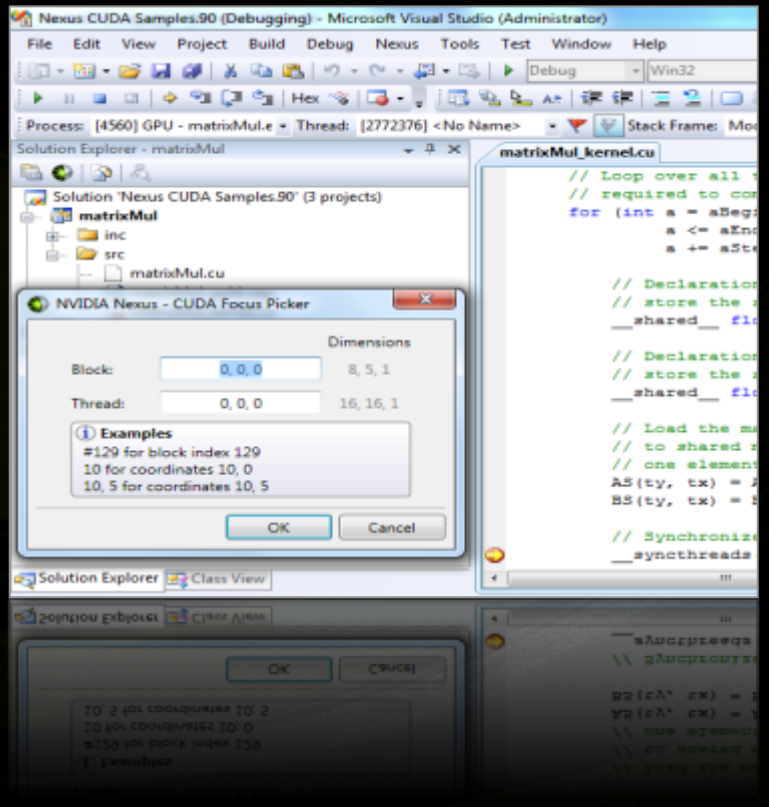
acceleware · GASS · STONE RIDGE TECHNOLOGY · ANEO · GPU Tech · TECH-X · CAPS · HPC · EM Photonics · SCALABLE GRAPHICS · WIPRO

OEM Solution Providers

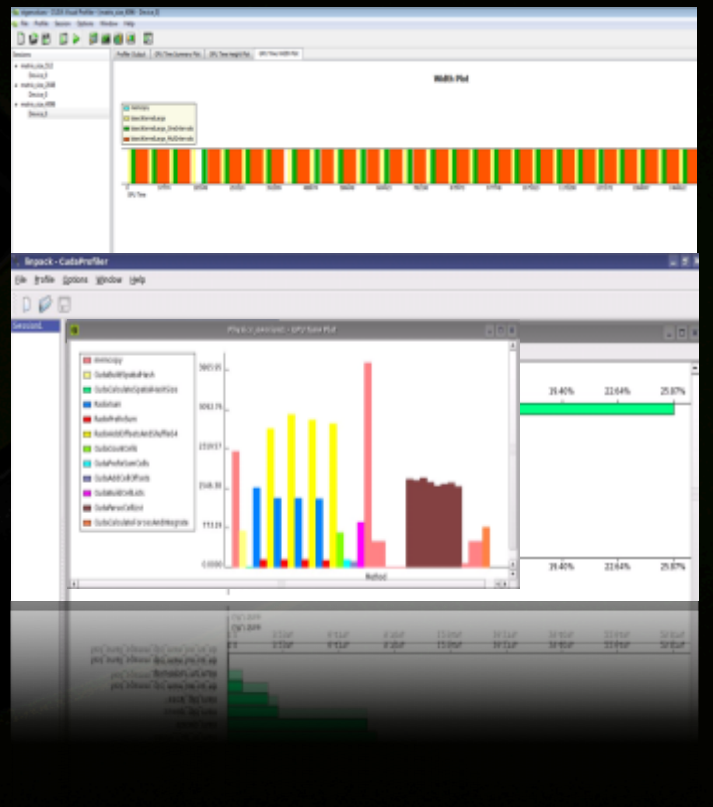
DELL · hp · IBM · CRAY · ASUS · SUPERMICRO · sgi · FUJITSU · BULL · APPRO · lenovo 联想 · NEC

© NVIDIA Corporation 2011

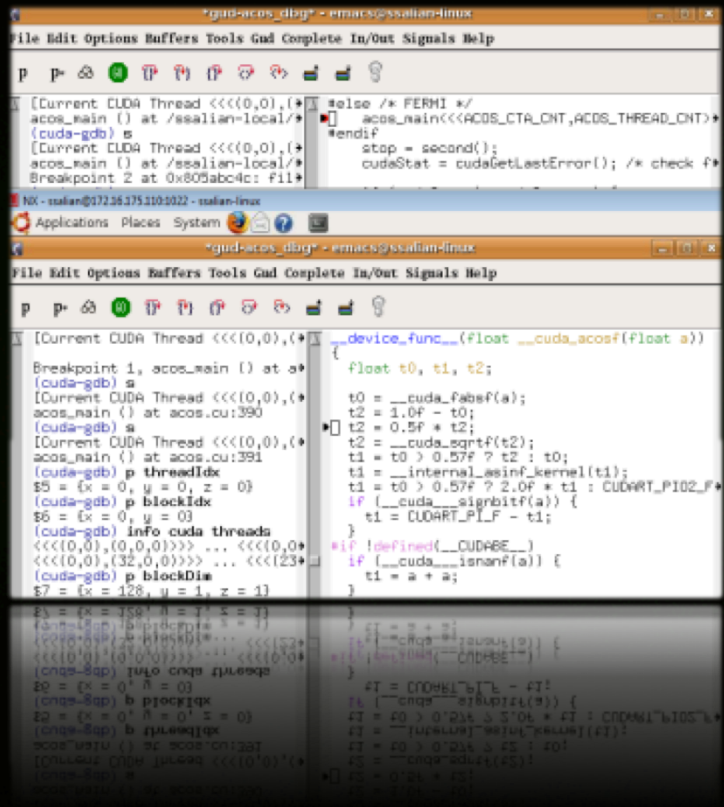
Parallel Nsight
Visual Studio



Visual Profiler
Windows/Linux



cuda-gdb
Linux/Mac



© NVIDIA Corporation 2011

Schedule

0900	Introduction
0915	CUDA C/C++ Basics Thomas Bradley, NVIDIA
1015	Break
1045	CUDA Libraries and CUDA Fortran Massimilano Fatica, NVIDIA
1145	Analysis and Optimization part 1 Tim Schröder, NVIDIA
1300	Lunch

Beginner

Beginner
Intermediate

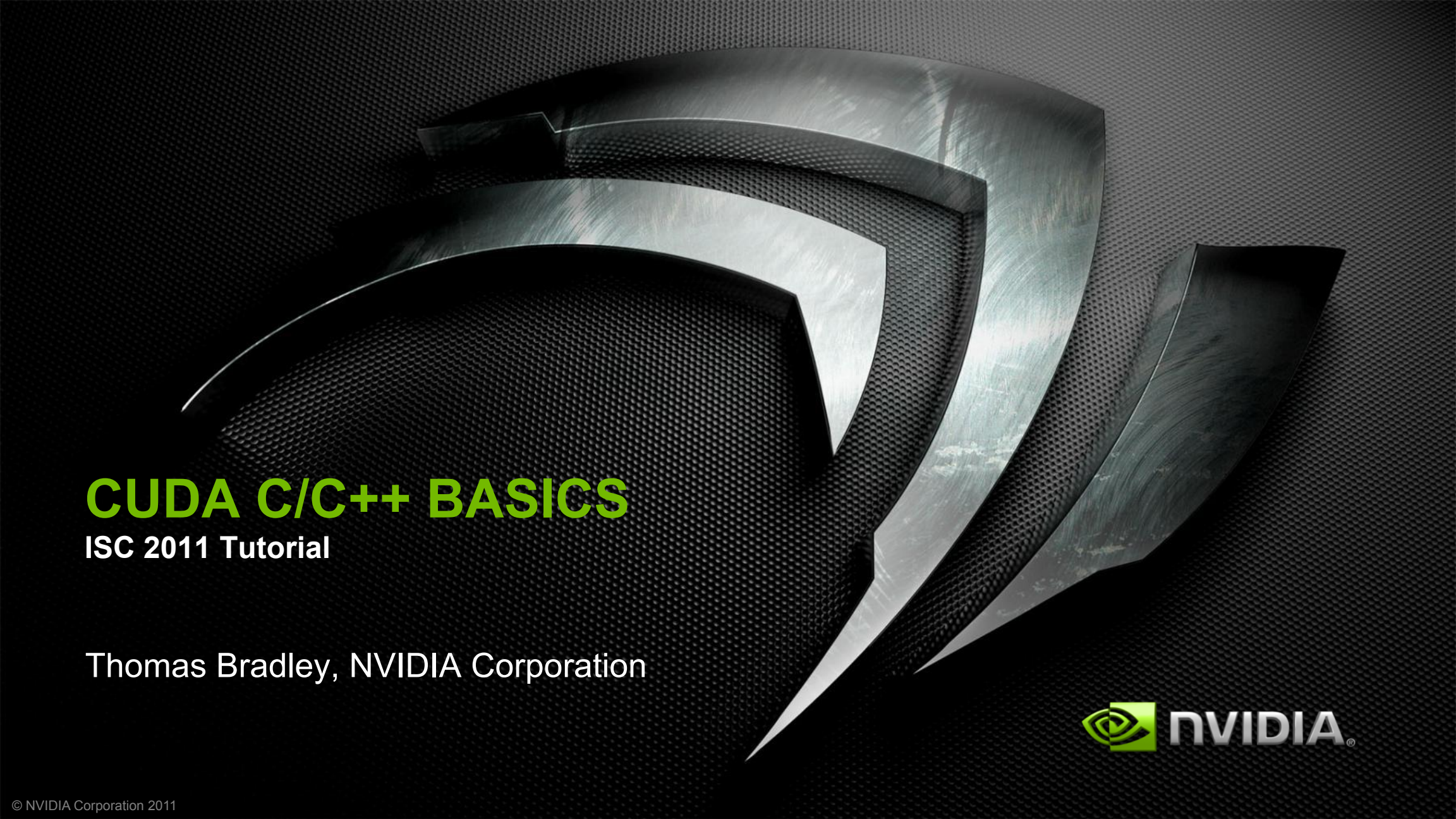
Intermediate
Advanced

© NVIDIA Corporation 2011

Schedule

1400	Analysis and Optimization part 2 Gernot Ziegler, NVIDIA
1515	Break
1545	Optimising Stencils for Finite Volume CFD Tobias Brandvik
1615	The Texture Unit as a Performance Booster in 3D Volume Reconstruction Karl Schwarz
1645	Optimisation Myths and Facts as Seen in Statistical Physics Massimo Bernaschi
1715	Putting Branching to Work in Real-time Visualization of Medical Images Erik Steen
1745	Close

© NVIDIA Corporation 2011



© NVIDIA Corporation 2011

What is CUDA?

- CUDA Architecture
  - Expose GPU computing for general purpose
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

© NVIDIA Corporation 2011

Introduction to CUDA C/C++

- What will you learn in this session?
  - Start from “Hello World!”
  - Write and launch CUDA C/C++ kernels
  - Manage GPU memory
  - Manage communication and synchronization

© NVIDIA Corporation 2011

Prerequisites

- You (probably) need experience with C or C++
- You don’t need GPU experience
- You don’t need parallel programming experience
- You don’t need graphics experience

© NVIDIA Corporation 2011

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

© NVIDIA Corporation 2011

HELLO WORLD!

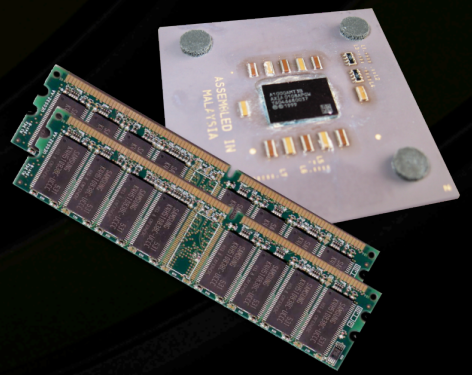
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

© NVIDIA Corporation 2011

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

© NVIDIA Corporation 2011

Heterogeneous Computing

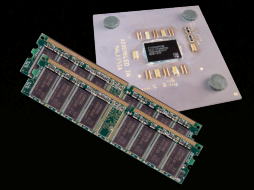
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

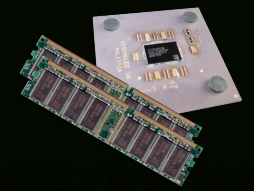
parallel fn

serial code

parallel code

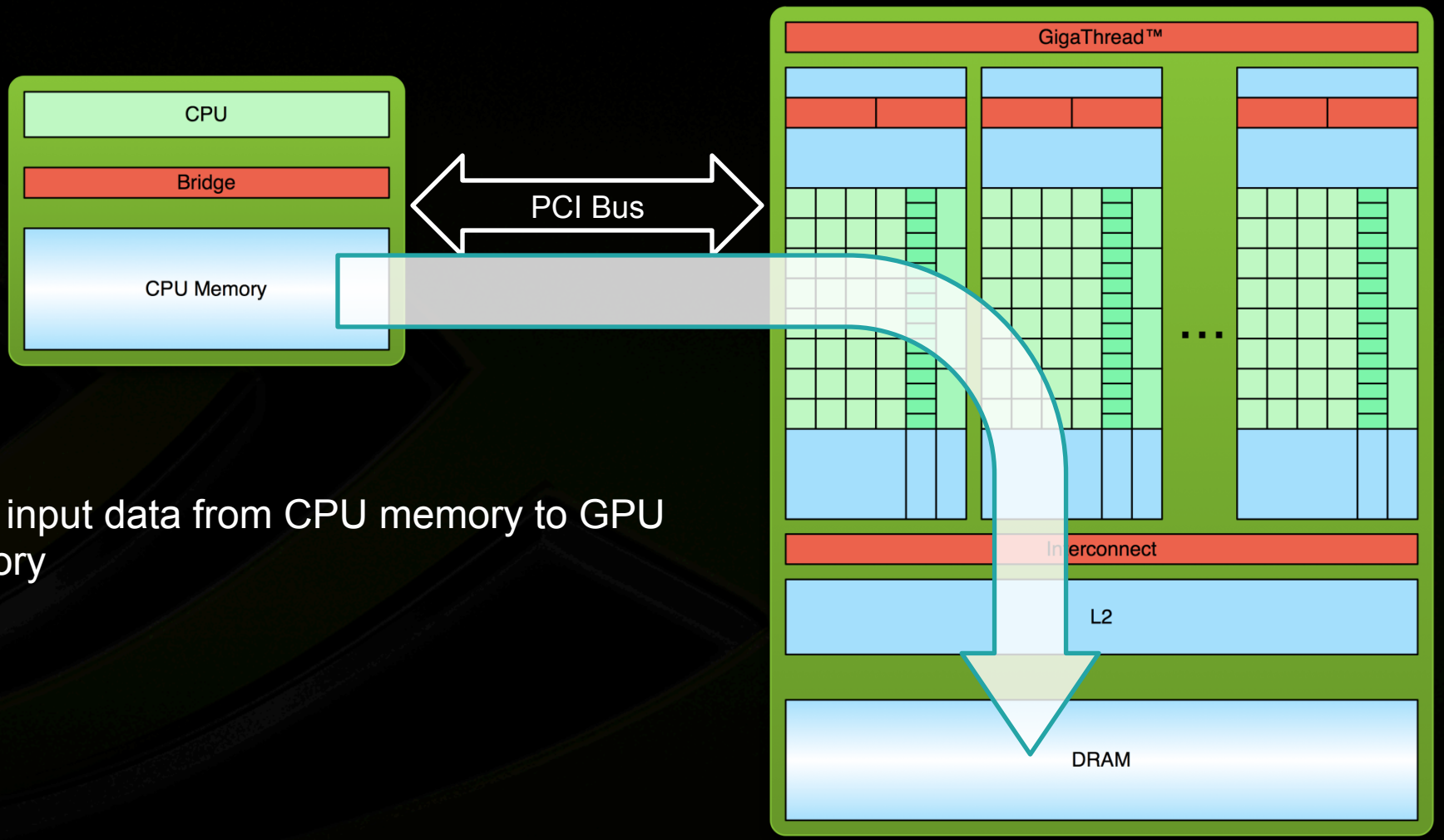
serial code





© NVIDIA Corporation 2011

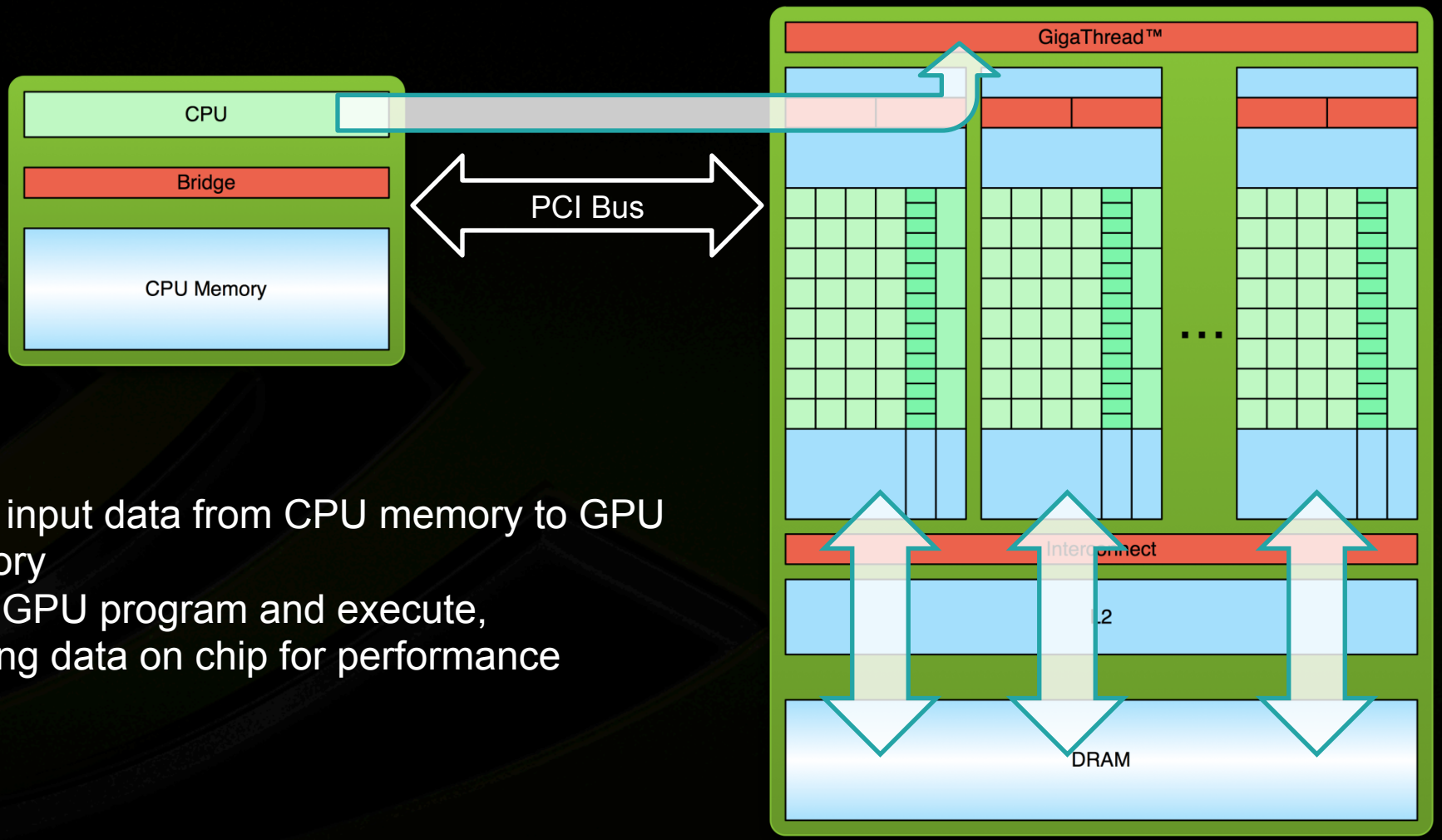
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory

© NVIDIA Corporation 2011

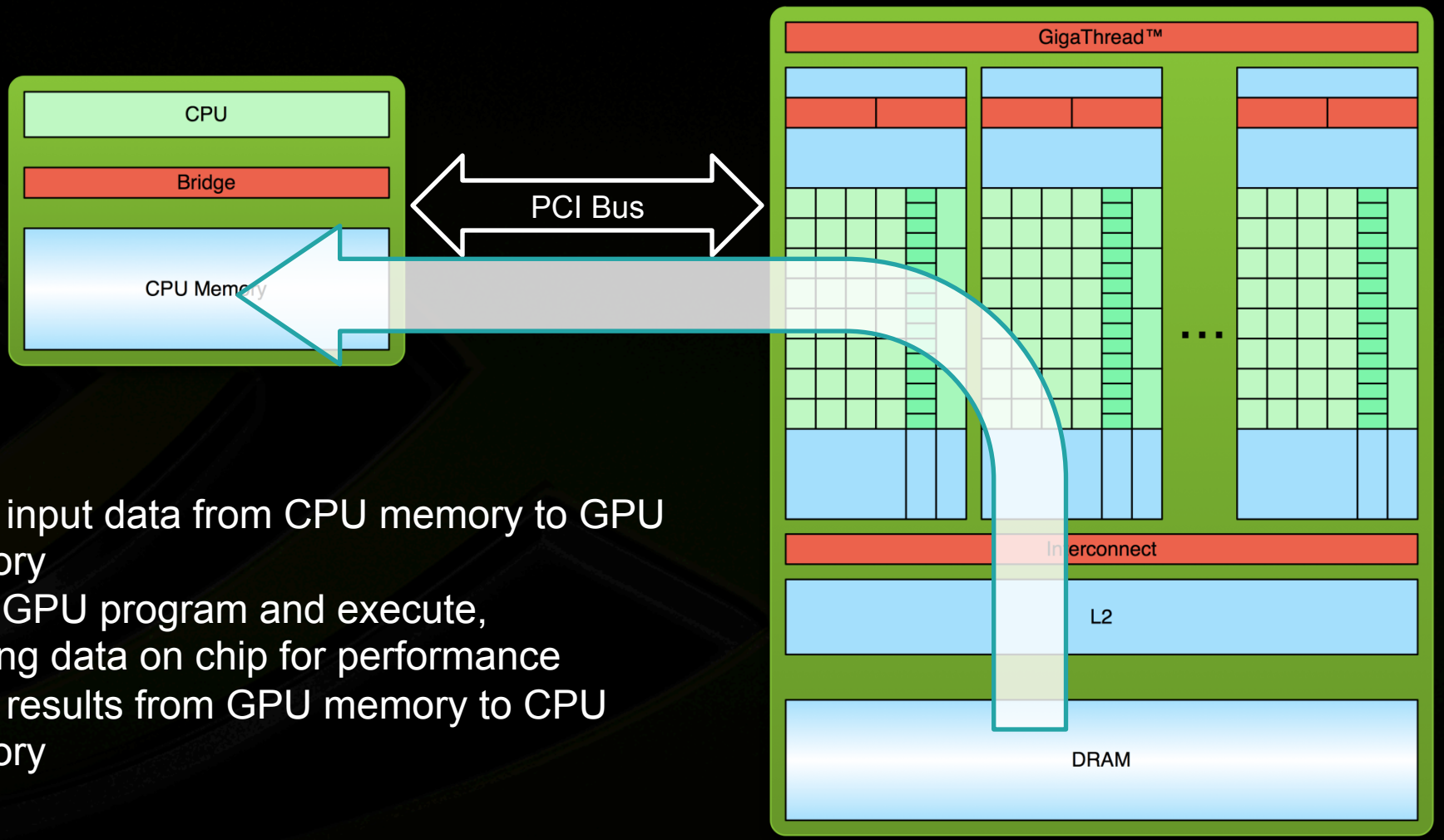
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

© NVIDIA Corporation 2011

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

© NVIDIA Corporation 2011

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc
hello_world.cu
$ a.out
Hello World!
$
```

© NVIDIA Corporation 2011

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

- Two new syntactic elements…

© NVIDIA Corporation 2011

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`

© NVIDIA Corporation 2011

Hello World! with Device COde

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

© NVIDIA Corporation 2011

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

Output:

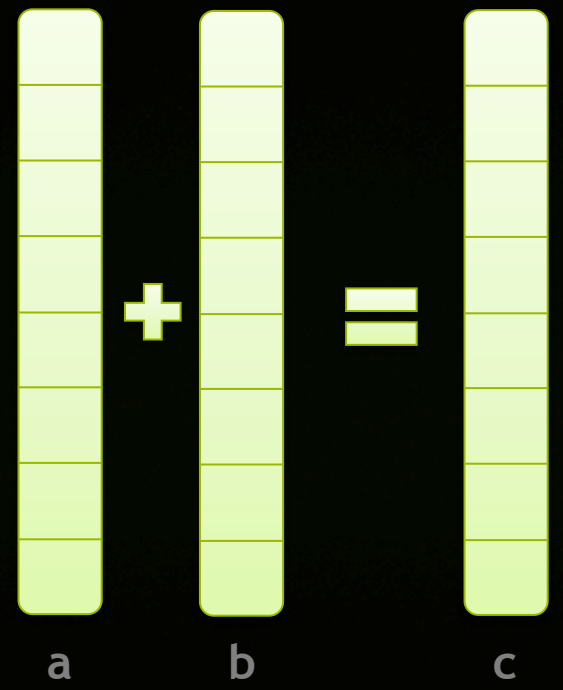
```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- `mykernel()` does nothing, somewhat anticlimactic!

© NVIDIA Corporation 2011

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We'll start by adding two integers and build up to vector addition



© NVIDIA Corporation 2011

Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

© NVIDIA Corporation 2011

Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

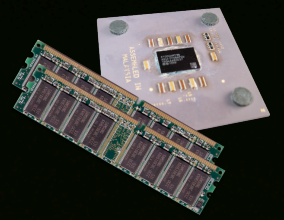
- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

© NVIDIA Corporation 2011

Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

© NVIDIA Corporation 2011

Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- Let's take a look at main()…

© NVIDIA Corporation 2011

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

© NVIDIA Corporation 2011

Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

© NVIDIA Corporation 2011

RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- **Blocks**
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

© NVIDIA Corporation 2011

Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

© NVIDIA Corporation 2011

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

© NVIDIA Corporation 2011

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0]  = a[0] + b[0];
```

Block 1

```
c[1]  = a[1] + b[1];
```

Block 2

```
c[2]  = a[2] + b[2];
```

Block 3

```
c[3]  = a[3] + b[3];
```

© NVIDIA Corporation 2011

Vector Addition on the Device: `add()`

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let's take a look at main()…

© NVIDIA Corporation 2011

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
        int *a, *b, *c;                 // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

© NVIDIA Corporation 2011

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

© NVIDIA Corporation 2011

Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host

- Passing parameters from host code to a device function

© NVIDIA Corporation 2011

Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

© NVIDIA Corporation 2011

INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- **Threads**
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

© NVIDIA Corporation 2011

CUDA Threads

- Terminology: a block can be split into parallel threads
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()`…

© NVIDIA Corporation 2011

Vector Addition Using Threads: `main()`

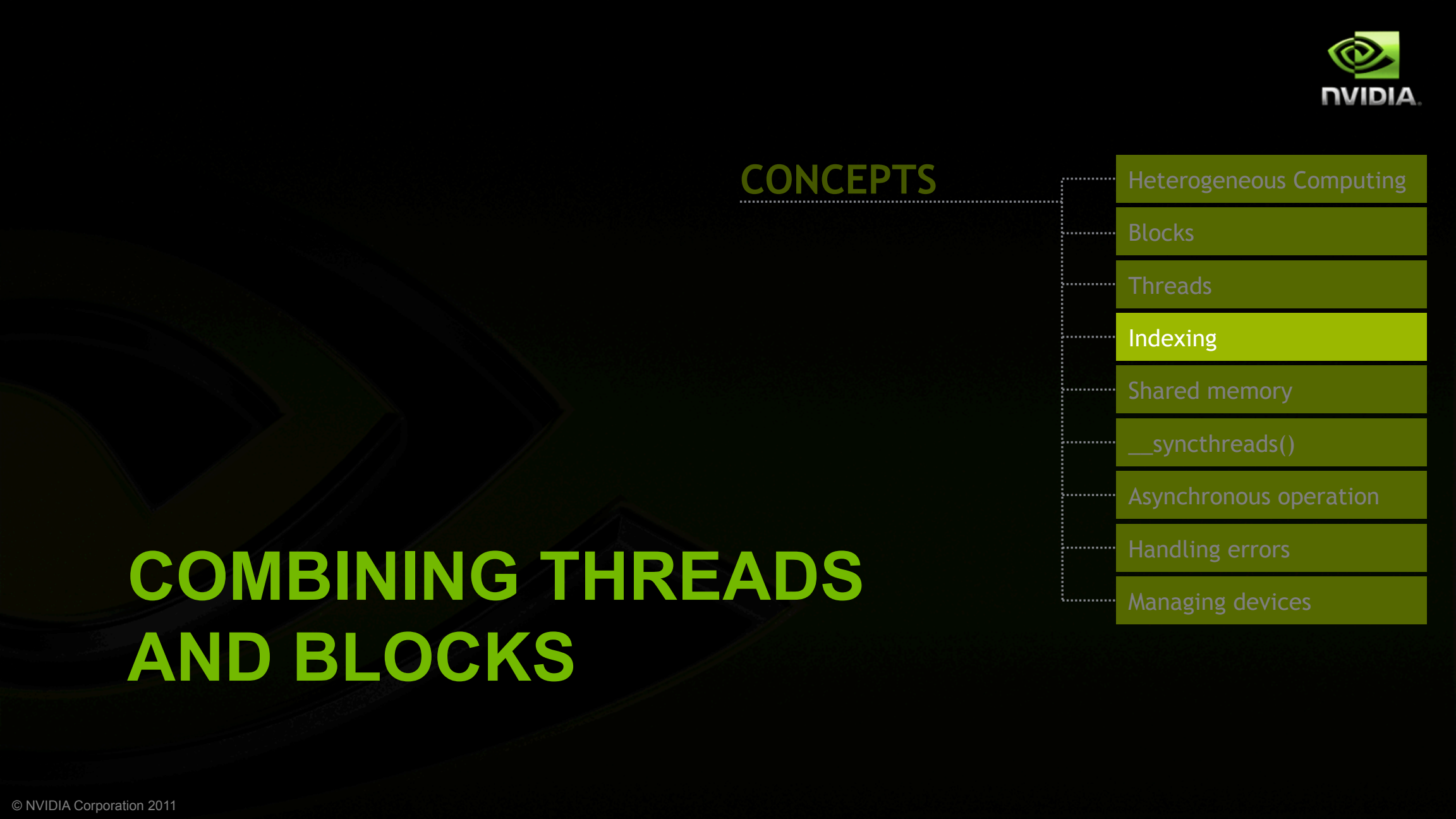
```
#define N 512
int main(void) {
        int *a, *b, *c;                 // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

© NVIDIA Corporation 2011

Vector Addition Using Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

© NVIDIA Corporation 2011



COMBINING THREADS AND BLOCKS

© NVIDIA Corporation 2011

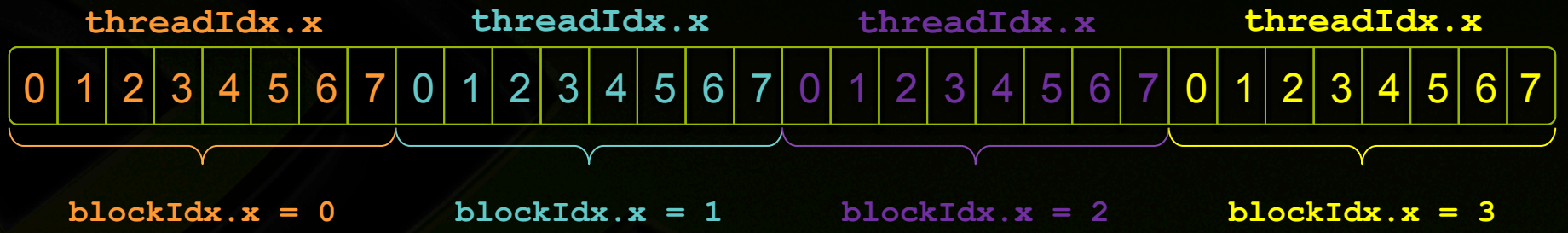
Combining Blocks *and* Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both *blocks* and *threads*
- Why? We'll come to that…
- First let's discuss data indexing…

© NVIDIA Corporation 2011

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)



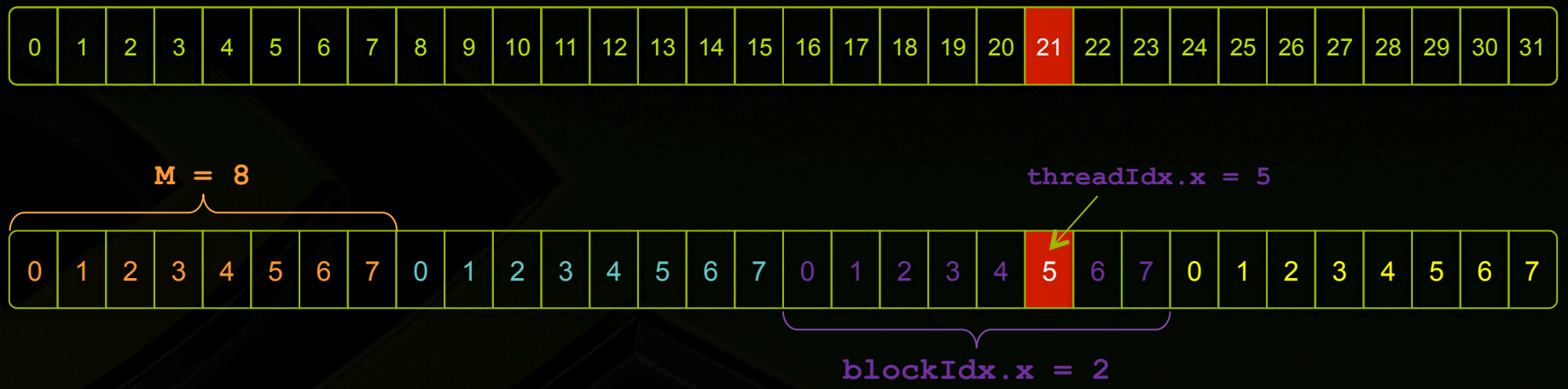
- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

© NVIDIA Corporation 2011

Indexing Arrays: Example

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2      * 8;
          = 21;
```

© NVIDIA Corporation 2011

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?

© NVIDIA Corporation 2011

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
        int *a, *b, *c;                 // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

© NVIDIA Corporation 2011

Addition with Blocks and Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

© NVIDIA Corporation 2011

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        if (index < n)
            c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

© NVIDIA Corporation 2011

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example…

© NVIDIA Corporation 2011

Review

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N/M,M>>>(…);`
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

© NVIDIA Corporation 2011

COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- **Shared memory**
- **__syncthreads()**
- Asynchronous operation
- Handling errors
- Managing devices

© NVIDIA Corporation 2011

1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

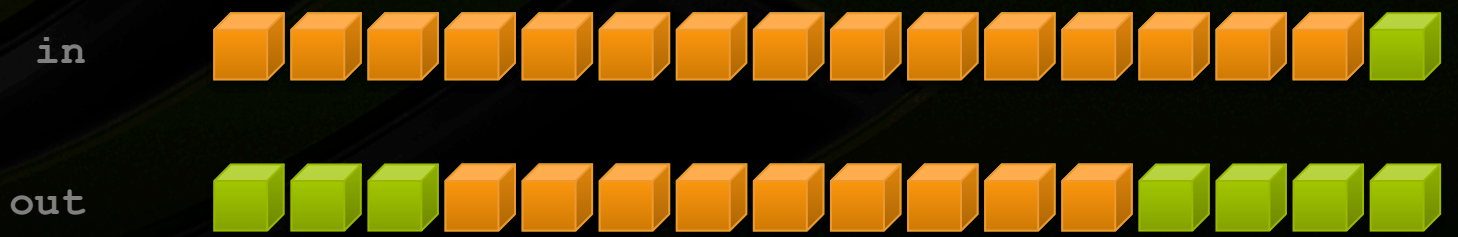
- If radius is 3, then each output element is the sum of 7 input elements:

© NVIDIA Corporation 2011

Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times



© NVIDIA Corporation 2011

Sharing Data Between Threads

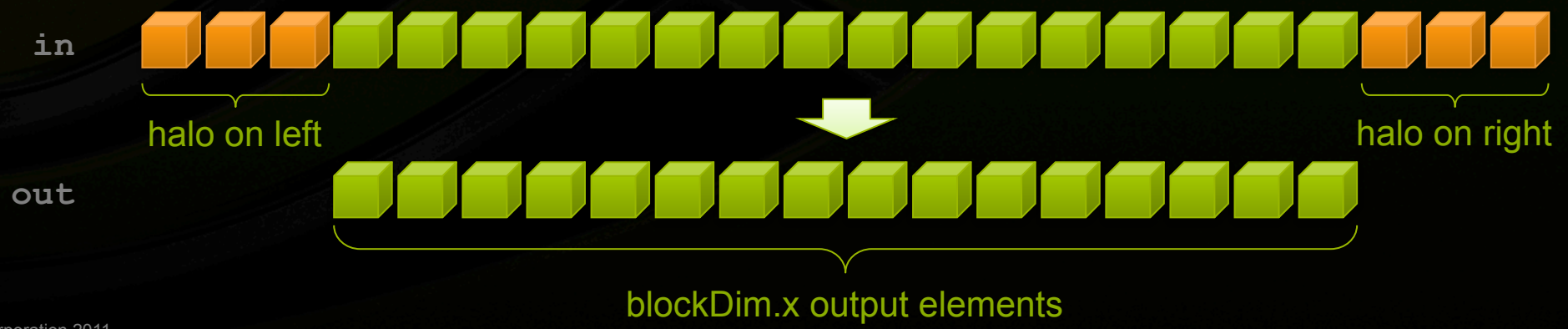
- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

© NVIDIA Corporation 2011

Implementing With Shared Memory

- Cache data in shared memory
  - Read `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory

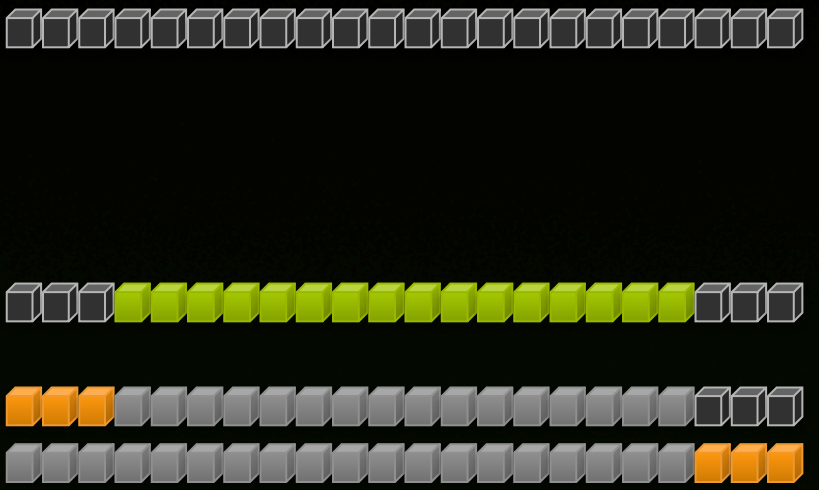
- Each block needs a halo of `radius` elements at each boundary



© NVIDIA Corporation 2011

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
```



© NVIDIA Corporation 2011

Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

© NVIDIA Corporation 2011

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

© NVIDIA Corporation 2011

__syncthreads()

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

© NVIDIA Corporation 2011

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
```

© NVIDIA Corporation 2011

Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

© NVIDIA Corporation 2011

Review (1 of 2)

- Launching parallel threads
  - Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block

- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

© NVIDIA Corporation 2011

Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

© NVIDIA Corporation 2011

MANAGING THE DEVICE

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

© NVIDIA Corporation 2011

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

© NVIDIA Corporation 2011

Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself

    OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

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

© NVIDIA Corporation 2011

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device

- A single thread can manage multiple devices

  `cudaSetDevice(i)` to select current device

  `cudaMemcpy(…)` for peer-to-peer copies†

† requires OS and device support

© NVIDIA Corporation 2011

Introduction to CUDA C/C++

- What have we learnt?
  - Write and launch CUDA C/C++ kernels
    - `__global__`, `blockIdx.x`, `threadIdx.x`, `<<<>>>`
  - Manage GPU memory
    - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`
  - Manage communication and synchronization
    - `__shared__`, `__syncthreads()`
    - `cudaMemcpy()` **VS** `cudaMemcpyAsync()`, `cudaDeviceSynchronize()`

© NVIDIA Corporation 2011

Topics we skipped

- We skipped some details, you can learn more:
  - CUDA Programming Guide
  - CUDA Zone – tools, training, webinars and more

    http://developer.nvidia.com/cuda

- Need a quick primer for later:
  - Compute capability
  - Multi-dimensional indexing
  - Textures

© NVIDIA Corporation 2011

Compute Capability

- The compute capability of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

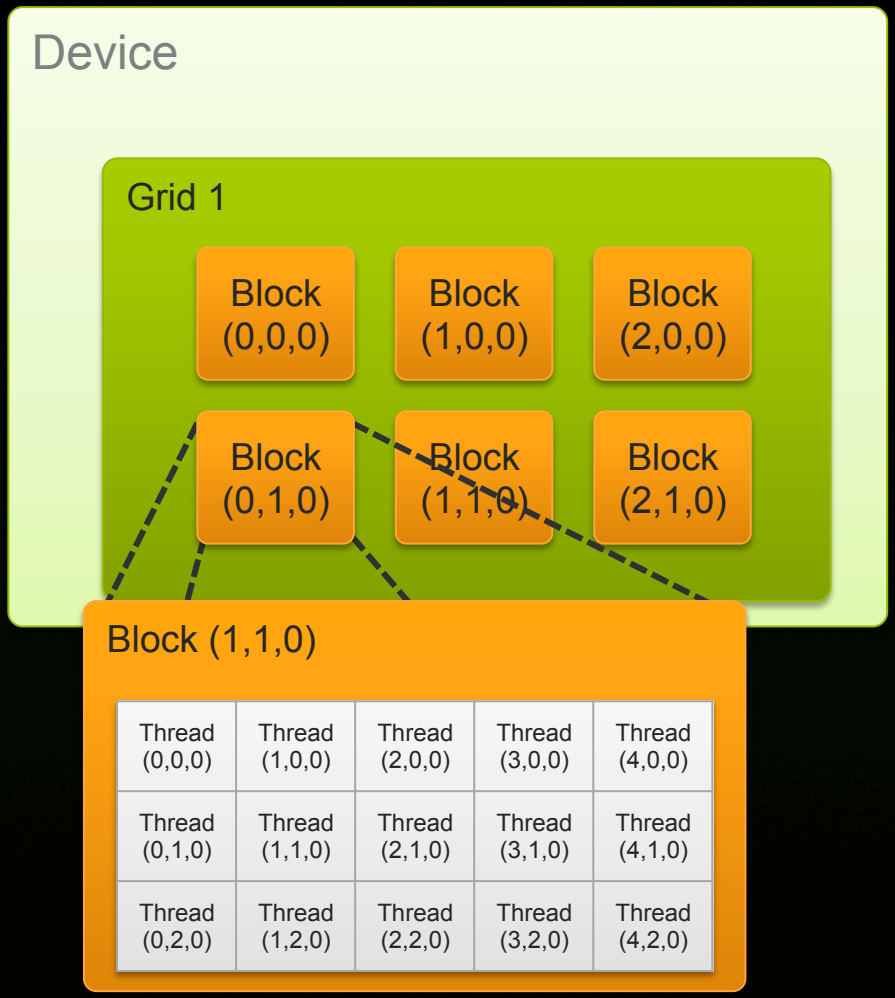
- The following presentations concentrate on Fermi devices
  - Compute Capability >= 2.0

© NVIDIA Corporation 2011

IDs and Dimensions

- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - We showed only one dimension (`x`)

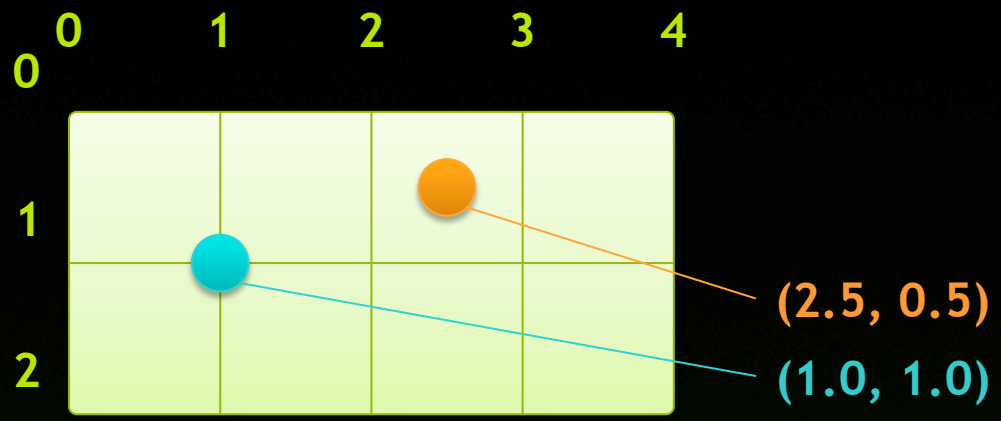
- Built-in variables:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



© NVIDIA Corporation 2011

Textures

- Read-only object
  - Dedicated cache
- Dedicated filtering hardware
  (Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling
  (Wrap, clamp)



© NVIDIA Corporation 2011

What's Next?

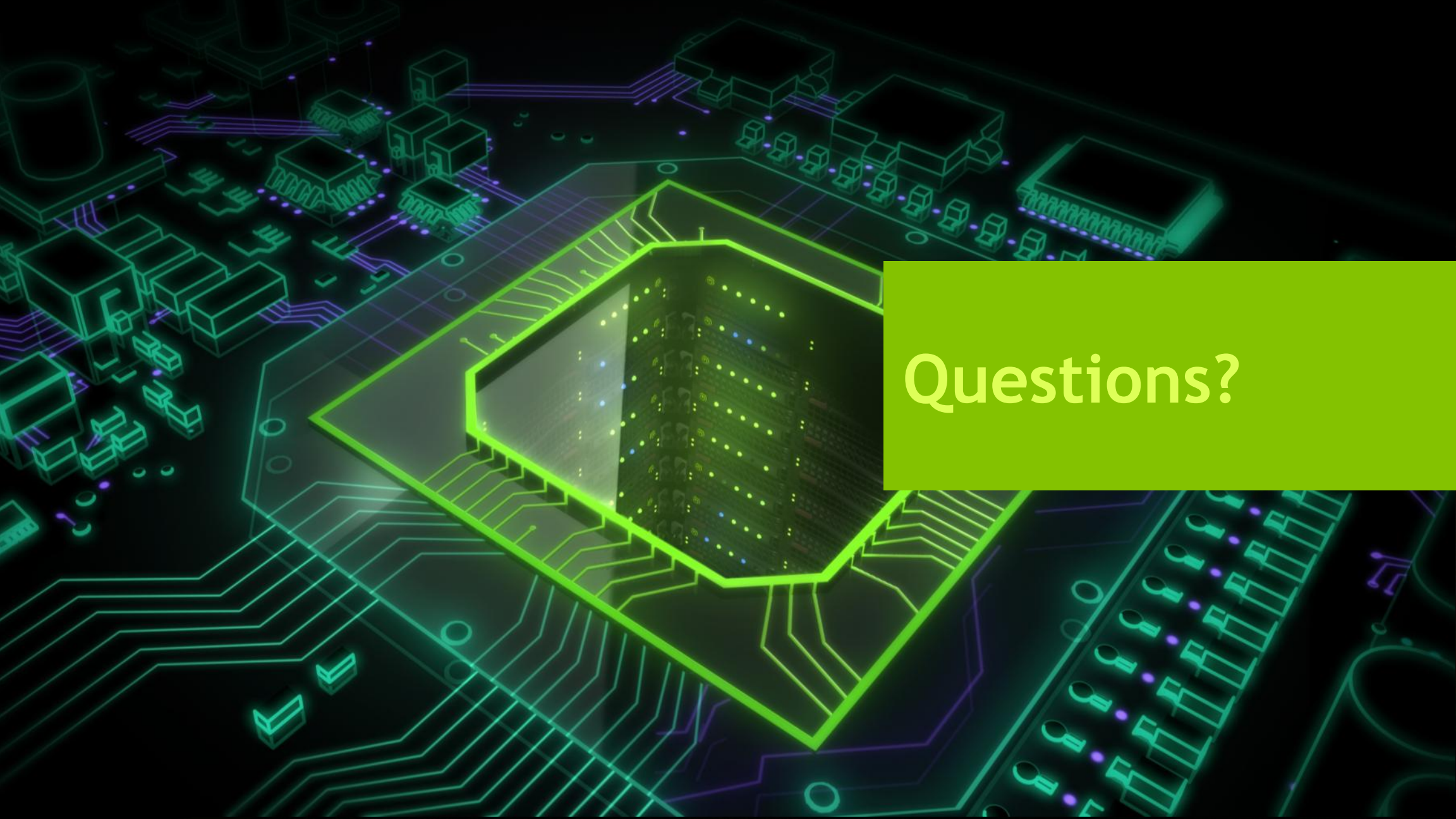
Morning

- CUDA Libraries and CUDA Fortran
- Analysis and Optimization

Afternoon

- Analysis and Optimization
- Real-world Examples:
  - Optimising Stencils for Finite Volume CFD
  - The Texture Unit as a Performance Booster in 3D Volume Reconstruction
  - Optimisation Myths and Facts as Seen in Statistical Physics
  - Putting Branching to Work in Real-time Visualization of Medical Images

© NVIDIA Corporation 2011


Questions?

CUDA Libraries and CUDA Fortran

Massimiliano Fatica
NVIDIA Corporation

NVIDIA CUDA Libraries

CUDA Toolkit includes several libraries:

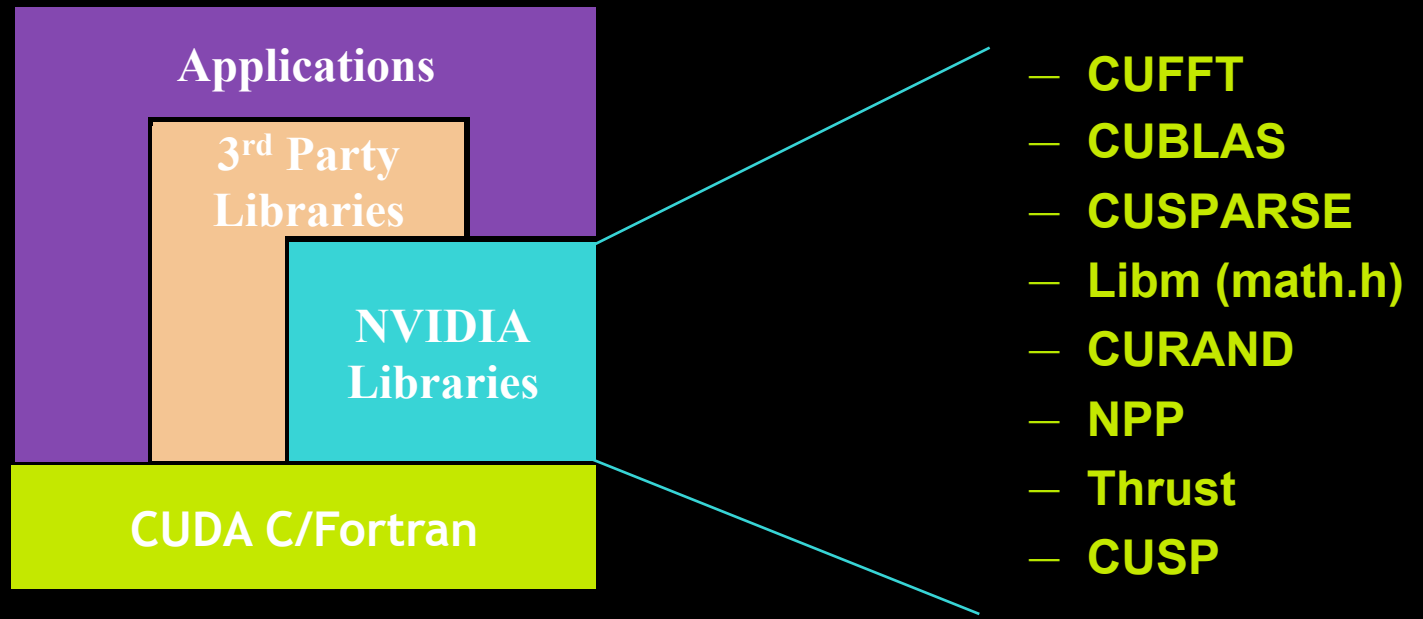
- CUFFT: Fourier transforms
- CUBLAS: Dense Linear Algebra
- CUSPARSE : Sparse Linear Algebra
- LIBM: Standard C Math library
- CURAND: Pseudo-random and Quasi-random numbers
- NPP: Image and Signal Processing
- Thrust : Template Library

Several open source and commercial* libraries:

- MAGMA: Linear Algebra
- CULA Tools*: Linear Algebra
- CUSP: Sparse Linear Solvers
- NAG*: Computational Finance

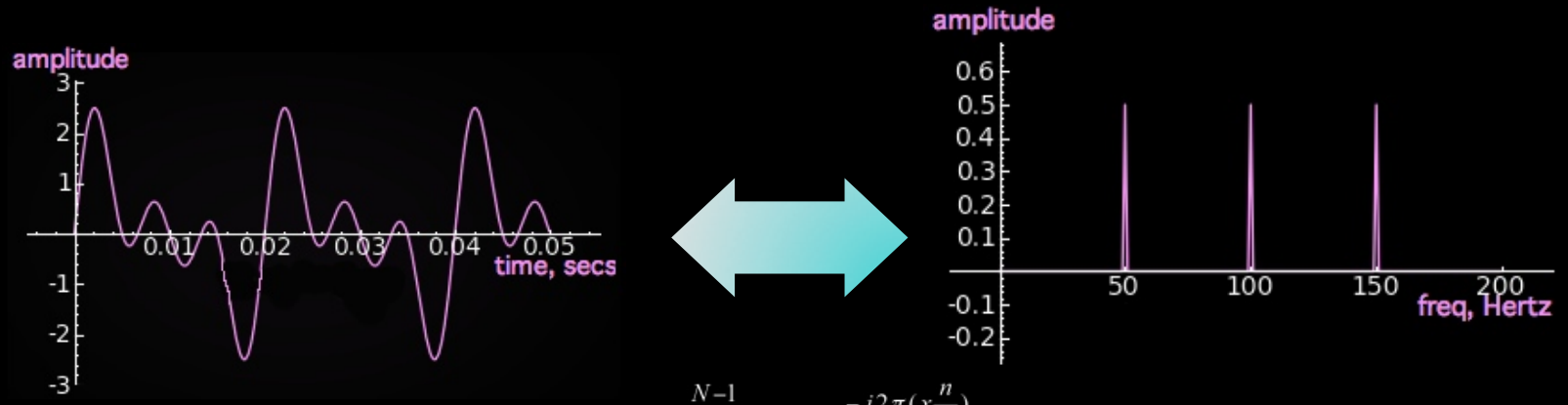
- OpenVidia: Computer Vision
- OpenCurrent: CFD
- ……..

NVIDIA CUDA Libraries



CUFFT Library

CUFFT is a GPU based Fast Fourier Transform library

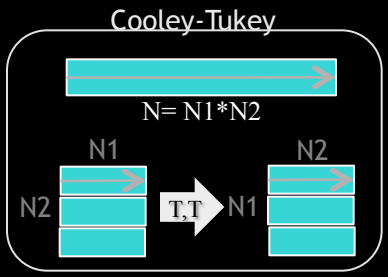


$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$

$$f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x) e^{j2\pi(x\frac{n}{N})}$$

CUFFT Library Features

- Algorithms based on Cooley-Tukey ($n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d$) and Bluestein
- Simple interface similar to FFTW
- 1D, 2D and 3D transforms of complex and real data
- Row-major order (C-order) for 2D and 3D data
- Single precision (SP) and Double precision (DP) transforms
- In-place and out-of-place transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- Streamed asynchronous execution
- Non normalized output: IFFT(FFT(A))=len(A)*A



CUFFT in 4 easy steps

Step 1 – Allocate space on GPU memory

Step 2 – Create plan specifying transform configuration like the size and type (real, complex, 1D, 2D and so on).

Step 3 – Execute the plan as many times as required, providing the pointer to the GPU data created in Step 1.

Step 4 – Destroy plan, free GPU memory

Example CUFFT Program

```
#include <stdlib.h>
#include <stdio.h>
#include “cufft.h“

#define NX 256
#define NY 128

main()
{
    cufftHandle plan;
    cufftComplex *idata, *odata;
    int i;

    cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
    cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
    for( i=0; i<NX*NY; i++ ) {
            idata[i].x = (float)rand() / (float)RAND_MAX;
            idata[i].y = (float)rand() / (float)RAND_MAX;
     }

    /* Create a 2D FFT plan. */
    cufftPlan2d(&plan, NX, NY, CUFFT_C2C);

    /* Use the CUFFT plan to transform the signal out of place.
     * Note: idata != odata indicates an out of place
     * transformation to CUFFT at execution time. */
    cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

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

    /* Destroy the CUFFT plan. */
    cufftDestroy(plan);
    cudaFree(idata);
    cudaFree(odata);

    return 0;
}
```

CUBLAS Library

- Implementation of BLAS (Basic Linear Algebra Subprograms)
  - Self-contained at the API level
- Supports all the BLAS functions
  - Level1 (vector,vector): O(N)
    - AXPY : y = alpha.x + y
    - DOT : dot = x.y
  - Level 2(matrix,vector): $O(N^{2})$
    - Vector multiplication by a General Matrix : GEMV
    - Triangular solver : TRSV
  - Level3(matrix,matrix): $O(N^{3})$
    - General Matrix Multiplication : GEMM
    - Triangular Solver : TRSM
- Following BLAS convention, CUBLAS uses column-major storage

CUBLAS Features

- Support of 4 types :
  - Float, Double, Complex, Double Complex
  - Respective Prefixes : S, D, C, Z
- Contains 152 routines : S(37), D(37), C(41), Z(41)
- Function naming convention: cublas + BLAS name
- Example: cublasSGEMM
  - S: single precision (float)
  - GE: general
  - M: multiplication
  - M: matrix

Using CUBLAS

- Interface to CUBLAS library is in cublas.h
- Function naming convention
  - cublas + BLAS name
  - Eg., cublasSGEMM
- Error handling
  - CUBLAS core functions do not return error
    - CUBLAS provides function to retrieve last error recorded
  - CUBLAS helper functions do return error
- Helper functions:
  - Memory allocation, data transfer

Calling CUBLAS from C

```
#include <stdlib.h>
#include <stdio.h>
#include "cublas.h"

main()
{
    float *a, *b, *c;
    float *d_a, *d_b, *d_c;
    int lda, ldb, ldc;
    int i, j, n;
    struct timeval t1, t2, t3, t4;
    double dt1, dt2, flops;

    cublasInit();
    printf( " n      t1      t2    GF/s  GF/s\n" );

    for( n=512; n<5120; n+=512 ) {

        lda = ldb = ldc = 2*n;

        cudaMallocHost( (void**)&a, n*lda*sizeof(float) );
        cudaMallocHost( (void**)&b, n*ldb*sizeof(float) );
        cudaMallocHost( (void**)&c, n*ldc*sizeof(float) );

        for( j=0; j<n; j++ ) {
            for( i=0; i<n; i++ ) {
                a[i+j*lda] = (float)rand()/(float)RAND_MAX;
                b[i+j*ldb] = (float)rand()/(float)RAND_MAX;
                c[i+j*ldc] = (float)rand()/(float)RAND_MAX;
            }
        }

        cublasAlloc( n*lda, sizeof(float), (void **)&d_a );
        cublasAlloc( n*ldb, sizeof(float), (void **)&d_b );
        cublasAlloc( n*ldc, sizeof(float), (void **)&d_c );

        gettimeofday (&t1, NULL);
        cublasSetMatrix( n, n, sizeof(float), a, lda, d_a, lda );
        cublasSetMatrix( n, n, sizeof(float), b, ldb, d_b, ldb );
        gettimeofday (&t2, NULL);
        cublasSgemm( 'N', 'N', n, n, n, 1.0, d_a, lda, d_b, ldb, 0.0, d_c, ldc );
        cudaThreadSynchronize();
        gettimeofday (&t3, NULL);
        cublasGetMatrix( n, n, sizeof(float), d_c, ldc, c, ldc );
        gettimeofday (&t4, NULL);

        cublasFree( d_a );
        cublasFree( d_b );
        cublasFree( d_c );

        cudaFreeHost( a );
        cudaFreeHost( b );
        cudaFreeHost( c );

        tdiff1 = t4.tv_sec - t1.tv_sec + 1.0e-6 * (t4.tv_usec - t1.tv_usec);
        tdiff2 = t3.tv_sec - t2.tv_sec + 1.0e-6 * (t3.tv_usec - t2.tv_usec);
        flops = 2.0 * (double)n * (double)n * (double)n;
        printf( "%4d %8.5f %8.5f %5.0f %5.0f\n", n, dt1, dt2, 1.0e-9*flops/tdiff1, 1.0e-9*flops/tdiff2 );
    }

    cublasShutdown();
    return 0;
}
```

Calling CUBLAS from FORTRAN

- **Two interfaces:**
  - **Thunking**
    - Allows interfacing to existing applications without any changes
    - 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
    - Intended for light testing due to call overhead
  - **Non-Thunking** (default)
    - Intended for production code
    - Substitute device pointers for vector and matrix arguments in all BLAS functions
    - 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)

SGEMM example (THUNKING)

```
program example_sgemm
! Define  3  single precision matrices A, B, C
real, dimension(:,:),allocatable:: A(:,:),B(:,:),C(:,:)
integer:: n=16
allocate (A(n,n),B(n,n),C(n,n))
! Initialize A, B and C
…
#ifdef CUBLAS
 ! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
 ! memory allocation on device and data movement)
 call cublas_SGEMM('n','n', n,n,n,1.,A,n,B,n,1.,C,n)
#else
! Call SGEMM in  host BLAS library
  call SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#endif
print *,c(n,n)
end program example_sgemm
```

To use the host BLAS routine:

g95 –O3 code.f90 –L/usr/local/lib -lblas

To use the CUBLAS routine (fortran_thunking.c is included in the toolkit /usr/local/cuda/src):

nvcc -O3 -c fortran_thunking.c

g95 -O3 -DCUBLAS code.f90 fortran_thunking .o -L/usr/local/cuda/lib64 -lcudart -lcublas

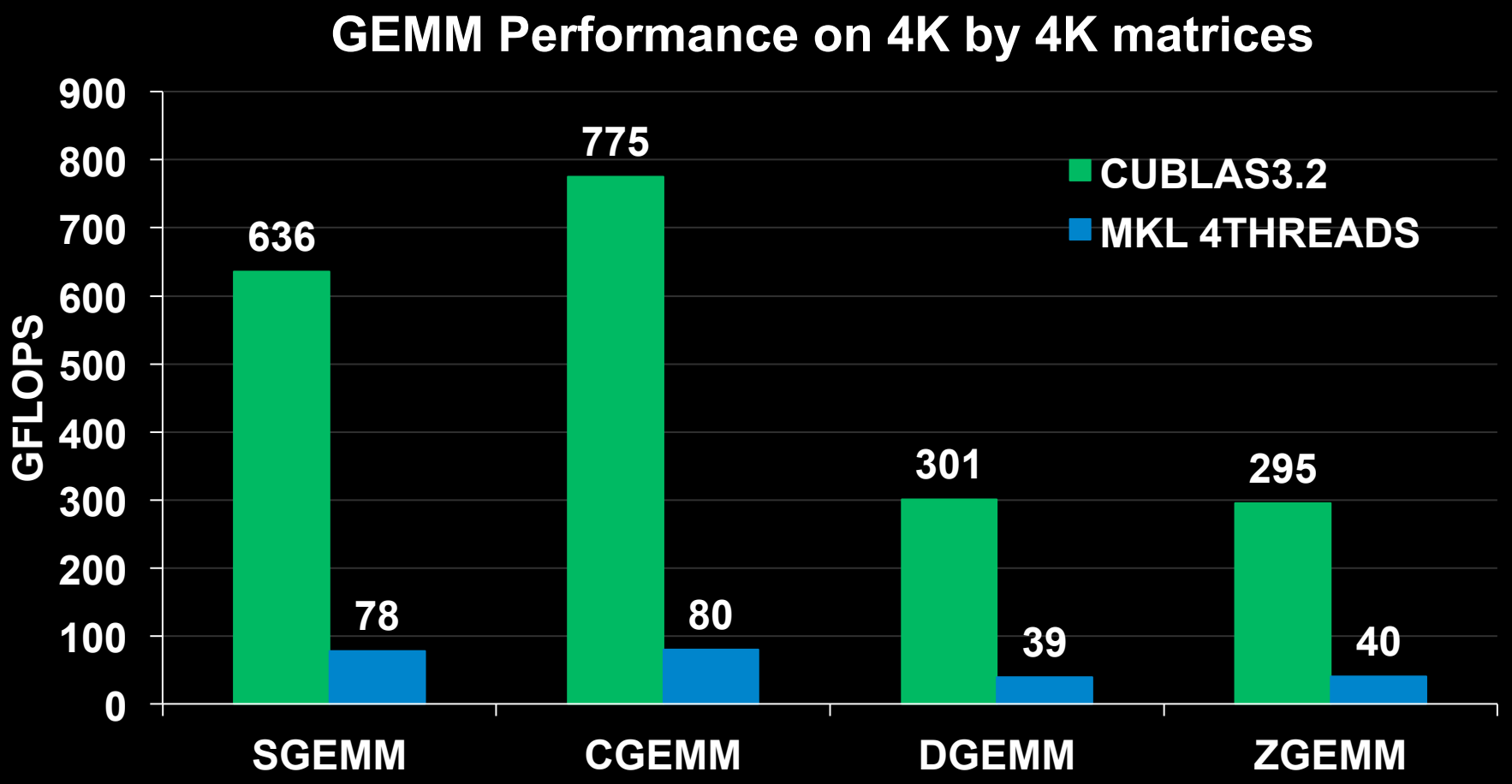
SGEMM example (NON-THUNKING)

```
program example_sgemm
real, dimension(:,:),allocatable:: A(:,:),B(:,:),C(:,:)
integer*8:: devPtrA, devPtrB, devPtrC
integer:: n=16, size_of_real=16
allocate (A(n,n),B(n,n),C(n,n))
call cublas_Alloc(n*n,size_of_real, devPtrA)
call cublas_Alloc(n*n,size_of_real, devPtrB)
call cublas_Alloc(n*n,size_of_real, devPtrC)
! Initialize A, B and C
…
! Copy data to GPU
call cublas_Set_Matrix(n,n,size_of_real,A,n,devPtrA,n)
call cublas_Set_Matrix(n,n,size_of_real,B,n,devPtrB,n)
call cublas_Set_Matrix(n,n,size_of_real,C,n,devPtrC,n)
! Call SGEMM in CUBLAS library
call cublas_SGEMM('n','n', n,n,n,1.,devPtrA,n,devPtrB,n,1.,devPtrC,n)
! Copy data from GPU
call cublas_Get_Matrix(n,n,size_of_real,devPtrC,n,C,n)
print *,c(n,n)
call cublas_Free(devPtrA)
call cublas_Free(devPtrB)
call cublas_Free(devPtrC)
end program example_sgemm
```

To use the CUBLAS routine (fortran.c is included in the toolkit /usr/local/cuda/src):

```
nvcc -O3 -c  fortran.c
g95 -O3  code.f90 fortran.o -L/usr/local/cuda/lib64 -lcudart  -lcublas
```

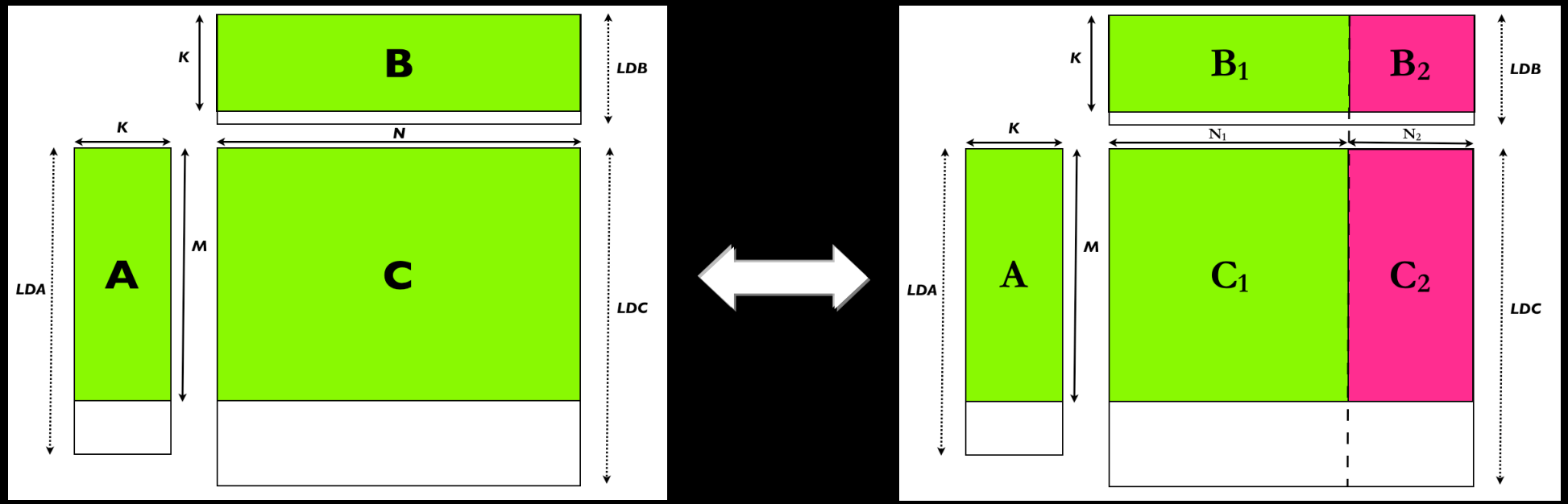
GEMM Performance



Performance may vary based on OS version and motherboard configuration

cuBLAS 3.2, Tesla C2050 (Fermi), ECC on
MKL 10.2.3, 4-core Corei7 @ 2.66 GHz

Using CPU and GPU concurrently



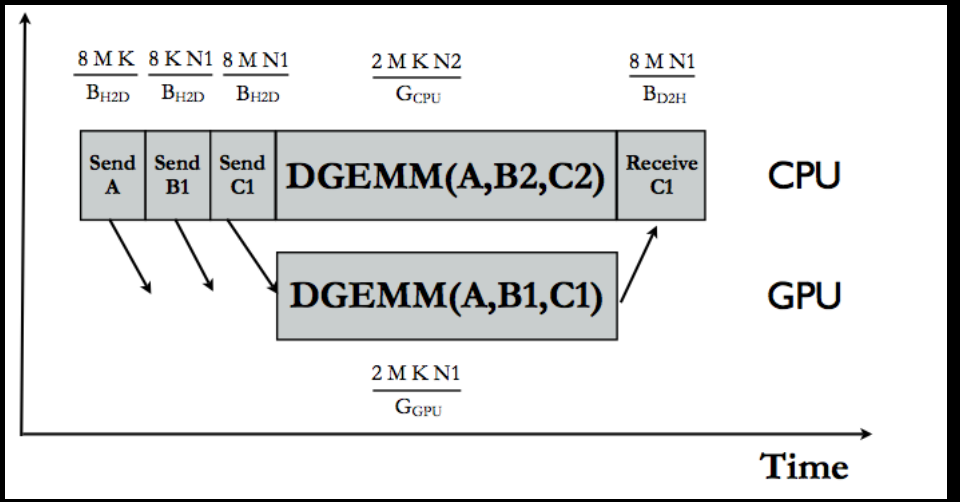
DGEMM(A,B,C) = DGEMM(A,B1,C1) U DGEMM(A,B2,C2)
(GPU) (CPU)

The idea can be extended to multi-GPU configuration and to handle huge matrices

Find the optimal split, knowing the relative performances of the GPU and CPU cores on DGEMM

Overlap DGEMM on CPU and GPU

```
// Copy A from CPU memory to GPU memory devA
   status = cublasSetMatrix (m, k    , sizeof(A[0]), A, lda, devA, m_gpu);
// Copy B1  from CPU memory to GPU memory devB
   status = cublasSetMatrix (k ,n_gpu, sizeof(B[0]), B, ldb, devB, k_gpu);
// Copy C1  from CPU memory to GPU memory devC
   status = cublasSetMatrix (m, n_gpu, sizeof(C[0]), C, ldc, devC, m_gpu);

// Perform  DGEMM(devA,devB,devC) on GPU
// Control immediately return to CPU
   cublasDgemm('n', 'n', m, n_gpu, k, alpha, devA, m,devB, k, beta, devC, m);

// Perform  DGEMM(A,B2,C2) on CPU
   dgemm('n','n',m,n_cpu,k, alpha, A, lda,B+ldb*n_gpu, ldb, beta,C+ldc*n_gpu, ldc);

// Copy devC  from GPU memory to CPU memory C1
   status = cublasGetMatrix (m, n, sizeof(C[0]), devC, m, C, *ldc);
```



Using CUBLAS, it is very easy to express the workflow in the diagram

Changes to CUBLAS API in CUDA 4.0

- New header file cublas_v2.h: defines new API.
- Add a handle argument: gives the control necessary to manage multiple host threads and multiple GPUs. Manage handle with cublasCreate(), cublasDestroy().
- Pass and return scalar values by reference to GPU memory.
- All functions return an error code.
- Rename cublasSetKernelStream() to cublasSetStream() for consistency with other CUDA libraries.

CUSPARSE

- New library for sparse basic linear algebra
- Conversion routines for dense, COO, CSR and CSC formats
- Optimized sparse matrix-vector multiplication
- Building block for sparse linear solvers

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \alpha \begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

CUDA Libm features

High performance and high accuracy implementation:

- C99 compatible math library, plus extras
- Basic ops: x+y, x*y, x/y, 1/x, sqrt(x), FMA (IEEE-754 accurate in single, double)
- Exponentials: exp, exp2, log, log2, log10, ...
- Trigonometry: sin, cos, tan, asin, acos, atan2, sinh, cosh, asinh, acosh, ...
- Special functions: lgamma, tgamma, erf, erfc
- Utility: fmod, remquo, modf, trunc, round, ceil, floor, fabs, ...
- Extras: rsqrt, rcbrt, exp10, sinpi, sincos, erfinv, erfcinv, ...

Improvements

- Continuous enhancements to performance and accuracy

CUDA 3.1 erfinvf (single precision)
accuracy
5.43 ulp → 2.69 ulp
performance
1.7x faster than CUDA 3.0

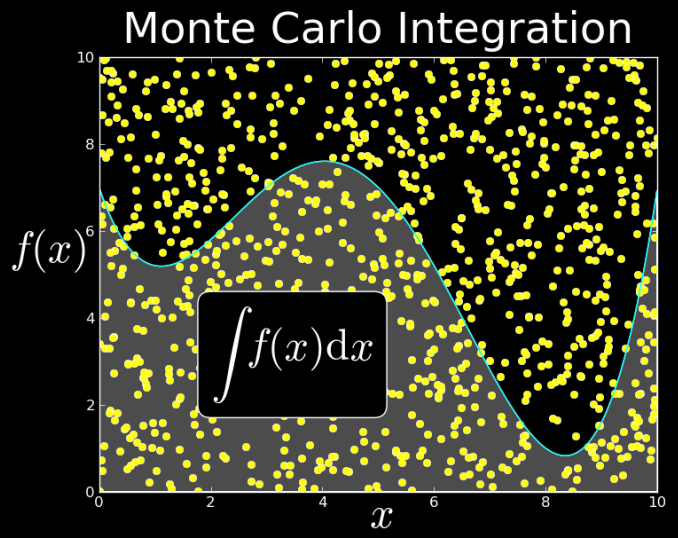
$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

CUDA 3.2 1/x (double precision)
performance
1.8x faster than CUDA 3.1

Double-precision division, rsqrt(), erfc(), & sinh() are all >~30% faster on Fermi

CURAND Library

- Library for generating random numbers



- Features:
  - XORWOW pseudo-random generator
  - Sobol' quasi-random number generators
  - Host API for generating random numbers in bulk
  - Inline implementation allows use inside GPU functions/kernels
  - Single- and double-precision, uniform, normal and log-normal distributions

CURAND Features

- Pseudo-random numbers
  George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14), 2003. Available at http://www.jstatsoft.org/v08/i14/paper.
- Quasi-random 32-bit and 64-bit Sobol' sequences with up to 20,000 dimensions.
- Host API: call kernel from host, generate numbers on GPU, consume numbers on host or on GPU.
- GPU API: generate and consume numbers during kernel execution.

CURAND use

1. **Create a generator:**
   curandCreateGenerator()
2. **Set a seed:**
   curandSetPseudoRandomGeneratorSeed()
3. **Generate the data from a distribution:**
   curandGenerateUniform()/(curandGenerateUniformDouble(): Uniform
   curandGenerateNormal()/cuRandGenerateNormalDouble(): Gaussian
   curandGenerateLogNormal/curandGenerateLogNormalDouble(): Log-Normal
4. **Destroy the generator:**
   curandDestroyGenerator()

Example CURAND Program: Host API

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

main()
{
    int i, n = 100;
    curandGenerator_t gen;
    float *devData, *hostData;

    /* Allocate n floats on host */
    hostData = (float *)calloc(n, sizeof(float));

    /* Allocate n floats on device */
    cudaMalloc((void **)&devData, n * sizeof(float));

    /* Create pseudo-random number generator */
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);

    /* Set seed */
    curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
    /* Generate n floats on device */
    curandGenerateUniform(gen, devData, n);
    /* Copy device memory to host */
    cudaMemcpy(hostData, devData, n * sizeof(float),
                    cudaMemcpyDeviceToHost);

    /* Show result */
    for(i = 0; i < n; i++) {
        printf("%1.4f ", hostData[i]);
    }
    printf("\n");

    /* Cleanup */
    curandDestroyGenerator(gen);
    cudaFree(devData);
    free(hostData);

    return 0;
}
```

Example CURAND Program: Run on CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

main()
{
    int i, n = 100;
    curandGenerator_t gen;
    float *hostData;

    /* Allocate n floats on host */
    hostData = (float *)calloc(n, sizeof(float));

    /* Create pseudo-random number generator */
    curandCreateGeneratorHost(&gen,
        CURAND_RNG_PSEUDO_DEFAULT);

    /* Set seed */
    curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
    /* Generate n floats on host */
    curandGenerateUniform(gen, hostData, n);

    /* Show result */
    for(i = 0; i < n; i++) {
        printf("%1.4f ", hostData[i]);
    }
    printf("\n");

    /* Cleanup */
    curandDestroyGenerator(gen);
    free(hostData);

    return 0;
}
```

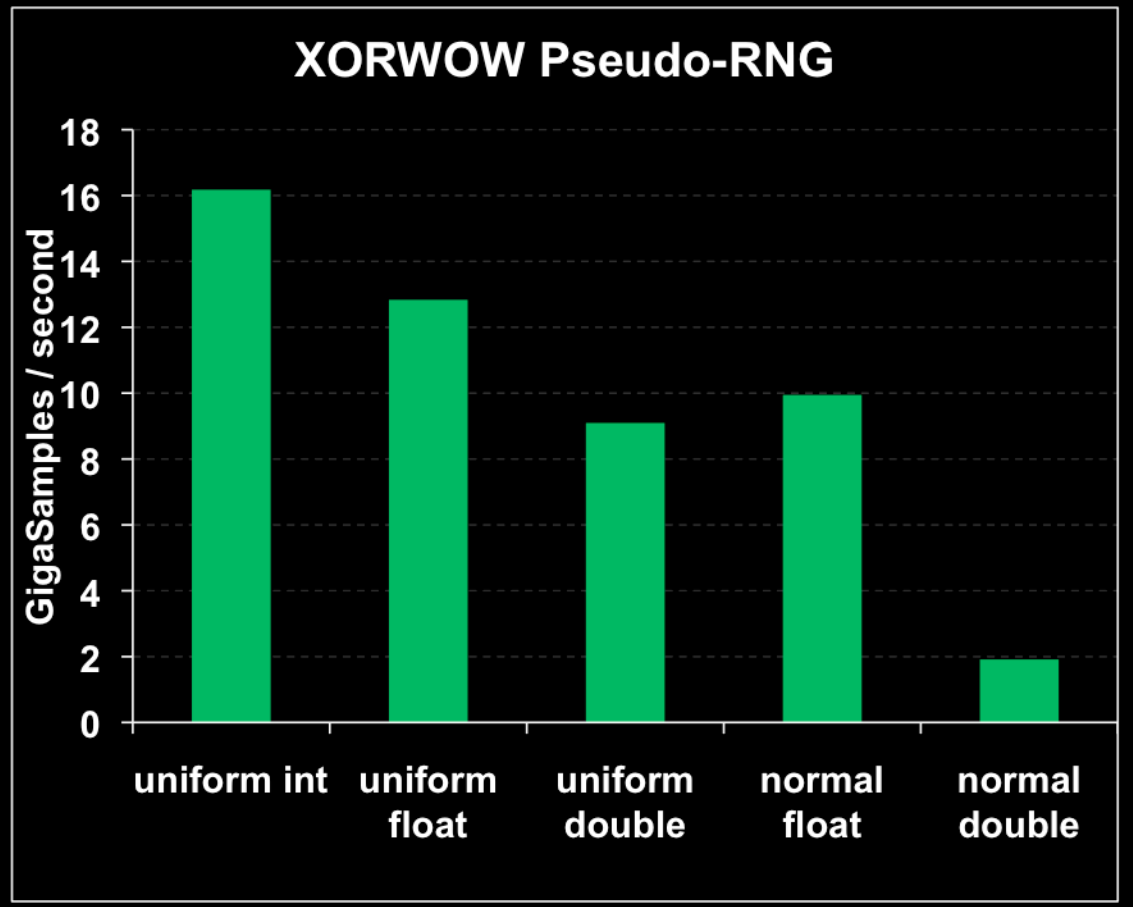
Example CURAND Program: Device API

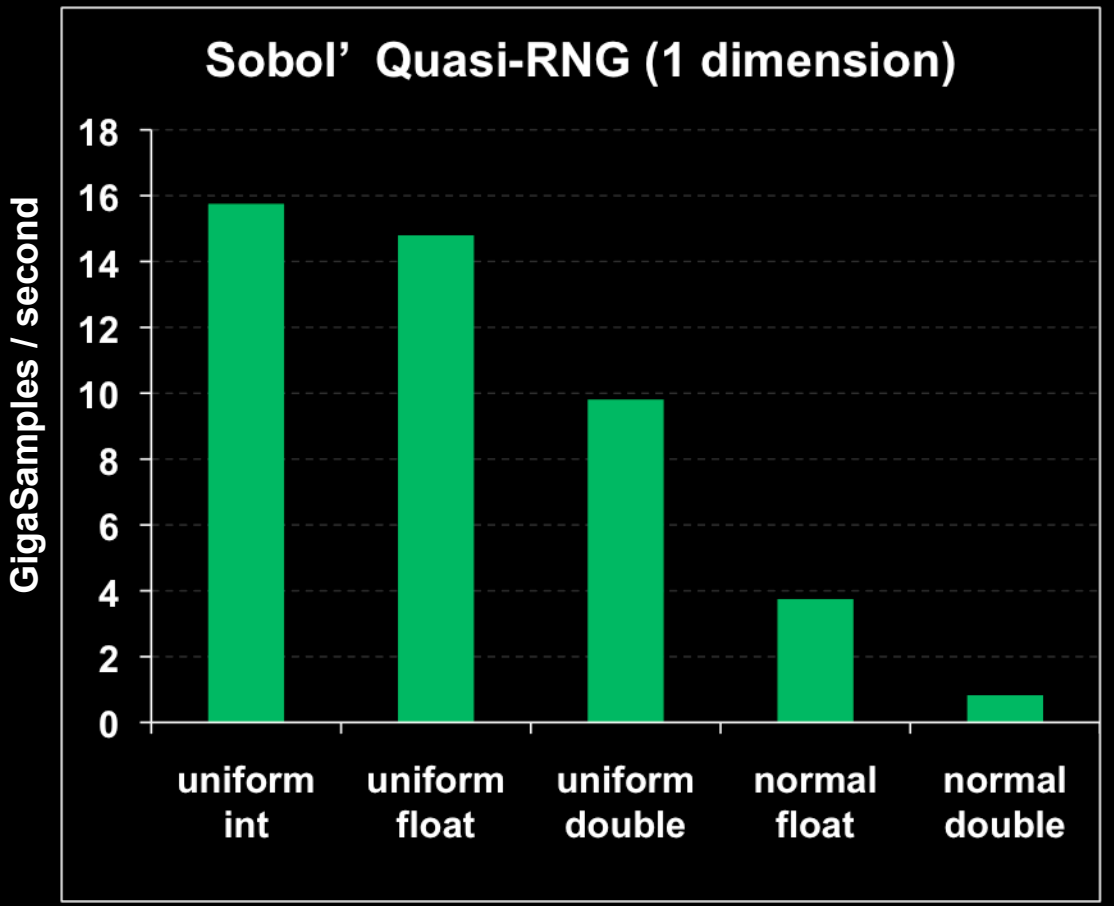
```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>

__global__ void setup_kernel(curandState *state)
{
    int id = threadIdx.x + blockIdx.x * 64;
    /* Each thread gets same seed,
       a different sequence number, no offset */
    curand_init(1234, id, 0, &state[id]);
}

__global__ void generate_kernel(curandState *state, int *result)
{
    int id = threadIdx.x + blockIdx.x * 64;
    int count = 0;
    unsigned int x;
    /* Copy state to local memory for efficiency */
    curandState localState = state[id];

    /* Generate pseudo-random unsigned ints */
    for(int n = 0; n < 100000; n++) {
        x = curand(&localState);
        /* Check if low bit set */
        if(x & 1) count++;
    }

    /* Copy state back to global memory */
    state[id] = localState;

    /* Store results */
    result[id] += count;
}
```

Example CURAND Program: Device API

```
main()
{
    int i, total;
    curandState *devStates;
    int *devResults, *hostResults;

    /* Allocate space for results on host */
    hostResults = (int *)calloc(64 * 64, sizeof(int));

    /* Allocate space for results on device */
    cudaMalloc((void **)&devResults, 64 * 64 * sizeof(int));

    /* Set results to 0 */
    cudaMemset(devResults, 0, 64 * 64 * sizeof(int));

    /* Allocate space for prng states on device */
    cudaMalloc((void **)&devStates, 64 * 64 * sizeof(curandState));

    /* Setup prng states */
    setup_kernel<<<64, 64>>>(devStates);

    /* Generate and use pseudo-random */
    for(i = 0; i < 10; i++) {
        generate_kernel<<<64, 64>>>(devStates, devResults);
    }

    /* Copy device memory to host */
    cudaMemcpy(hostResults, devResults, 64 * 64 * sizeof(int),
                cudaMemcpyDeviceToHost);

    /* Show result */
    total = 0;
    for(i = 0; i < 64 * 64; i++) {
        total += hostResults[i];
    }
    printf("Fraction with low bit set was %10.13f\n",
    (float)total / (64.0f * 64.0f * 100000.0f * 10.0f));

    /* Cleanup */
    cudaFree(devStates);
    cudaFree(devResults);
    free(hostResults);

    return 0;
}
```

CURAND Performance





Performance may vary based on OS version and motherboard configuration

CURAND 3.2, NVIDIA C2050 (Fermi), ECC on

NVIDIA Performance Primitives (NPP)

- C library of functions (primitives)
  - well optimized
  - low level API:
    - easy integration into existing code
    - algorithmic building blocks
  - actual operations execute on CUDA GPUs
- Approximately 350 image processing functions
- Approximately 100 signal processing functions

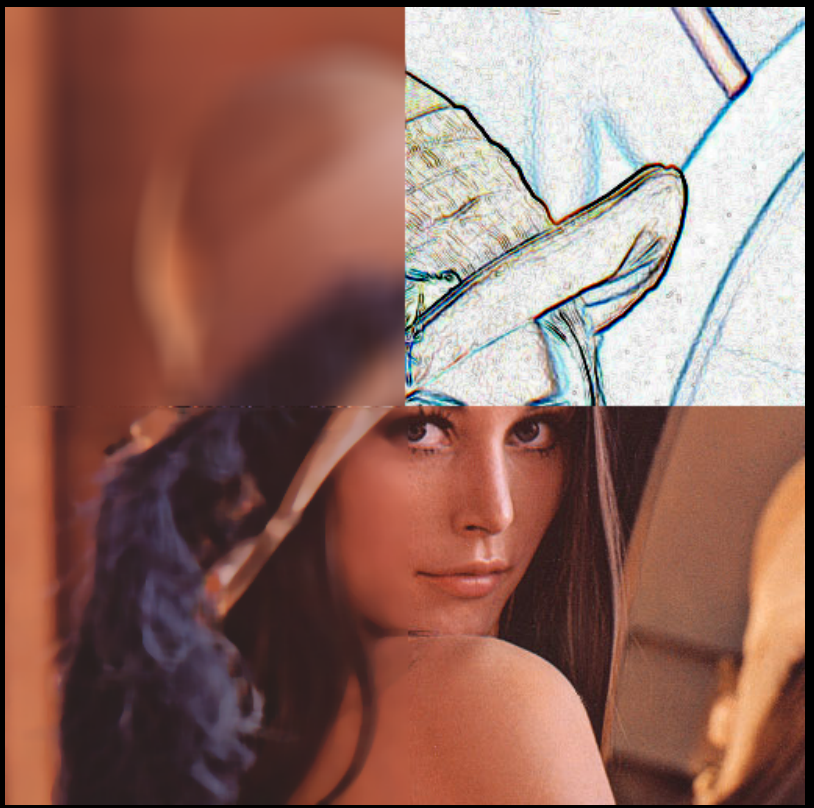


Image Processing Primitives

- Data exchange & initialization
  - Set, Convert, CopyConstBorder, Copy, Transpose, SwapChannels
- Arithmetic & Logical Ops
  - Add, Sub, Mul, Div, AbsDiff
- Threshold & Compare Ops
  - Threshold, Compare
- Color Conversion
  - RGB To YCbCr (& vice versa), ColorTwist, LUT_Linear
- Filter Functions
  - FilterBox, Row, Column, Max, Min, Dilate, Erode, SumWindowColumn/Row
- Geometry Transforms
  - Resize , Mirror, WarpAffine/Back/Quad, WarpPerspective/Back/Quad
- Statistics
  - Mean, StdDev, NormDiff, MinMax, Histogram, SqrIntegral, RectStdDev
- Segmentation
  - Graph Cut

Thrust

- A template library for CUDA
  - Mimics the C++ STL
- Containers
  - Manage memory on host and device: thrust::host_vector<T>, thrust:device_vector<T>
  - Help avoid common errors
- Iterators
  - Know where data lives
  - Define ranges: d_vec.begin()
- Algorithms
  - Sorting, reduction, scan, etc: thrust::sort()
  - Algorithms act on ranges and support general types and operators

Thrust Example

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib.h>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Algorithms

- **Elementwise operations**

  `for_each`, `transform`, `gather`, `scatter` …

- **Reductions**

  `reduce`, `inner_product`, `reduce_by_key` …

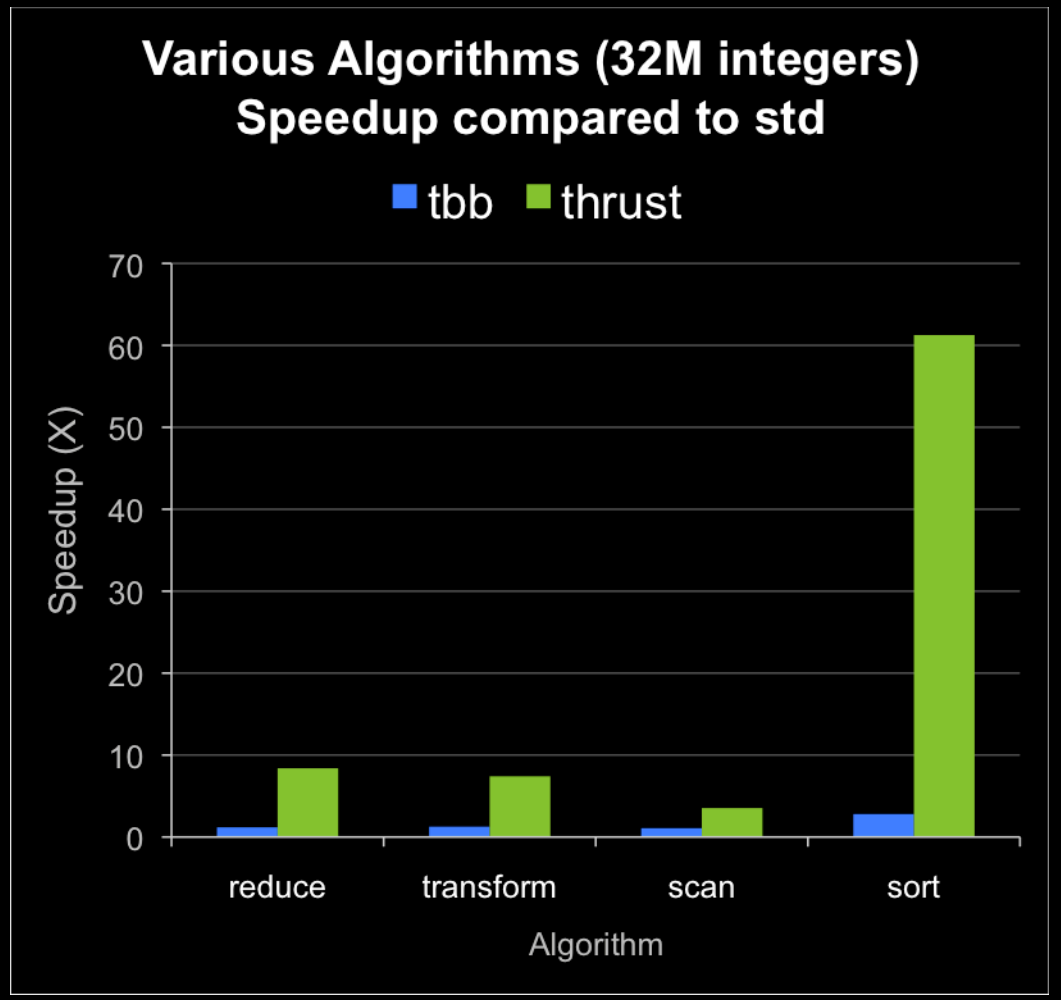
- **Prefix-Sums**

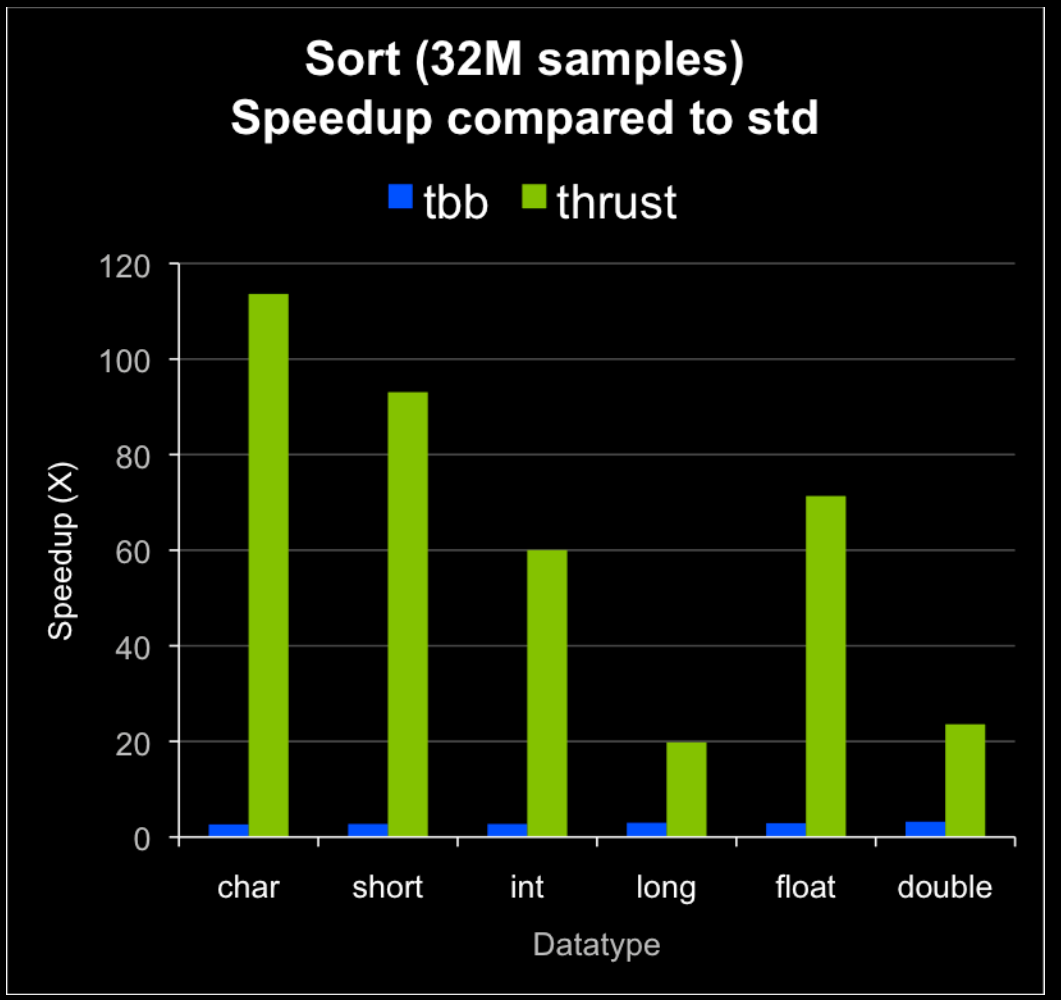
  `inclusive_scan`, `inclusive_scan_by_key` …

- **Sorting**

  `sort`, `stable_sort`, `sort_by_key` …

Thrust Algorithm Performance





* Thrust 4.0, NVIDIA Tesla C2050 (Fermi)

* Core i7 950 @ 3.07GHz

Interoperability (from Thrust to C/CUDA)

- Convert iterators to raw pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< N / 256, 256 >>>(N, ptr);

// Note: ptr cannot be dereferenced on the host!
```

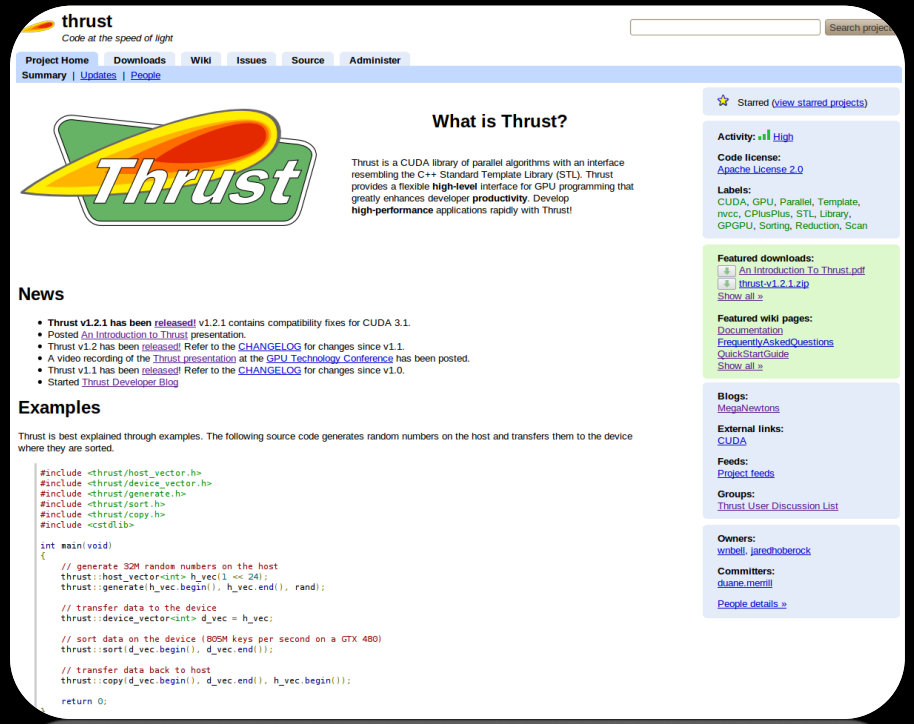
Interoperability (from C/CUDA to Thrust)

- Wrap raw pointers with `device_ptr`

```
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof int));

// wrap raw pointer with a device_ptr
device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```

Thrust on Google Code

- Quick Start Guide
- Examples
- Documentation
- Mailing List (thrust-users)



CUDA Fortran

- PGI / NVIDIA collaboration
- Same CUDA programming model as CUDA-C with Fortran syntax
- Strongly typed – variables with device-type reside in GPU memory
- Use standard allocate, deallocate
- Copy between CPU and GPU with assignment statements:

```
GPU_array = CPU_array
```

- Kernel loop directives (CUF Kernels) to parallelize loops with device data

CUDA Fortran example

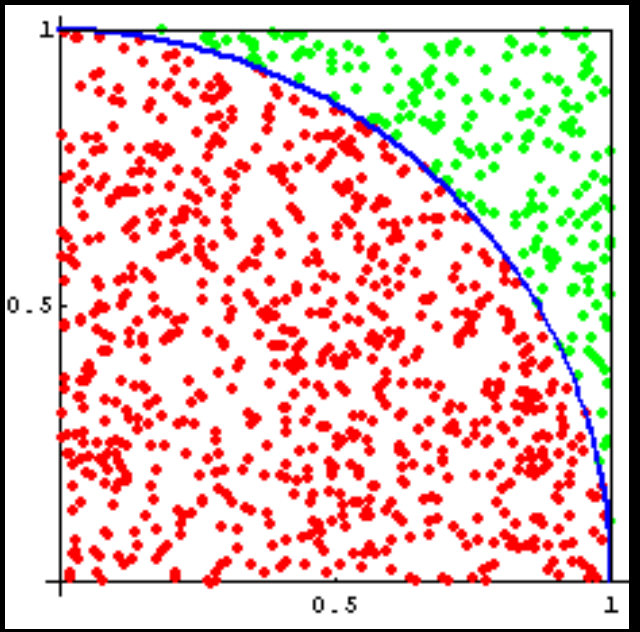
```
program gpu_example
 use cudafor
 real, allocatable :: cpu_array(:,:)
 real, allocatable, device :: gpu_array(:,:)
 type(dim3):: grid_size, block_size
! Use standard allocate for CPU and GPU arrays
 allocate(cpu_array(n,m), gpu_array(n,m))
 call initialize(cpu_array)
! Move data with simple assignment
 gpu_array = cpu_array
! Call CUDA kernel
  …
  block_size=dim3(16,16,1)
  call gpu_func<<<grid_size, block_size>>> ( gpu_array, n,m )
  cpu_array = gpu_array
  deallocate(cpu_array, gpu_array)
end program
```

CUDA Fortran example

```
attributes(global) subroutine gpu_func(gpu_array, n, m)
real:: gpu_array(:,:)
integer, value:: n, m
integer:: i,j
real::  p_ref=1.0

! Indices start from 1
i = threadIdx%x+ (blockIdx%x -1) * blockDim%x
j = threadIdx%y+ (blockIdx%y -1) * blockDim%y
If ( i <= n .and. j<=m)
            gpu_array(i,j) = SQRT( 2)*( gpu_array(i,j) + p_ref )
end if

end subroutine gpu_func
```

Computing π with CUDA Fortran

$$\pi = 4 * (\Sigma \text{ red points})/(\Sigma \text{ points})$$



Simple example:

- **Generate random numbers (CURAND)**
- **Compute sum using of kernel loop directive**
- **Compute sum using two stages reduction with Cuda Fortran kernels**
- **Compute sum using single stage reduction with Cuda Fortran kernel**
- **Accuracy**

CUDA Libraries from CUDA Fortran

- CUBLAS, CUFFT and CURAND have C interfaces
- Use F90 interfaces and ISO C Binding to call them directly from CUDA Fortran

```
interface curandGenerateUniform

!curandGenerateUniform(curandGenerator_t generator, float *outputPtr, size_t num);
subroutine curandGenerateUniform(generator, odata, numele) bind(C,name='curandGenerateUniform')
 use iso_c_binding
 integer(c_size_t),value:: generator
!pgi$ ignore_tr odata
 real(c_float), device:: odata(*)
 integer(c_size_t),value:: numele
end subroutine curandGenerateUniform

!curandGenerateUniformDouble(curandGenerator_t generator, double *outputPtr, size_t num);
subroutine curandGenerateUniformDouble(generator, odata, numele) bind(C,name='curandGenerateUniformDouble')
 use iso_c_binding
 integer(c_size_t),value:: generator
!pgi$ ignore_tr odata
 real(c_double), device:: odata(*)
 integer(c_size_t),value:: numele
end subroutine curandGenerateUniformDouble

end interface curandGenerateUniform
```

Computing π with CUF kernel

```
! Compute pi using a Monte Carlo method
program compute_pi
use precision
use cudafor   ! CUDA Fortran runtime
use curand    ! CURAND interface
implicit none
real(fp_kind), allocatable, pinned:: hostData(:)
real(fp_kind), allocatable, device:: deviceData(:)
real(fp_kind):: pival
integer :: inside_cpu,inside, i, iter, Nhalf
integer(kind=8) :: gen, N, seed=1234

N=2000 ! Define how many numbers we want to generate
Nhalf=N/2
! Allocate arrays on CPU and GPU
 allocate(hostData(N), deviceData(N))
! Create pseudonumber generator
 call curandCreateGenerator(gen, CURAND_RNG_PSEUDO_DEFAULT)
! Set seed
 call curandSetPseudoRandomGeneratorSeed( gen, seed)

! Generate N floats or double on device
call curandGenerateUniform(gen, deviceData, N)

! Copy the data back to CPU to check result later
 hostData=deviceData
! Perform the test on GPU using CUF kernel
 inside=0
!$cuf kernel do <<<*,*>>>
 do i=1,Nhalf
  if( (deviceData(i)**2+deviceData(i+Nhalf)**2) <= 1._fp_kind ) inside=inside+1
 end do
! Perform the test on CPU
 inside_cpu=0
 do i=1,Nhalf
  if( (hostData(i)**2+hostData(i+Nhalf)**2) <= 1._fp_kind) inside_cpu=inside_cpu+1
 end do
! Check the results
 if (inside_cpu .ne. inside) print *,"Mismatch between CPU/GPU", inside_cpu,inside
! Print the value of pi and the error
 pival= 4._fp_kind*real(inside,fp_kind)/real(Nhalf,fp_kind)
 print"(t3,a,i10,a,f10.8,a,e11.4)","Samples=",Nhalf,"  Pi=", pival,"  Error=", &
          abs(pival-2.0_fp_kind*asin(1.0_fp_kind))
! Deallocate data on CPU and GPU
 deallocate(hostData,deviceData)
! Destroy the generator
 call curandDestroyGenerator(gen)
end program compute_pi
```

Computing π

pgf90 -Mcuda=3.2 -O3 -Mpreprocess -o pi_gpu precision_module.cuf curand_module.cuf pi.cuf -lcurand

```
Compute pi  in single precision (seed 1234)
 Samples=       10000  Pi=3.11120009  Error= 0.3039E-01
 Samples=      100000  Pi=3.13632011  Error= 0.5273E-02
 Samples=     1000000  Pi=3.14056396  Error= 0.1029E-02
 Samples=    10000000  Pi=3.14092445  Error= 0.6683E-03
 Samples= 100000000  Pi=3.14158082  Error= 0.1192E-04
```

```
Compute pi  in single precision (seed 1234567)
 Samples=       10000  Pi=3.16720009  Error= 0.2561E-01
 Samples=      100000  Pi=3.13919997  Error= 0.2393E-02
 Samples=     1000000  Pi=3.14109206  Error= 0.5007E-03
 Samples=    10000000  Pi=3.14106607  Error= 0.5267E-03
  Mismatch between CPU/GPU     78534862     78534859
 Samples= 100000000  Pi=3.14139414  Error= 0.1986E-03
```

Where is the error coming from?

```
if( (    hostData(i)**2+   hostData(i+Nhalf)**2) <= 1._fp_kind) inside_cpu=inside_cpu+1  (CPU)
if( (deviceData(i)**2+deviceData(i+Nhalf)**2) <= 1._fp_kind ) inside=inside+1              (GPU)
```

–Sum of the point inside the circle is done with integers (no issues due to floating point arithmetic)
–Computation of the distance from the origin (x*x+y*y), no special functions just + and *

GPU accuracy

- FERMI GPUs are IEEE-754 compliant, both for single and double precision
- Support for Fused Multiply-Add instruction (IEEE 754-2008)
- Results with FMA could be different* from results without FMA
- In CUDA Fortran is possible to toggle FMA on/off with a compiler switch:
  -Mcuda=nofma
- Extremely useful to compare results to “golden” CPU output
- FMA is being supported in future CPUs

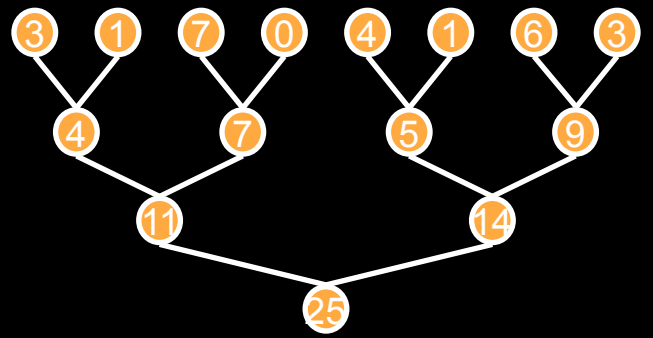
```
Compute pi  in single precision (seed=1234567 FMA disabled)
  Samples=        10000  Pi=3.16720009  Error= 0.2561E-01
  Samples=       100000  Pi=3.13919997  Error= 0.2393E-02
  Samples=      1000000  Pi=3.14109206  Error= 0.5007E-03
  Samples=     10000000  Pi=3.14106607  Error= 0.5267E-03
  Samples=    100000000  Pi=3.14139462  Error= 0.1981E-03
```

*GPU results with FMA are identical to CPU if operations are done in double precision

Reductions on GPU

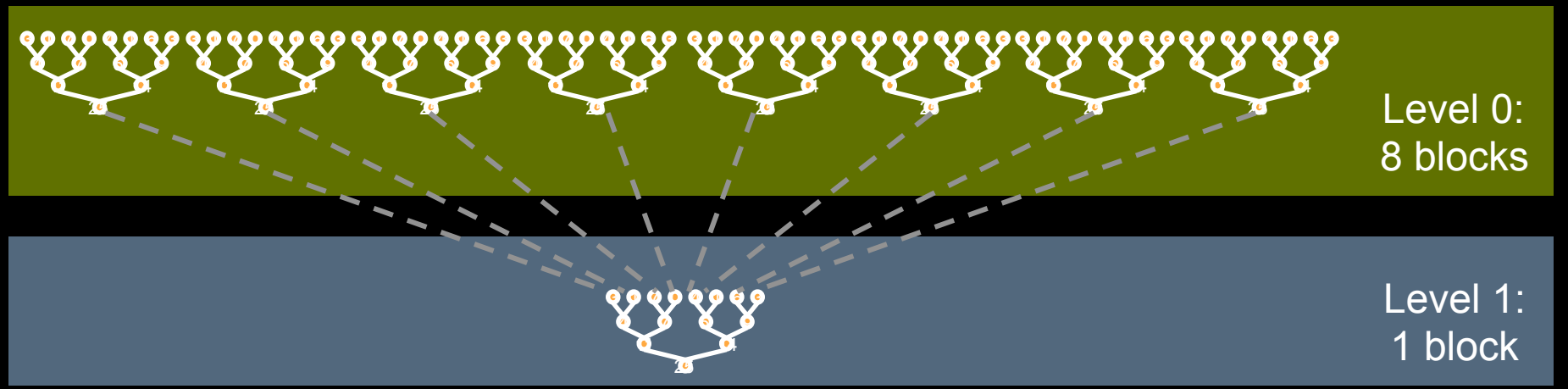
- Need to use multiple blocks

- Need to use multiple threads in a block:

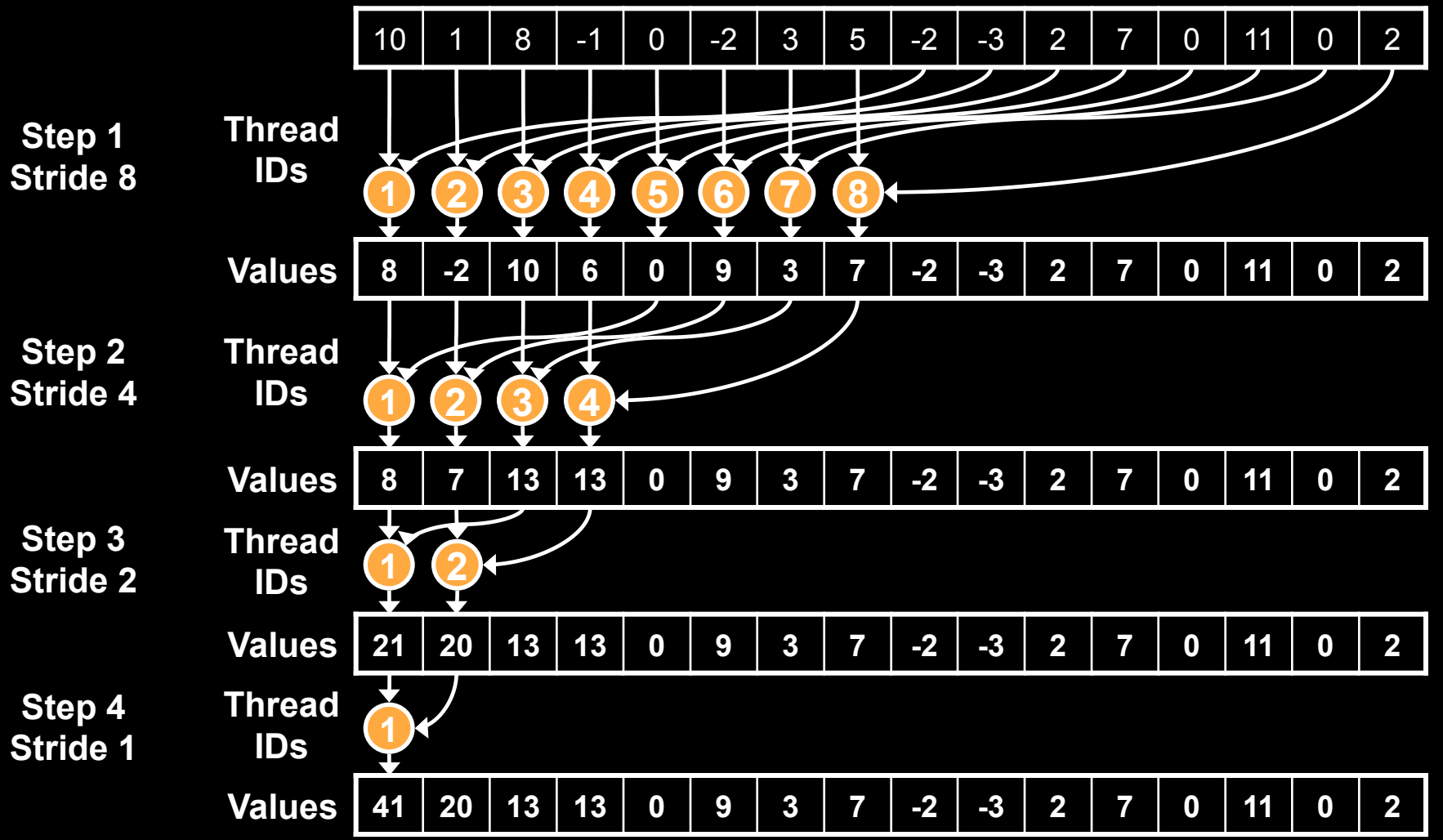


-No global synchronization:

two stages approach, same code for both stages



Parallel Reduction: Sequential Addressing



Computing π with CUDA Fortran kernels (1/2)

```
attributes(global) subroutine partial_sum(input,partial,N)
 real(fp_kind) :: input(N)
 integer :: partial(256)
 integer, shared, dimension(256) :: psum
 integer(kind=8),value :: N
 integer :: i,index, inext,interior

 index=threadIdx%x+(BlockIdx%x-1)*BlockDim%x
! Check if the point is inside the circle and increment local counter
 interior=0
 do i=index,N/2,BlockDim%x*GridDim%x
  if( (input(i)**2+input(i+N/2)**2) <= 1._fp_kind) interior=interior+1
 end do

 ! Local reduction per block
 index=threadIdx%x
 psum(index)=interior
 call syncthreads()

 inext=blockDim%x/2
 do while ( inext >=1 )
  if (index <=inext) psum(index)=psum(index)+psum(index+inext)
  inext = inext /2
  call syncthreads()
 end do

! Each block writes back its partial sum
 if (index == 1) partial(BlockIdx%x)=psum(1)
end subroutine
```

```
! Compute the partial sums with 256 blocks of 512 threads
  call partial_sum<<<256,512,512*4>>>(deviceData,partial,N)
! Compute the final sum with 1 block of 256 threads
  call final_sum<<<1,256,256*4>>>(partial,inside_gpu)
```

Computing π with CUDA Fortran kernels (2/2)

```
attributes(global) subroutine final_sum(partial,nthreads,total)
 integer, intent(in) :: partial(nthreads)
 integer, intent(out) :: total
 integer, shared :: psum(*)
 integer :: index, inext

 index=threadIdx%x

! load partial sums in shared memory
 psum(index)=partial(index)
 call syncthreads()

 inext=blockDim%x/2
 do while ( inext >=1 )
  if (index <=inext) psum(index)=psum(index)+psum(index+inext)
  inext = inext /2
  call syncthreads()
 end do

! First thread has the total sum, writes it back to global memory
 if (index == 1) total=psum(1)
end subroutine
```

Computing π with atomic lock

Instead of storing back the partial sum:

```
! Each block writes back its partial sum
if (index == 1) partial(BlockIdx%x)=psum(1)
```

use atomic lock to ensure that one block at the time updates the final sum:

```
if (index == 1) then
 do while ( atomiccas(lock,0,1) == 1) !set lock
 end do
 partial(1)=partial(1)+psum(1) ! atomic update of partial(1)
 call threadfence()            ! Wait for memory transaction to be  visible to all the other threads
 lock =0                       ! release lock
end if
```

```
partial(1)=0
call sum<<<64,256,256*4>>>(deviceData,partial,N)
inside=partial(1)
```

Calling Thrust from CUDA Fortran

C wrapper for Thrust: csort.cu

```
#include <thrust/device_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

extern "C" {
//Sort for integer arrays
void sort_int_wrapper( int *data, int N)
{
  // Wrap raw pointer with a device_ptr
    thrust::device_ptr<int> dev_ptr(data);
 // Use device_ptr in Thrust sort algorithm
    thrust::sort(dev_ptr, dev_ptr+N);
}

//Sort for single precision arrays
void sort_float_wrapper( float *data, int N)
{
   thrust::device_ptr<float> dev_ptr(data);
   thrust::sort(dev_ptr, dev_ptr+N);
}

//Sort for double precision arrays
void sort_double_wrapper( double *data, int N)
{
   thrust::device_ptr<double> dev_ptr(data);
   thrust::sort(dev_ptr, dev_ptr+N);
}
}
```

Calling Thrust from CUDA Fortran

Fortran interface to C wrapper using ISO C Bindings

```
module thrust

 interface thrustsort

 subroutine sort_int( input,N) bind(C,name="sort_int_wrapper")
  use iso_c_binding
  integer(c_int),device:: input(*)
  integer(c_int),value:: N
 end subroutine

 subroutine sort_float( input,N) bind(C,name="sort_float_wrapper")
  use iso_c_binding
  real(c_float),device:: input(*)
  integer(c_int),value:: N
 end subroutine

 end interface
 end module

subroutine sort_double( input,N) bind(C,name="sort_double_wrapper")
  use iso_c_binding
  real(c_double),device:: input(*)
  integer(c_int),value:: N
 end subroutine
```

CUDA Fortran sorting with Thrust

```
program testsort
use thrust
real, allocatable :: cpuData(:)
real, allocatable, device :: gpuData(:)
integer:: N=10
!Allocate CPU and GPU arrays
  allocate(cpuData(N),gpuData(N))
!Fill the host array with random data
 do i=1,N
   cpuData(i)=random(i)
 end do
! Print unsorted data
  print *, cpuData
! Send data to GPU
 gpuData = cpuData
!Sort the data
 call thrustsort(gpuData,N)
!Copy the result back
 cpuData = gpuData
! Print sorted data
  print *, cpuData
!Deallocate arrays
 deallocate(cpuData,gpuData)
end program testsort
```

```
nvcc -c -arch sm_20 csort.cu
pgf90 -rc=rc4.0 -Mcuda=cc20 -O3 -o testsort thrust_module.cuf testsort.cuf csort.o
```

```
 $ ./testsort
Before sorting 4.1630346E-02 0.9124327 0.7832350 0.6540373 100.0000 0.3956419 0.2664442 0.13724658.0488138E-03 0.8788511
After sorting 8.0488138E-03 4.1630346E-02 0.1372465 0.26644420.3956419 0.6540373 0.7832350 0.87885110.9124327 100.0000
```

CUDA Fortran sorting with Thrust

```
program timesort
use cudafor
use thrust
real, allocatable :: cpuData(:)
real, allocatable, device :: gpuData(:)
integer:: N=100000000,istat
! cuda events for elapsing
type ( cudaEvent ) :: startEvent , stopEvent
real :: time, random
!Allocate CPU and GPU arrays
  allocate(cpuData(N),gpuData(N))
!Fill the host array with random data
 do i=1,N
   cpuData(i)=random(i)
 end do
```

```
! Create events
istat = cudaEventCreate ( startEvent )
istat = cudaEventCreate ( stopEvent )
! Send data to GPU
gpuData = cpuData
!Sort the data
istat = cudaEventRecord ( startEvent , 0)
 call thrustsort(gpuData,N)
istat = cudaEventRecord ( stopEvent , 0)
istat = cudaEventSynchronize ( stopEvent )
istat = cudaEventElapsedTime ( time , startEvent , stopEvent )
!Copy the result back
cpuData = gpuData
print *," Sorted array in:",time," (ms)"
print *,"After sorting", cpuData(1:5),cpuData(N-4:N)
!Deallocate arrays
 deallocate(cpuData,gpuData)
end program timesort
```

```
$ ./timesort
Sorting array of     100000000  single precision
Sorted array in:    194.6642      (ms)
After sorting   7.0585919E-09   1.0318221E-08   1.9398616E-08   3.1738640E-08  4.4078664E-08   0.9999999       0.9999999       1.000000     1.000000       1.000000
```

© NVIDIA Corporation 2011

Outline

- **Kernel optimizations**
  - Launch configuration
  - Global memory throughput
  - Shared memory access
  - Instruction throughput / control flow
- **Optimization of CPU-GPU interaction**
  - Maximizing PCIe throughput
  - Overlapping kernel execution with memory copies

© NVIDIA Corporation 2011

Launch Configuration

© NVIDIA Corporation 2011

Launch Configuration

- **Key to understanding:**
  - Instructions are issued in order
  - A thread stalls when one of the operands isn't ready:
    - Memory read by itself doesn't stall execution
  - Latency is hidden by switching threads
    - GMEM latency: 400-800 cycles
    - Arithmetic latency: 18-22 cycles
- **How many threads/threadblocks to launch?**
- **Conclusion:**
  - Need enough threads to hide latency

© NVIDIA Corporation 2011

Launch Configuration

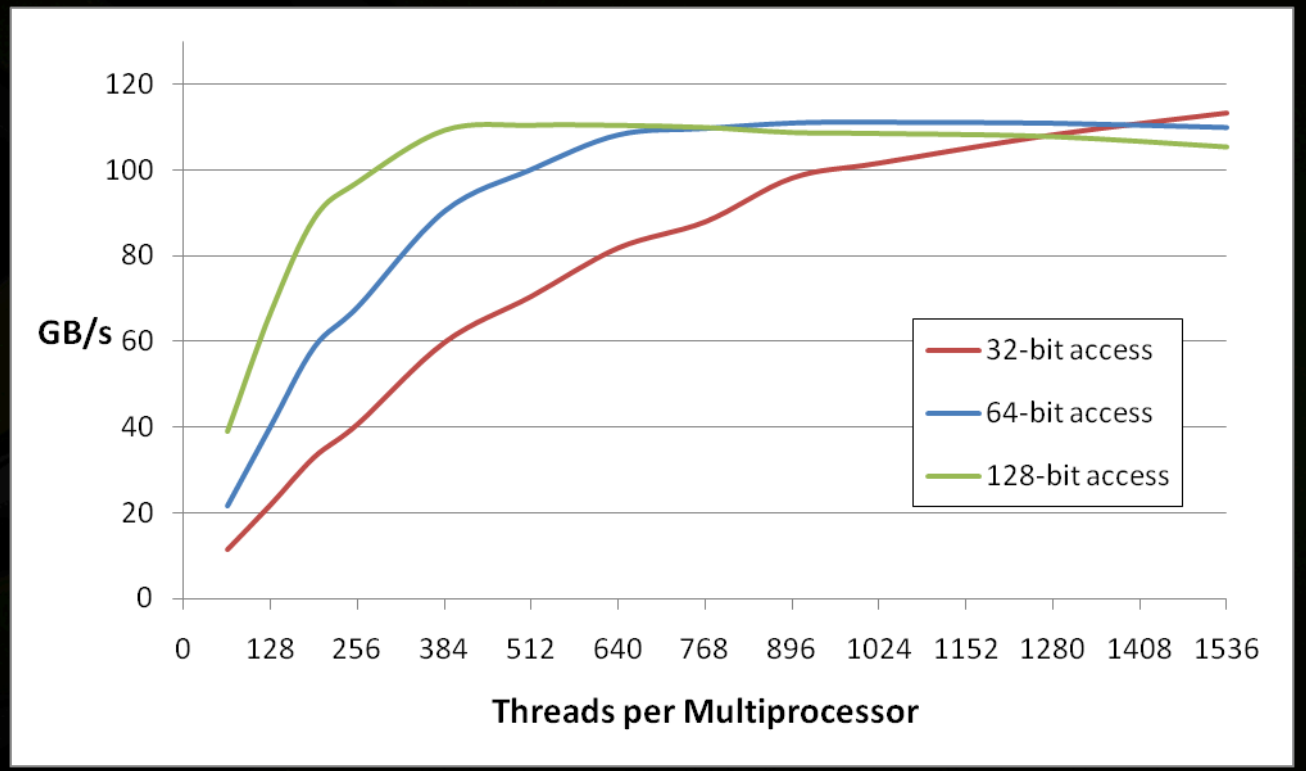
- **Hiding arithmetic latency:**
  - Need ~18 warps (576) threads per SM
  - Or, latency can also be hidden with independent instructions from the same warp
    - For example, if instruction never depends on the output of preceding instruction, then only 9 warps are needed, etc.

- **Maximizing global memory throughput:**
  - Depends on the access pattern, and word size
  - Need enough memory transactions in flight to saturate the bus
    - Independent loads and stores from the same thread
    - Loads and stores from different threads
    - Larger word sizes can also help (float2 is twice the transactions of float, for example)

© NVIDIA Corporation 2011

Maximizing Memory Throughput

- **Increment of an array of 64M elements**
  - Two accesses per thread (load then store)
  - The two accesses are dependent, so really 1 access per thread at a time
- **Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s**



Several independent smaller accesses have the same effect as one larger one.

For example:

Four 32-bit ~= one 128-bit

© NVIDIA Corporation 2011

Launch Configuration: Summary

- **Need enough total threads to keep GPU busy**
  - Typically, you'd like 512+ threads per SM
    - More if processing one fp32 element per thread
  - Of course, exceptions exist
- **Threadblock configuration**
  - Threads per block should be a multiple of warp size (32)
  - SM can concurrently execute up to 8 threadblocks
    - Really small threadblocks prevent achieving good occupancy
    - Really large threadblocks are less flexible
    - I generally use 128-256 threads/block, but use whatever is best for the application
- **For more details:**
  - Vasily Volkov's GTC2010 talk "Better Performance at Lower Occupancy" (http://www.gputechconf.com/page/gtc-on-demand.html#session2238)

© NVIDIA Corporation 2011

Global Memory Throughput

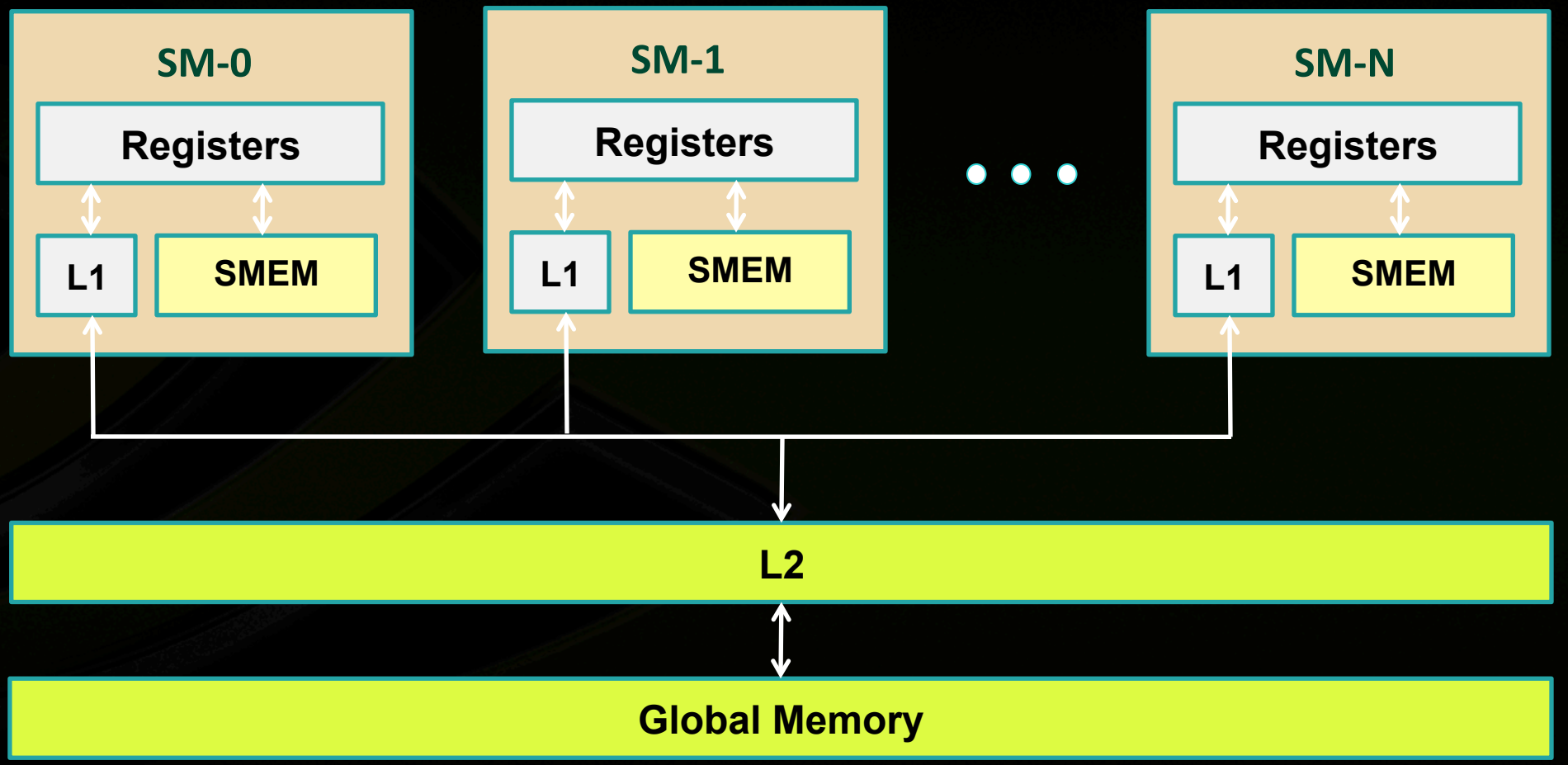
© NVIDIA Corporation 2011

Memory Hierarchy Review

- **Local storage**
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
- **Shared memory / L1**
  - Program configurable: 16KB shared / 48 KB L1 OR 48KB shared / 16KB L1
  - Shared memory is accessible by the threads in the same threadblock
  - Very low latency
  - Very high throughput: 1+ TB/s aggregate
- **L2**
  - All accesses to global memory go through L2, including copies to/from CPU host
- **Global memory**
  - Accessible by all threads as well as host (CPU)
  - High latency (400-800 cycles)
  - Throughput: up to 177 GB/s

© NVIDIA Corporation 2011

Fermi Memory Hierarchy Review



© NVIDIA Corporation 2011

GMEM Operations

- **Two types of loads:**
  - Caching
    - Default mode
    - Attempts to hit in L1, then L2, then GMEM
    - Load granularity is 128-byte line
  - Non-caching
    - Compile with *–Xptxas –dlcm=cg* option to nvcc
    - Attempts to hit in L2, then GMEM
      - Do not hit in L1, invalidate the line if it's in L1 already
    - Load granularity is 32-bytes
- **Stores:**
  - Invalidate L1, write-back for L2

© NVIDIA Corporation 2011

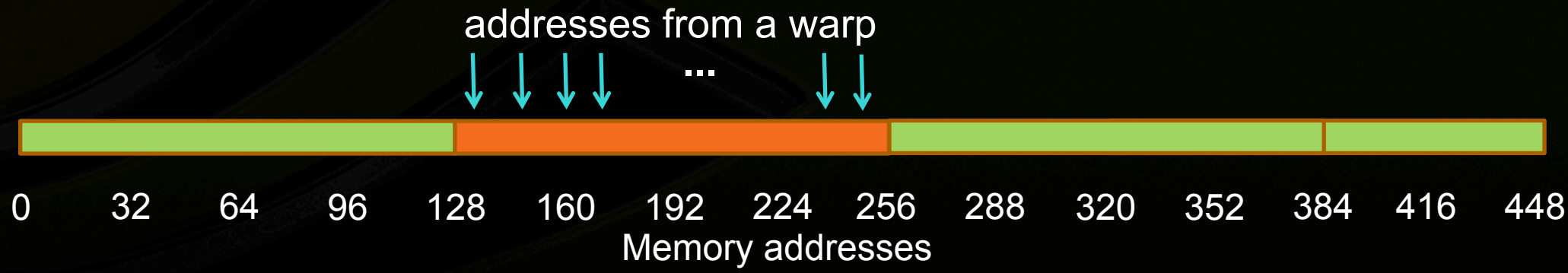
Load Operation

- **Memory operations are issued per warp (32 threads)**
  - Just like all other instructions
- **Operation:**
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

© NVIDIA Corporation 2011

Caching Load

- **Warp requests 32 aligned, consecutive 4-byte words**
- **Addresses fall within 1 cache-line**
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



© NVIDIA Corporation 2011

Non-caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

© NVIDIA Corporation 2011

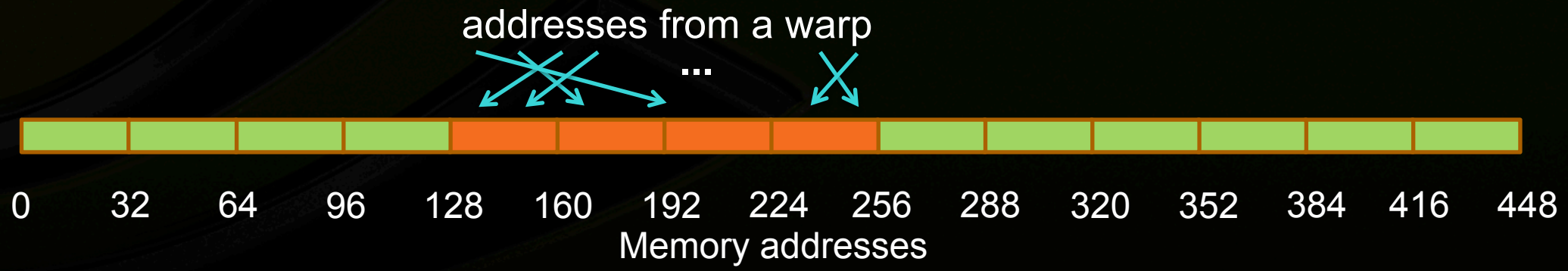
Caching Load

- **Warp requests 32 aligned, permuted 4-byte words**
- **Addresses fall within 1 cache-line**
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

© NVIDIA Corporation 2011

Non-caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 4 segments
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%



© NVIDIA Corporation 2011

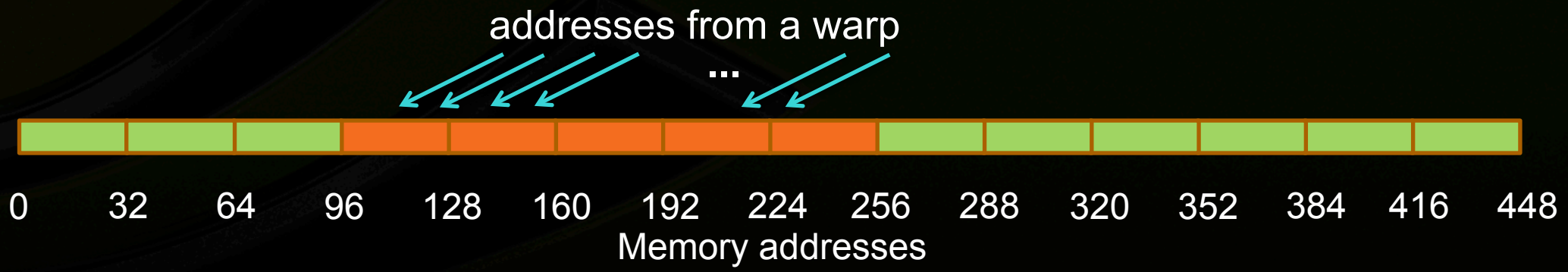
Caching Load

- **Warp requests 32 misaligned, consecutive 4-byte words**
- **Addresses fall within 2 cache-lines**
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%

© NVIDIA Corporation 2011

Non-caching Load

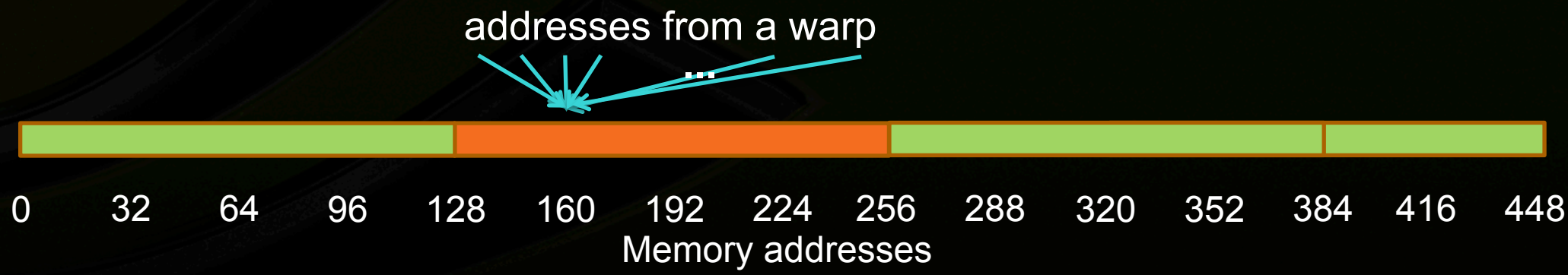
- **Warp requests 32 misaligned, consecutive 4-byte words**
- **Addresses fall within at most 5 segments**
  - Warp needs 128 bytes
  - 160 bytes move across the bus on misses
  - Bus utilization: at least 80%
    - Some misaligned patterns will fall within 4 segments, so 100% utilization



© NVIDIA Corporation 2011

Caching Load

- **All threads in a warp request the same 4-byte word**
- **Addresses fall within a single cache-line**
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%



© NVIDIA Corporation 2011

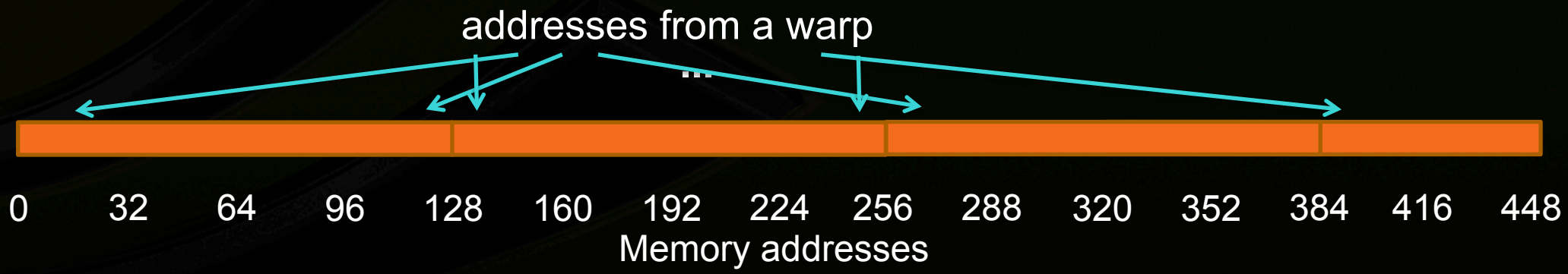
Non-caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single segment
  - Warp needs 4 bytes
  - 32 bytes move across the bus on a miss
  - Bus utilization: 12.5%

© NVIDIA Corporation 2011

Caching Load

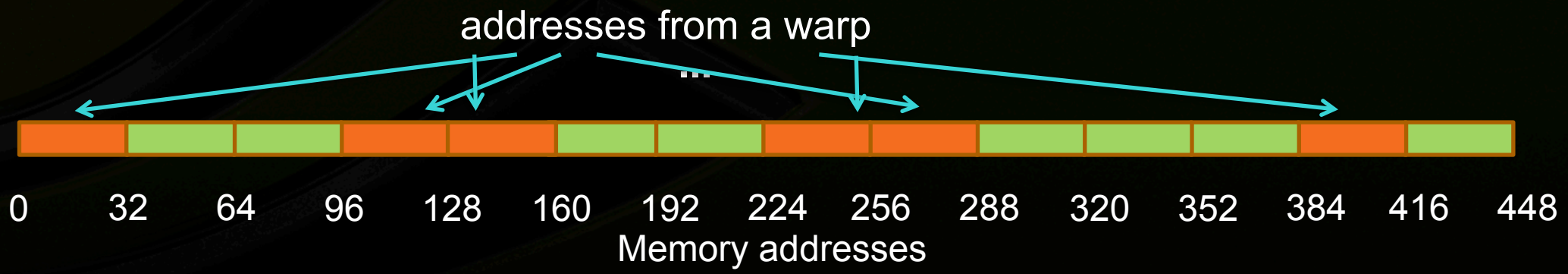
- **Warp requests 32 scattered 4-byte words**
- **Addresses fall within *N* cache-lines**
  - Warp needs 128 bytes
  - *N**128 bytes move across the bus on a miss
  - Bus utilization: 128 / (*N**128)



© NVIDIA Corporation 2011

Non-caching Load

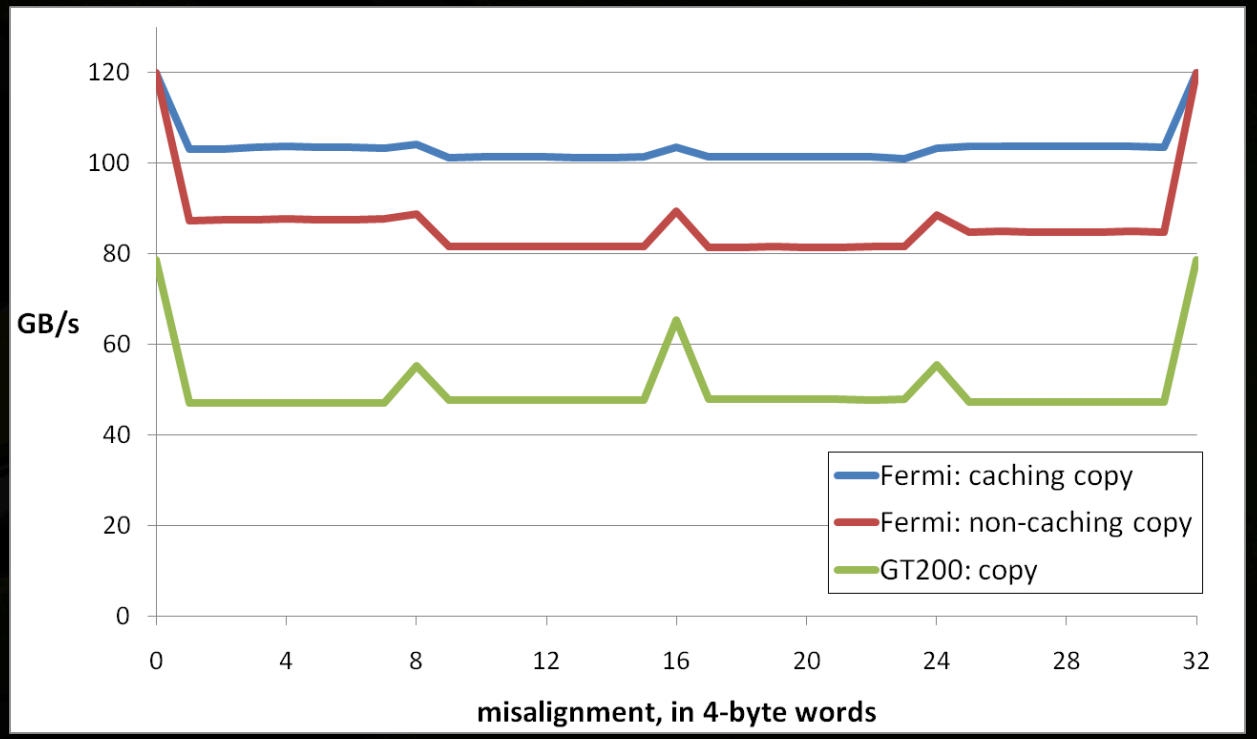
- **Warp requests 32 scattered 4-byte words**
- **Addresses fall within *N* segments**
  - Warp needs 128 bytes
  - *N**32 bytes move across the bus on a miss
  - Bus utilization: 128 / (*N**32)



© NVIDIA Corporation 2011

Impact of Address Alignment

- **Warps should access aligned regions for maximum memory throughput**
  - L1 can help for misaligned loads if several warps are accessing a contiguous region
  - ECC further significantly reduces misaligned store throughput



Experiment:

- Copy 16MB of floats
- 256 threads/block

Greatest throughput drop:

- CA loads: 15%
- CG loads: 32%

© NVIDIA Corporation 2011

GMEM Optimization Guidelines

- **Strive for perfect coalescing**
  - Align starting address (may require padding)
  - A warp should access within a contiguous region
- **Have enough concurrent accesses to saturate the bus**
  - Process several elements per thread
    - Multiple loads get pipelined
    - Indexing calculations can often be reused
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)
- **Try L1 and caching configurations to see which one works best**
  - Caching vs non-caching loads (compiler option)
  - 16KB vs 48KB L1 (CUDA call)

© NVIDIA Corporation 2011

Shared Memory

© NVIDIA Corporation 2011

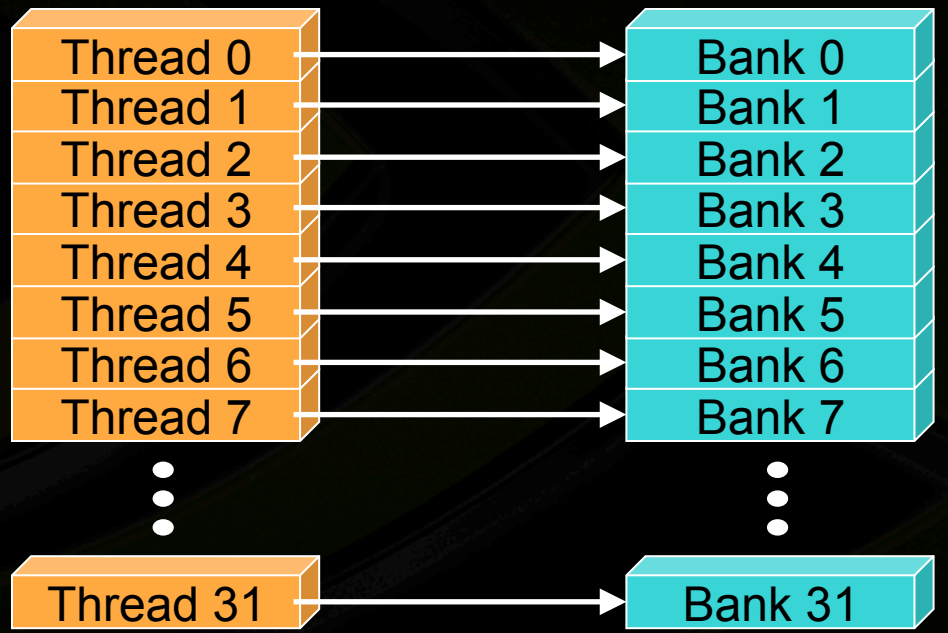
Shared Memory

- **Uses:**
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- **Organization:**
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks
- **Performance:**
  - 4 bytes per bank per 2 clocks per multiprocessor
  - smem accesses are issued per 32 threads (warp)
  - serialization: if *n* threads of 32 access different 4-byte words in the same bank, *n* accesses are executed serially
  - multicast: *n* threads access the same word in one fetch
    - Could be different bytes within the same word

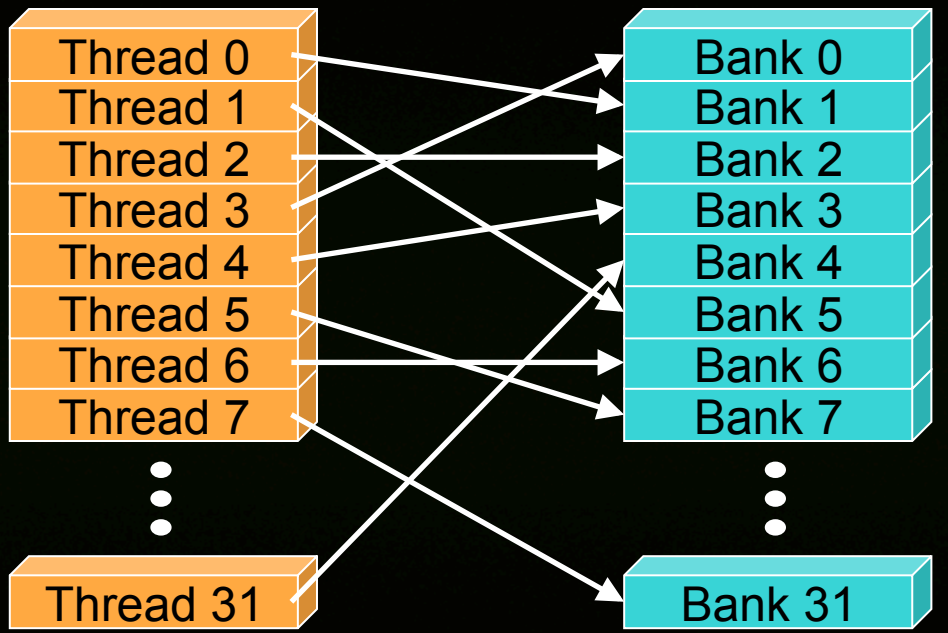
© NVIDIA Corporation 2011

Bank Addressing Examples

- No Bank Conflicts

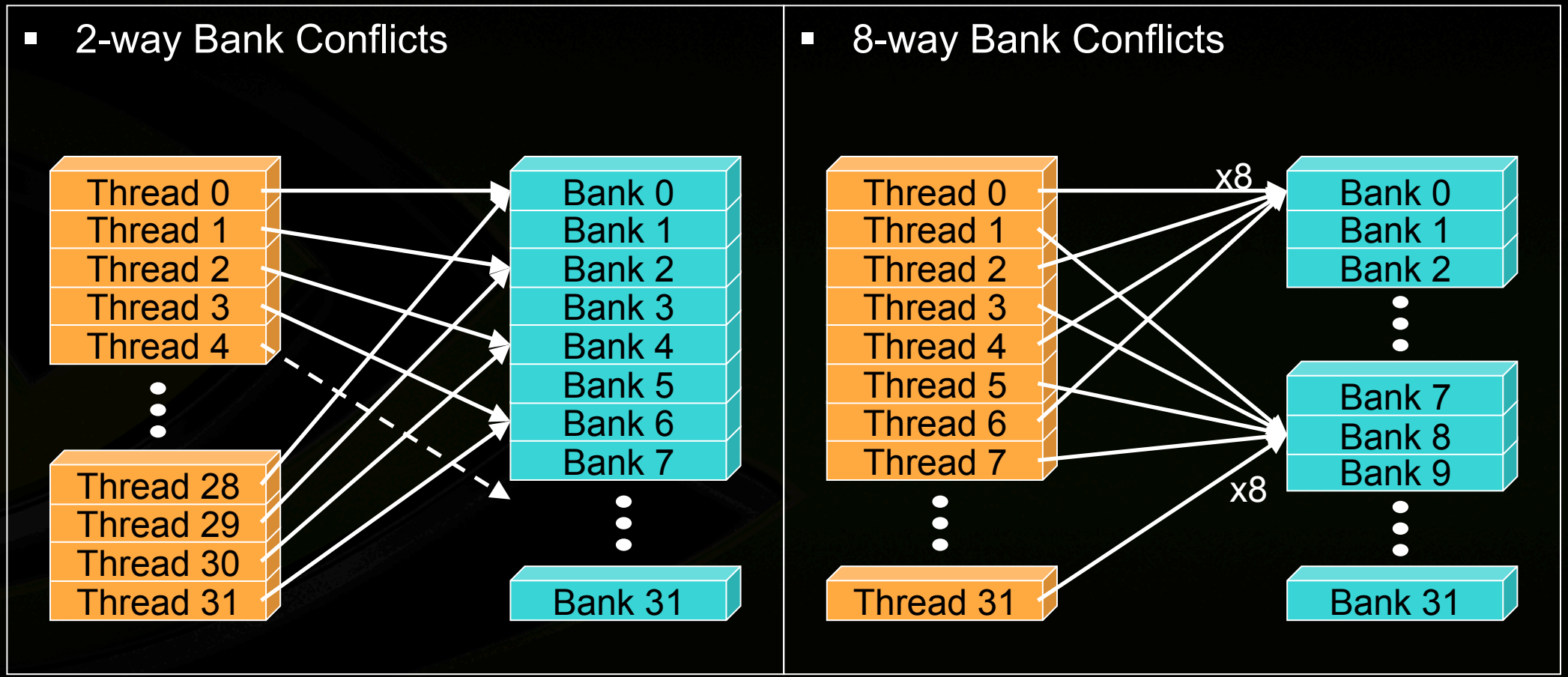


- No Bank Conflicts



© NVIDIA Corporation 2011

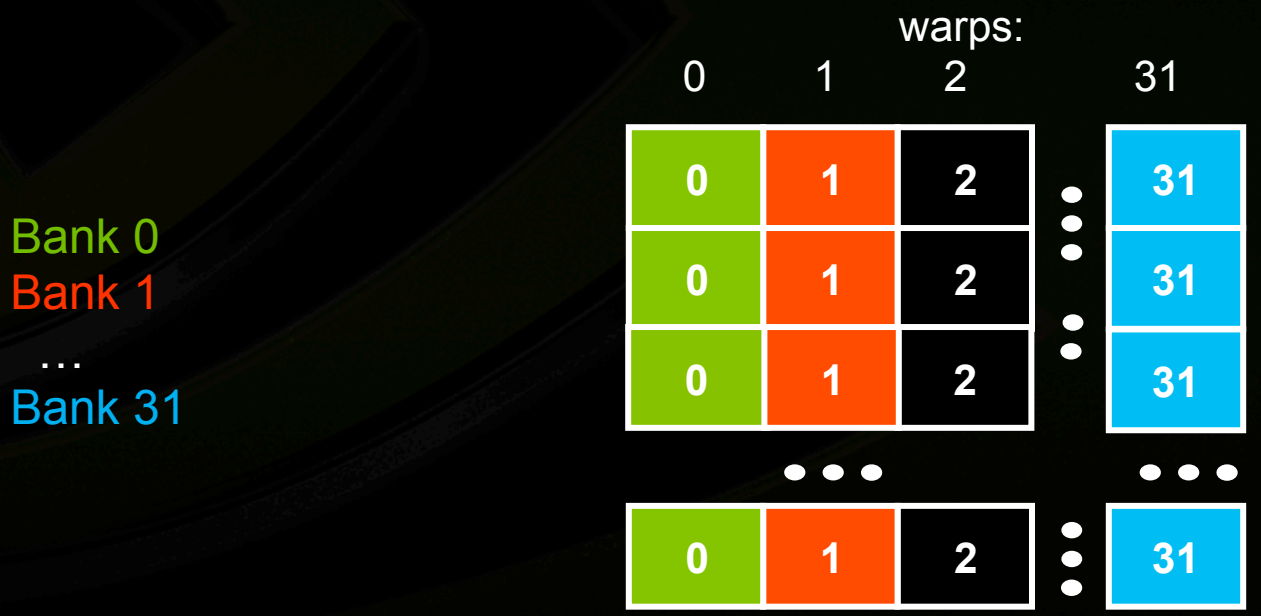
Bank Addressing Examples



© NVIDIA Corporation 2011

Shared Memory: Avoiding Bank Conflicts

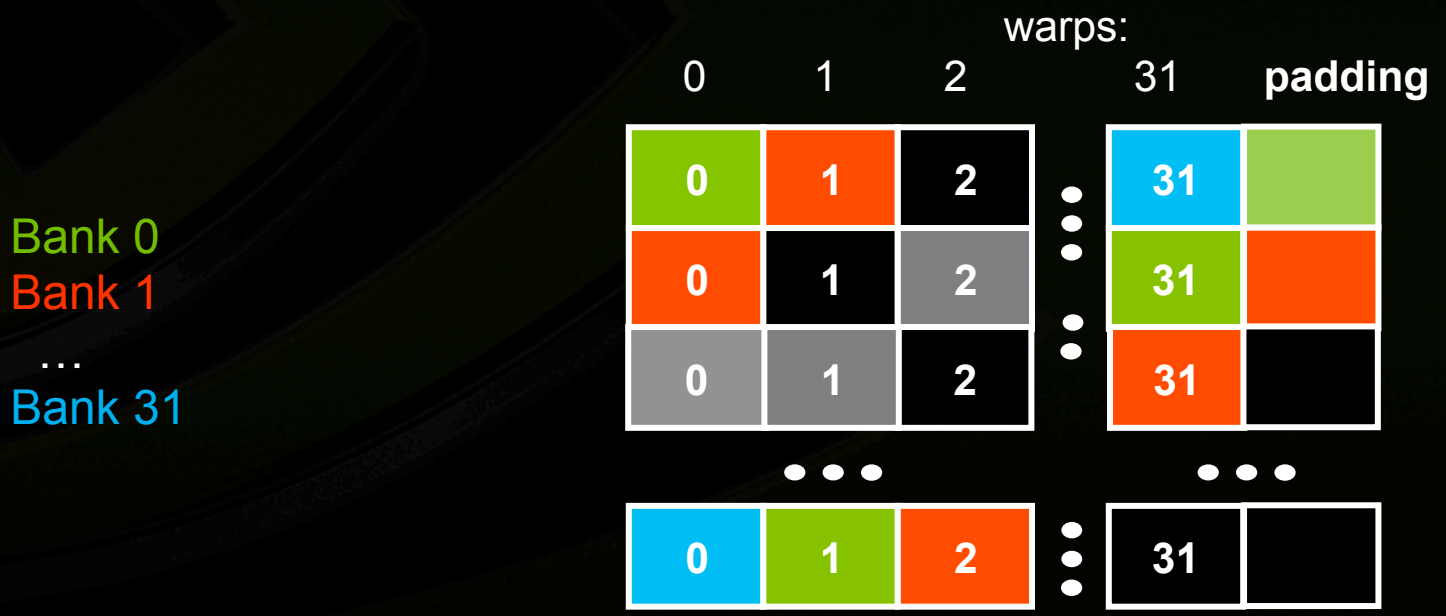
- 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank)



© NVIDIA Corporation 2011

Shared Memory: Avoiding Bank Conflicts

- **Add a column for padding:**
  - 32x33 SMEM array
- **Warp accesses a column:**
  - 32 different banks, no bank conflicts



© NVIDIA Corporation 2011

Additional “memories”

- *Texture* and *constant*
- Read-only
- Data resides in global memory
- Read through different caches

© NVIDIA Corporation 2011

Texture

- **Separate cache**
- **Dedicated texture cache hardware provides:**
  - Out-of-bounds index handling
    - clamp or wrap-around
  - Optional interpolation
    - Think: using fp indices for arrays
    - Linear, bilinear, trilinear
      - Interpolation weights are 9-bit
  - Optional format conversion
    - {char, short, int} -> float
  - All of these are “free”

© NVIDIA Corporation 2011

Instruction Throughput / Control Flow

© NVIDIA Corporation 2011

Runtime Math Library and Intrinsics

- **Two types of runtime math library functions**
  - __func(): many map directly to hardware ISA
    - Fast but lower accuracy (see CUDA Programming Guide for full details)
    - Examples: __sinf(x), __expf(x), __powf(x, y)
  - func(): compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x, y)
- **A number of additional intrinsics:**
  - __sincosf(), __frcp_rz(), ...
  - Explicit IEEE rounding modes (rz,rn,ru,rd)

© NVIDIA Corporation 2011

Control Flow

- **Instructions are issued per 32 threads (warp)**
- **Divergent branches:**
  - Threads within a single warp take different paths
    - `if-else`, ...
  - Different execution paths within a warp are serialized
- **Different warps can execute different code with no impact on performance**
- **Avoid diverging within a warp**
  - Example with divergence:
    - `if (threadIdx.x > 2) {...} else {...}`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) {...} else {...}`
    - Branch granularity is a whole multiple of warp size

© NVIDIA Corporation 2011

CPU-GPU Interaction

© NVIDIA Corporation 2011

Pinned (non-pageable) memory

- **Pinned memory enables:**
  - faster PCIe copies
  - memcopies asynchronous with CPU
  - memcopies asynchronous with GPU
- **Usage**
  - cudaHostAlloc / cudaFreeHost
    - instead of malloc / free
  - cudaHostRegister / cudaHostUnregister
    - pin regular memory after allocation
- **Implication:**
  - pinned memory is essentially removed from host virtual memory

© NVIDIA Corporation 2011

Streams and Async API

- **Default API:**
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- **Streams and async functions provide:**
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
- **Stream = sequence of operations that execute in issue-order on GPU**
  - Operations from different streams may be interleaved
  - A kernel and memcopy from different streams can be overlapped

© NVIDIA Corporation 2011

Overlap kernel and memory copy

- **Requirements:**
  - D2H or H2D memcopy from pinned memory
  - Kernel and memcopy in different, non-0 streams
- **Code:**

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

potentially overlapped

© NVIDIA Corporation 2011

Call Sequencing for Optimal Overlap

- **CUDA calls are dispatched to the hw in the sequence they were issued**
- **Fermi can concurrently execute:**
  - Up to 16 kernels
  - Up to 2 memcopies, as long as they are in different directions (D2H and H2D)
- **A call is dispatched if both are true:**
  - Resources are available
  - Preceding calls in the same stream have completed
- **Scheduling:**
  - Kernels are executed in the order in which they were issued
  - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available
- **Note that if a call blocks, it blocks all other calls of the same type behind it, even in other streams**
  - Type is one of { kernel, memcopy}

© NVIDIA Corporation 2011

Stream Examples (current HW)

K:
kernel
M:
memcopy
Integer: stread ID

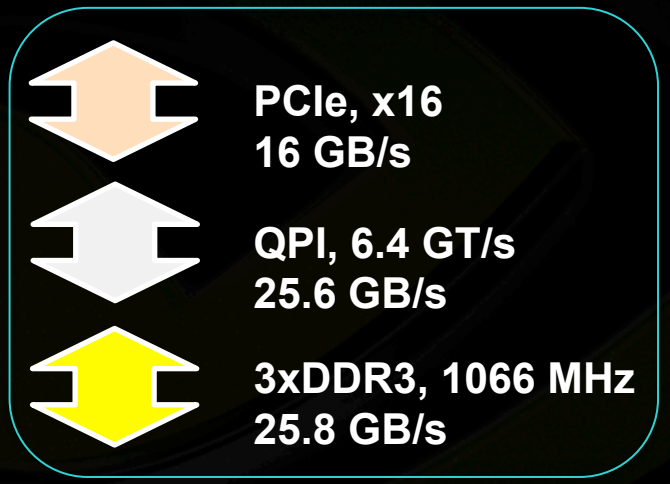
© NVIDIA Corporation 2011

More on Dual Copy

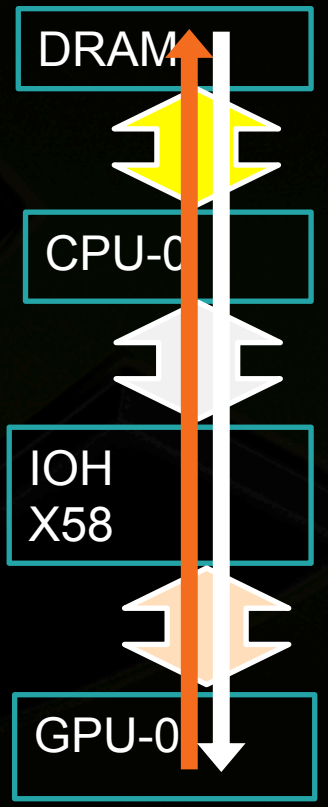
- **Fermi is capable of duplex communication with the host**
  - PCIe bus is duplex
  - The two memcopies must be in different streams, different directions
- **Not all current host systems can saturate duplex PCIe bandwidth:**
  - Likely issues with IOH chips
  - If this is important to you, test your host system

© NVIDIA Corporation 2011

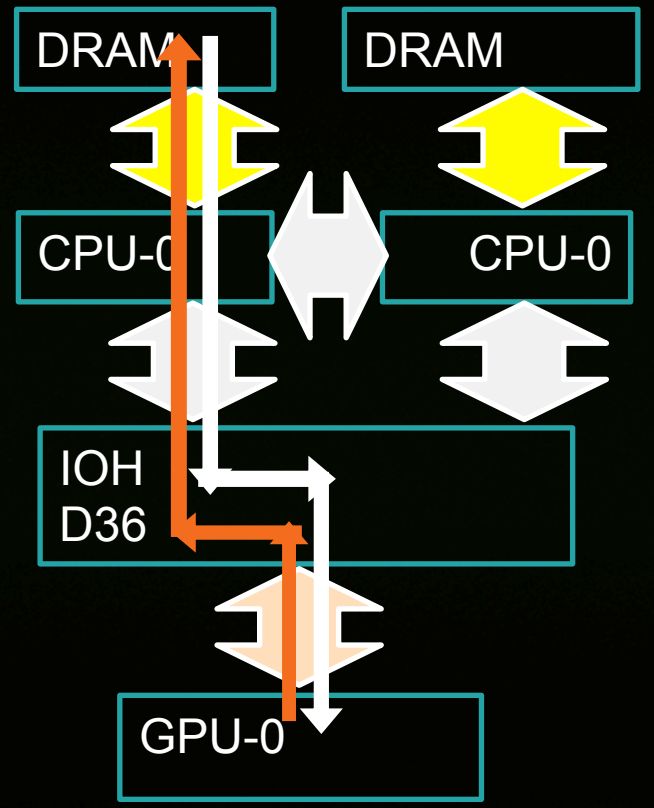
Duplex Copy: Experimental Results



10.8 GB/s

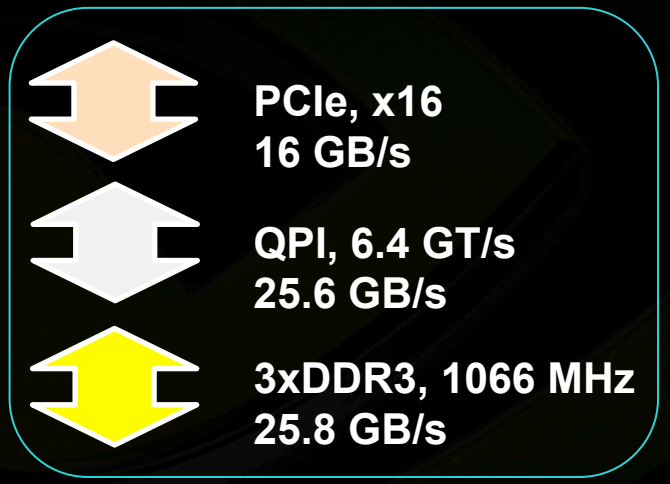


7.5 GB/s

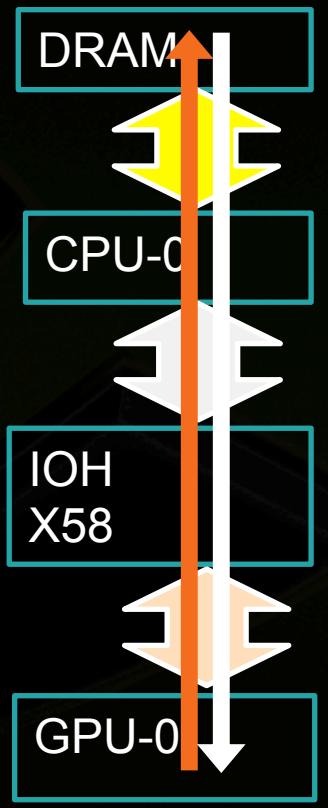


© NVIDIA Corporation 2011

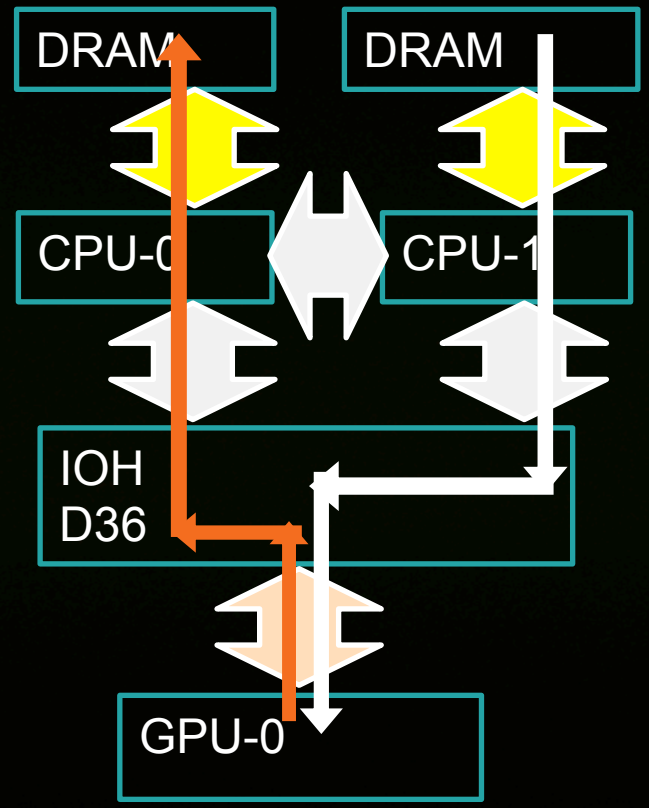
Duplex Copy: Experimental Results



10.8 GB/s



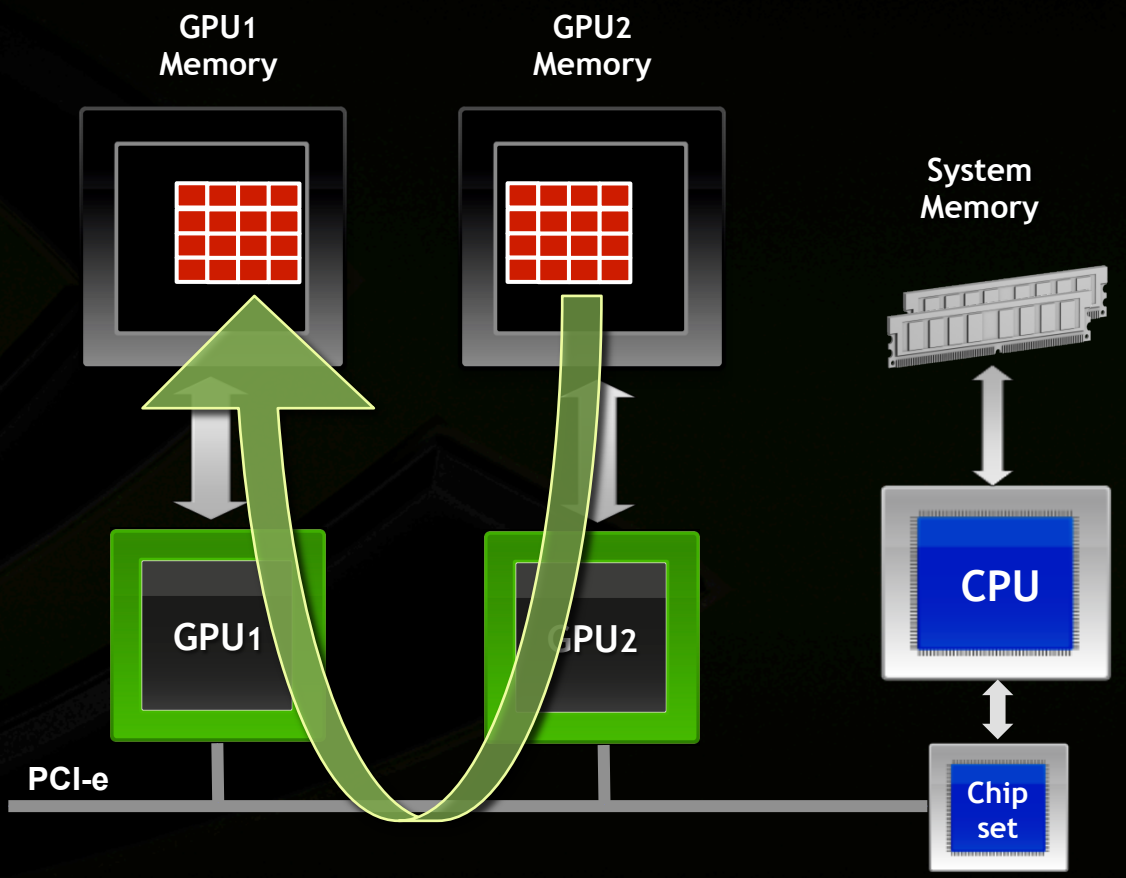
11 GB/s



© NVIDIA Corporation 2011

GPUDirect v2.0: Peer-to-Peer Communication

Direct Transfers b/w GPUs

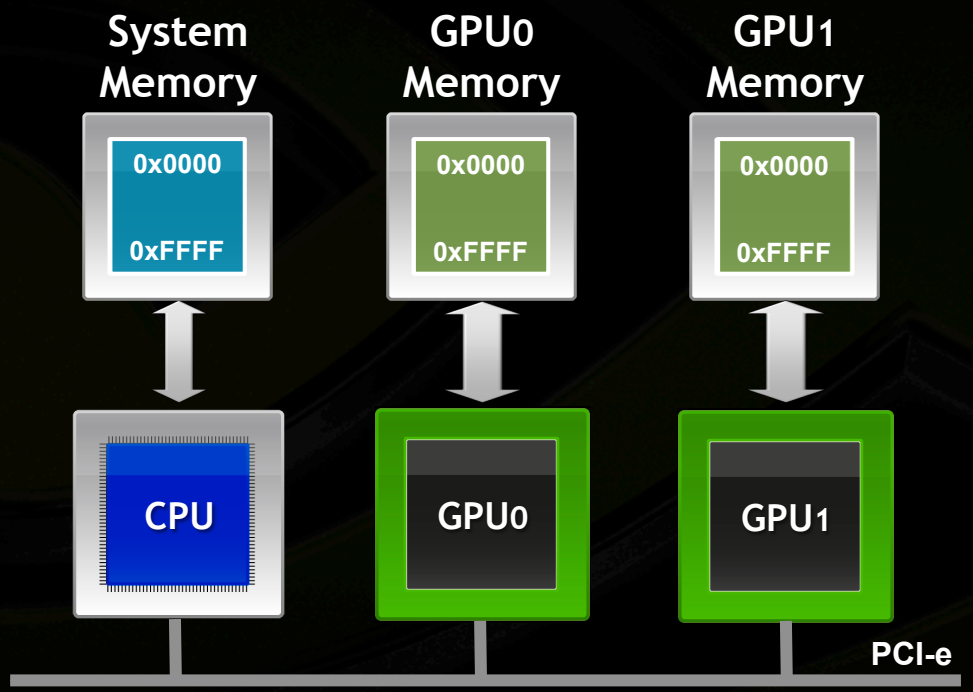


© NVIDIA Corporation 2011

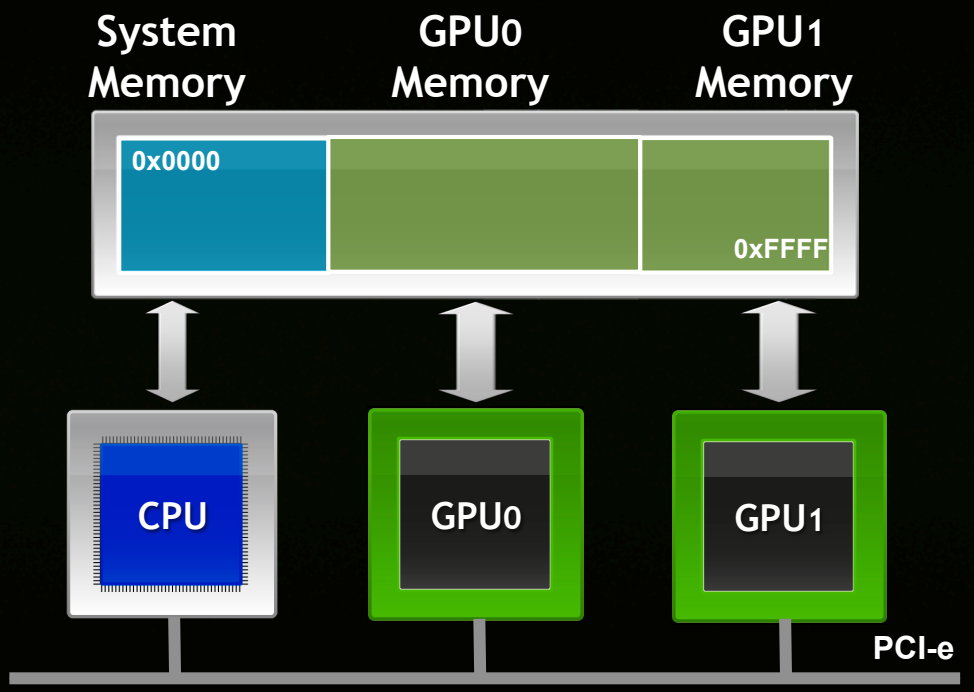
Unified Virtual Addressing

Easier to Program with Single Address Space

No UVA: Multiple Memory Spaces



UVA : Single Address Space

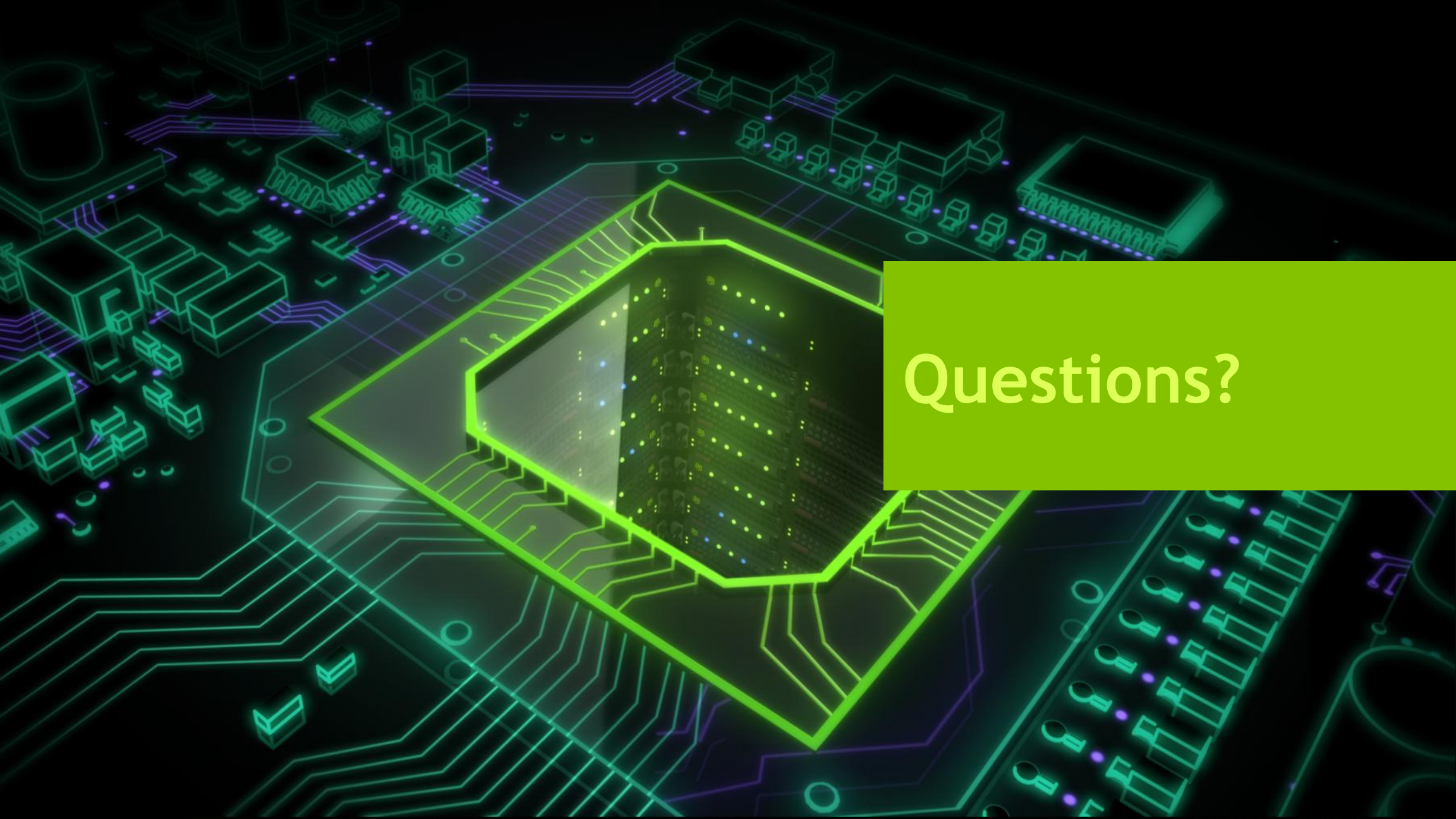


© NVIDIA Corporation 2011

Summary

- **Kernel Launch Configuration:**
  - Launch enough threads per SM to hide latency
  - Launch enough threadblocks to load the GPU
- **Global memory:**
  - Maximize throughput (GPU has lots of bandwidth, use it effectively)
- **Use shared memory when applicable (over 1 TB/s bandwidth)**
- **GPU-CPU interaction:**
  - Minimize CPU/GPU idling, maximize PCIe throughput
- **Use analysis/profiling when optimizing:**
  - “Analysis-driven Optimization” part of the tutorial following

© NVIDIA Corporation 2011


Questions?

Analysis-Driven Optimization

ISC 2011 Tutorial

Gernot Ziegler, NVIDIA Corporation

© NVIDIA Corporation 2011

Performance Optimization Process

- **Determine the limits for kernel performance**
  - Memory throughput
  - Instruction throughput
  - Latency
  - Combination of the above
- **Use appropriate performance metric for each kernel**
  - For example, for a memory bandwidth-bound kernel, Gflops/s don't make sense
- **Address the limiters in the order of importance**
  - Determine how close resource usage is to the HW limits
  - Analyze for possible inefficiencies
  - Apply optimizations
    - Often these will just be obvious from how HW operates

© NVIDIA Corporation 2011

Presentation Outline

- **Identifying performance limiters**
- **Analyzing and optimizing :**
  - Memory-bound kernels
  - Instruction (math) bound kernels
  - Kernels with poor latency hiding
  - Register spilling
- **For each:**
  - Brief background
  - How to analyze
  - How to judge whether particular issue is problematic
  - How to optimize
  - Some cases studies based on “real-life” application kernels
- **Most information is for Fermi GPUs**

© NVIDIA Corporation 2011

Notes on profiler

- **Most counters are reported per Streaming Multiprocessor (SM)**
  - Not entire GPU
  - Exceptions: L2 and DRAM counters
- **A single run can only collect a few counters**
  - Multiple runs are needed when profiling more counters
    - Done automatically by the Visual Profiler
    - Have to be done manually using command-line profiler
    - Use CUPTI API to have your application collect signals on its own
- **Counter values may not be exactly the same for repeated runs**
  - Threadblocks and warps are scheduled at run-time
  - So, “two counters being equal” usually means “two counters within a small delta”
- **See the profiler documentation for more information**

© NVIDIA Corporation 2011

Identifying Performance Limiters

© NVIDIA Corporation 2011

Limited by Bandwidth or Arithmetic?

- **Perfect fp32 instructions:bytes ratio for Fermi C2050:**
  - ~4.5 : 1 instructions/byte with ECC on
  - ~3.6 : 1 instructions/byte with ECC off
  - These assume fp32 instructions, throughput for other instructions varies
- **Algorithmic analysis:**
  - Rough estimate of arithmetic to bytes ratio
- **Code likely uses more instructions and bytes than algorithm analysis suggests:**
  - Instructions for loop control, pointer math, etc.
  - Address pattern may result in more memory fetches
  - Two ways to investigate:
    - Use the profiler (quick, but approximate)
    - Use source code modification (more accurate, more work intensive)

© NVIDIA Corporation 2011

Analysis with Profiler

- **Profiler counters:**
  - instructions_issued, instructions_executed
    - Both incremented by 1 per warp
    - "issued" includes instruction replays (instruction re-issue), "executed" does not
  - gld_request, gst_request
    - Incremented by 1 per warp for each gmem load/store instruction
    - Instruction may be counted if it is "predicated out"
  - l1_global_load_miss, l1_global_load_hit, global_store_transaction
    - Incremented by 1 per **L1 line** (line is 128B)
  - uncached_global_load_transaction
    - Incremented by 1 per group of 1, 2, 3, or 4 transactions
    - Better to look at L2_read_request counter (incremented by 1 per 32 bytes, per GPU)
- **For comparisons:**
  - 32 * instructions_issued /* 32 = warp size */
  - 128B * (global_store_transaction + l1_global_load_miss)

© NVIDIA Corporation 2011

A Note on Counting Global Memory Accesses

- **Load/store instruction count can be lower than the number of actual memory transactions**
  - Address pattern, different word sizes
- **Hence: Counting requests from L1 to the rest of the memory system makes the most sense**
  - Caching-loads: count L1 misses
  - Non-caching loads and stores: count L2 read requests
    - Note that L2 counters are for the entire chip, L1 counters are per SM
- **Assuming "coalesced" address patterns, some shortcuts:**
  - One 32-bit access instruction -> one 128-byte transaction per warp
  - One 64-bit access instruction -> two 128-byte transactions per warp
  - One 128-bit access instruction -> four 128-byte transactions per warp

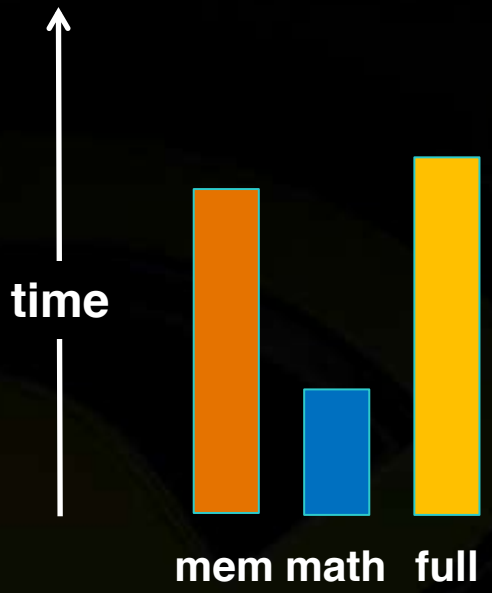
© NVIDIA Corporation 2011

Analysis with Modified Source Code

- **Time memory-only and math-only versions of the kernel**
  - Easier for codes that don't have data-dependent control-flow or addressing
  - Gives you good estimates for:
    - Time spent accessing memory
    - Time spent in executing instructions
- **Then, compare times for modified kernels**
  - Helps decide whether the kernel is mem or math bound
  - Shows how well memory operations are overlapped with arithmetic
    - Compare the sum of mem-only and math-only times to full-kernel time

© NVIDIA Corporation 2011

Some Example Scenarios



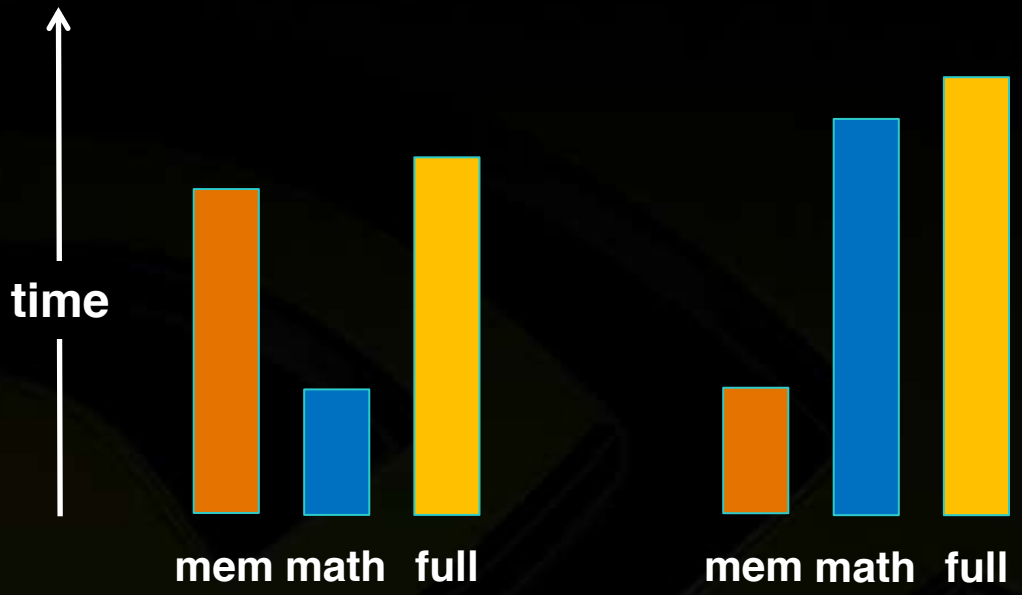
Memory-bound

Good mem-math overlap: latency not a problem

(assuming memory throughput is not low compared to HW theory)

© NVIDIA Corporation 2011

Some Example Scenarios



Memory-bound

Good mem-math overlap: latency not a problem

(assuming memory throughput is not low compared to HW theory)

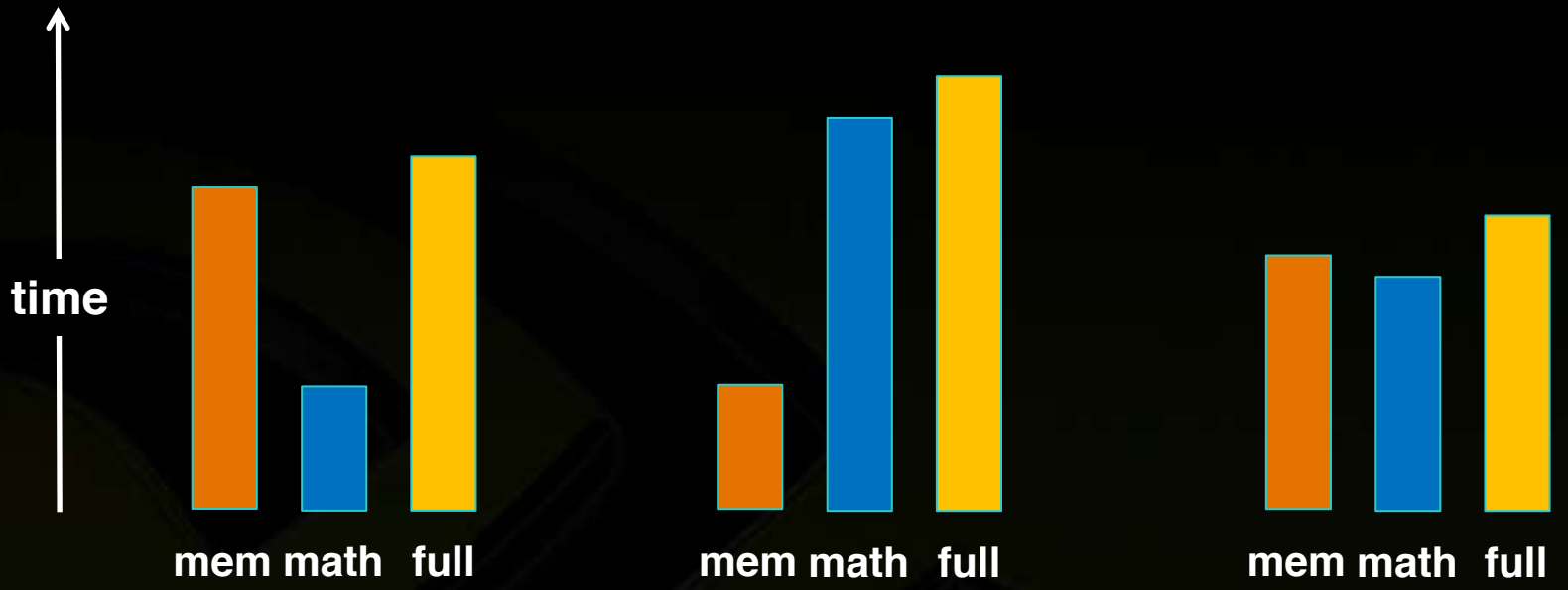
Math-bound

Good mem-math overlap: latency not a problem

(assuming instruction throughput is not low compared to HW theory)

© NVIDIA Corporation 2011

Some Example Scenarios



Memory-bound

Good mem-math overlap: latency not a problem

(assuming memory throughput is not low compared to HW theory)

Math-bound

Good mem-math overlap: latency not a problem

(assuming instruction throughput is not low compared to HW theory)

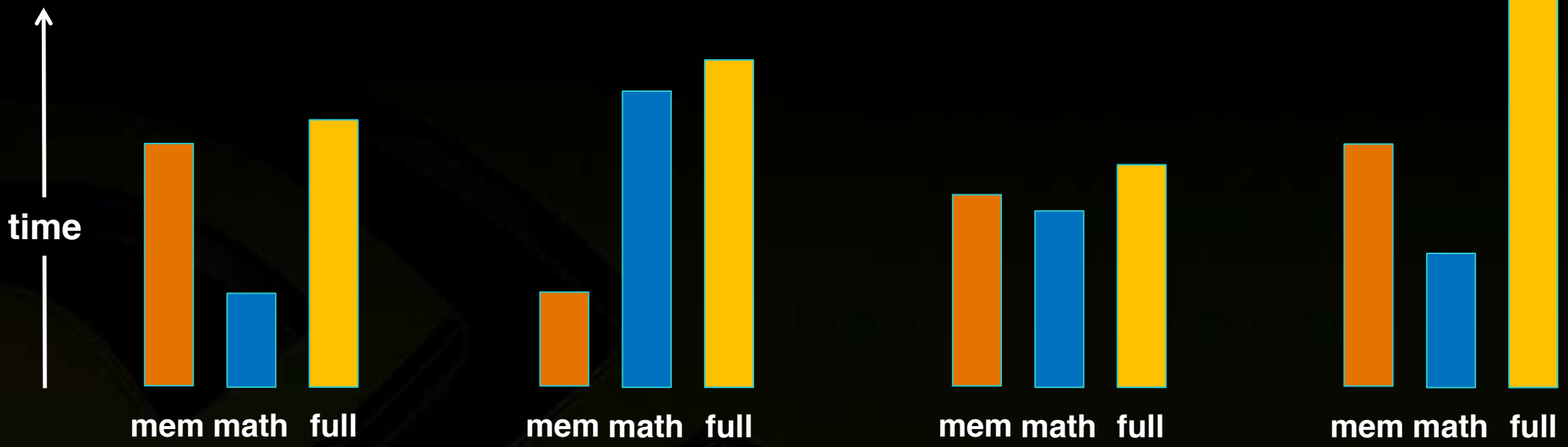
Balanced

Good mem-math overlap: latency not a problem

(assuming memory/instr throughput is not low compared to HW theory)

© NVIDIA Corporation 2011

Some Example Scenarios



Memory-bound

Good mem-math overlap: latency not a problem

(assuming memory throughput is not low compared to HW theory)

Math-bound

Good mem-math overlap: latency not a problem

(assuming instruction throughput is not low compared to HW theory)

Balanced

Good mem-math overlap: latency not a problem

(assuming memory/instr throughput is not low compared to HW theory)

Memory and latency bound

Poor mem-math overlap: Latency is a problem

© NVIDIA Corporation 2011

Source Modification

- **Memory-only:**
  - Remove as much arithmetic as possible
    - Without changing access pattern
    - Use the profiler to verify that load/store instruction count is the same
- **Store-only:**
  - Also remove the loads
- **Math-only:**
  - Remove global memory accesses
  - Need to trick the compiler:
    - Compiler throws away all code that it detects as not contributing to gmem stores
    - Put gmem stores inside conditionals that always evaluate to `false`
      - Condition outcome should not be known to the compiler (kernel parameter)
      - Condition should depend on the value about to be stored (prevents other optimizations)

© NVIDIA Corporation 2011

Source Modification for Math-only

- Condition outcome should not be known to the compiler
- Condition should depend on the value about to be stored (prevents other optimizations)

```
__global__ void fwd_3D( ..., int flag)
{
            ...
   value = temp + coeff * vsq;
   if( 1 == value * flag )
      g_output[out_idx] = value;
}
```

If you compare **only** the flag, then the compiler may **move the computation into the conditional** as well

© NVIDIA Corporation 2011

Source Modification and Occupancy

- **Removing pieces of code is likely to affect register count**
  - This could increase GPU occupancy, skewing the results
  - See slide 23 to see how that could affect throughput
- **Make sure to keep the same occupancy**
  - Check the occupancy with profiler before modifications
  - After modifications, if necessary add dummy shared memory to match the unmodified kernel's GPU occupancy

```
kernel<<< grid, block, smem, ...>>>(...)
```

© NVIDIA Corporation 2011

Case Study: Limiter Analysis

- 3DFD of the wave equation, fp32
- Time (ms):
  - Full-kernel: 35.39
  - Mem-only: 33.27
  - Math-only: 16.25
- Instructions issued:
  - Full-kernel: 18,194,139
  - Mem-only: 7,497,296
  - Math-only: 16,839,792
- Memory access transactions:
  - Full-kernel: 1,708,032
  - Mem-only: 1,708,032
  - Math-only: 0

- Analysis:
  - Instruction:Byte ratio = ~2.66
    - 32*18,194,139 / 128*1,708,032
  - Good overlap between math and mem:
    - 2.12 ms of math-only time (13%) are not overlapped with mem
  - App memory throughput: 62 GB/s
    - HW theory is 114 GB/s, so we're off optimum
- Conclusion:
  - Code is memory-bound
  - Latency could be an issue too
  - Optimizations should focus on memory throughput first
    - math contributes very little to total time
    - (2.12 out of 35.39ms)

© NVIDIA Corporation 2011

Case Study: Limiter Analysis

- 3DFD of the wave equation, fp32
- Time (ms):
  - Full-kernel: 35.39
  - Mem-only: 33.27
  - Math-only: 16.25
- Instructions issued:
  - Full-kernel: 18,194,139
  - Mem-only: 7,497,296
  - Math-only: 16,839,792
- Memory access transactions:
  - Full-kernel: 1,708,032
  - Mem-only: 1,708,032
  - Math-only: 0

- **Analysis:**
  - Instruction:Byte ratio = ~2.66
    - 32*18,194,139 / 128*1,708,032
  - Good overlap between math and mem:
    - 2.12 ms of math-only time (13%) are not overlapped with mem
  - App memory throughput: 62 GB/s
    - HW theory is 114 GB/s, so we're off optimum
- **Conclusion:**
  - Code is memory-bound
  - Latency could be an issue too
  - Optimizations should focus on memory throughput first
    - math contributes very little to total time (2.12 out of 35.39ms)

© NVIDIA Corporation 2011

Summary: Limiter Analysis

- **Rough algorithmic analysis:**
  - How many bytes needed, how many instructions
- **Profiler analysis:**
  - Instruction count, memory request/transaction count
- **Analysis with source modification:**
  - Memory-only version of the kernel
  - Math-only version of the kernel
  - Examine how these times relate and overlap

© NVIDIA Corporation 2011

Optimizations for Global Memory

© NVIDIA Corporation 2011

Memory Throughput Analysis

- **Throughput: from application point of view**
  - From app point of view: count bytes requested by the application
  - From HW point of view: count bytes moved by the hardware
  - The two can be different
    - Scattered/misaligned pattern: not all transaction bytes are utilized
    - Broadcast: the same small transaction serves many requests
- **Two aspects to analyze for performance impact:**
  - Addressing pattern
  - Number of concurrent accesses in flight

© NVIDIA Corporation 2011

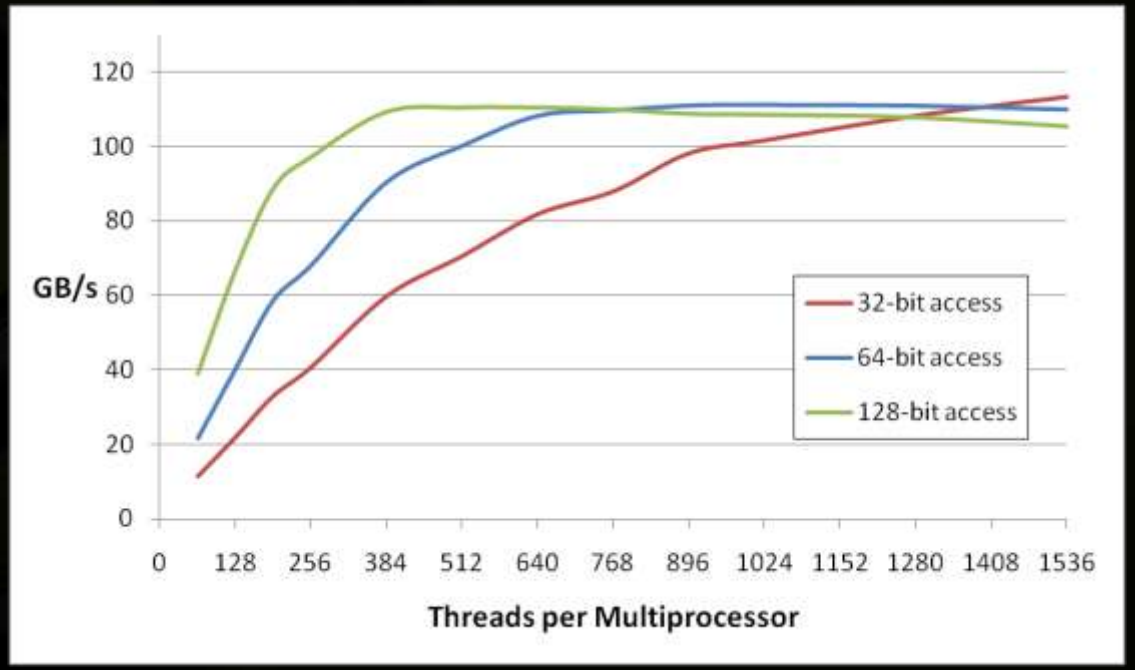
Memory Throughput Analysis

- **How to determine that access pattern is problematic:**
  - Profiler counters: access instruction count is significantly smaller than transaction count
    - gld_request < (l1_global_load_miss + l1_global_load_hit) * (word_size / 4B)
    - gst_request < 4 * l2_write_requests * (word_size / 4B)
    - Make sure to adjust the transaction counters for word size (see slide 8)
  - App throughput is much smaller than HW throughput
    - Use profiler to get HW throughput
- **How to tell that number of concurrent accesses is insufficient:**
  - Throughput from HW point of view is much lower than theoretical
- **CUDA 4.0 Visual Profiler does some of this automatically**

© NVIDIA Corporation 2011

Concurrent Accesses and Performance

- **Increment a 64M element array**
  - Two accesses per thread (load then store, but they are dependent)
    - Thus, each warp (32 threads) has one outstanding transaction at a time
- **Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s**



Several independent smaller accesses have the same effect as one larger one.

For example:

Four 32-bit ~= one 128-bit

© NVIDIA Corporation 2011

Optimization: Address Pattern

- **Coalesce the address pattern**
  - 128-byte lines for caching loads
  - 32-byte segments for non-caching loads, stores
  - A warp's address pattern is converted to transactions
    - Coalesce to maximize utilization of bus transactions
    - Refer to CUDA Programming Guide / Best Practices Guide / Fundamental Opt. talk
- **Try non-caching loads**
  - **Compiler option: -Xptxas –dlcm=cg or Inline PTX (CUDA 4.0)**
  - Smaller transactions (32B instead of 128B)
    - more efficient for scattered or partially-filled patterns
- **Try fetching data via texture unit**
  - Smaller transactions and different caching
  - Cache not polluted by other gmem loads

© NVIDIA Corporation 2011

Optimizing Access Concurrency

- **Have enough concurrent accesses to saturate the bus**
  - Need (mem_latency)x(bandwidth) bytes in flight (Little's law)
  - Fermi C2050 global memory:
    - 400-800 cycle latency, 1.15 GHz clock, 144 GB/s bandwidth, 14 SMs
    - Need 30-50 128-byte transactions in flight per SM
- **Ways to increase concurrent accesses:**
  - Increase occupancy
    - Adjust threadblock dimensions
      - To maximize occupancy at given register and smem requirements
    - Reduce register count (-maxrregcount option, or __launch_bounds__)
    - Use *CUDA Occupancy Calculator* (part of Toolkit)
  - Modify code to process several elements per thread

© NVIDIA Corporation 2011

Case Study: Access Pattern 1

- **Same 3DFD code as in the previous study**
- **Using caching loads (compiler default):**
  - Memory throughput: 62 / 74 GB/s for app / hw
  - Different enough to be interesting
- **Loads are coalesced:**
  - gld_request == (l1_global_load_miss + l1_global_load_hit)
- **There are halo loads that use only 4 threads out of 32**
  - For these transactions only 16 bytes out of 128 are useful
- **Solution: try non-caching loads**
  - Performance increase of 7%
    - Not bad for just trying a compiler flag, no code change
  - Memory throughput: 66 / 67 GB/s for app / hw

© NVIDIA Corporation 2011

Case Study: Accesses in Flight

- **Continuing with the FD code**
  - Throughput from both app and hw point of view is 66-67 GB/s
  - Now 30.84ms out of 33.71ms are due to mem
  - 1024 concurrent threads per SM
    - Due to register count (24 per thread)
    - But: At this thread count , simple copy kernel reaches ~80% of achievable mem throughput
- **Solution: increase accesses per thread**
  - Modified code so that each thread is responsible for 2 output points
    - Doubles the load and store count per thread, saves some indexing math
    - Doubles the tile size -> reduces bandwidth spent on halos
  - Further 25% increase in performance
    - App and HW throughputs are now 82 and 84 GB/s, respectively

© NVIDIA Corporation 2011

Case Study: Access Pattern 2

- **Kernel from climate simulation code**
  - Mostly fp64 (so, at least 2 x 128B transactions per warp's 32 thread access)
- **Profiler results:**
  - gld_request: 72,704
  - l1_global_load_hit: 439,072
  - l1_global_load_miss: 724,192
- **Analysis:**
  - L1 hit rate: 37.7%
  - **16** transactions per load instruction
    - **Indicates bad access pattern** (2 are expected due to 64-bit words)
    - Of the 16, 10 miss in L1 and contribute to mem bus traffic (compare: 2 optimal)
    - So, we fetch **5x** more bytes than needed by the app

© NVIDIA Corporation 2011

Case Study: Access Pattern 2

- **Looking closer at the access pattern:**
  - Each thread traverses a contiguous memory region - linearly!
  - Developer expecting CPU-like L1 caching
    - But remember what's been said about coding for L1 and L2
    - (Fundamental Optimizations, slide 11)
  - This is one of the worst access patterns for GPUs
- **Solution:**
  - Transposed the code so that each warp accesses a contiguous memory region
  - 2.17 transactions per load instruction
  - This and some other changes improved performance by 3x

© NVIDIA Corporation 2011

Optimizing w. Compression / datatype change

- **Consider compression/data type changes when**
  - **Every other aspect has been optimized**
  - **Kernel is limited by number of bytes needed**
- **Approaches:**
  - Int: conversion between 8-, 16-, 32-bit integers is 1 instruction (64-bit requires a couple)
  - FP: conversion between fp16, fp32, fp64 is one instruction
    - fp16 (1s5e10m) is storage only, no math instructions
  - Range-based compression:
    - Lower and upper limits are kernel arguments
    - Data is an index for interpolation between the limits
- **Application in practice:**
  - Clark *et al.* "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs"
  - http://arxiv.org/abs/0911.3191

© NVIDIA Corporation 2011

Summary: Memory Analysis and Optimization

- **Analyze:**
  - Access pattern:
    - Compare counts of access instructions and transactions
    - Compare throughput from app and hw point of view
  - Number of accesses in flight
    - Look at occupancy and independent accesses per thread
    - Compare achieved throughput to theoretical throughput
      - Also compare with simple memcpy throughput at the same occupancy
- **Optimizations:**
  - Coalesce address patterns per warp (nothing new here), consider texture
  - Process more words per thread (if insufficient accesses in flight to saturate bus)
  - Try 4 combinations of L1 size (16kb/48kb) and load type (caching and non-caching)
  - Consider compression

© NVIDIA Corporation 2011

Optimizations for Instruction Throughput

© NVIDIA Corporation 2011

Possible Limiting Factors

- **Raw instruction throughput**
  - Know the kernel instruction mix
  - fp32, fp64, int, mem, transcendentals, etc. have different throughputs
    - Refer to the CUDA Programming Guide / Best Practices Guide
  - Can examine assembly, if needed:
    - Can look at PTX (virtual assembly), though it's not the final optimized code
    - Can look at post-optimization machine assembly (--dump-sass, via cuobjdump)
- **Instruction serialization ("instruction replays" for warp's threads)**
  - Occurs when threads in a warp execute/issue the same instruction after each other
    - As opposed to the entire warp issuing the instruction at once
    - Think of it as "replaying" the same instruction for different threads in a warp
  - Some causes:
    - Shared memory bank conflicts
    - Constant memory bank conflicts

© NVIDIA Corporation 2011

Instruction Throughput: Analysis

- **Profiler counters (both incremented by 1 per warp):**
  - instructions executed: counts instructions encountered during execution
  - instructions issued: also includes additional issues due to serialization
  - Difference between the two: instruction issues that happened due to serialization, instruction cache misses, etc.
    - Will rarely be 0, concern only if it's a significant percentage of instructions issued
- **Compare achieved throughput to HW capabilities**
  - Peak instruction throughput is documented in the Programming Guide
  - Profiler also reports throughput:
    - GT200: as a fraction of theoretical peak for fp32 instructions
    - Fermi: as IPC (instructions per clock)

© NVIDIA Corporation 2011

Instruction Throughput: Optimization

- **Use intrinsics where possible (__sin(), __sincos(), __exp(), etc.)**
  - Available for a number of math.h functions
  - 2-3 bits lower precision, much higher throughput
    - Refer to the CUDA Programming Guide for details
  - Often a single instruction, whereas a non-intrinsic is a SW sequence
- **Additional compiler flags that also help (select GT200-level precision):**
  - -ftz=true : flush denormals to 0
  - -prec-div=false : faster fp division instruction sequence (some precision loss)
  - -prec-sqrt=false : faster fp sqrt instruction sequence (some precision loss)
- **Make sure you do fp64 arithmetic only where you mean it:**
  - fp64 throughput is lower than fp32
  - fp literals without an “f” suffix (34.7) are interpreted as fp64 per C standard

© NVIDIA Corporation 2011

Serialization: Profiler Analysis

- **Serialization is significant if**
  - instructions_issued is significantly higher than instructions_executed
  - CUDA 4.0: Instructions replayed %
- **Warp divergence**
  - Profiler counters: divergent_branch, branch
    Profiler derived: Divergent branches (%)).
  - However, this only counts the branches, not the rest of serialized instructions
  - Better: 'threads instruction executed' counter:
    Increments for every instruction by number of threads that executed the instruction.
  - If there is no divergence, then for every instruction it should increment by 32
    (and threads instruction executed = 32* instruction executed)
  - Thus: Control flow divergence% =
    100 * ((32 * instructions executed) – threads instruction executed)/(32* instructions executed)

© NVIDIA Corporation 2011

Serialization: Profiler Analysis

- **SMEM bank conflicts**
  - Profiler counters:
    - l1_shared_bank_conflict
      - incremented by 1 per warp for each replay
        (or: each n-way shared bank conflict increments by n-1)
      - double increment for 64-bit accesses
    - shared_load, shared_store: incremented by 1 per warp per instruction
  - Bank conflicts are significant if both are true:
    - instruction throughput affects performance
    - l1_shared_bank_conflict is significant compared to instructions_issued:
      - Shared bank conflict replay (%) =
        100 * (l1_shared_bank_conflict)/instructions_issued
      - Shared memory bank conflict per shared memory instruction (%) =
        100 * (l1 shared bank conflict)/(shared load + shared store)

© NVIDIA Corporation 2011

Serialization: Analysis with Modified Code

- **Modify kernel code to assess performance improvement if serialization were removed**
  - Helps decide whether optimizations are worth pursuing
- **Shared memory bank conflicts:**
  - Change indexing to be either broadcasts or just threadIdx.x
  - Should also declare smem variables as volatile
    - Prevents compiler from “caching” values in registers
- **Warp divergence:**
  - Change the if-condition to have all threads take the same path
  - Time both paths to see what each costs

© NVIDIA Corporation 2011

Serialization: Optimization

- **Shared memory bank conflicts:**
  - Pad SMEM arrays
    - For example, when a warp accesses a 2D array's column
    - See CUDA Best Practices Guide, Transpose SDK whitepaper
  - Rearrange data in SMEM
- **Warp serialization:**
  - Try grouping threads that take the same path into same warp
    - Rearrange the data, pre-process the data
    - Rearrange how threads index data (may affect memory perf)

© NVIDIA Corporation 2011

Case Study: SMEM Bank Conflicts

- **A different climate simulation code kernel, fp64**
- **Profiler values:**
  - Instructions:
    - Executed / issued: 2,406,426 / 2,756,140
    - Difference: 349,714 (**12.7%** of instructions issued were “replays”)
  - GMEM:
    - Total load and store transactions: 170,263
    - Instr:byte ratio: 4
      - Suggests that instructions are a significant limiter (especially since there is a lot of fp64 math)
  - SMEM:
    - Load / store: 421,785 / 95,172
    - Bank conflict: 674,856 (really 337,428 because of double-counting for fp64)
      - This means a total of 854,385 SMEM access instructions (421,785 +95,172+337,428) , of which **39%** replays
- **Solution: Pad shared memory array**

  Performance increased by 15%
    - replayed instructions reduced down to 1%

© NVIDIA Corporation 2011

Instruction Throughput: Summary

- **Analyze:**
  - Check achieved instruction throughput
  - Compare to HW peak (note: must take instruction mix into consideration)
  - Check percentage of instructions due to serialization
- **Optimizations:**
  - Intrinsics, compiler options for expensive operations
  - Group threads that are likely to follow same execution path
  - Avoid SMEM bank conflicts (pad, rearrange data)

© NVIDIA Corporation 2011

Optimizations for Latency

© NVIDIA Corporation 2011

Latency: Analysis

- **Suspect latency issues if:**
  - Neither memory nor instruction throughput rates are close to HW theoretical rates
  - Poor overlap between mem and math
    - Full-kernel time is significantly larger than max{mem-only, math-only}
- **Two possible causes:**
  - Insufficient concurrent threads per multiprocessor to hide latency
    - Occupancy too low
    - Too few threads in kernel launch to load the GPU
      - Indicator: elapsed time doesn't change if problem size is increased (and with it the number of blocks/threads)
  - Too few concurrent threadblocks per SM when using `__syncthreads()`
    - `__syncthreads()` can prevent overlap between math and mem within the same threadblock

© NVIDIA Corporation 2011

Simplified View of Latency and Syncs

Memory-only time
Math-only time

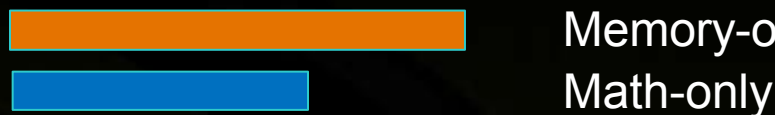
Kernel where most math cannot be executed until all data is loaded by the threadblock

Full-kernel time, one large threadblock per SM

time

© NVIDIA Corporation 2011

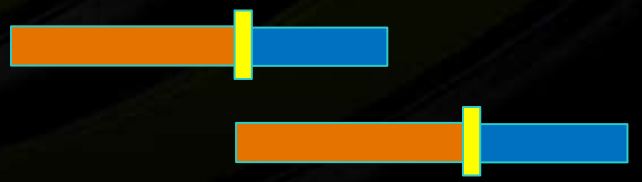
Simplified View of Latency and Syncs



Memory-only time
Math-only time

Kernel where most math cannot be executed until all data is loaded by the threadblock

Full-kernel time, one large threadblock per SM



Full-kernel time, two threadblocks per SM
(each half the size of one large one)

© NVIDIA Corporation 2011

Latency: Optimization

- **Insufficient threads or workload:**
  - Best: Increase the level of parallelism (more threads)
  - Alternative: Process several output elements per thread – gives more independent memory and arithmetic instructions (which get pipelined) - downside: code complexity
- **Synchronization Barriers:**
  - Can assess impact on perf by commenting out __syncthreads()
    - Incorrect result, but gives upper bound on improvement
  - Try running several smaller threadblocks
    - Less hogging of SMs; think of it as SM “pipelining” blocks
    - In some cases that costs extra bandwidth due to more halos
- **More information:**
  - Vasily Volkov, GTC2010: “Better Performance at Lower Occupancy” http://www.gputechconf.com/page/gtc-on-demand.html#session2238

© NVIDIA Corporation 2011

Register Spilling

© NVIDIA Corporation 2011

Register Spilling

- **Compiler "spills" registers to local memory when register limit exceeded**
  - Fermi HW limit is 63 registers per thread
  - Spills also possible < 63regs if register limit is programmer-specified
    - Common when trying to achieve certain GPU occupancy with -maxrregcount compiler flag or __launch_bounds__ in source
  - lmem is like gmem memory-bus-load-wise, except that writes are cached in L1
    - lmem load hit in L1 -> no bus traffic
    - lmem load miss in L1 -> bus traffic (128 bytes per miss)
  - Compiler flag –Xptxas –v gives the register and lmem usage per thread
- **Potential impact on performance**
  - Additional bandwidth pressure if evicted from L1
  - Additional instructions
  - Not always a problem, easy to investigate with quick profiler analysis

© NVIDIA Corporation 2011

Register Spilling: Analysis

- **Profiler counters:** l1_local_load_hit, l1_local_load_miss
- **Impact on instruction count:**
  - Compare L1 localmem transactions to total instructions issued
- **Impact on memory throughput:**
  - Misses add 128 bytes per warp
  - Compare 2*l1_local_load_miss count to gmem access count (stores + loads)
    - Multiply lmem load misses by 2: missed line must have been evicted -> store across bus
    - If kernel uses caching loads: consider only gmem misses in L1
    - If kernel uses non-caching loads: consider all loads

© NVIDIA Corporation 2011

Optimization for Register Spilling

- **Try increasing the limit of registers per thread**
  - Use a higher limit in –maxrregcount, or lower thread count for __launch_bounds__
  - Will likely decrease occupancy, potentially making gmem accesses less efficient
  - However, may still be an overall win – lmem transactions potentially reduced, thus fewer total bytes being accessed in gmem
- **Use shared memory for less-used variables**
- **Non-caching loads for gmem**
  - potentially fewer contentions with spilled registers in L1
- **Increase L1 size to 48KB**
  - default is 16KB L1 / 48KB smem

© NVIDIA Corporation 2011

Register Spilling: Case Study

- **FD kernel, (3D-cross stencil)**
  - fp32, so all gmem accesses are 4-byte words
    - Needed higher occupancy to saturate memory bandwidth
  - Coalesced, non-caching loads
    - one gmem request = 128 bytes
    - all gmem loads result in bus traffic
  - Larger threadblocks mean lower gmem pressure
    - Halos (ghost cells) are smaller as a percentage
- **Aiming to have 1024 concurrent threads per SM**
  - Means no more than 32 registers per thread
  - Compiled with –maxrregcount=32

© NVIDIA Corporation 2011

Case Study: Register Spilling 1

- **10th order in space kernel (31-point stencil)**
  - 32 registers per thread : 68 bytes of lmem per thread : upto 1024 threads per SM
- **Profiled counters:**
  - l1_local_load_miss = 36 inst_issued = 8,308,582
  - l1_local_load_hit = 70,956 gld_request = 595,200
  - local_store = 64,800 gst_request = 128,000
- **Conclusion: spilling is not a problem in this case**
  - Ratio of gmem to lmem bus traffic approx 10,044 : 1 (hardly any bus traffic is due to spills)
    - L1 contains most of the spills (99.9% hit rate for lmem loads)
  - Only 1.6% of all instructions are due to spills
- **Comparison:**
  - 42 registers per thread : no spilling : upto 768 threads per SM
    - Single 512-thread block per SM : 24% perf decrease
    - Three 256-thread blocks per SM : 7% perf decrease

© NVIDIA Corporation 2011

Case Study: Register Spilling 2

- **12th order in space kernel (37-point stencil)**
  - 32 registers per thread : 80 bytes of lmem per thread : up to 1024 threads per SM
- **Profiled counters:**
  - l1_local_load_miss = 376,889 inst_issued = 10,154,216
  - l1_local_load_hit = 36,931 gld_request = 550,656
  - local_store = 71,176 gst_request = 115,200
- **Conclusion: spilling is a problem for this case**
  - The ratio of gmem to lmem bus traffic is approx 6 : 7 (53% of bus traffic is due to spilling)
    - L1 does not contain the spills (8.9% hit rate for lmem loads)
  - Only 4.1% of all instructions are due to spills
- **Solution: increase register limit per thread**
  - 42 registers per thread : no spilling : upto 768 threads per SM
  - Single 512-thread block per SM : 13% perf increase
  - Three 256-thread blocks per SM : 37% perf increase

© NVIDIA Corporation 2011

Register Spilling: Summary

- **Doesn't always decrease performance, but when it does it's due to:**
  - Increased pressure on the memory bus (due to lmem transactions not L1 cached)
  - Increased instruction count
- **Use the profiler to examine the impact by comparing:**
  - 2*l1_local_load_miss to all gmem accesses that don't hit in L1:
    Local memory bus traffic (%) =
    (#SMs * 2 * l1 local load miss * 128 * 100)/((l2 read requests + l2 write requests)* 32)
  - Local access count to total instructions issued:
    Local memory replay (%) =
    100 * (l1 local load miss + l1 local store miss)/ instructions_issued
- **Register Spilling is significant if:**
  - Memory-bound code:
    lmem *misses* are significant percentage of total bus traffic
  - Instruction-bound code:
    lmem *accesses* are significant percentage of all instructions

© NVIDIA Corporation 2011

Summary

- **Determining what limits your kernel most:**
  - Arithmetic, memory bandwidth, latency
- **Address the bottlenecks in the order of importance**
  - *Analyze* for inefficient use of hardware
  - *Estimate* the impact on overall performance
  - *Optimize* to use hardware most efficiently
- **More resources:**
  - Talk on Fundamental Optimizations
  - Prior CUDA tutorials at Supercomputing
    - http://gpgpu.org/{sc2007,sc2008,sc2009,sc2010}
  - GTC2010 talks: http://www.nvidia.com/gtc2010-content
  - CUDA Programming Guide, CUDA Best Practices Guide
  - CUDA webinars

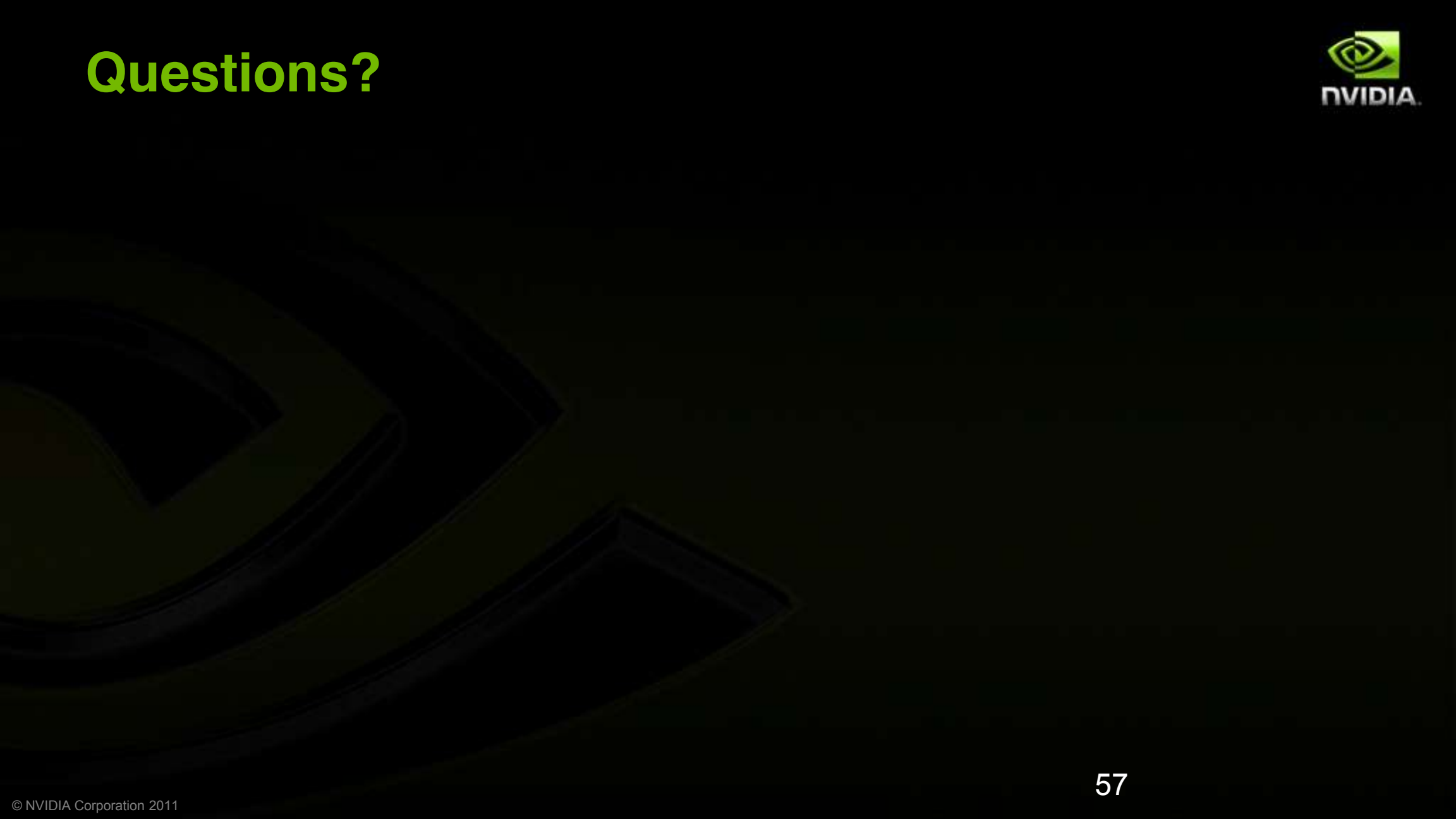
© NVIDIA Corporation 2011

New Profiler API (CUPTI)

- CUPTI = CUDA Performance Tools Interface
- Define profiled region of application
- Can change config/log file while profiling this region
- API calls in CUDA reference manual:
  - cuProfilerInitialize
  - cuProfilerStart
  - cuProfilerStop

© NVIDIA Corporation 2011

Questions?



© NVIDIA Corporation 2011

Stencil operations in CUDA

Tobias Brandvik
Whittle Laboratory
University of Cambridge

Motivation

We want to simulate the flow of air through turbomachines

Overview

1. Basic stencil operations in CUDA
2. Optimisation strategies
3. A software framework for stencil applications
4. Case study: A turbomachinery flow solver

A simple stencil

Second derivative:

$$\frac{\partial^2 u}{\partial x^2}$$

Simple Fortran implementation:

```
DO K=2,NK-1
  DO J=2,NJ-1
    DO I=2,NI-1
      D2UDX2(I,J,K) = (U(I+1,J,K) - 2.0*U(I,J,K) +
     &           U(I-1,J,K))/(DX*DX)
    END DO
  END DO
END DO
```

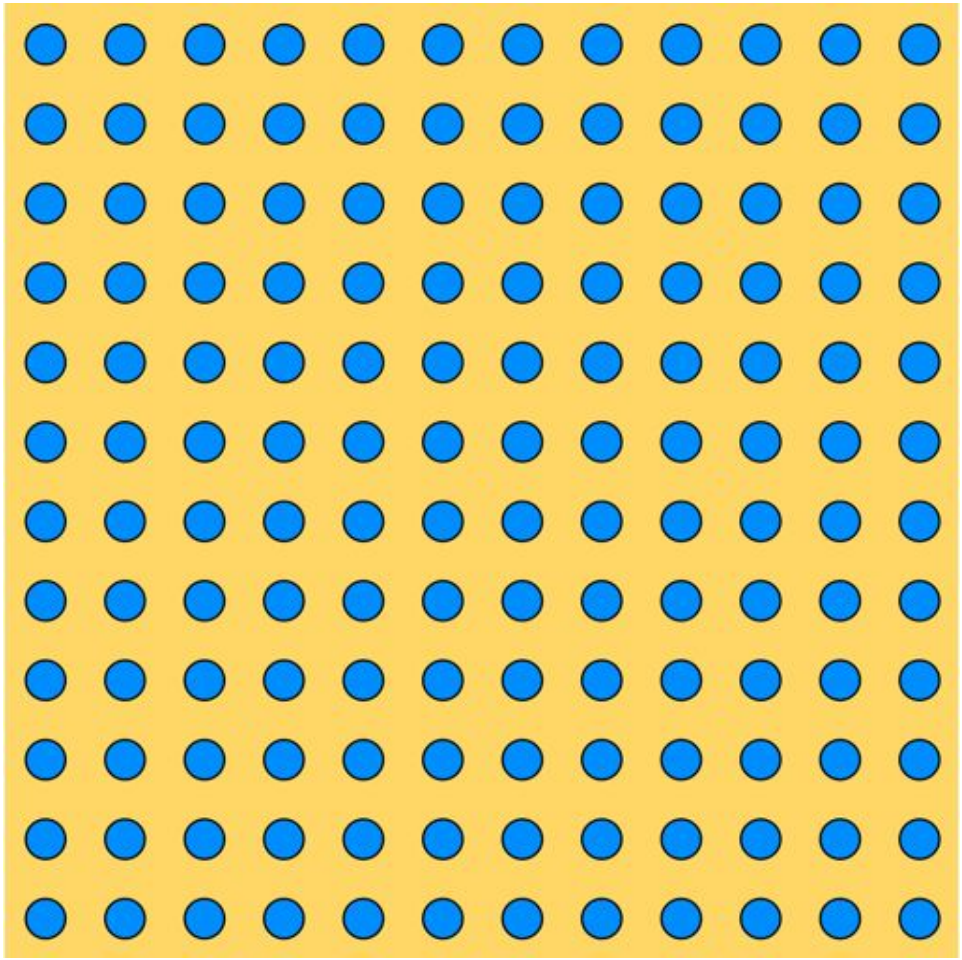
Structured grid

- Stencil should be evaluated for every grid node in a large 3D grid (e.g. 100^3)

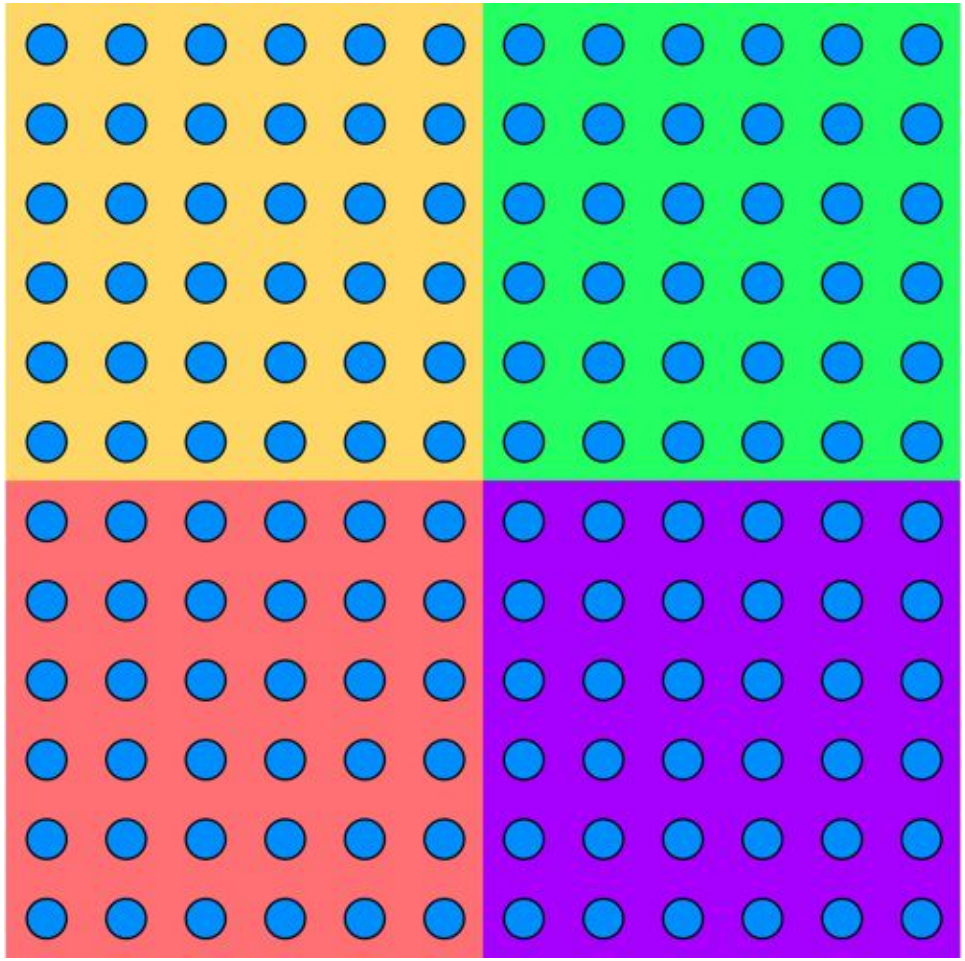
CUDA thread block parallelism

- The grid is divided into “sub-blocks” that are assigned to a CUDA thread block

Grid is divided into sub-blocks



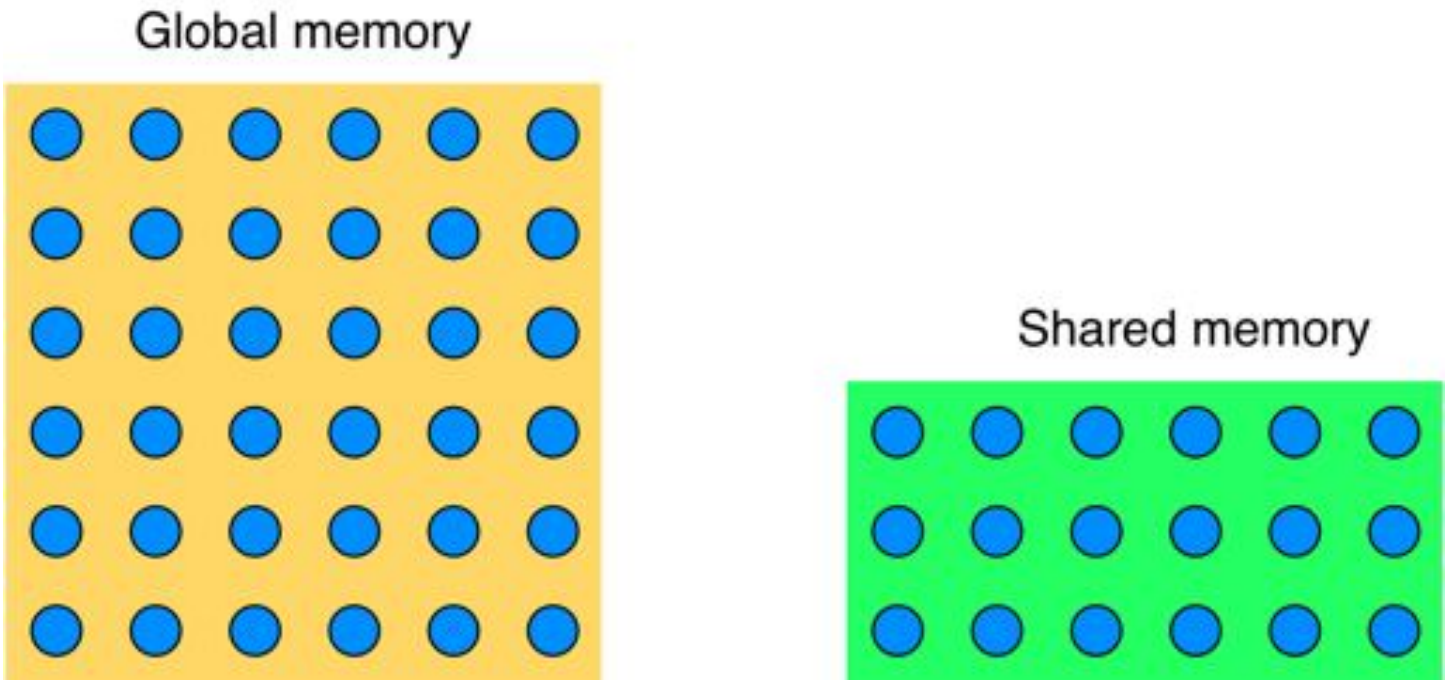
Grid is divided into sub-blocks



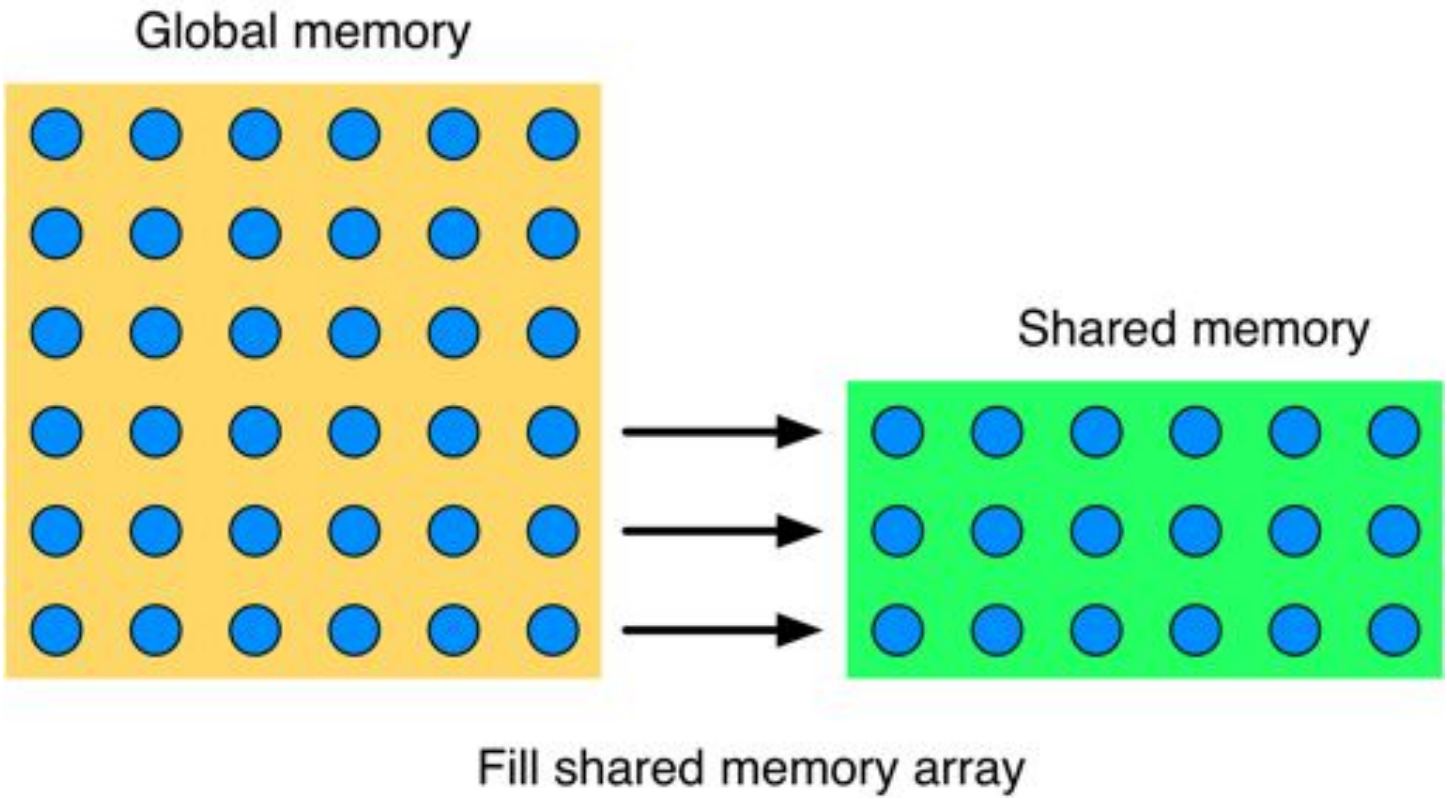
Sub-block implementation (after Datta et al.)

- Each thread in a thread block reads sub-block data from global device memory to SM shared memory (coalesced reads for maximum bandwidth)
- Synch threads
- Update nodes in sub-block using shared memory and output result back to global memory
- But shared memory and max threads per block are limited, so best plan is to march through sub-domain plane-by-plane…

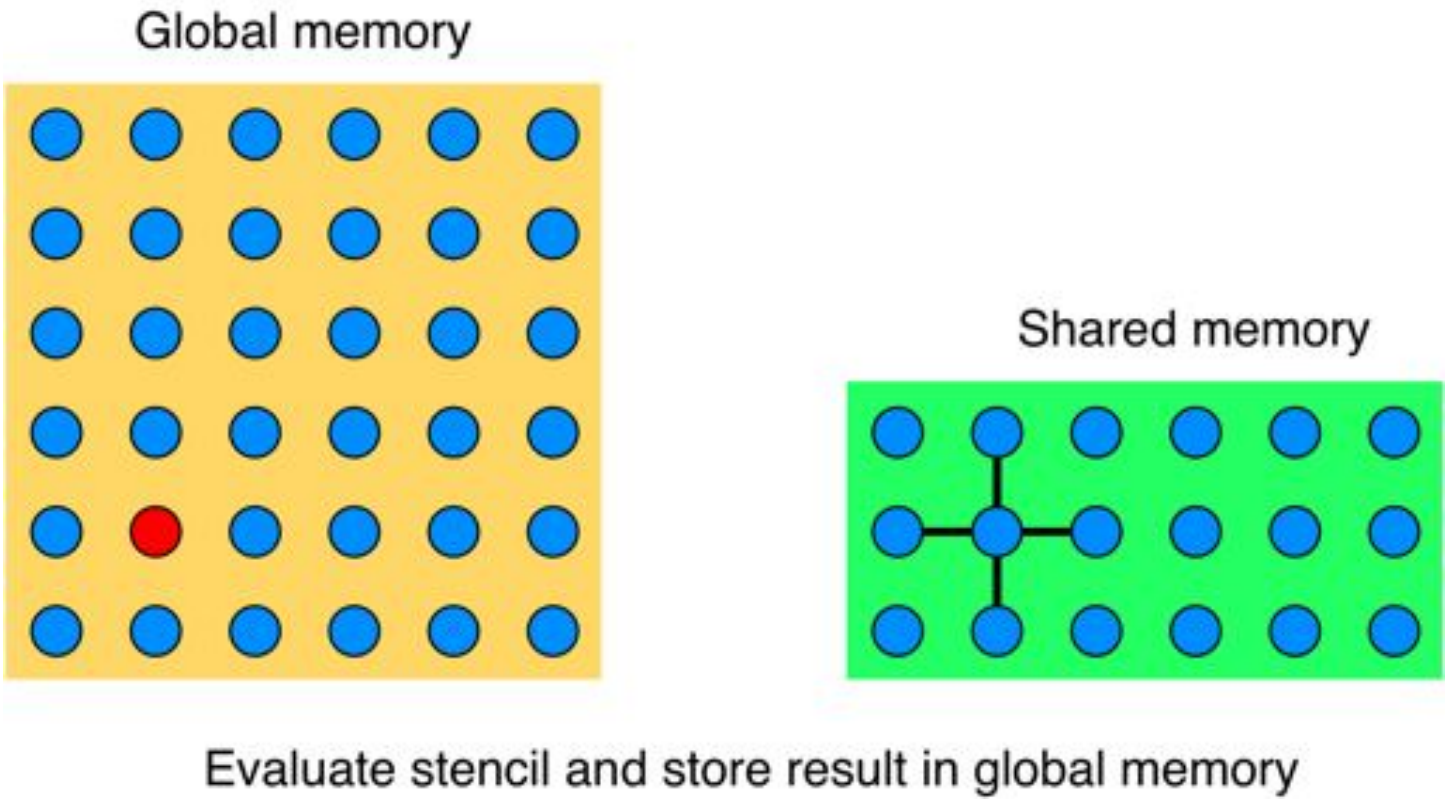
Sub-block implementation



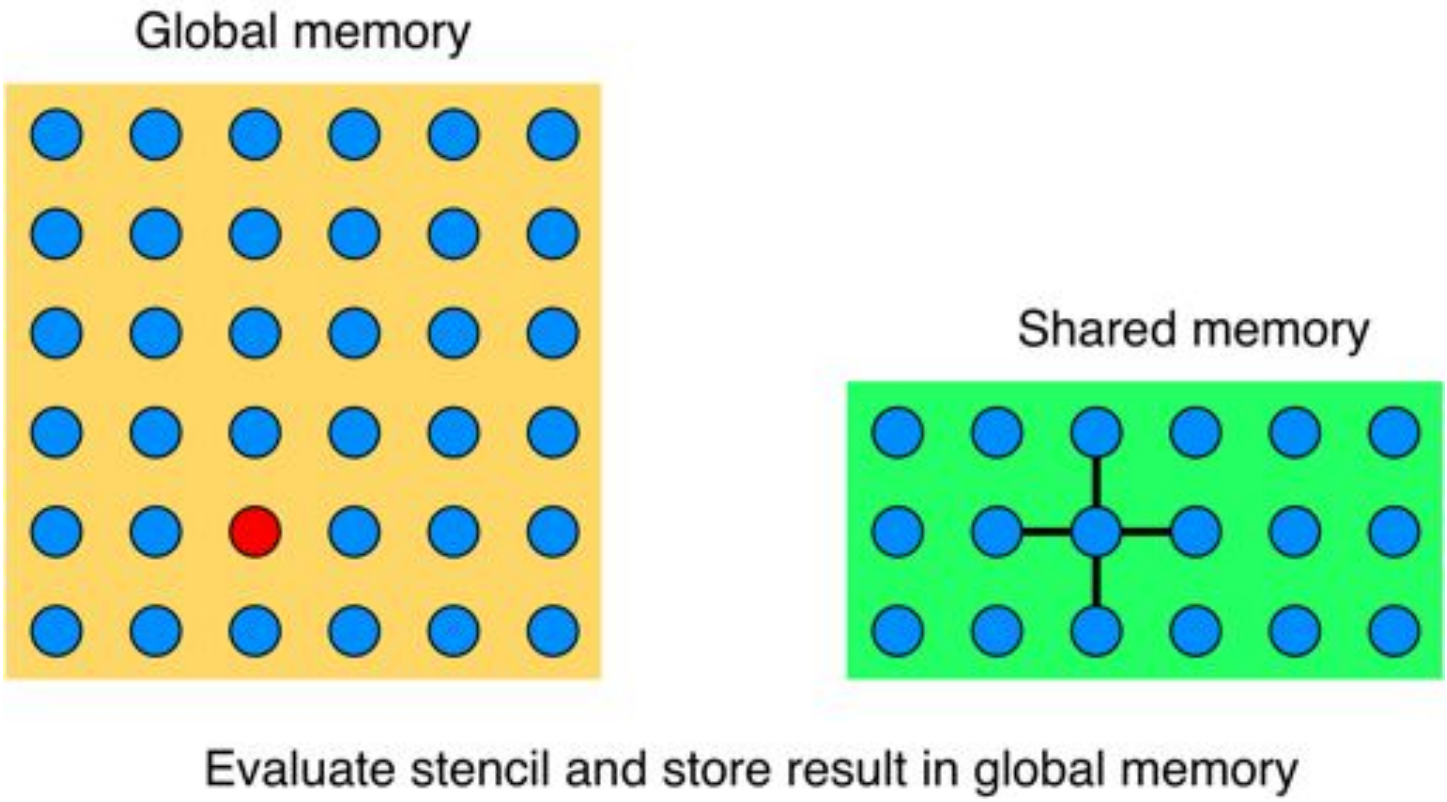
Sub-block implementation



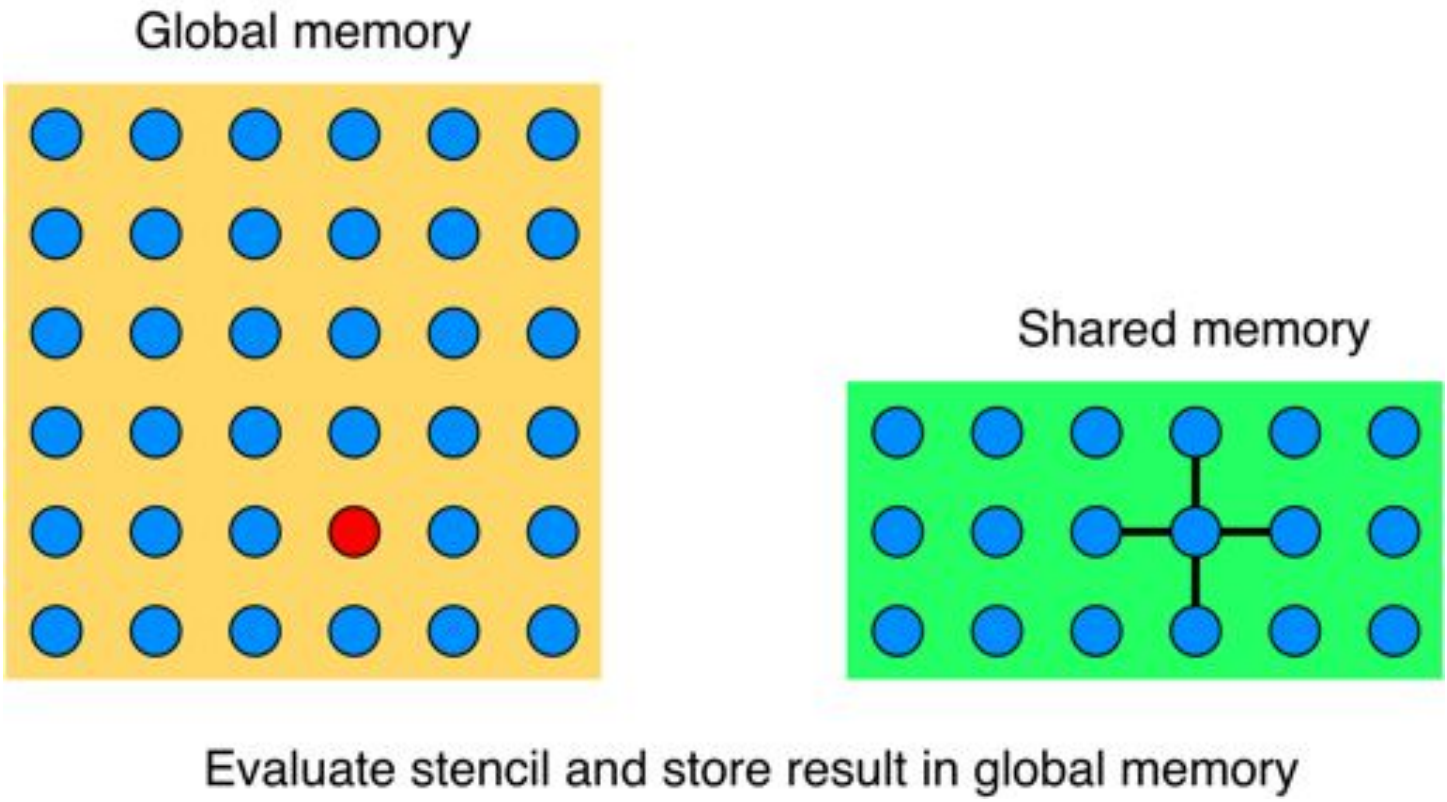
Sub-block implementation



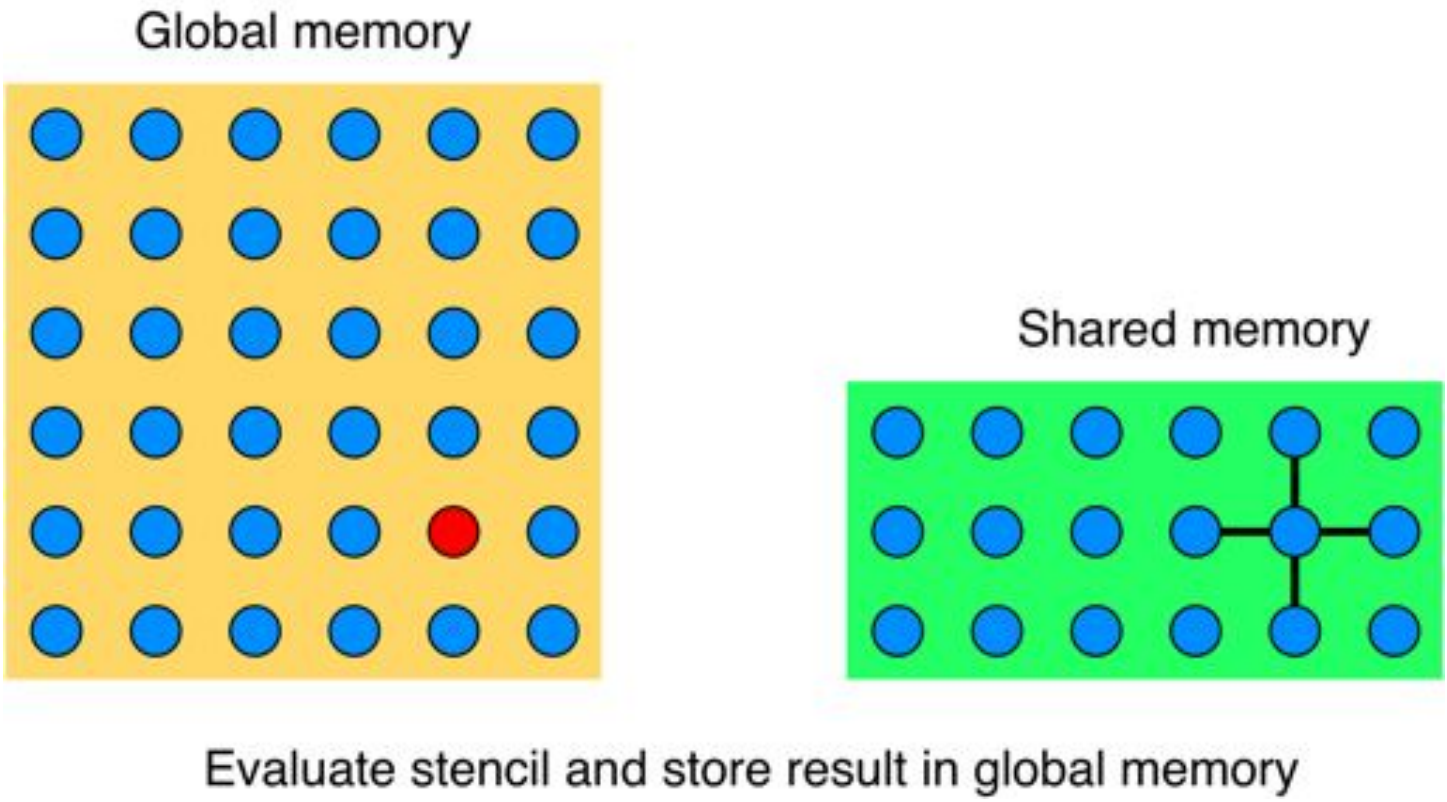
Sub-block implementation



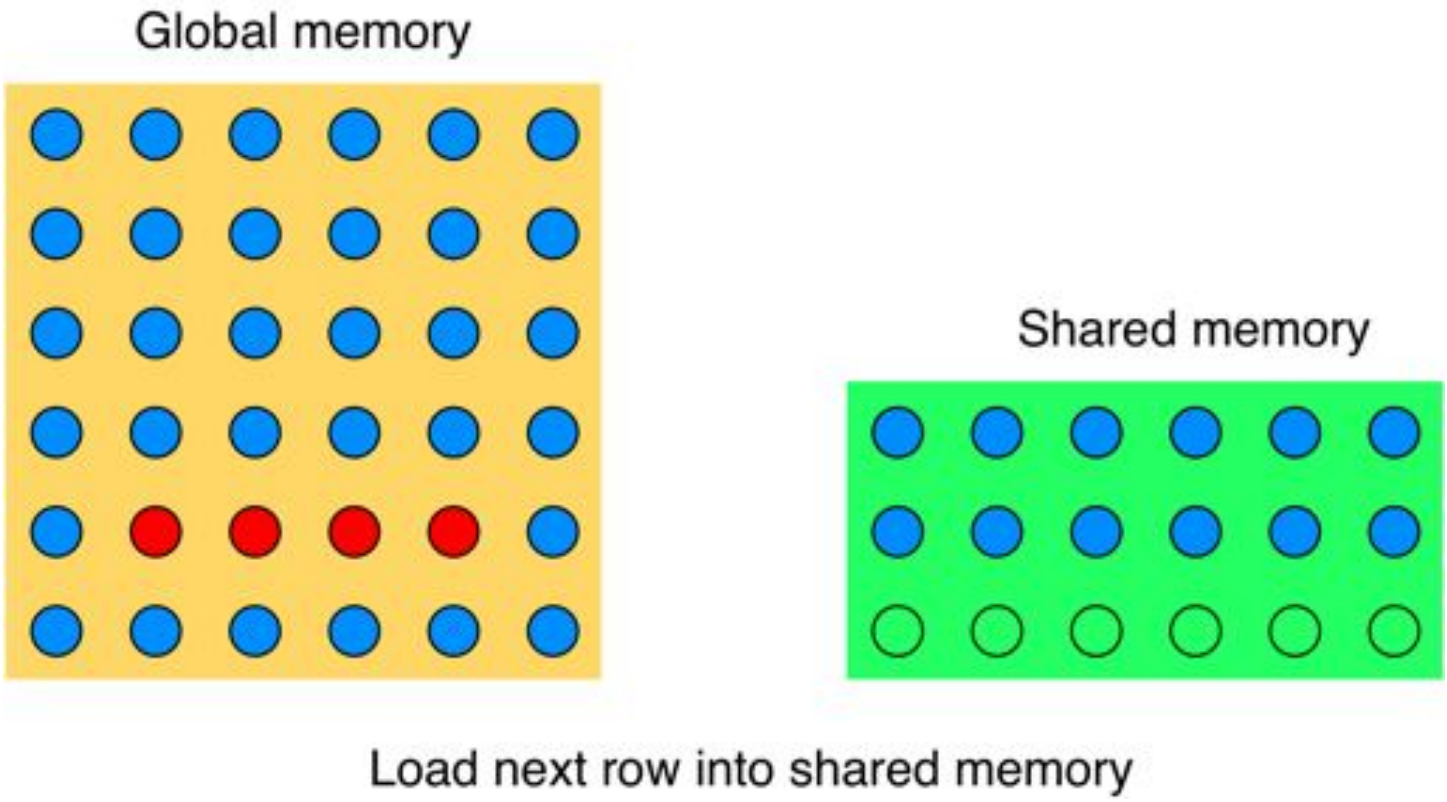
Sub-block implementation



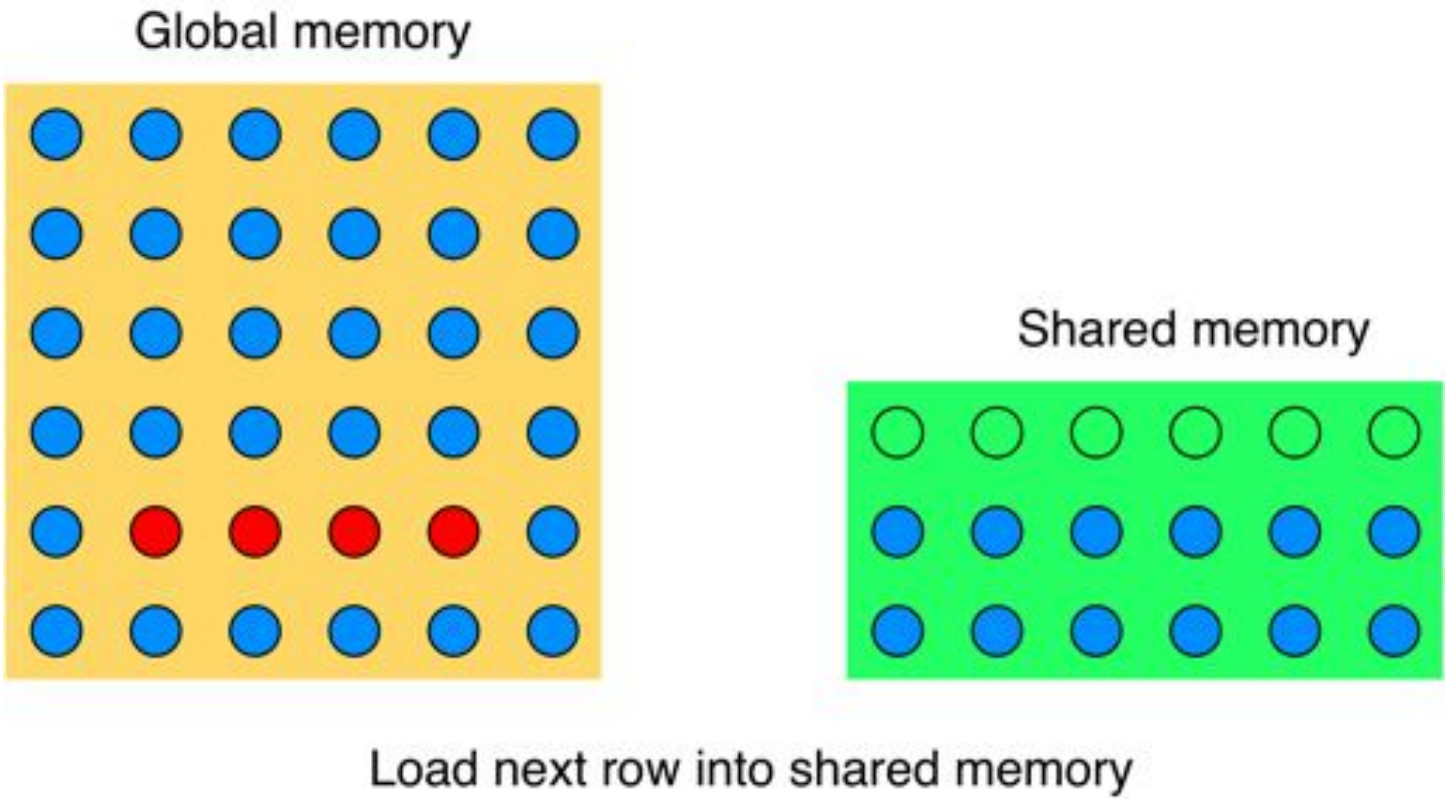
Sub-block implementation



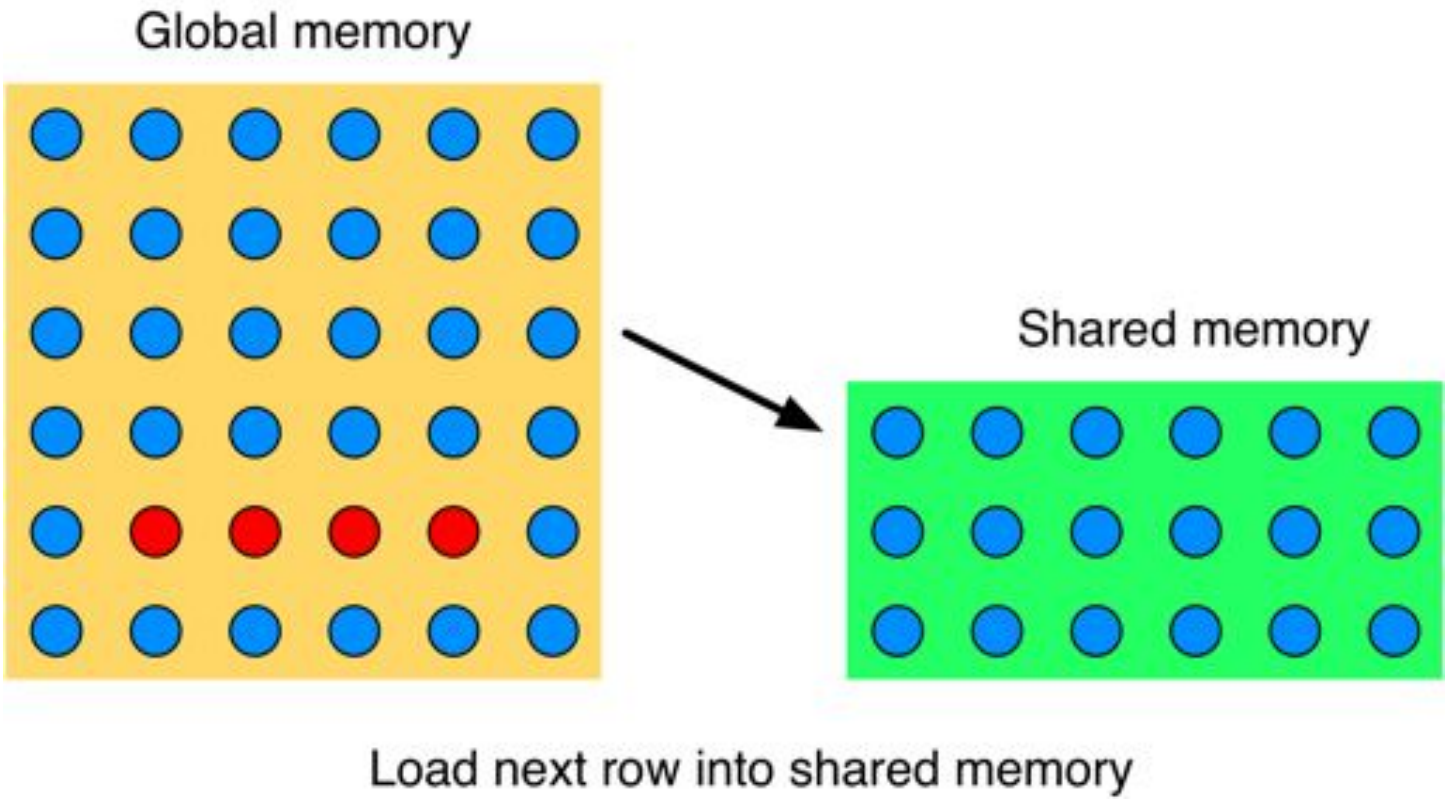
Sub-block implementation



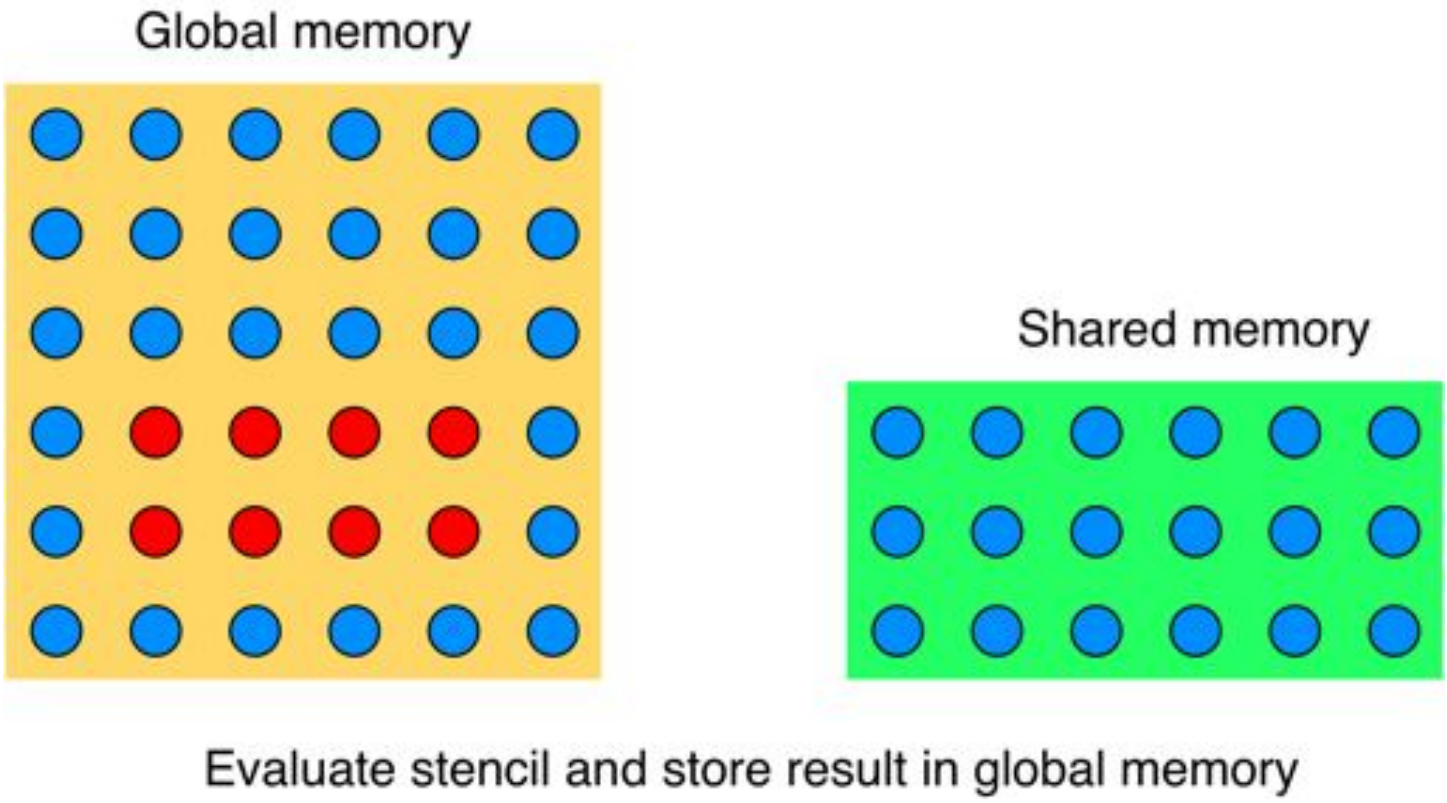
Sub-block implementation



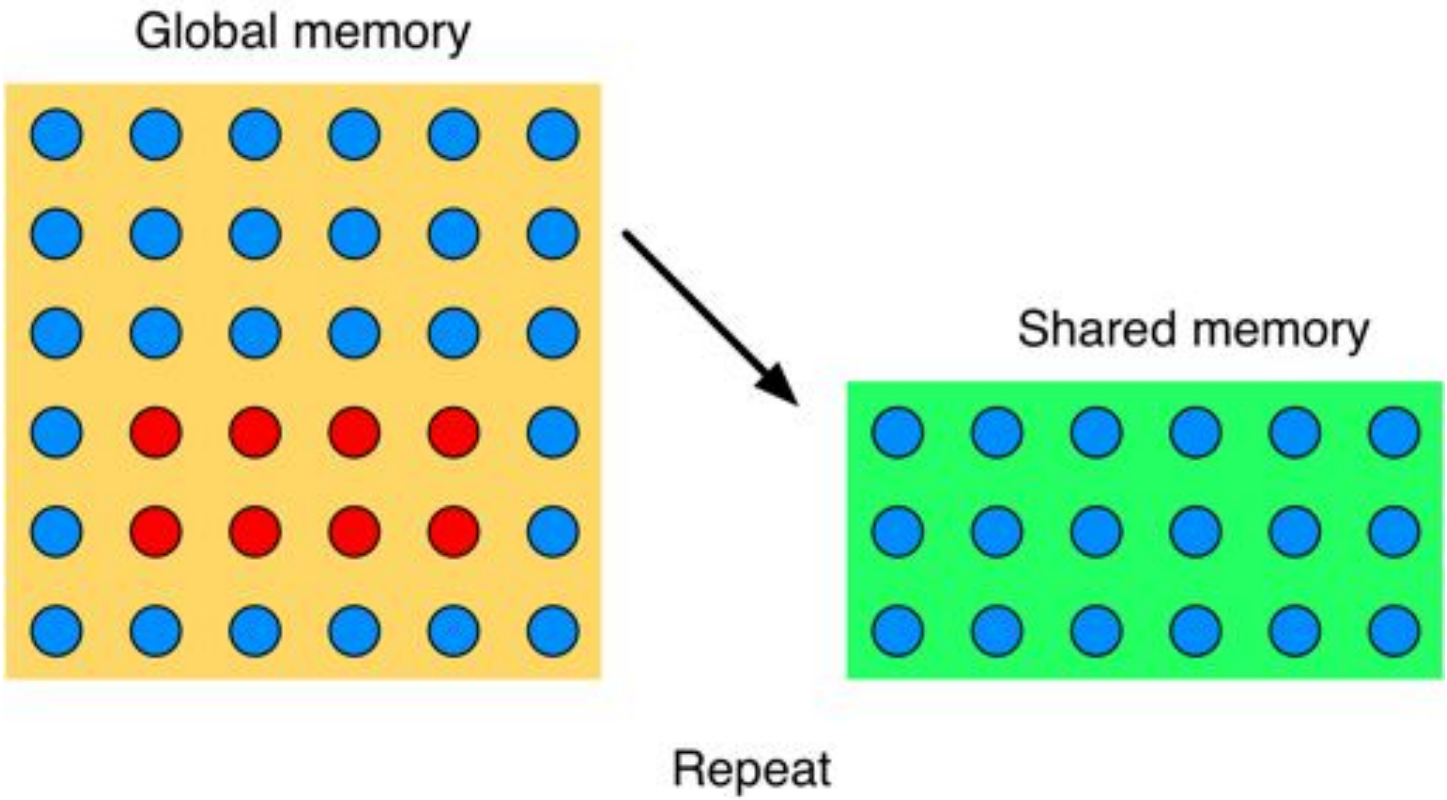
Sub-block implementation



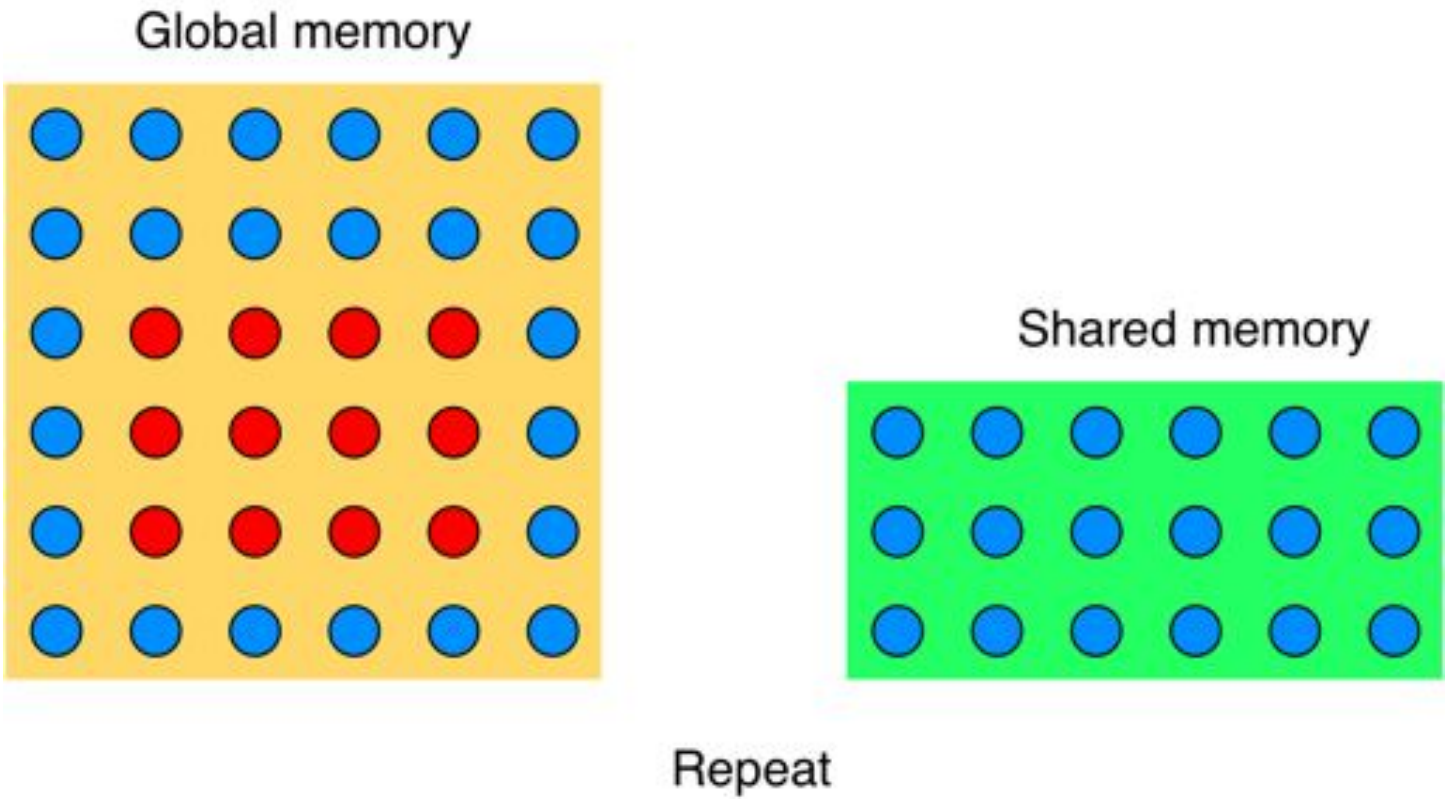
Sub-block implementation



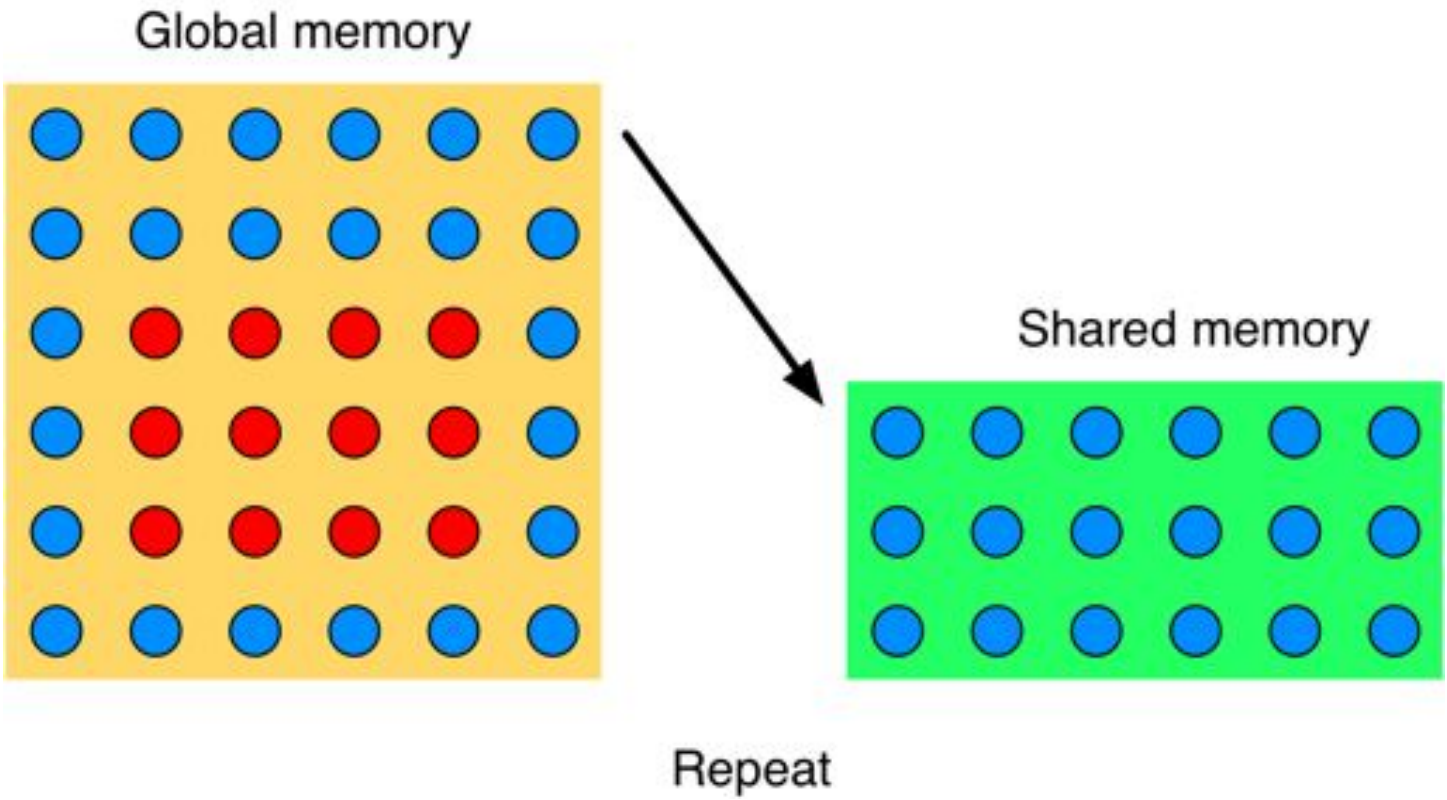
Sub-block implementation



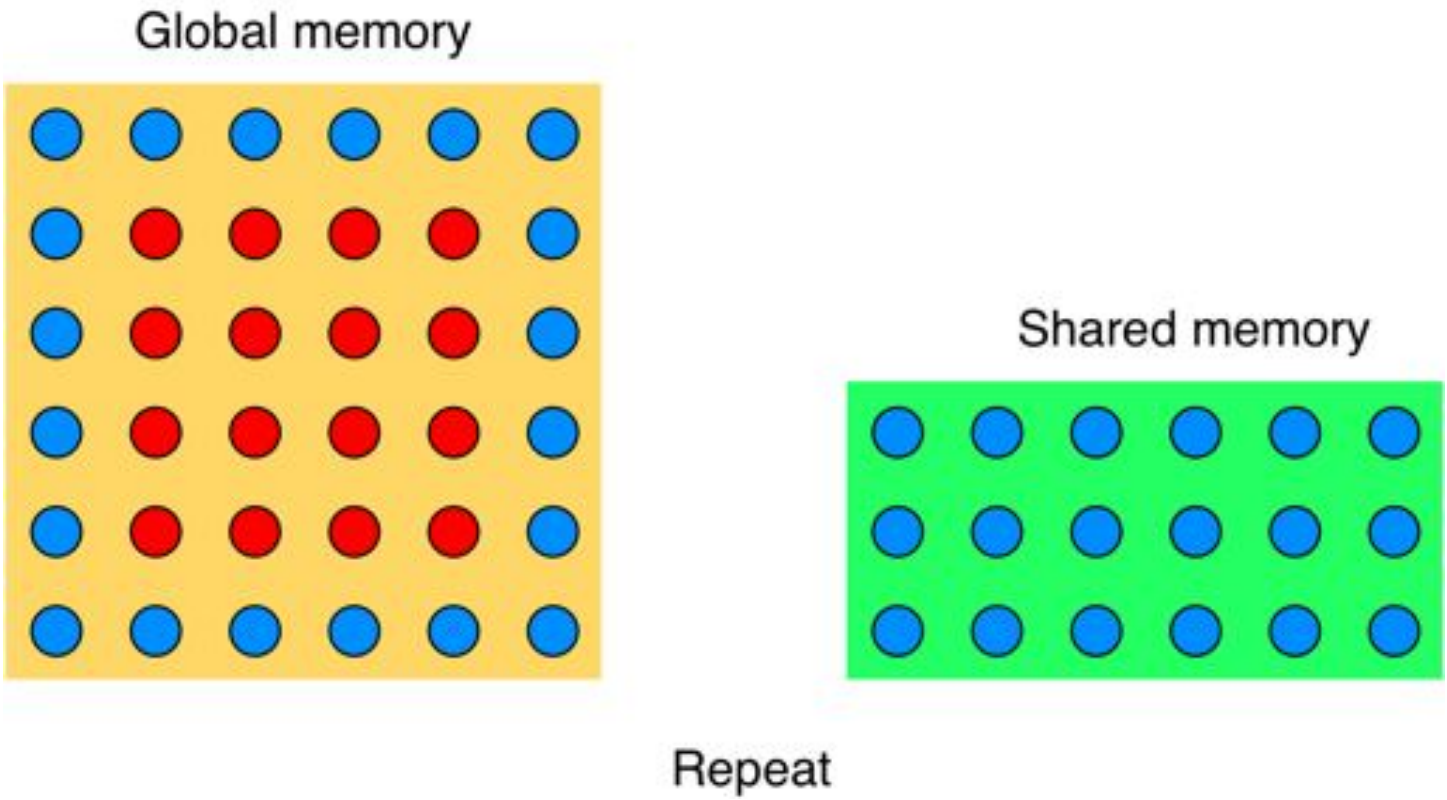
Sub-block implementation



Sub-block implementation



Sub-block implementation



CUDA Example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
```

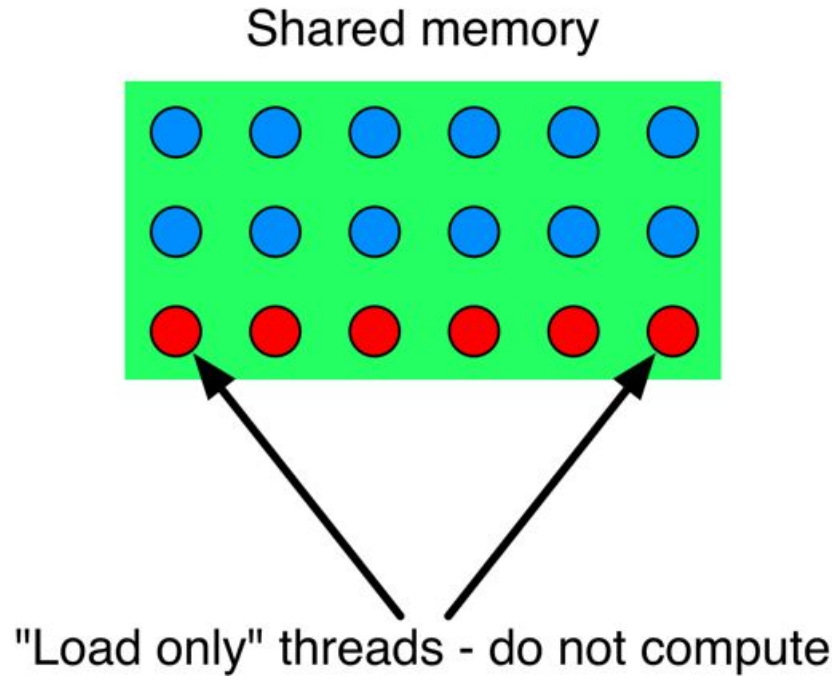
CUDA Example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
a[i][j][0] = a_d[i0m10];       // fetch first three planes
a[i][j][1] = a_d[i000];
a[i][j][2] = a_d[i0p10];
__syncthreads();               // make sure planes are loaded
```

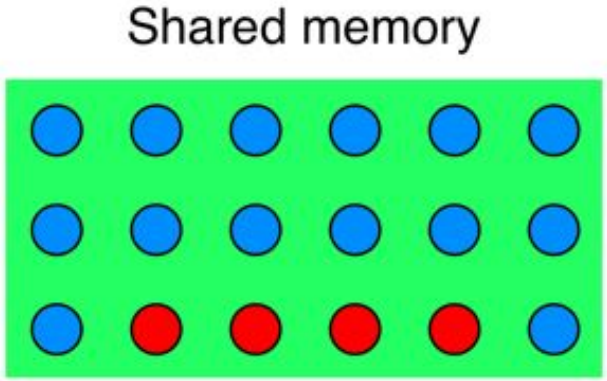
CUDA Example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
a[i][j][0] = a_d[i0m10];       // fetch first three planes
a[i][j][1] = a_d[i000];
a[i][j][2] = a_d[i0p10];
__syncthreads();               // make sure planes are loaded
// compute the stencil: //
b_d[i000]  = sf1*a[i][j][1] +
           + sfd6*(a[i-1][j][1] + a[i+1][j][1]
                 + a[i][j][0]   + a[i][j][2]
                 + a[i][j-1][1] + a[i][j+1][1])
// load next "k" plane and repeat //
```

Optimisaton #1: No load-only threads

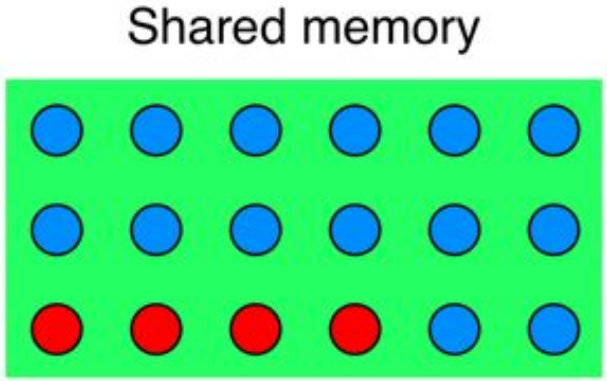


Optimisaton #1: No load-only threads



Have only compute threads - two load operations

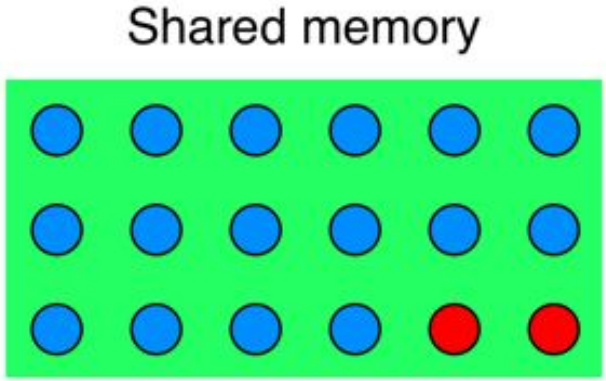
Optimisaton #1: No load-only threads



First load - all threads read from memory

Have only compute threads - two load operations

Optimisaton #1: No load-only threads



Second load - only two threads read from memory

Have only compute threads - two load operations

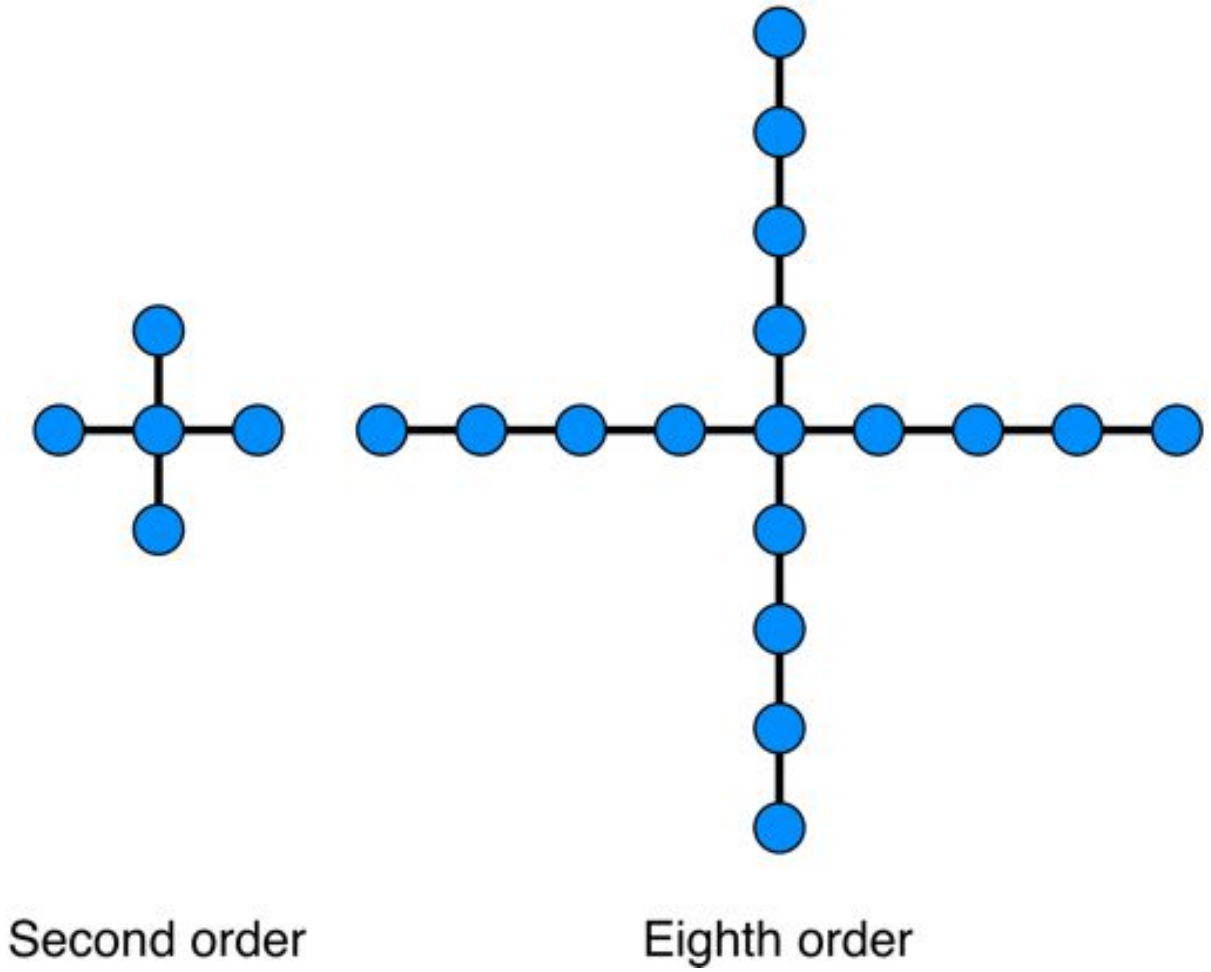
Optimisation #2: Use all on-chip memory

- Real applications often have large stencils
- Inputs: A, B, C, D, E
- Outputs: Y1, Y2, Y3
- Stencils: f1, f2, f3
- Y1 = f1(A, B, C) + f2(D, E)
- Y2 = Y1 + F3(B, C, D)
- Y3 = Y1 + Y2 + F4(A, E)

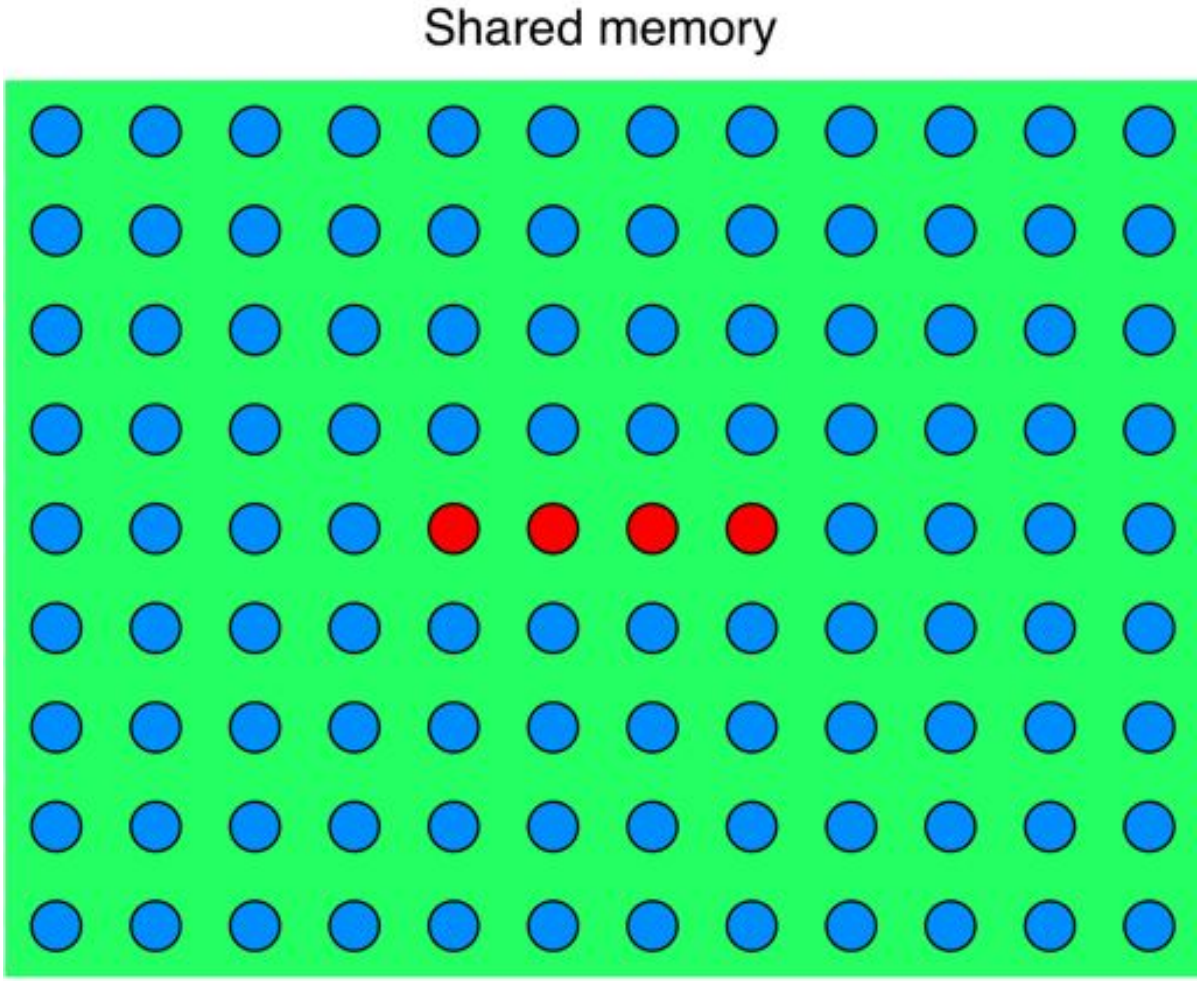
Optimisation #2: Use all on-chip memory

- Many arrays lead to large pressure on shared memory
- Sub-blocks become small
- Halo nodes become a large fraction of total nodes in sub-block
- Place some inputs in texture cache (GT200) or normal cache (Fermi)

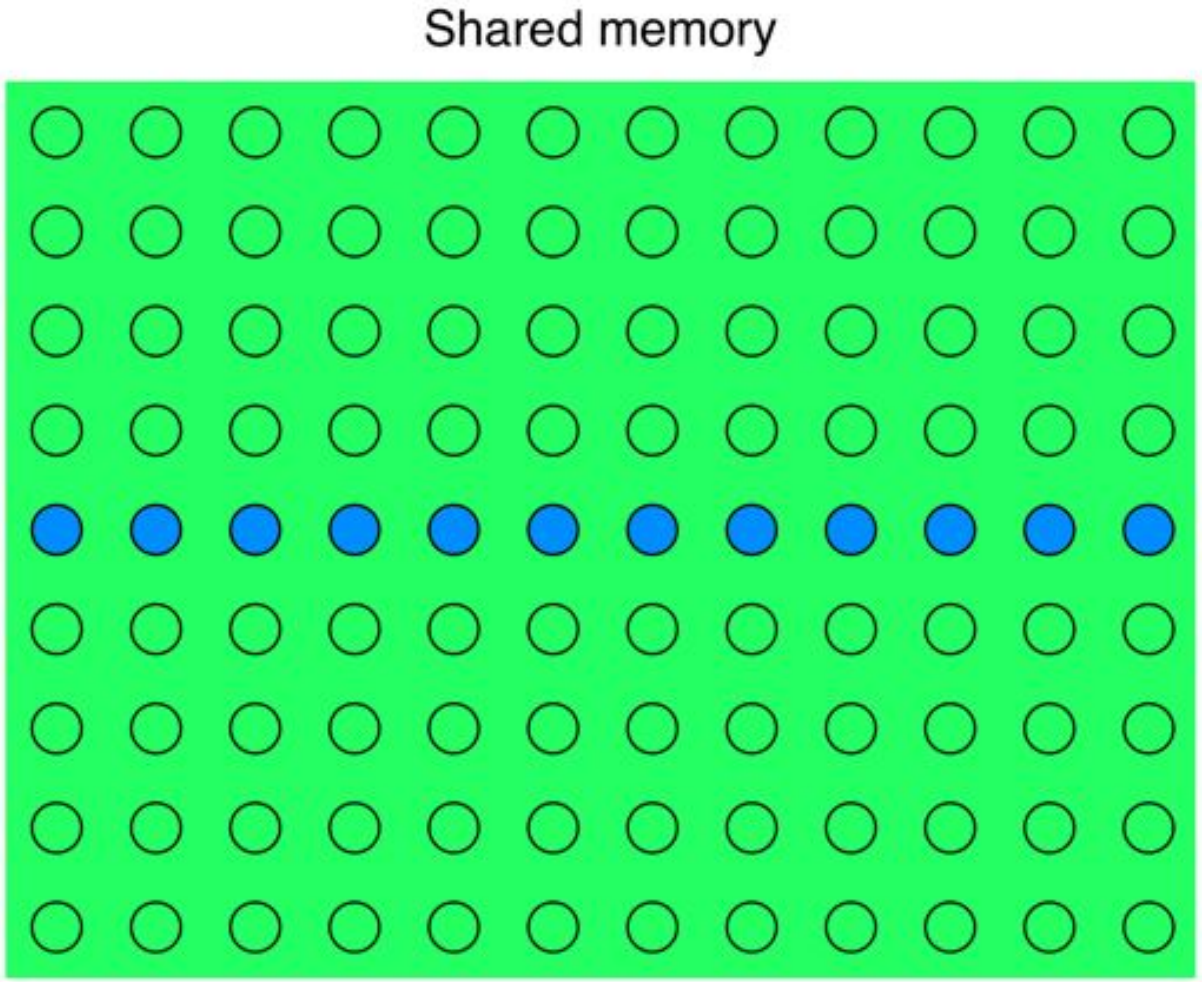
Optimisation #3: High-order stencils



Optimisation #3: High-order stencils

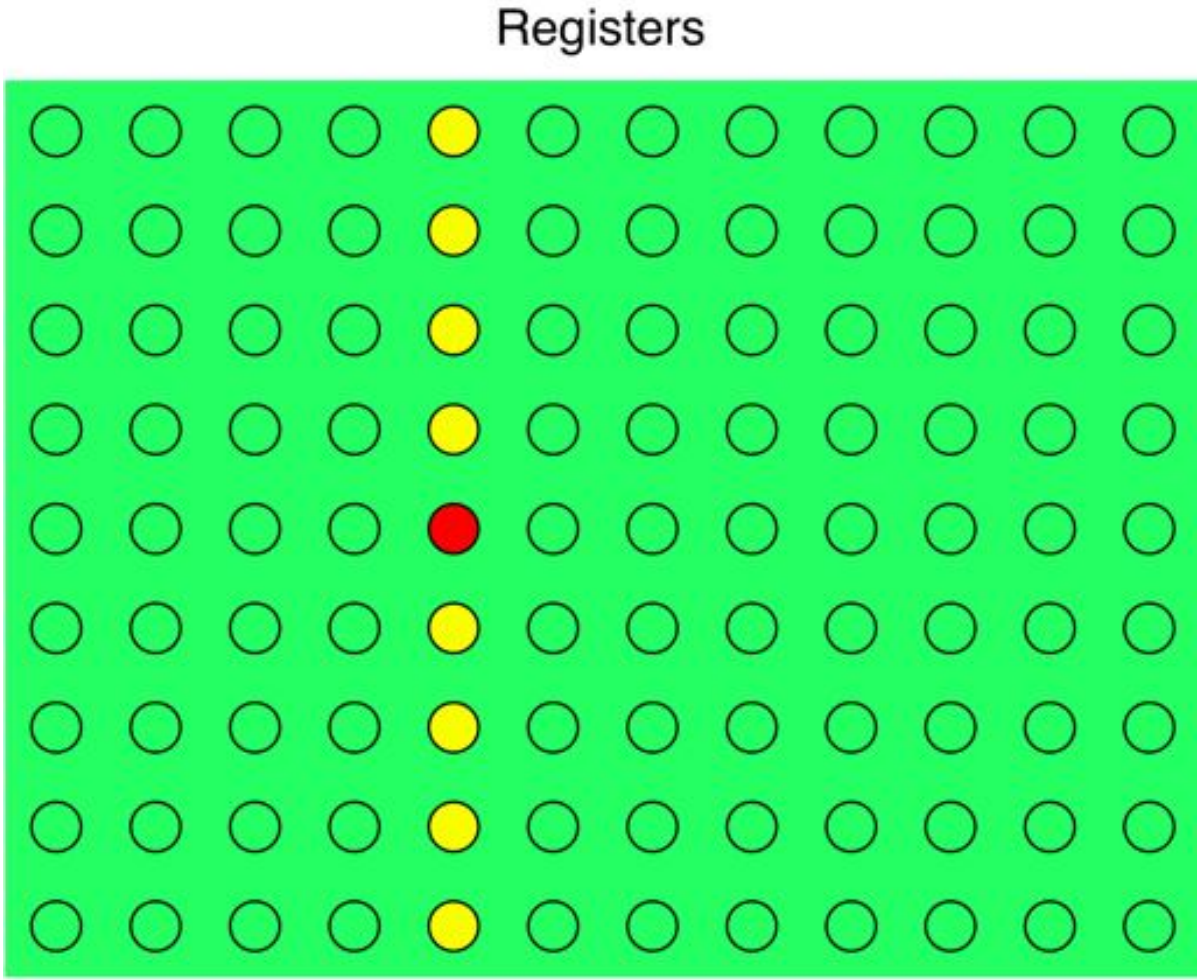


Optimisation #3: High-order stencils



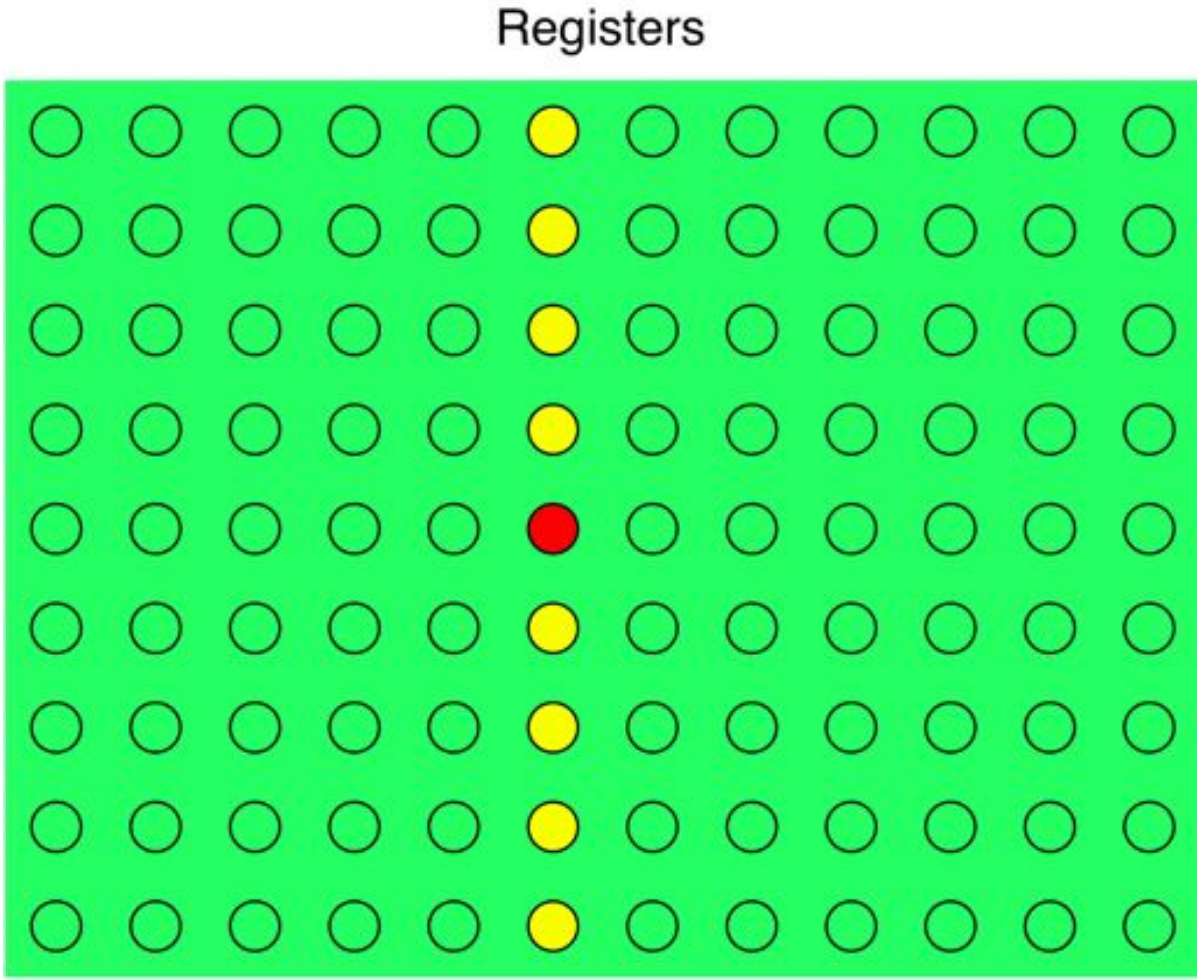
Only the grid nodes in one plane need to be shared to evaluate the stencil - store these in shared memory

Optimisation #3: High-order stencils



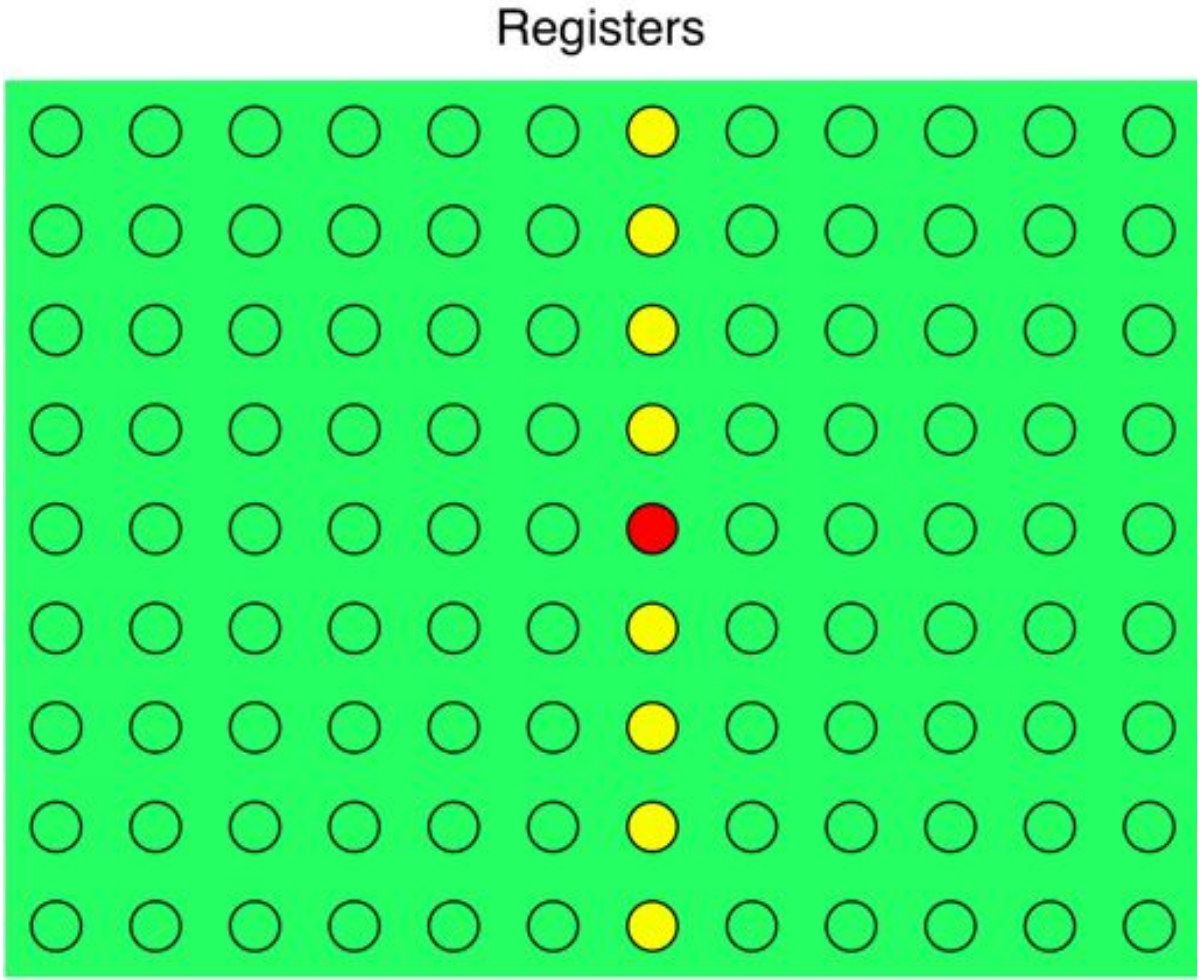
The other grid nodes can be stored in registers for each thread

Optimisation #3: High-order stencils



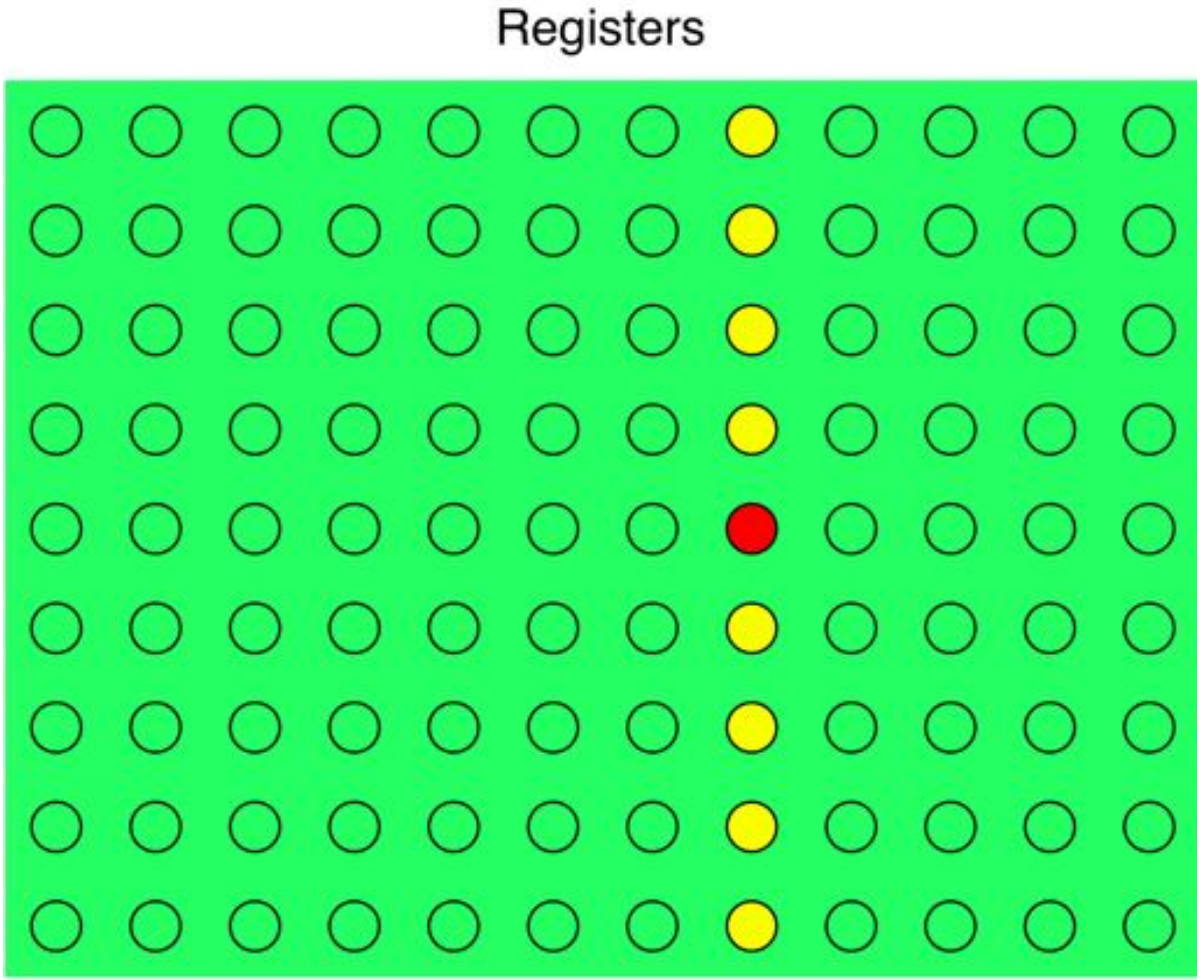
The other grid nodes can be stored in registers for each thread

Optimisation #3: High-order stencils



The other grid nodes can be stored in registers for each thread

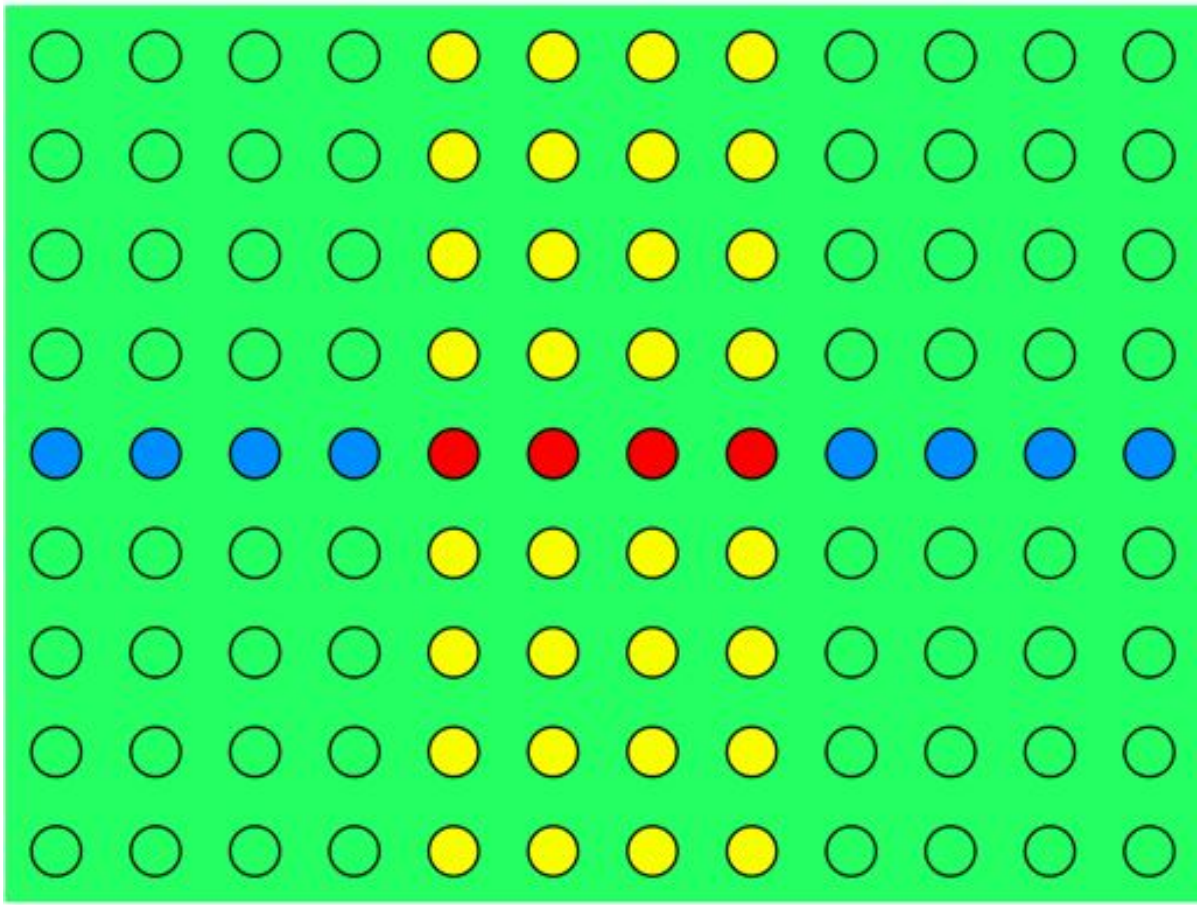
Optimisation #3: High-order stencils



The other grid nodes can be stored in registers for each thread

Optimisation #3: High-order stencils

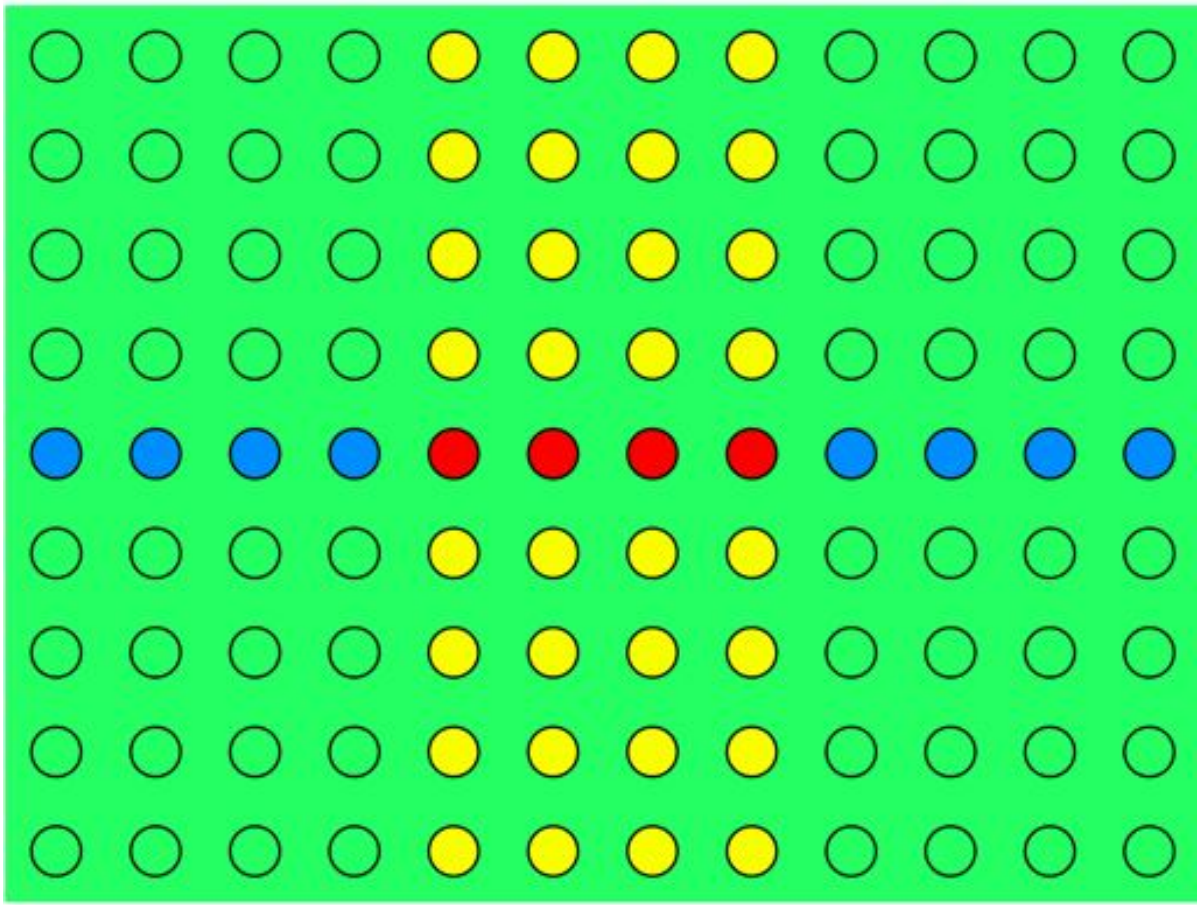
Shared memory and registers



Final mix of shared memory and register storage

Optimisation #3: High-order stencils

Shared memory and registers



Final mix of shared memory and register storage

Real stencil applications

- Real stencil applications:
  - Many complicated stencils
  - Should run on any processor
  - Should scale across multiple processors

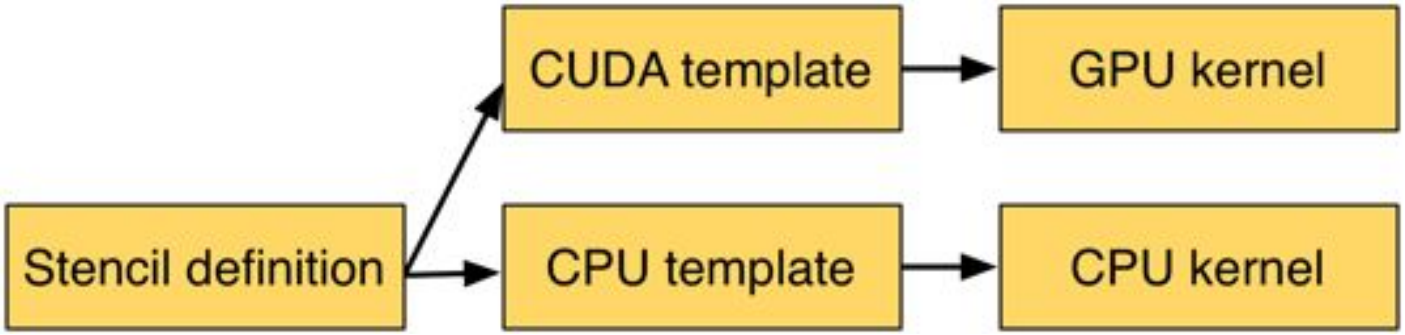
A framework of stencil applications

- We have created a software for applications that use stencil operations
- “SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms”, T. Brandvik and G. Pullan, CIT 2010.
- Consists of two components:
  - Run-time library (memory allocation, MPI)
  - Kernel generator (source-to-source compiler)

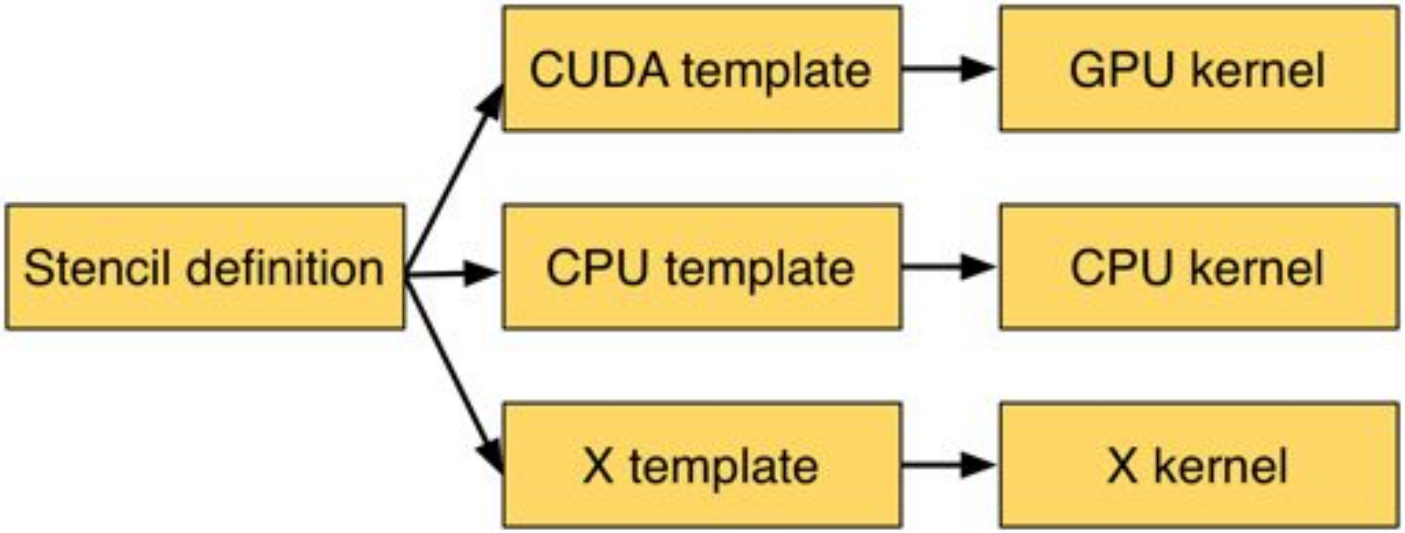
SBLOCK kernel generator

- Different implementation strategies are necessary for different processors:
  - NVIDIA GPUs (CUDA)
  - CPUs (pthreads, SSE)
  - Other GPUs (OpenCL)
- Hand-coding each stencil is not feasible
- Kernel generator transforms a high-level definition of a stencil into optimised low-level source code

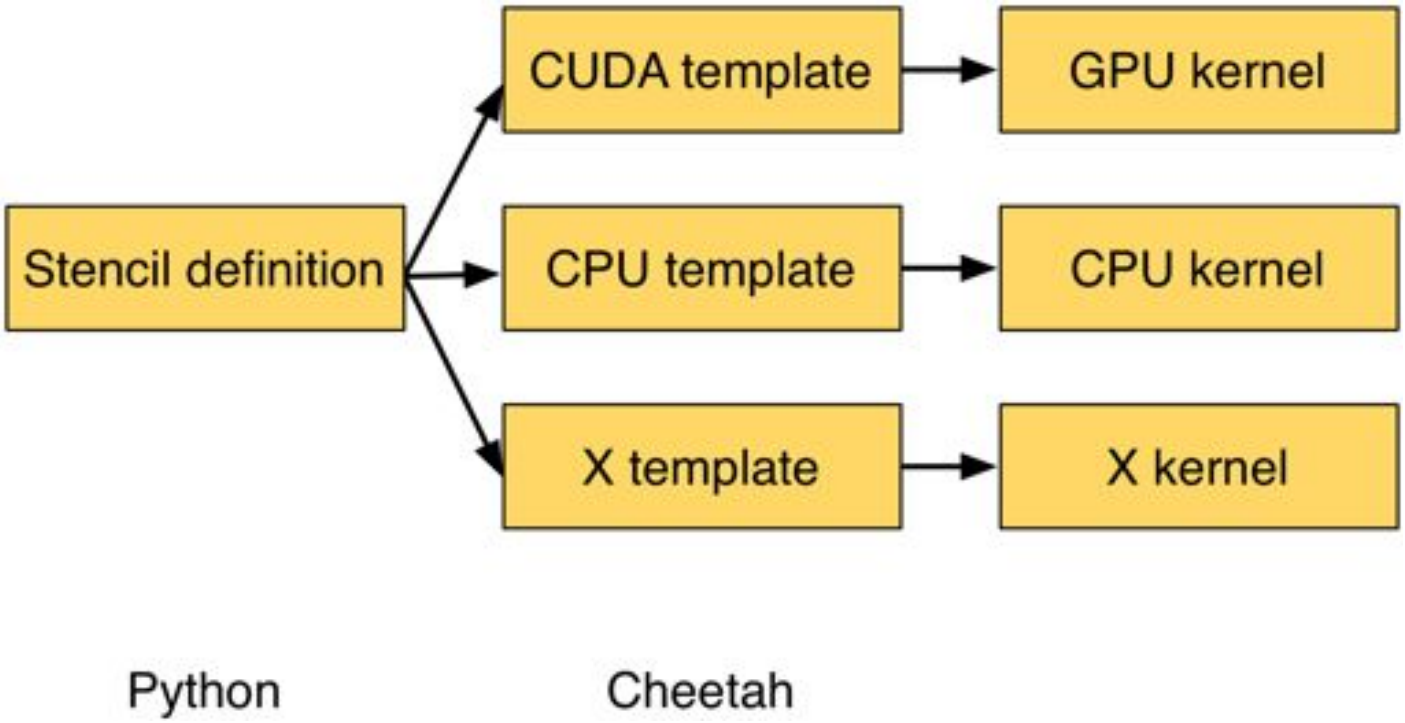
SBLOCK kernel generator



SBLOCK kernel generator



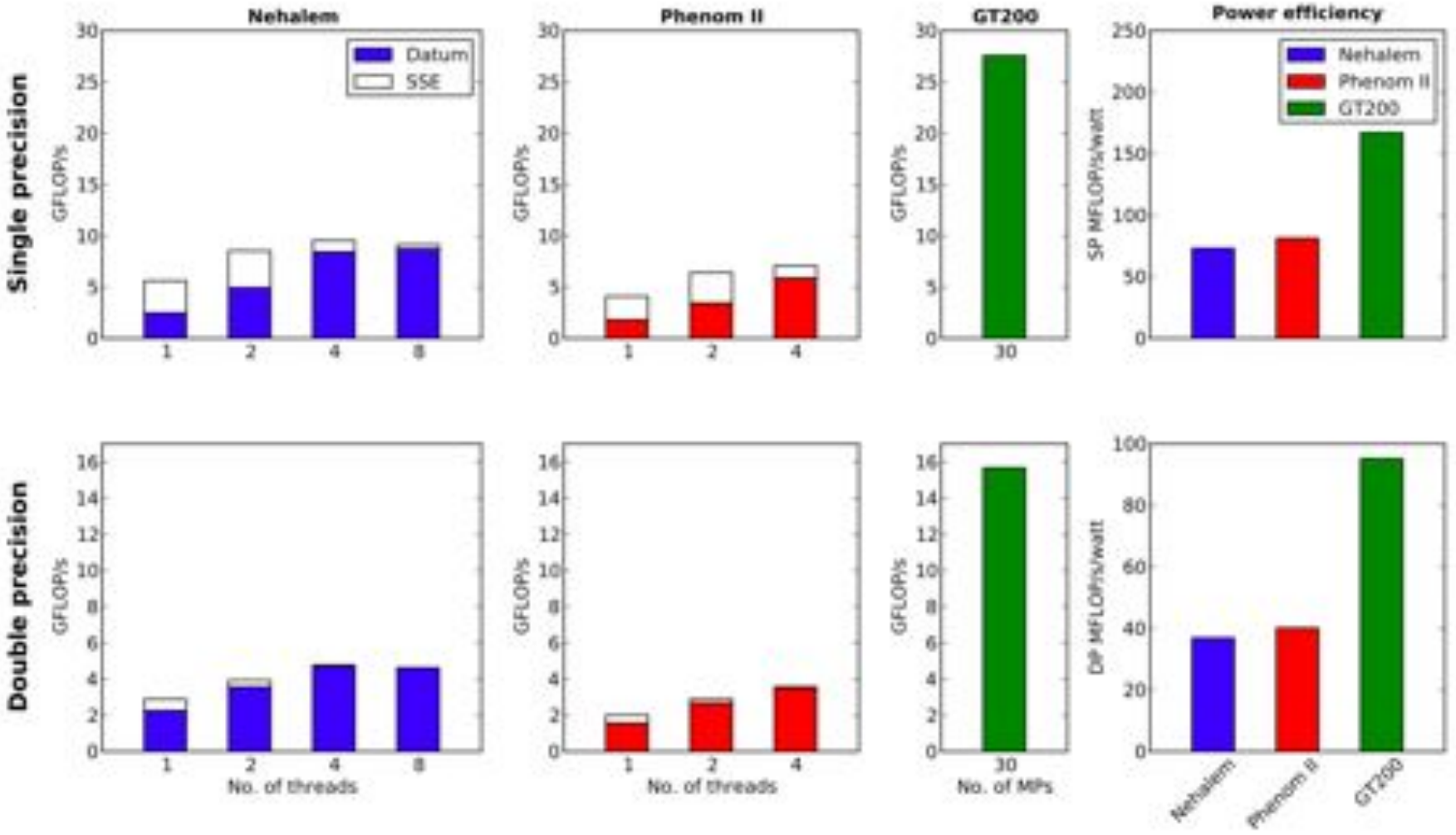
SBLOCK kernel generator



SBLOCK performance

- Laplace stencil
- Intel Nehalem (quad-core, 2.66 GHz)
- AMD Phenom II (quad-core, 3.0 GHz)
- NVIDIA Tesla (GT200)

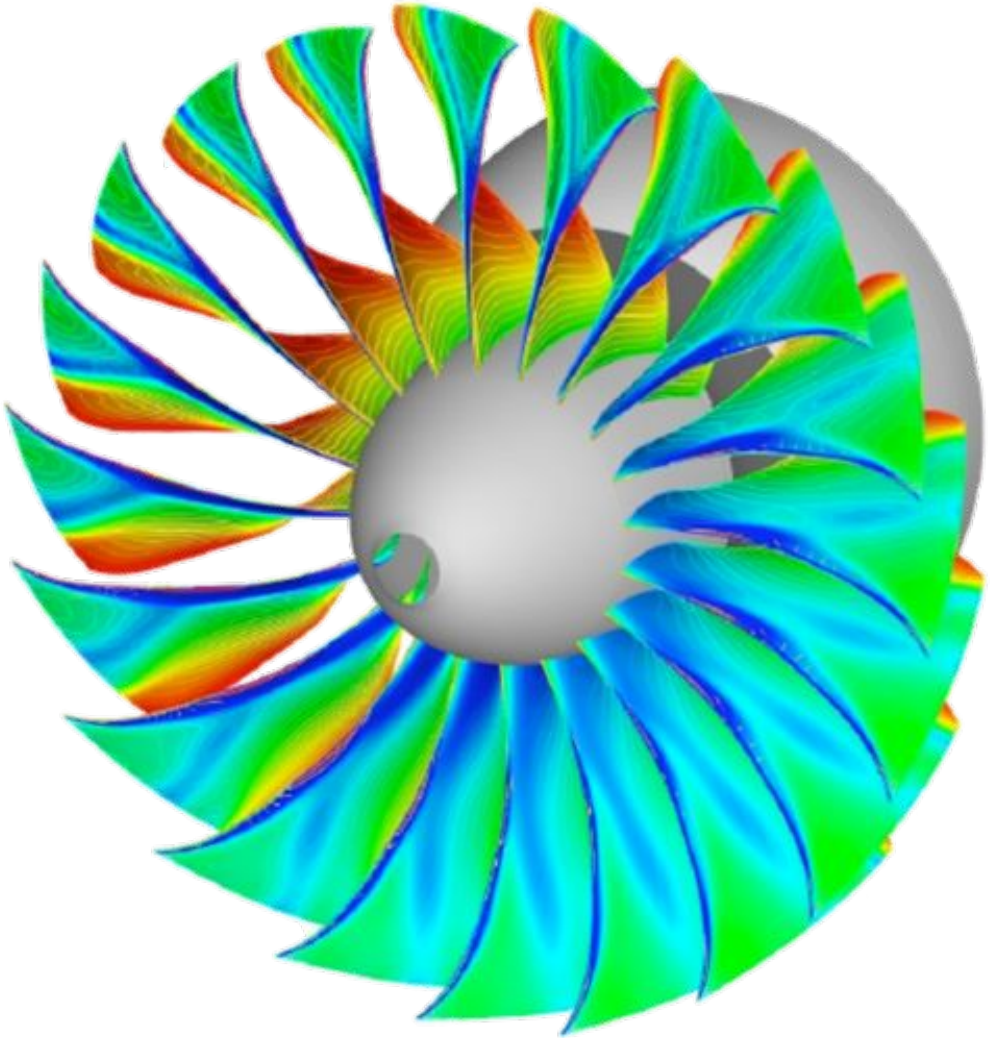
SBLOCK performance



Turbostream: A CFD solver implemented with SBLOCK

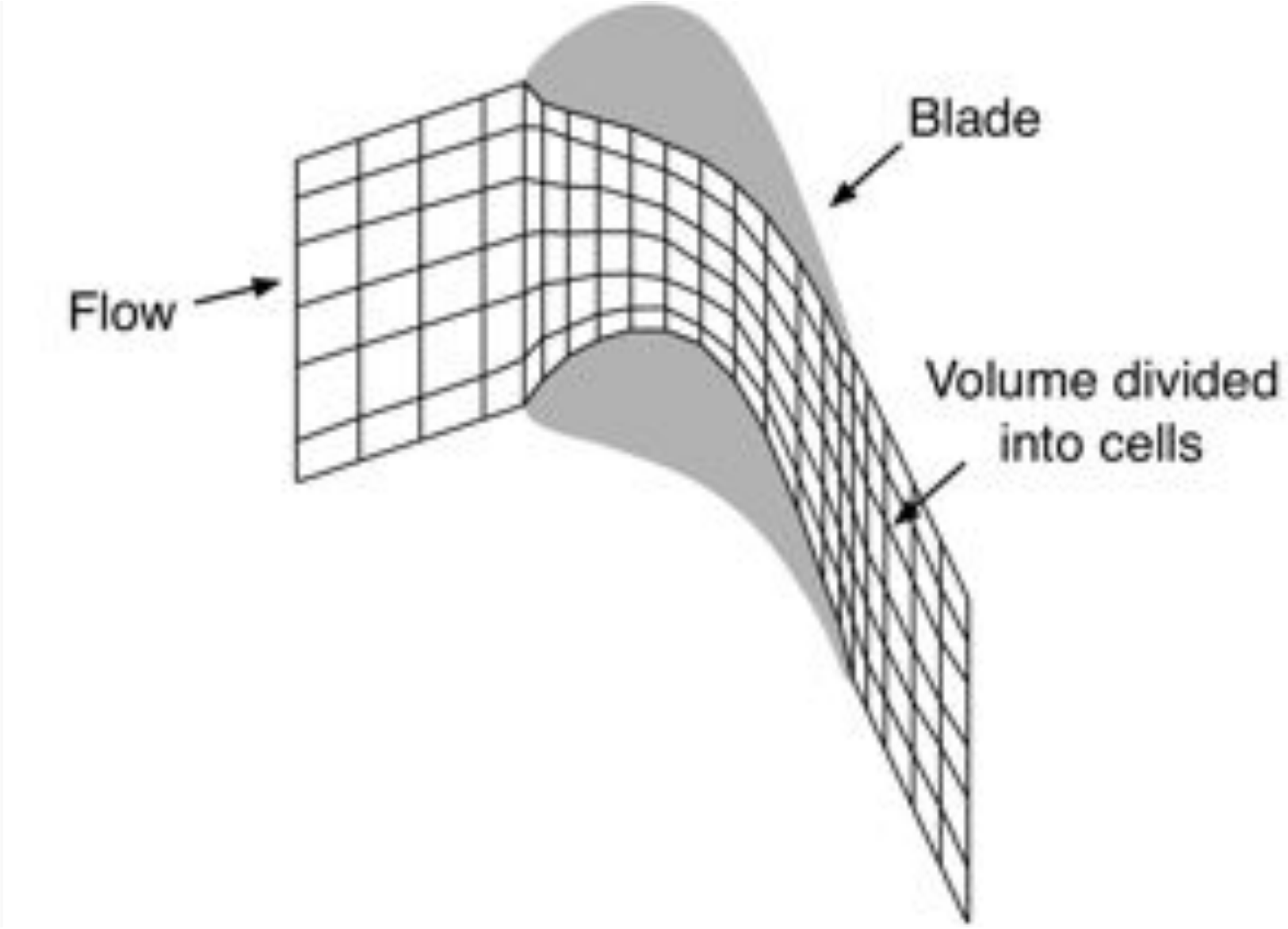
- Turbostream is a new CFD solver for the flow in turbomachines
- Based on the existing “Denton codes” from John Denton (Fortran+MPI)
- The Denton codes date back to the 1970s and are used in some part of the design process of most turbomachinery manufacturers

CFD of a jet engine

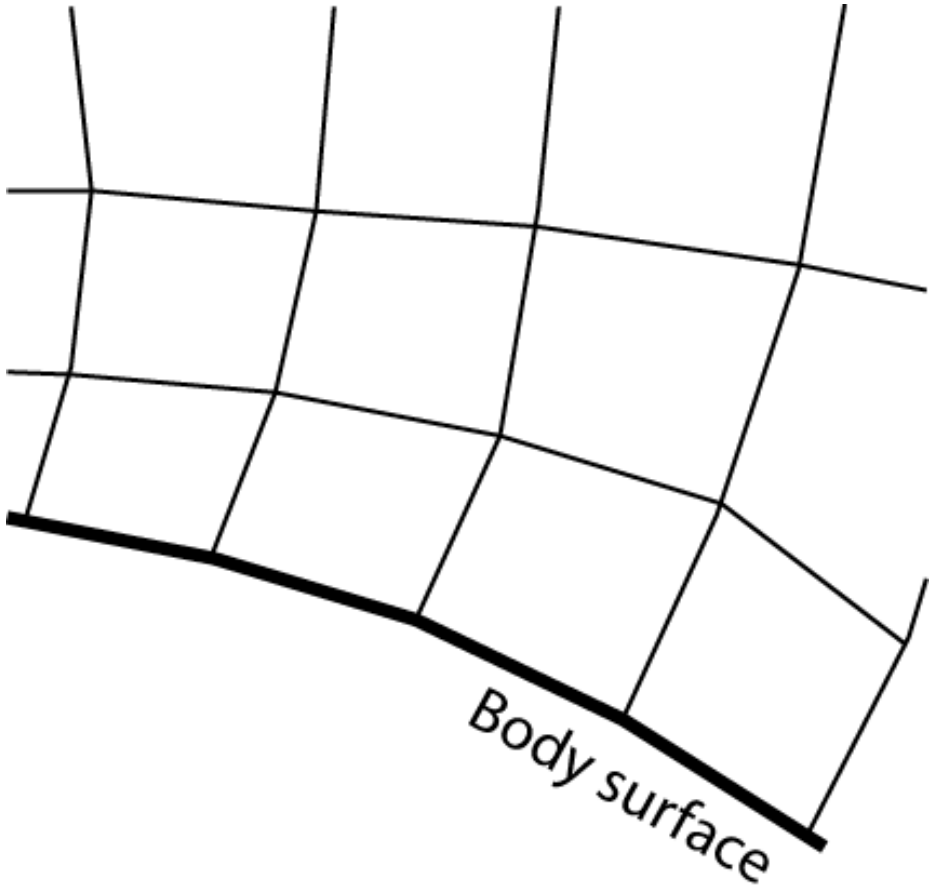


Blades coloured by pressure

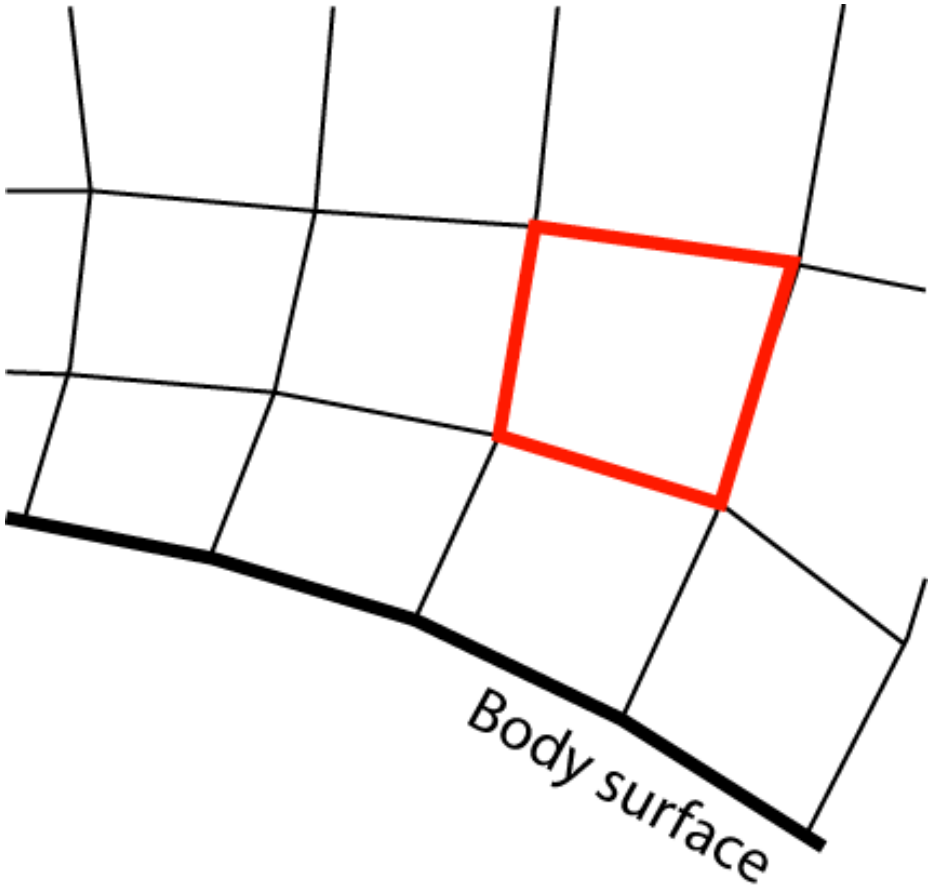
Introduction to CFD



Governing equations for each cell



Governing equations for each cell

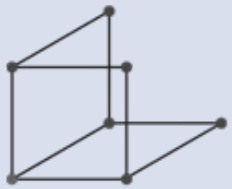
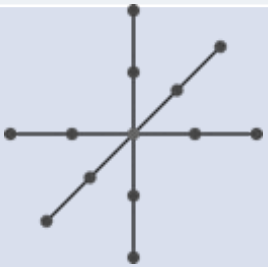


Conserve:
- Mass
- Momentum
- Energy

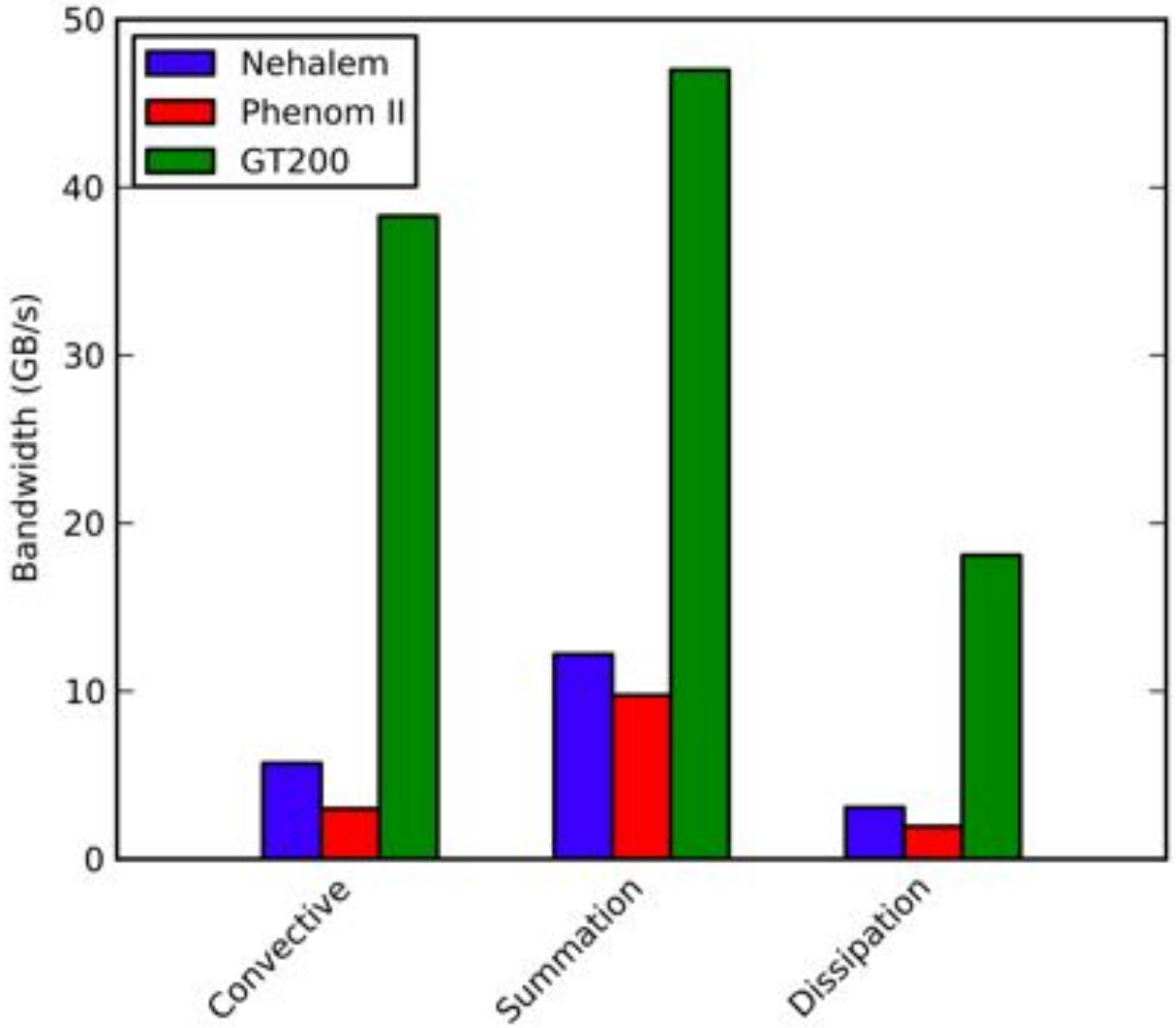
Turbostream overview

- 20 stencil kernels
- Many different boundary conditions
- 7,500 lines of Python stencil definitions
- 20,000 lines of low-level source code after templates

Turbostream kernels

Kernel	Stencil	SP Flops	Bytes	AI
Convective fluxes		260	139	1.87
Flux summation		8	28	0.29
Artificial dissipation		242	44	5.5

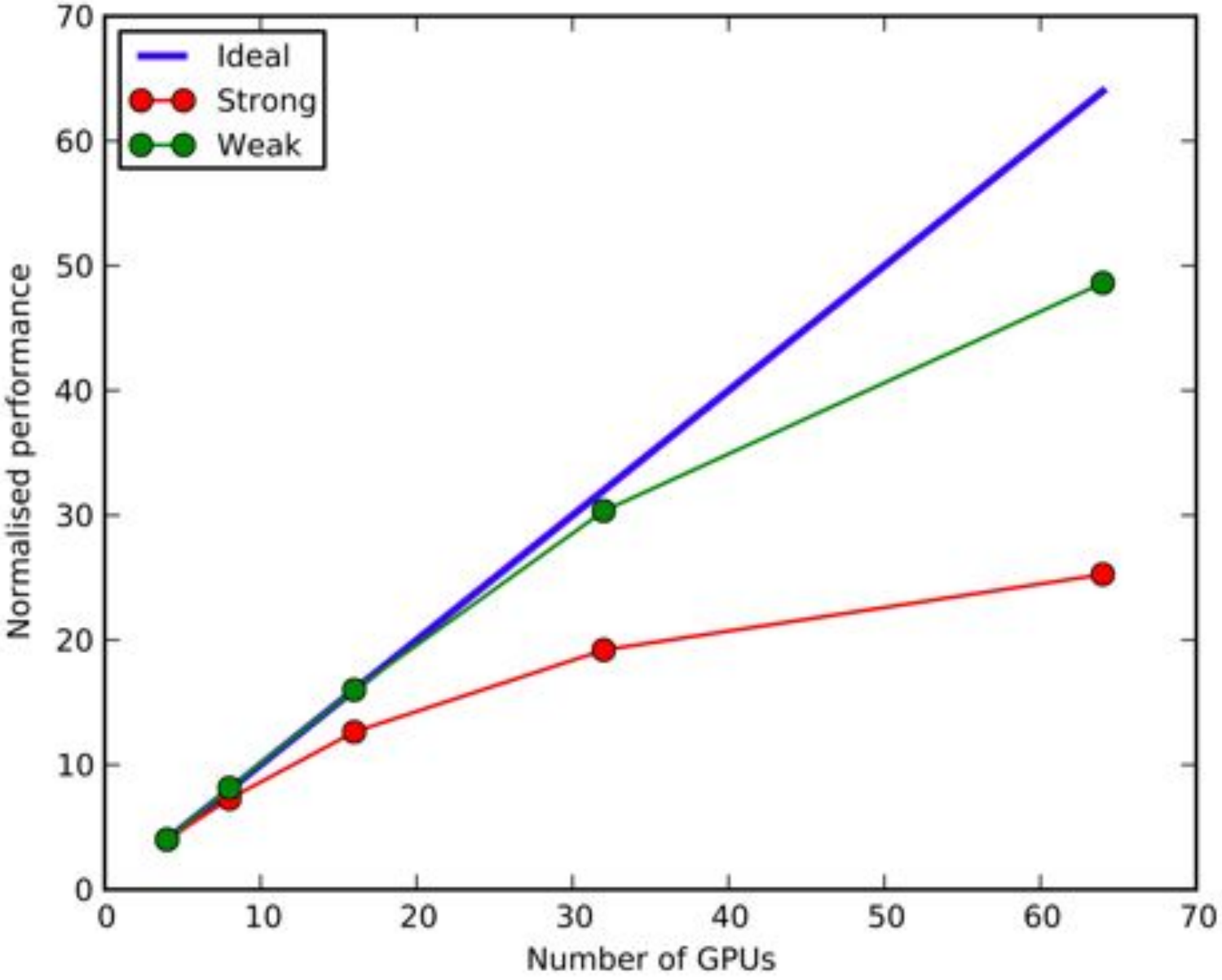
Turbostream kernels



Single-processor performance

Processor	Original Fortran	Turbostream
Phenom II	1	0.89
Nehalem	1.21	1.48
GT 200	-	10.2

Multi-GPU performance



Impact on Turbomachinery CFD

- Typical design calculations:
  - 1-2 million grid nodes
  - Hours -> 10 minutes
  - Many more design iterations
- High-resolution simulations:
  - 20–200 million grid nodes
  - Months -> 1-2 weeks
  - Can be used in industry for the first time

Turbostream Ltd

- A spin-off company has been formed around the Turbostream solver
- www.turbostream-cfd.com
  - Case studies
  - All papers and presentations

Case study

- Full-annulus unsteady simulation of a compressor test rig at Siemens, UK
- 160 million grid nodes
- 32 NVIDIA C1060 GPUs
- 24 hours per revolution

Animation

- http://vimeo.com/20403928

SIEMENS

The Texture Unit as Performance Booster in CT-Reconstruction

Dr.-Ing. Karl Schwarz

Siemens AG

Healthcare

Computed Tomography

H IM CR R&D SE

Forchheim

Germany

Siemens AG 2011

SIEMENS

Computed Tomography

CT milestones:

from 1895

- Discovery of X-rays
- Radon transformation
- Axial CT
- Spiral CT
- Multislice CT
- 3D-Backprojection
- Dual Tube
- Iterative Reconstruction

until today

Somatom Definition Flash

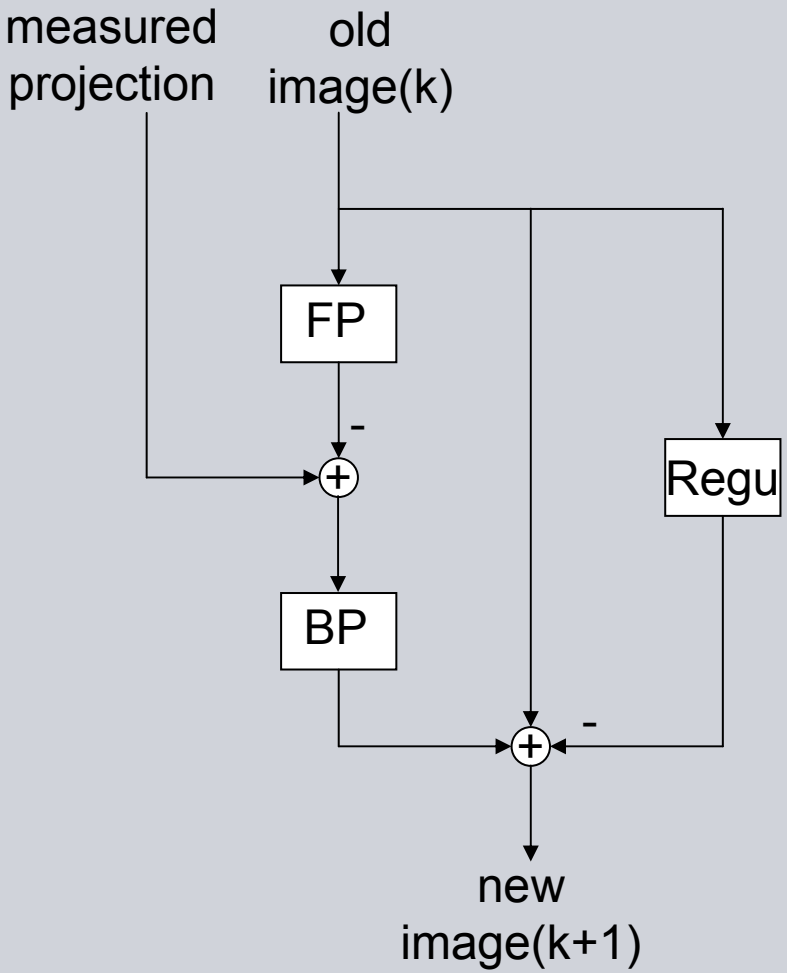
SIEMENS

The Current Challenge in CT: Iterative Reconstruction

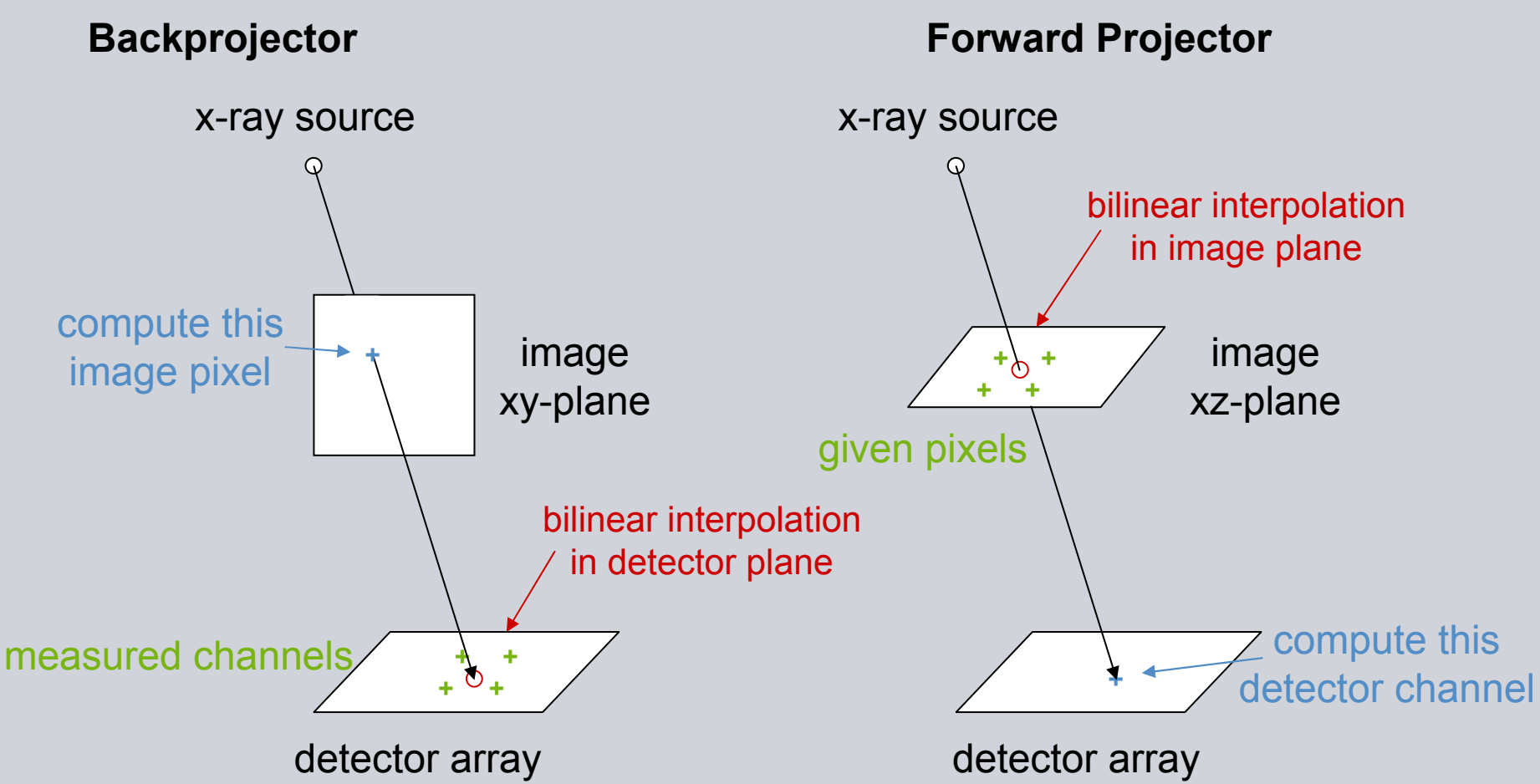
Main Building Blocks in Iterative Reconstruction :

- Backprojector (BP)
  - computes the image
- Forward Projector (FP)
  - simulates the measurement
- Regularization Filter (Regu)
  - noise reduction

One Iteration Step



Bilinear Interpolation



SIEMENS

Subtitle :

“With a Little Help from” [1]

the Texture Unit

[1] Beatles: 2nd title on album “Sgt. Pepper’s Lonely Hearts Club Band”, Abbey Road Studios, London, 1967

Siemens AG 2011

SIEMENS

Texture Features

linear interpolation :	**tex1d**(vector, x)	= 1 clock cycle
bilinear interpolation :	**tex2d**(matrix, x, y)	= 1 clock cycle
trilinear interpolation :	**tex3d**(volume, x, y, z)	= 3 clock cycles

- nearest neighbour access or interpolation
- border handling (clamp or wrap)
- format conversion (short to float, half to float)
- separate texture cache

SIEMENS

Peak Performance Comparison

	Tesla C1060	**Tesla C2050**	speedup %
Shader	240 x 1.296 GHz	448 x 1.15 GHz	
GCycles/sec	311	515	+ 65
GFlops*	622	1030	
Texture	80 x 0.61 GHz	56 x 0.575 GHz	
GTexel/sec	48.8	32.2	- 34
GOps**	976	644	
Sum GOps	1598	1674	+ 5
Texture/Shader Cycle Ratio	1 / 6.4	1 / 16	- 60

Comparison of theoretical peak performance

* one fused multiply add per shader clock cycle

** 1 Texel = 20 operations

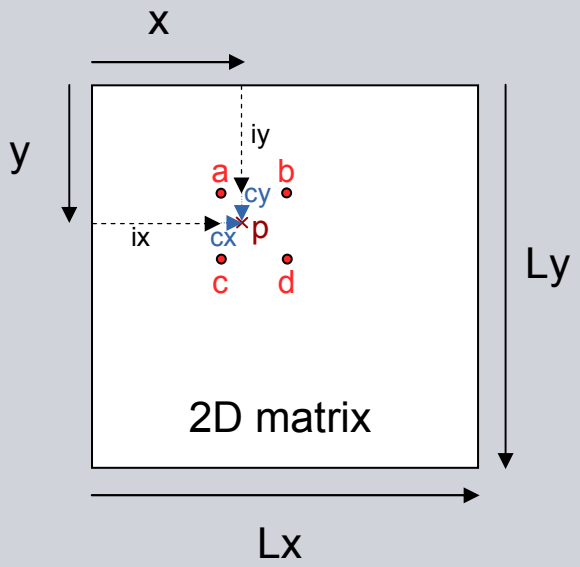
SIEMENS

Bilinear Interpolation

CUDA:

p = tex2D(matrix,x+0.5f,y+0.5f);

Conventional C:	operations	moves
ix = (int) x;	1	
cx = x - (float)ix;	2	
iy = (int) y;	1	
cy = y - (float)iy;	2	
addr = iy * Lx + ix;	2	
a = matrix[addr];		1
b = matrix[addr + 1];	1	1
addr = addr + Lx;	1	
c = matrix[addr];		1
d = matrix[addr + 1];	1	1
g = a + cx * (b - a);	3	
h = c + cx * (d - c);	3	
p = g + cy * (h - g);	3	
sum:	**20**	**4**

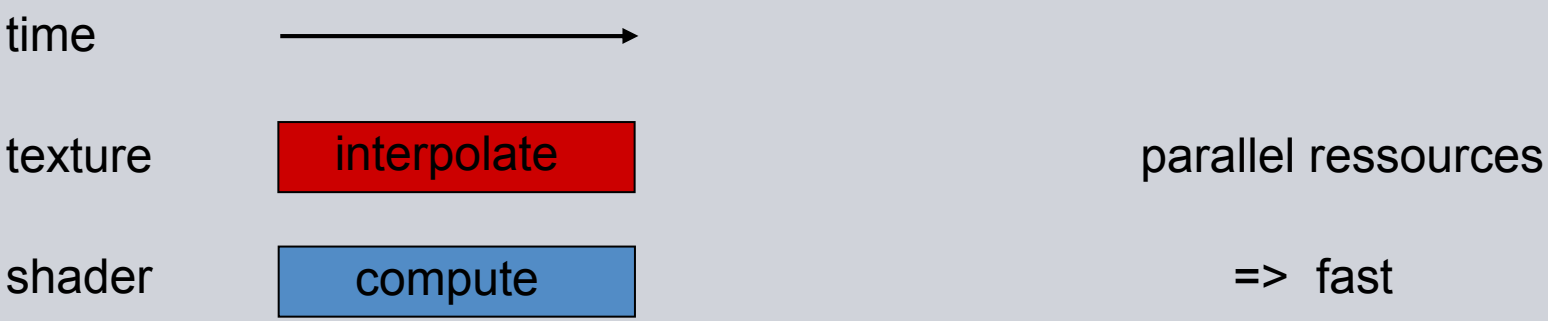


rewrite linear interpolation:
a*(1-c) + b*c = a + c*(b-a)

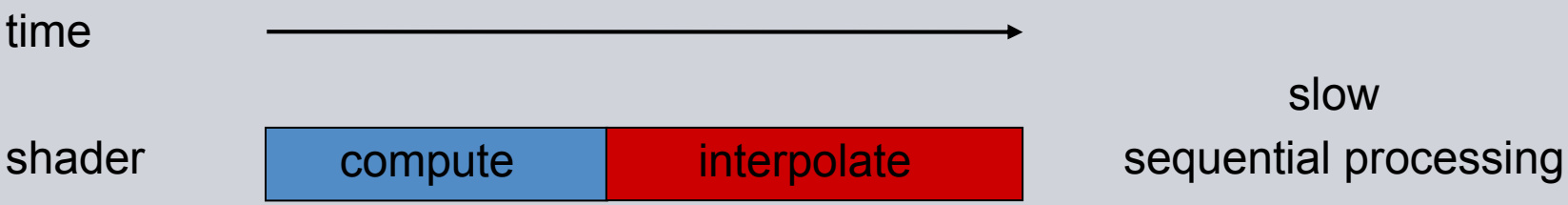
SIEMENS

Texture Discussion

Textures and shader work in parallel:



If all work is done only by shader (no textures used):



SIEMENS

Texture Discussion (GF100)

Additional advantages of textures:

- **separate texture cache** for every SM (Streaming Multiprocessor)
- **border handling** (clamp, wrap)
- **format conversion** (16 bit half float to 32 bit float, 16 bit short to 32 bit float)

Handicaps of textures:

- precision of interpolation coefficients: 8 bit resolution
- limited texture resources => multiple textures, texture atlas
- +0.5 in texture argument: tex2D(matrix,x+0.5f,y+0.5f);

Optimization Hints

Texture cache access basics (GF100):

- 4 subsequent threads access the texture memory at the same time
- only one cache line is transferred to this texture block per clock cycle
- a texture cache line is 32 bytes x 4

Use a more local texture memory access pattern:

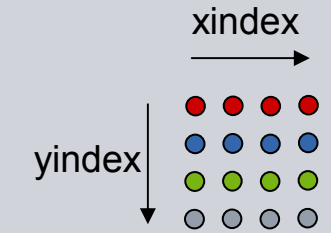
SIEMENS

Optimization Hints

Resort thread coordinates at the beginning of kernel:

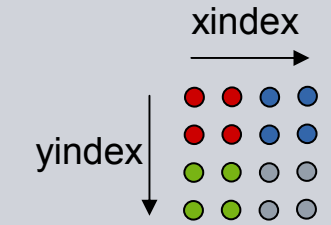
Before:

```
xindex = (blockIdx.x * blockDim.x) + threadIdx.x;
yindex = (blockIdx.y * blockDim.y) + threadIdx.y;
```



After:

```
x = threadIdx.x;
y = threadIdx.y;
x1 = (x>>1)&1;
y1 = y&1;
x = (x & ~2) | (y1 << 1);
y = (y & ~1) | x1;
xindex = (blockIdx.x * blockDim.x) + x;
yindex = (blockIdx.y * blockDim.y) + y;
```



more local texture access, but
storage of results is no longer coalescent

SIEMENS

Optimization Hints

If possible:

Reduce input data wordlength from 4 to 2 Byte.

from float to half float or
from float to short

- reduces main memory size by 2
- reduces necessary load bandwidth from memory by 2
- doubles content of the texture cache
- doubles content of one cache line (64 words against 32 words)

SIEMENS

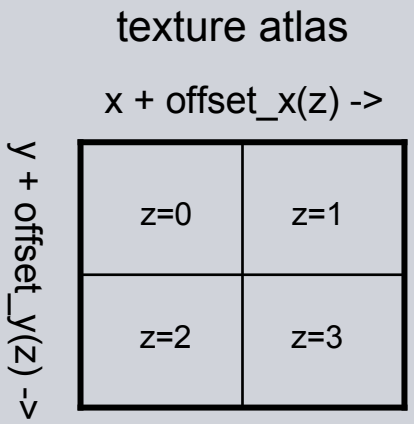
Texture Atlas

Shortcoming until CUDA 3.2

Currently we have to work with a **texture atlas**

tex2D(tex_atlas,x+**offset_x(z)**,y+**offset_y(z)**)

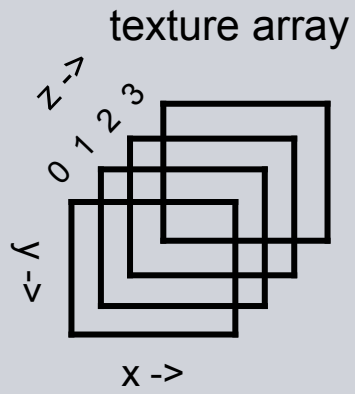
But we need **vectors of textures** to simplify handling



Solved with CUDA 4.0 : layered textures

tex2DLayered(tex_array,x,y,**z**)

z-dimension is integer index (up to 2^11),
similar to 3D texture,
but with bilinear interpolation only.



SIEMENS

Texture Precision (GF100)

tex2D(matrix,x,y)

Maximum 2D texture width and height is 65535 x 65536 (=2^{16})

Interpolation coefficient is a fraction between [0 , 1[.

The interpolation coefficient is represented with 8 bit.

32 bit floating point number representation according IEEE 754:

$x = (-1)^S 2^{E-127} (1.M)$

x = 33000.1 = 0x 4700 E81A = 0100 0111 0000 0000 1110 1000 0001 1010

sign exponent mantissa

S = 0 , E = 0x8E = 142 , M = 0x00E81A = 59418

1.M = 0x80E81A * 2^{-23} = 8448026 * 2^{-23}

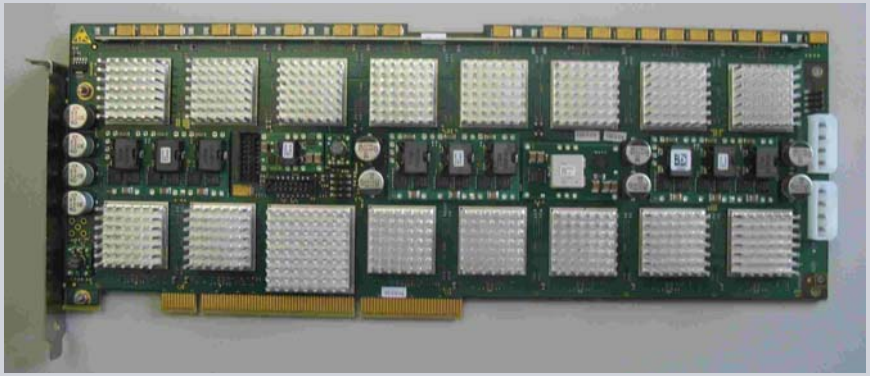
floor(1.M * 2^{15}) = 0x80E8 = 33000 = index

frac(1.M * 2^{15}) = 0x1A * 2^{-8} = 0.1015625 = coefficient

There are only 8 bit remaining for the coefficient !

SIEMENS

Mind the Fermi Gap

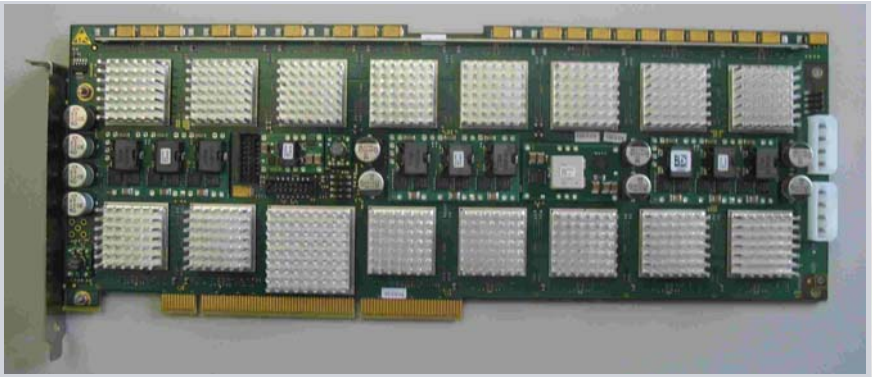


	CBR	Tesla C2050	increase %
Memory / GBytes	8	3	- 62
GTexel / sec	76.3	32.2	- 58
images / sec	40	25	- 37
power / W	200	238	+ 19
year	2006	2010	+ 0.2
technology	FPGA	GPU	
manufacturer	Siemens	Nvidia	

3D-Backprojector comparison: CBR versus Tesla C2050

SIEMENS

Solve the Fermi Gap



	1 x CBR	3 x Tesla C2050	increase %
Memory / GBytes	8	9	+ 13
GTexel / sec	76.3	96.6	+ 27
images / sec	40	64	+ 60
power / W	200	714	+ 257
year	2006	2010	+ 0.2
technology	FPGA	GPU	
manufacturer	Siemens	Nvidia	

3D-Backprojector comparison: 1 x CBR versus 3 x Tesla C2050

SIEMENS

Motivation

A GPU offers:

- programmable in software
- can run several algorithms
- low development costs
- fast development cycles for algorithms
- free performance upgrade with next GPU generation

SIEMENS

Accelerator Technologies

- **ASIC** : highest performance
  very expensive development
  longest development cycle
  no flexibility
- **FPGA** : high performance
  expensive development
  long development cycle
  programmable in firmware
- **DSP** : programmable in software
  low power consumption
- **CPU** : mainstream
  programmable with all kind of languages and tools
- **GPU** : fastest software programmable solution
  programmable in CUDA and OpenCL

SIEMENS

Take Home Messages

- Medical imaging depends on bilinear interpolations.
  => Textures

- Texture units are powerful hardware resources.
  Therefore use them !

- Texture units can deliver a performance increase by a factor ≥ 2 !

Computed Tomography
SOMATOM Definition Flash

SIEMENS

Thank You

Courtesy of: Friedrich-Alexander University Erlangen-Nuremberg - Institute of Medical Physics / Erlangen, Germany

Optimisation Myths and Facts as Seen in Statistical Physics

Massimo Bernaschi
Institute for Applied Computing
National Research Council
&
Computer Science Department
University "La Sapienza"
Rome - ITALY
m.bernaschi@iac.cnr.it

Presentation Outline

- Update by overrelaxation of Heisenberg spin-glass
- Survey on possible implementations for GPU
- Results on GPU and comparison with a high-end multicore CPU
- Multi-GPU: possible implementations and results
- Conclusions

The (3D) Heisenberg Spin Glass Model

A system whose energy is given by:

$$-\sum J_{[x,y,z],[x\pm1,y\pm1,z\pm1]}\sigma_{[x,y,z]}\sigma_{[x\pm1,y\pm1,z\pm1]}$$

where the sums run on nearest neighbors.

- $\sigma_{[x,y,z]}$ is a 3-component vector $[a, b, c]$
  - $[a, b, c] \in \mathbb{R}$
  - $||\sigma|| = \sqrt{a^2 + b^2 + c^2} = 1$
- $J_{[x,y,z],[x\pm1,y\pm1,z\pm1]} \in \mathbb{R}$ are scalar random variables with Gaussian distribution:
  - average value equal to 0
  - variance equal to 1
  - for $J > 0$ spins tend to align but for $J < 0$ spins tend to misalign

spins are frustrated... and simulations take long times!

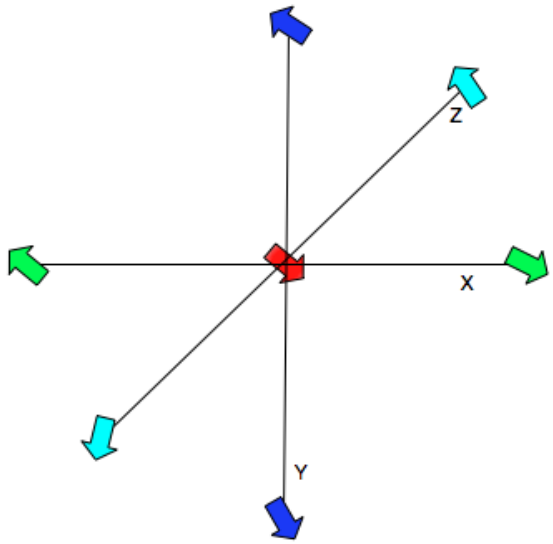
Update by Overrelaxation

The overrelaxation is the maximal move that leaves the system energy invariant:

```
foreach σ⃗ in (SPIN SYSTEM){
    H⃗_σ = 0⃗
     foreach n in (NEIGHBORS OF σ⃗){
         H⃗_σ = H⃗_σ + J_n σ⃗_n
     }
    σ⃗ = 2(H⃗_σ · σ⃗ / H⃗_σ · H⃗_σ) H⃗_σ − σ⃗
}
```

a set of scalar products that involve only simple floating point arithmetic operations.

In 3D, for a single spin:

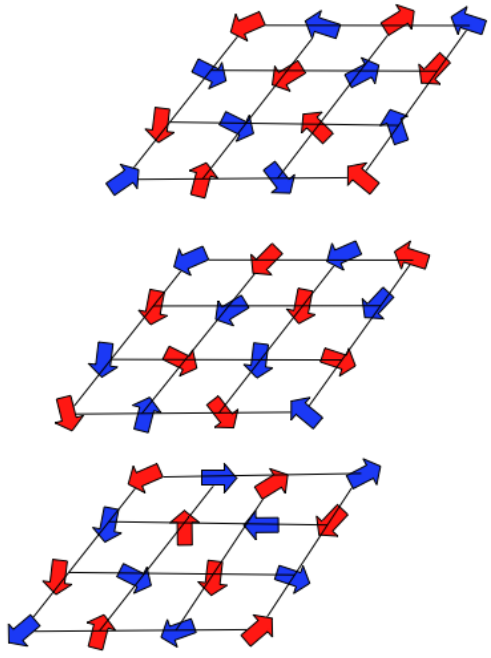


- 20 sums
- 3 differences
- 28 products
- one division ($\sim$ 10 Flops).

Check-board decomposition

Spins can be updated in parallel if they don't interact directly.

- Simple *(check-board)* decompositions of the domain can be used to guarantee the consistency of the update procedure.

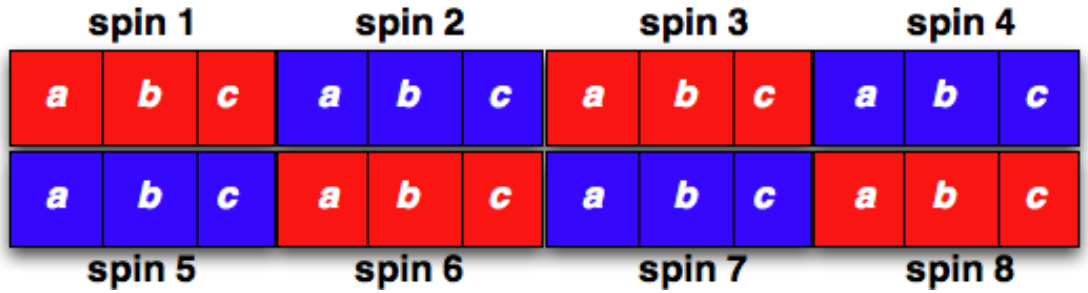


Data structures

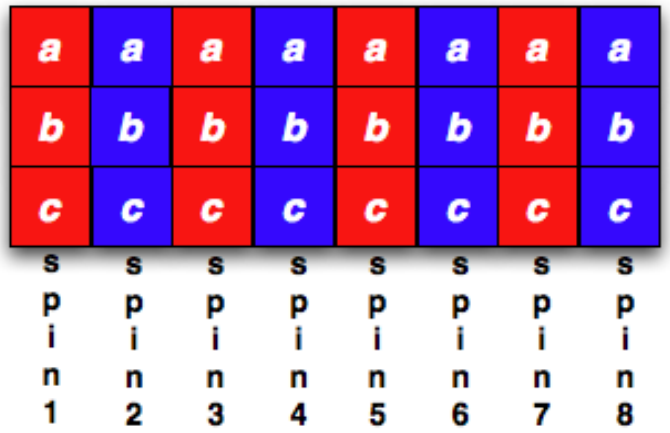
Spins: each spin is a 3-component ($[a, b, c]$) vector.

At least two alternatives (assuming C-like memory ordering).

- array of structures
  (good for data locality)



- structure of arrays
  (good for thread locality)



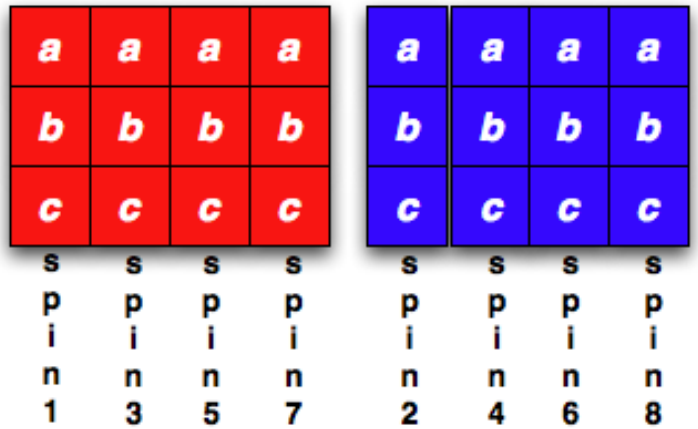
Couplings: each coupling is a scalar but there is a different coupling for each spin in each direction (x, y, z).
So we have the same possible alternatives:

Structure of Arrays (SoA) on GPU and Array of Structures (AoS) on CPU.

A survey of possible techniques: GPU global memory + registers (1)

A straightforward approach: each thread loads data directly from GPU global memory to GPU registers (no shared memory variables)

- Use the *structure of arrays* data layout with red and blue spins stored separately
- Two kernel invocations:
  one for red spins;
  one for blue spins.



- Very limited resource usage: 50 registers. No shared memory.
- Up to 640 threads per SM on Fermi GPUs.

A survey of possible techniques: GPU global memory + registers (2)

In spite of its simplicity this scheme produces a lot of data traffic:

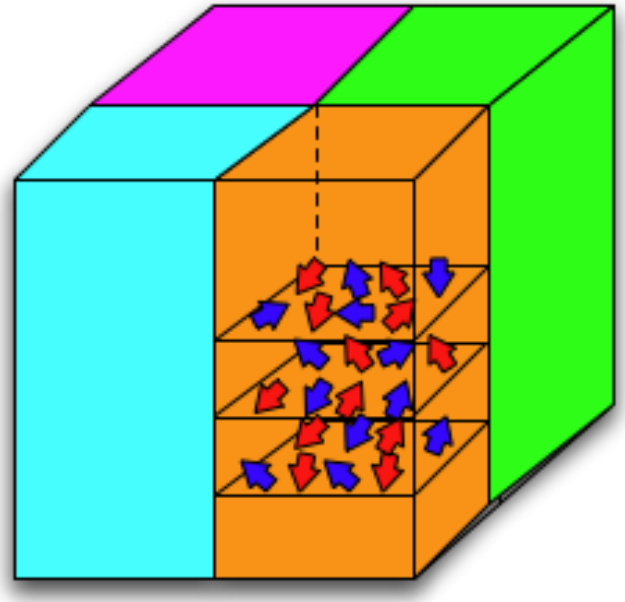
- each spin has 6 neighbors;
- each spin loaded exactly **7** times from the Global Memory (assuming periodic boundary conditions).
- one time, by one thread, for its update;
- six times (by six other threads) to contribute to the update of its neighbors.

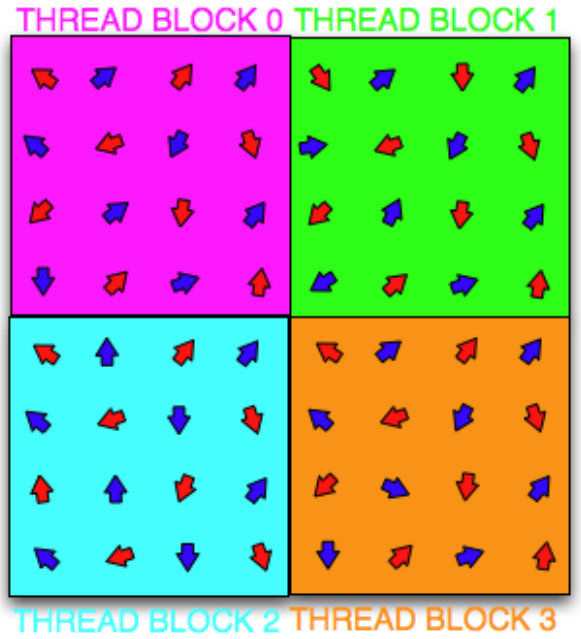
It looks like a situation where GPU *shared memory* could come to rescue...

A survey of possible techniques: GPU shared memory (3-plane version) (1)

An approach originally proposed to solve the Laplace equation

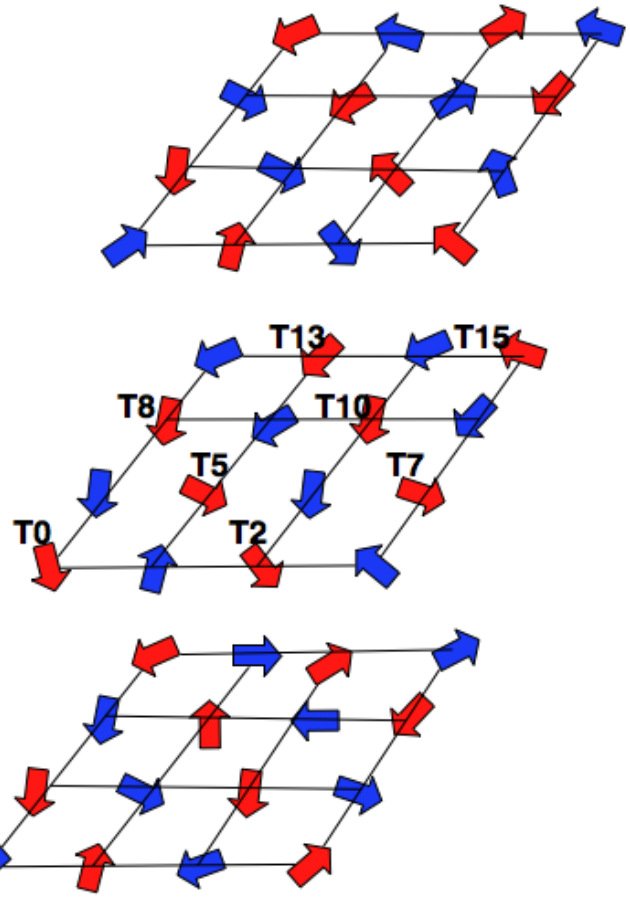
- The domain is divided in columns.
- A single block of threads is in charge of a column





A survey of possible techniques: GPU shared memory (3-plane version) (2)

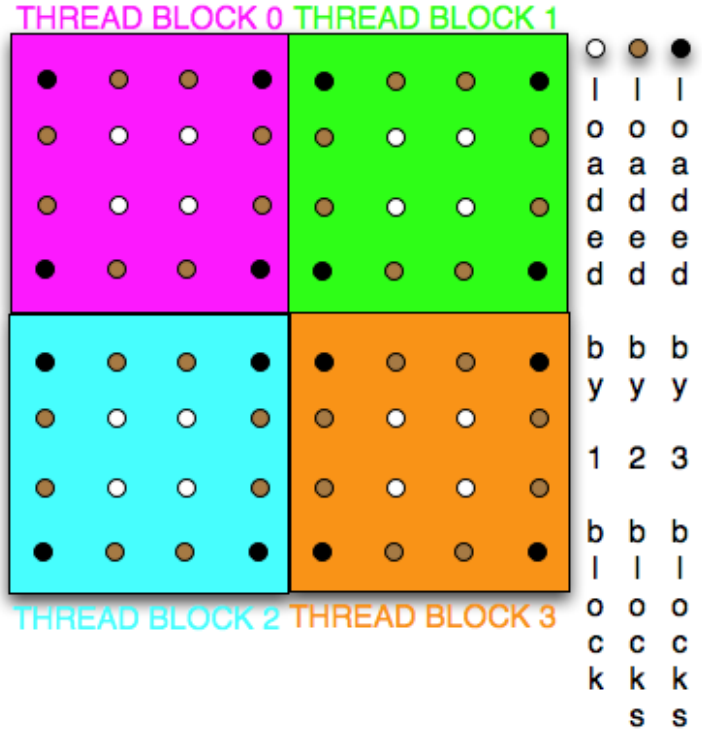
Once data are in shared memory, all red spins of the active plane can be updated concurrently then...



A survey of possible techniques: GPU shared memory (3-plane version) (3)

Data traffic between Global Memory and GPU processors is dramatically reduced!

- Only the spins on the boundaries of the subdomains need to be loaded more than once.
- A spin needs to be loaded three times, at most, instead of seven!



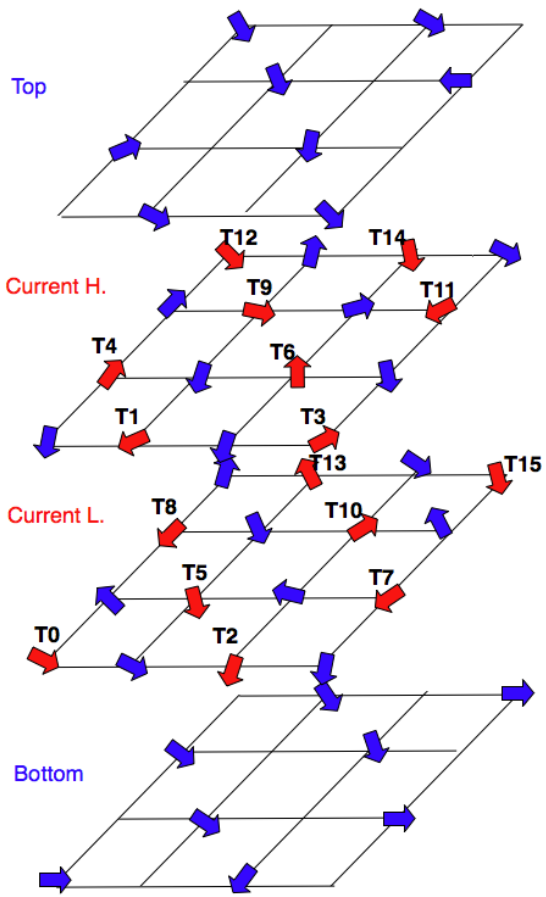
A survey of possible techniques: GPU shared memory (3-plane version) (4)

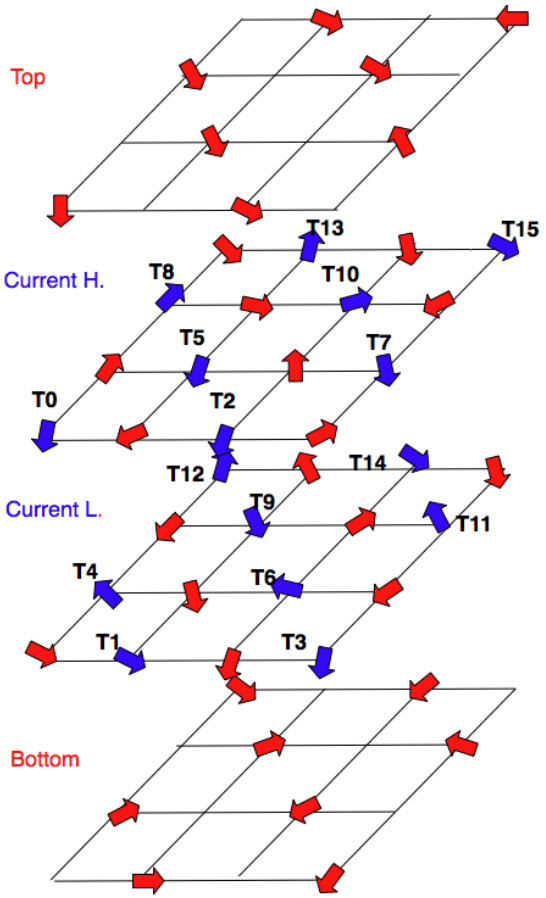
Resources usage:

- Shared memory (single precision):
  - 6 variables *per* spin: 24 bytes
    (3 spin components and the couplings along the 3 directions)
  - each thread loads 3 spins $->$ 72 bytes of shared memory.
- Registers: 62 (no local memory).
- On Fermi GPUs, having up to 48 Kbytes of shared memory, the number of threads is limited by the number of registers (32768).

Even more important... half of the threads are idle after data loading!
What is a possible solution?

A survey of possible techniques: GPU shared memory (4-plane version)





Working on two planes allows to keep all threads busy!

In the end **two** planes have been updated.

- 4 planes need to be resident in shared memory.
- More shared memory *per* thread is required.
- Fewer threads *per* SM: max **384** on Fermi GPUs
  (the global memory version could have up to **640** threads).

Further variants are possible by replacing plain (global) memory accesses with *texture fetches* operations:

- exploit *texture* caching capabilities;
- make easier to manage (periodic) boundary conditions.

Results

Source codes and test case available from

`http://www.iac.rm.cnr.it/~massimo/hsgfiles.tgz`

The test case is a system having 128^3 spins.

- spins and couplings are single precision variables;
- energy is accumulated in a double precision variable.

The performance measure is the time in nanoseconds required for a single spin update (lower is better)

$$T_{upd} = \frac{TotalTime \text{ (excluding I/O)}}{(\#\ of\ iterations)\times(\#\ of\ spins)}$$

GM: simple Global Memory version;

ShM: Shared Memory based (3 planes);

ShM4P: Shared Memory based (4 planes);

TEXT: simple Global Memory using *texture* operations for data fetching.

Platform	*# threads*	T_{upd}
Intel X5680 (SSE instr.)	1	~ 13.5 ns
Intel X5680 (SSE instr. + OpenMP)	8	~ 3.4 ns
Tesla C1060 GM	320	1.9 ns
Tesla C1060 ShM	128	2.5 ns
Tesla C1060 ShM4P	64	2.2 ns
Tesla C1060 TEXT	320	1.8 ns
GTX 480 (Fermi) GM	320	0.66 ns
GTX 480 (Fermi) ShM	256	1.3 ns
GTX 480 (Fermi) ShM4P	192	0.86 ns
GTX 480 (Fermi) TEXT	480	0.63 ns

Single vs. Double precision

Platform	*# threads*	$T_{upd}(s.p.)$	$T_{upd}(d.p.)$
GTX 480 (Fermi) GM	320	0.66 ns	1.3 ns
GTX 480 (Fermi) ShM4P	192/96	0.86 ns	2.2 ns
Intel X5680 3.33 Ghz (SSE instr.)	1	13.5 ns	26.6 ns
Intel X5680 3.33 Ghz (SSE instr.) (SSE instr. + OpenMP)	8	3.5 ns	6.8 ns

GM: simple Global Memory version;

ShM4P: Shared Memory based (4 planes).

The Error Correcting Code effect

Platform	*ECC on*	*ECC off*
C2050 (Fermi) GM	1.0 ns	0.84 ns
C2050 (Fermi) ShM	1.63 ns	1.66 ns
C2050 (Fermi) ShM4P	1.4 ns	1.3 ns
C2050 (Fermi) TEXT	1.0 ns	0.78 ns

The C2050 is slightly slower than the GTX 480 (14 instead of 15 SM and lower clock rate).

Intel X5680 "Westmere" (3.33 Ghz). Intel compiler: v. 11.1

# of *cores*	T_{upd} (s.p.) no SSE	T_{upd} (s.p.)	T_{upd} (d.p.)
1	32.6 ns	13.6 ns	26.6 ns
2	17.8 ns	7.1 ns (96%)	15.6 ns
4	8.5 ns	4.7 ns (72%)	9.0 ns
6	6.3 ns	3.7 ns (61%)	7.4 ns
8	4.6	3.3 ns (51%)	6.8 ns
10	4.2	3.3 ns (41%)	7.0 ns
12	3.9	3.4 ns (33%)	7.0 ns

System size 128^3. Very minor differences for 256^3.

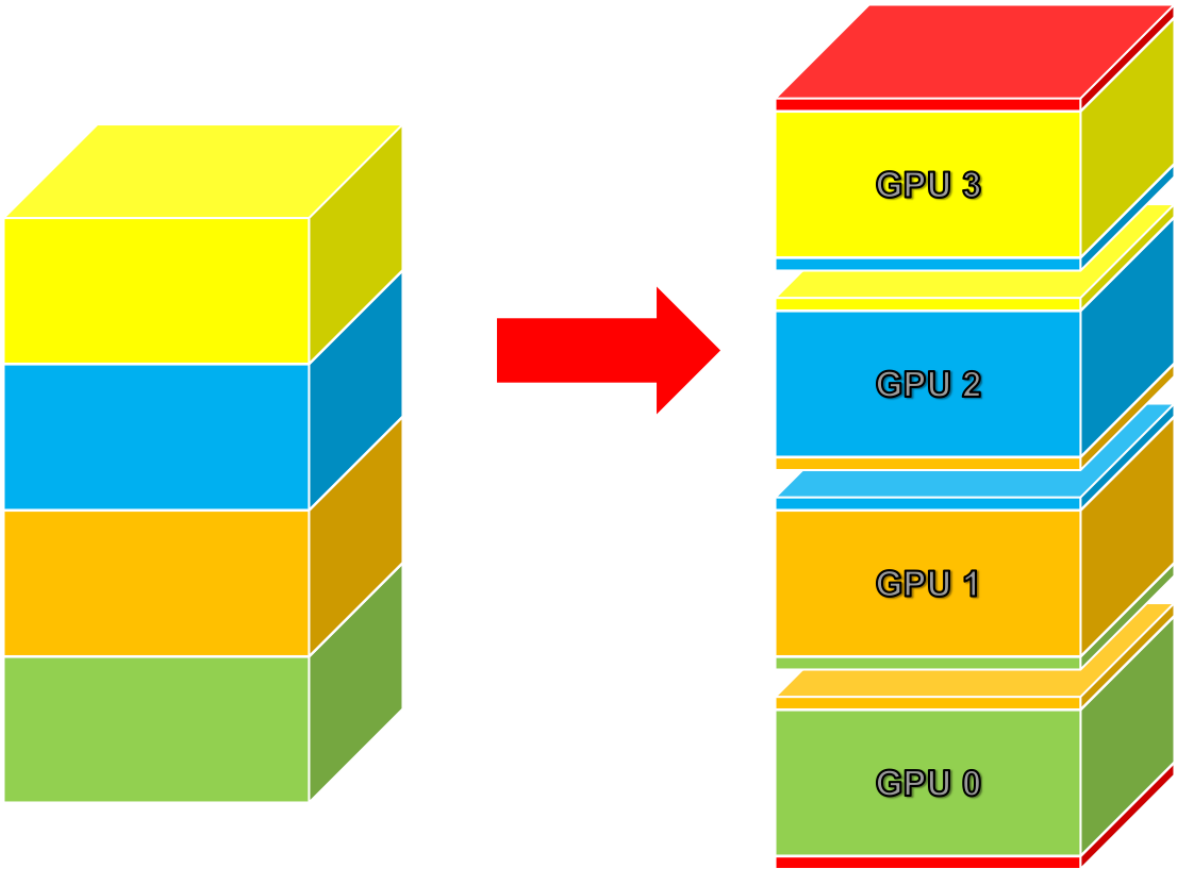
Parallelization based on OpenMP. To "convince" the compiler that the innermost loop can be safely vectorized at source code level...

```
REAL * __restrict__ anspx=an+spxoff,
     * __restrict__ ansmx=an+smxoff;
REAL * __restrict__ anspy=an+spyoff,
     * __restrict__ ansmy=an+smyoff;
REAL * __restrict__ anspz=an+spzoff,
     * __restrict__ ansmz=an+smzoff;
REAL * __restrict__ bnspx=bn+spxoff,
     * __restrict__ bnsmx=bn+smxoff;
REAL * __restrict__ bnspy=bn+spyoff,
     * __restrict__ bnsmy=bn+smyoff;
REAL * __restrict__ bnspz=bn+spzoff,
     * __restrict__ bnsmz=bn+smzoff;
REAL * __restrict__ cnspx=cn+spxoff,
     * __restrict__ cnsmx=cn+smxoff;
REAL * __restrict__ cnspy=cn+spyoff,
     * __restrict__ cnsmy=cn+smyoff;
REAL * __restrict__ cnspz=cn+spzoff,
     * __restrict__ cnsmz=cn+smzoff;
REAL * __restrict__ JpxWO=Jpx+yzoff,
     * __restrict__ JmxWO=Jmx+yzoff;
REAL * __restrict__ JpyWO=Jpy+yzoff,
     * __restrict__ JmyWO=Jmy+yzoff;
REAL * __restrict__ JpzWO=Jpz+yzoff,
     * __restrict__ JmzWO=Jmz+yzoff;
REAL * __restrict__ at=a+yzoff,
     * __restrict__ bt=b+yzoff,
     * __restrict__ ct=c+yzoff;
REAL * __restrict__ ta=t_a+yzoff,
     * __restrict__ tb=t_b+yzoff,
     * __restrict__ tc=t_c+yzoff;
```

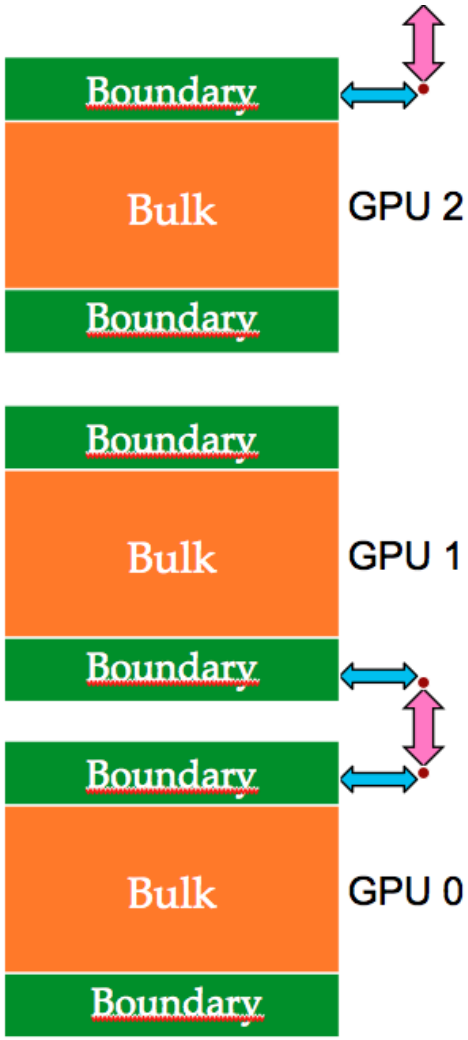
```
REAL as, bs, cs, factor;
int ix;
for(ix=istart; ix<iend; ix++) {
    as=anspx[ix]*JpxWO[ix]+
    anspy[ix]*JpyWO[ix]+
    anspz[ix]*JpzWO[ix]+
    ansmx[ix]*JmxWO[ix]+
    ansmy[ix]*JmyWO[ix]+
    ansmz[ix]*JmzWO[ix];

    bs=bnspx[ix]*JpxWO[ix]+
    bnspy[ix]*JpyWO[ix]+
    bnspz[ix]*JpzWO[ix]+
    bnsmx[ix]*JmxWO[ix]+
    bnsmy[ix]*JmyWO[ix]+
    bnsmz[ix]*JmzWO[ix];

    cs=cnspx[ix]*JpxWO[ix]+
    cnspy[ix]*JpyWO[ix]+
    cnspz[ix]*JpzWO[ix]+
    cnsmx[ix]*JmxWO[ix]+
    cnsmy[ix]*JmyWO[ix]+
    cnsmz[ix]*JmzWO[ix]+h;

    factor=two*(as*at[ix]+bs*bt[ix]+cs*ct[ix])/
                (as*as+bs*bs+cs*cs);
    ta[ix]=as*factor-at[ix];
    tb[ix]=bs*factor-bt[ix];
    tc[ix]=cs*factor-ct[ix];
}
```

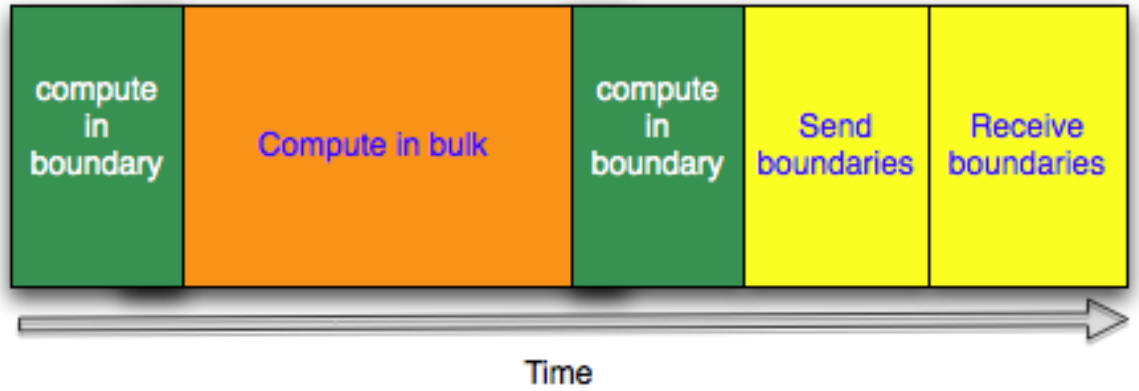
Multi-GPU

Simple domain decomposition along one axis



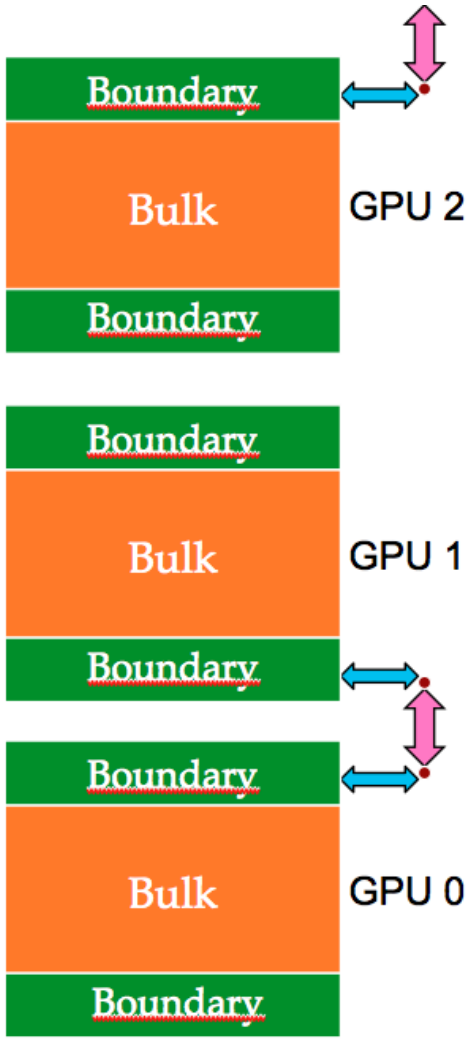
Multi-GPU: a first simple approach



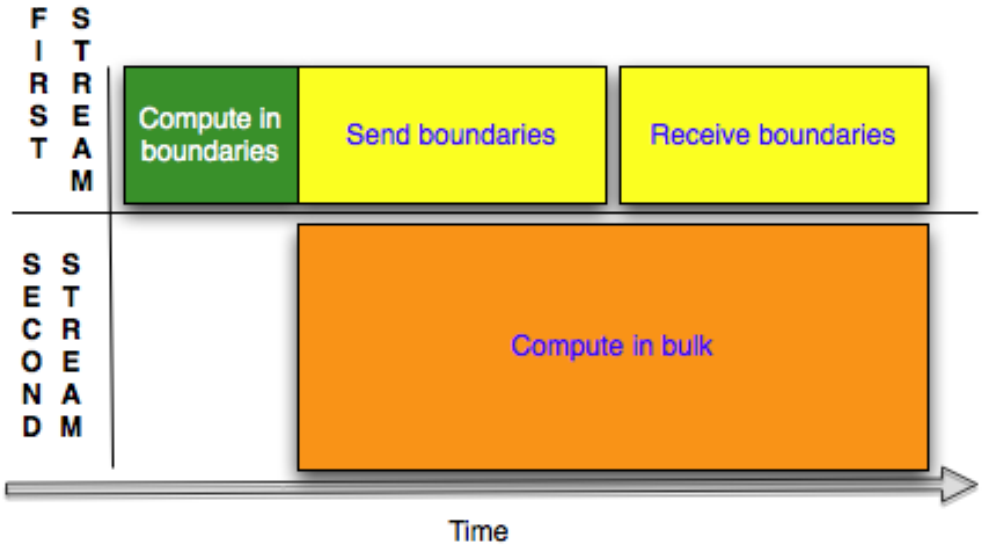


1. compute in boundaries;
2. compute in bulk;
3. exchange data (by using CPUs!);
4. goto to 1.

Multi-GPU: Overlap of communication and computation



- Compute the overrelaxation for the *bulk* and, at the same time, exchange data for the *boundaries*. Requires:
  - Cuda Streams
  - Asynchronous Memory Copy operations



1. Create 2 streams: one for the boundaries, one for the bulk;
2. concurrently:

stream one starts to update the boundaries;

stream two starts to update the bulk
(actually the latter starts only when the SM is available);

3. concurrently:

stream one copies data in the boundaries from the GPU to the CPU;
- exchanges data among CPUs by using MPI;
- copies data to the boundaries from the CPU to the GPU;

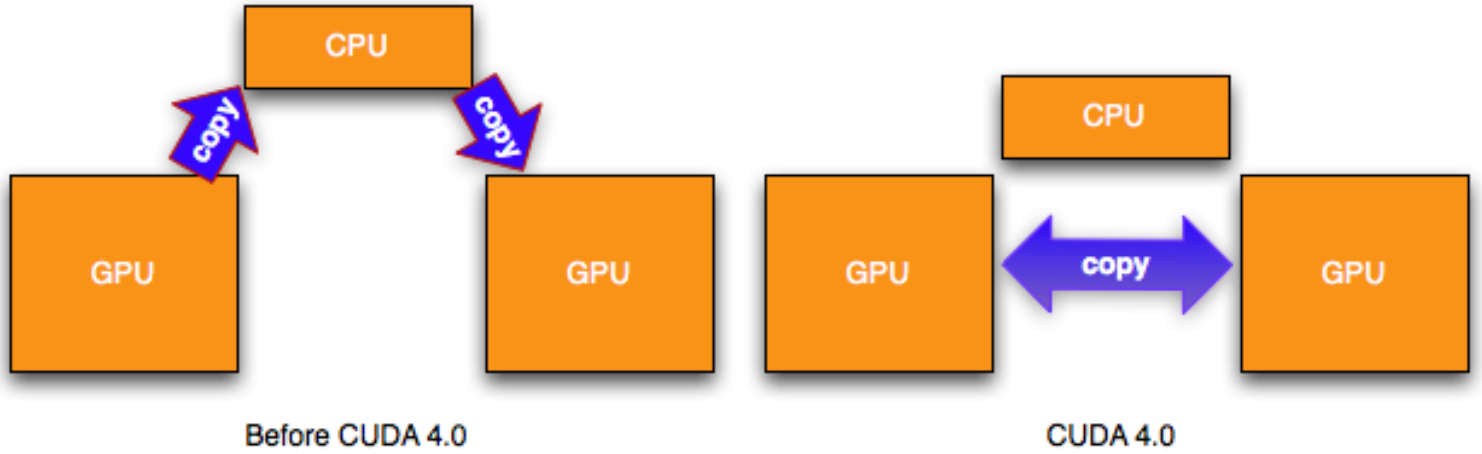
stream two continues to update the bulk;

4. go to 2.

The CPU acts as a MPI-coprocessor of the GPU!

Multi-GPU: Peer-to-Peer Memory Copy

- Starting on CUDA 4.0, memory copies can be performed between two different GPU connected to the same PCI-e root complex.
- If *peer-to-peer* access is enabled then the copy operation no longer needs to be staged through the CPU and is therefore faster!



- By using streams it remains possible to overlap communication and computation.

MPI based multi-GPU results

- CPU: Dual Intel Xeon(R)X5550 @ 2.67GHz. GPU: S2050.
- QDR Infiniband connection among nodes.
- MPI intra node communication *via* shared memory.

# of *GPU*	T_{upd} naive	T_{upd} with overlap	Efficiency with overlap
1	0.66 ns	0.66 ns	N.A.
2	0.45 ns	0.36 ns	91%
4	0.3 ns	0.19 ns	86%
8	0.22 ns	0.16 ns	51%

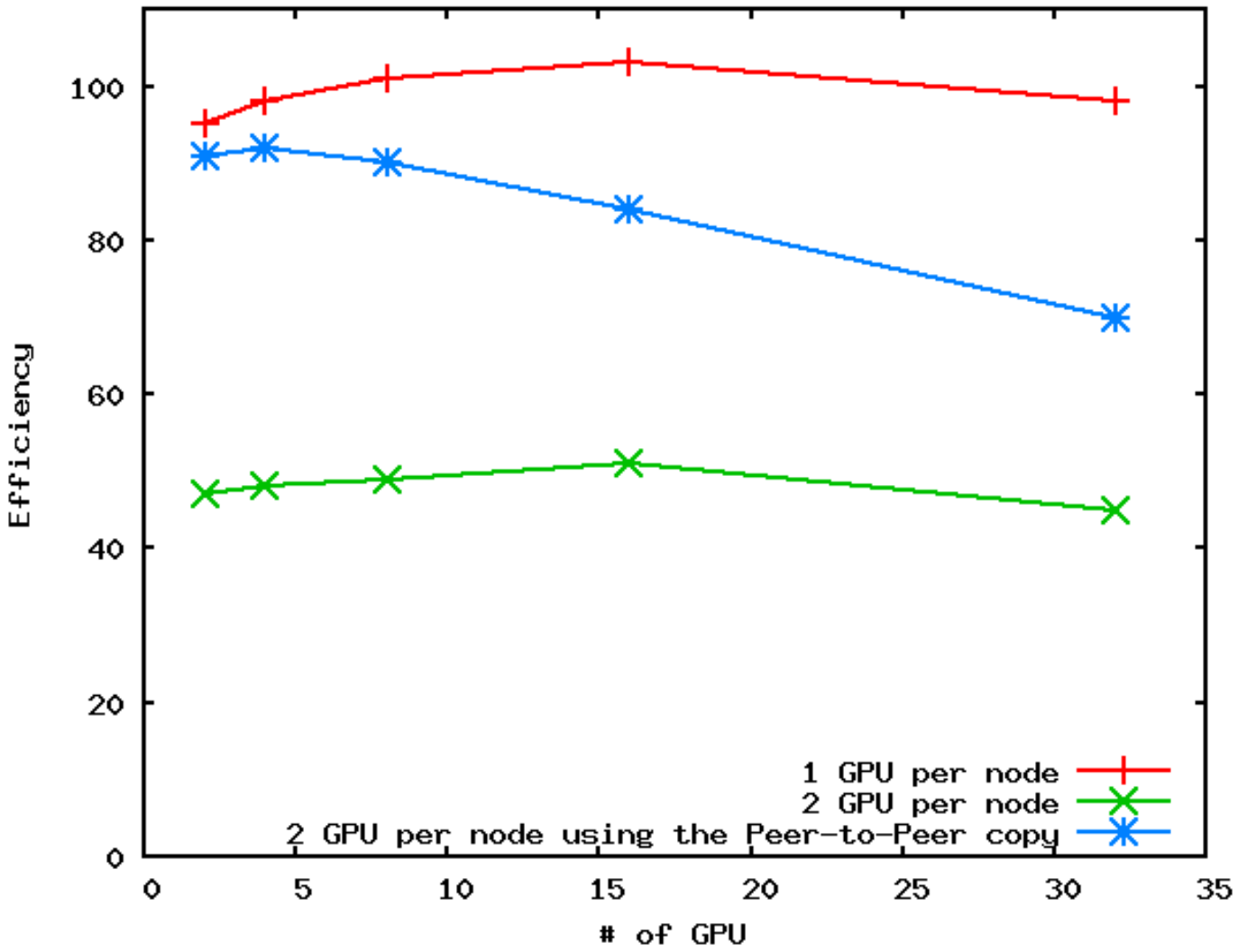
System size 256^3. Single Precision.

MPI, P2P and Hybrid multi-GPU results

- 8 2050 GPU with the same Hw configuration;
- Reference time from the 256^3 test case
  - 512^3 does not fit in a single GPU.

MPI	P2P	Speedup	Efficiency
8	1	5.02	63%
4	2	5.91	74%
2	4	6.80	85%
1	8	7.70	96%

System size 512^3. Single Precision.



Efficiency with up to 32 (2070) GPU for a 512^3 system.

Conclusions

Disclaimer: What follows applies to the Heisenberg spin glass model and very likely to other spins systems.
Any inference for other computational problems is at your own risk!

- GPU implementations outperform any CPU implementation.
- Vector instructions are absolutely required to get higher performances on multi-core CPUs.
- High end multi-core CPUs may show poor scalability.
- Multi-GPU programming is no more complex than hybrid OpenMP/vector programming but...
  - both Peer-to-Peer copy and MPI explicit message passing must be overlapped with GPU kernels execution.

Thanks!

Utilizing texture units for Image Processing

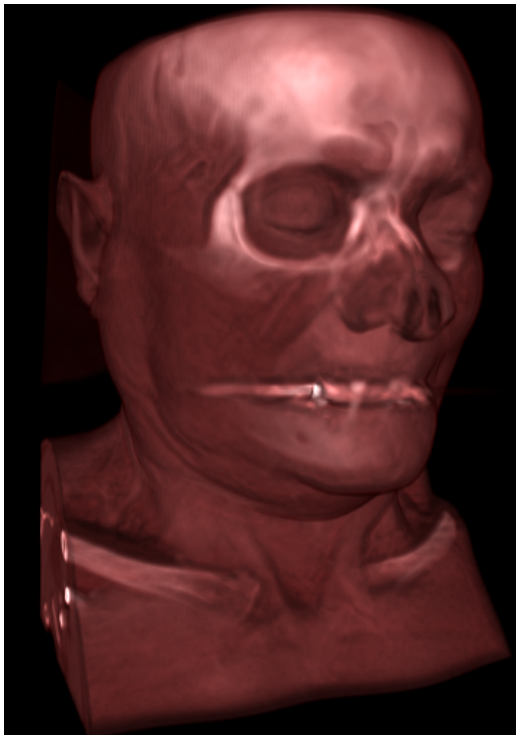
Erik N. Steen PhD

Principal Engineer

GE Vingmed Ultrasound

Why use GPUs for Image processing

- GPU: A tremendous additional «computer within the computer»
  - *Hardware support for basic operations used in image processing algorithms*
  - *Many image processing algorithms are parallel in nature*
  - *Amount of code can sometimes be greatly reduced ..*



Texture access

- Global memory access through separate texture cache
  - ***L2 cache used as well on Fermi GPUs***
- Efficient for localized (1D, 2D 3D) memory access patterns
  - ***Automated handling of borders***
- Includes interpolators
  - ***Interpolators offer additional computational resources***

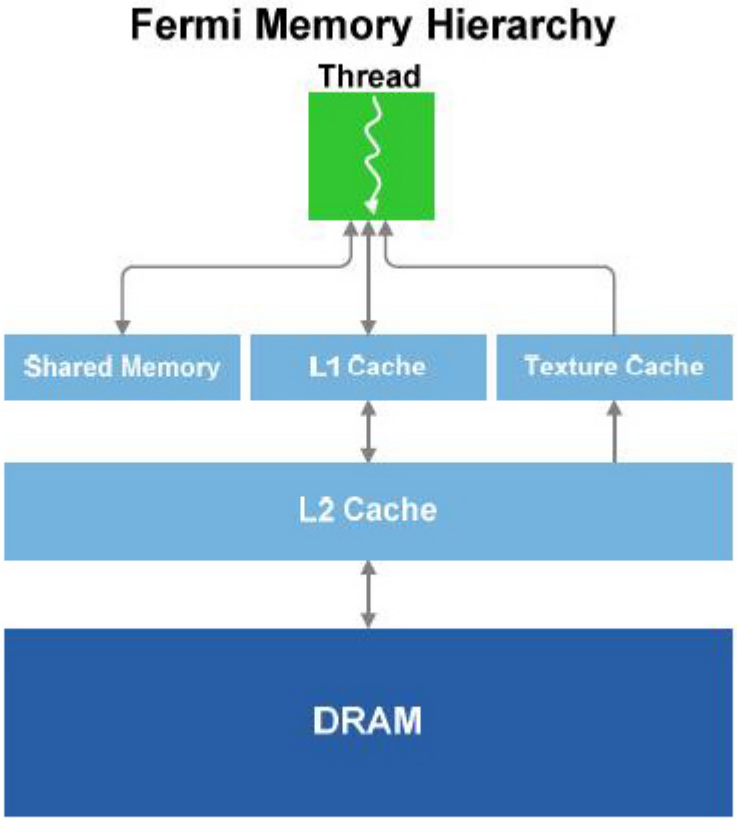

Fermi Memory Hierarchy
Thread
Shared Memory
L1 Cache
Texture Cache
L2 Cache
DRAM

Image processing with texturing

- Image data can be copied to pitched linear memory

OR

- Image data can be copied to a cudaArray

- BUT:
-

  GPU kernels can only work in parallel with data transfers if:

  - *Copies are made asynchronous (using CUDA streams)*
  - *Pitch linear memory is used.*

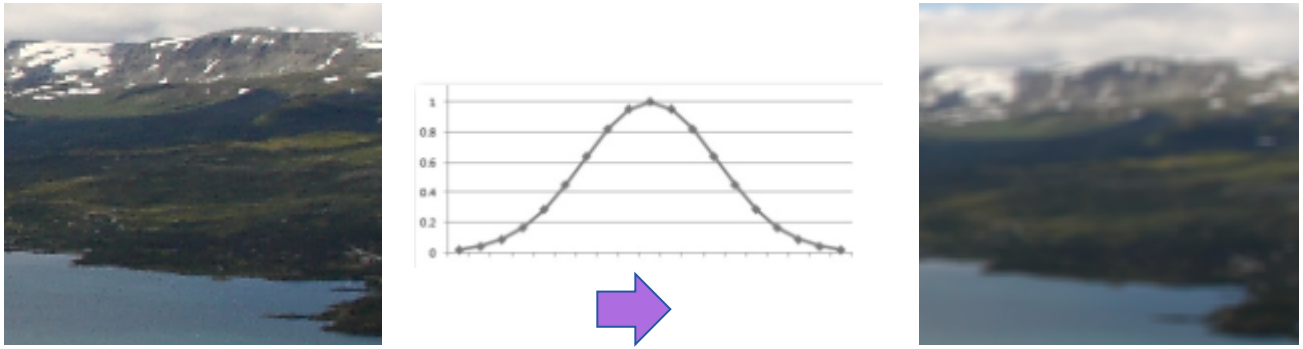
Case 1: Convolution filters

- Convolution filters are very frequently used in image processing for tasks such as:
  - Smoothing
  - Edge enhancement
  - Resampling
- The CUDA SDK contains example code based on
  - Constant memory for coefficients
  - Data access using either texturing or shared memory
- The 2D Gaussian filter can be implemented efficiently as a 2-step separable 1D filter (row-wise & column-wise)

Convolution filters

s(x-4)	s(x-3)	s(x-2)	s(x-1)	s(x)	s(x+1)	s(x+2)	s(x+3)	s(x+4)

$$o(x) = \sum_{i=-4}^{4} w(i) \cdot s(x+i)$$



Example: Gaussian Blur

Filter coefficients can be stored in constant memory

1D convolution (from SDK)

```
__global__ void convolutionRowsKernel(
    float *d_Dst,
    int imageW,
    int imageH )
{
    const   int ix = IMAD(blockDim.x, blockIdx.x, threadIdx.x);
    const   int iy = IMAD(blockDim.y, blockIdx.y, threadIdx.y);

    const float  x = (float)ix + 0.5f;
    const float  y = (float)iy + 0.5f;

    if(ix >= imageW || iy >= imageH)  return;
     float sum = 0;
      // Note template implementation in SDK is more efficient

       for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
           sum += tex2D(texSrc, x + (float)k, y) * c_Kernel[KERNEL_RADIUS -
       k];

      d_Dst[IMAD(iy, imageW, ix)] = sum;
}
```

Fast 1D convolution filters

- In several cases, two and two filter taps can be combined:
  - Consider the following filter coefficients (taps):

$$y_0 = 0.0625 \cdot x_0 + 0.250 \cdot x_1 + 0.375 \cdot x_2 + 0.250 \cdot x_3 + 0.0625 \cdot x_4$$

  - Trivially implemented with 5 texture fetches & 5 MADDs
  - Can be re-written using two (texture based) linear interpolations

$$y_0 = 0.3125 \cdot (0.25 \cdot x_0 + 0.75 \cdot x_1) + 0.375 \cdot x_2 + 0.3125 \cdot (0.75 \cdot x_3 + 0.25 \cdot x_4)$$

Now implemented with 3 texture fetches & 3 MADDs

-> Significant speedup, especially for larger filter kernels

Optimized 1D convolution code

SDK Example: «convolutionTexture»

```
__global__ void convolutionRowsKernel(
    float *d_Dst,
    int imageW,
    int imageH )
{
    .
    .
    .
    .
      for(int k = -KERNEL_RADIUS k <= KERNEL_RADIUS; k++)
        // Relative sample position stored in constant memory
         sum +=


        tex2D(texSrc, x + c_Offset[KERNEL_RADIUS – k], y) * c_Kernel
       [KERNEL_RADIUS - k];


     d_Dst[IMAD(iy, imageW, ix)] = sum;
}
```

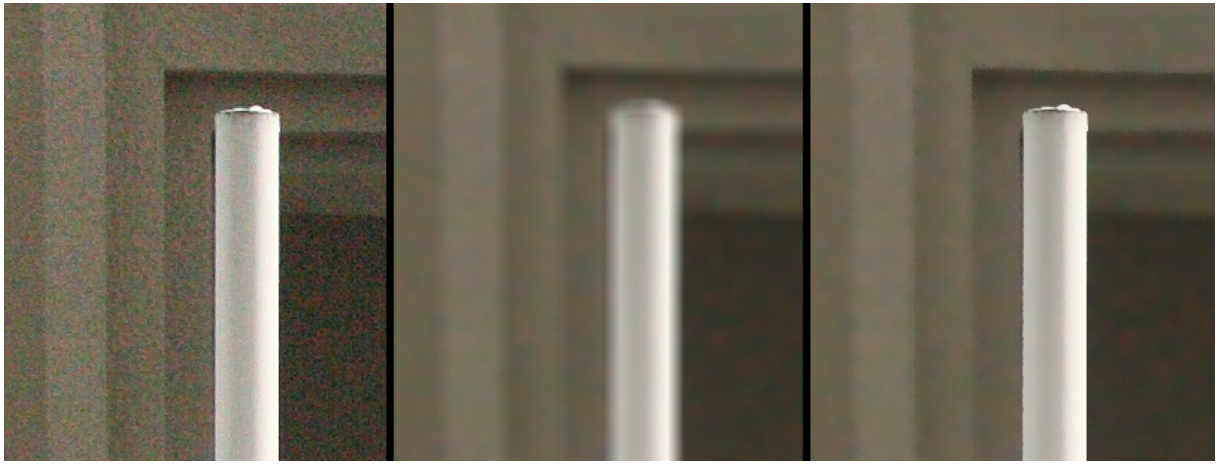
Bilateral (edge preserving) filtering

Gaussian filtering:

- Filtered pixel influenced more by pixels close in space

Bilateral filtering includes one additional criterion:

- Filtered pixel influenced more by pixels close in value



Original Image | Gaussian blur | Bilateral filtering

Bilateral (edge preserving) filtering

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in S} G_{\sigma_s}\left(\| \mathbf{p} - \mathbf{q} \|\right) G_{\sigma_r}\left(| I_{\mathbf{p}} - I_{\mathbf{q}} |\right) I_{\mathbf{q}}$$

Weights based on distance as well as difference in value

space σ_s : Spatial extent of the kernel

range σ_r : Edge amplitude

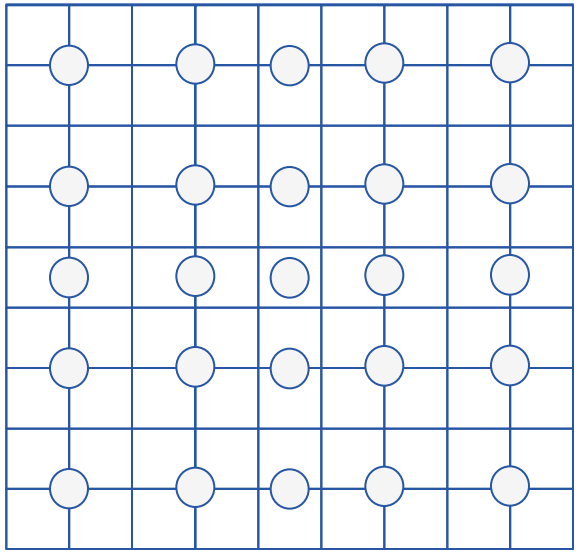
SDK example can be found in "bilateralfilter"

Fast Bilateral filtering

- Signal dependent weights can be stored into a 1D texture
  - Or computed analytically
- Constant memory used for spatial weights and offsets

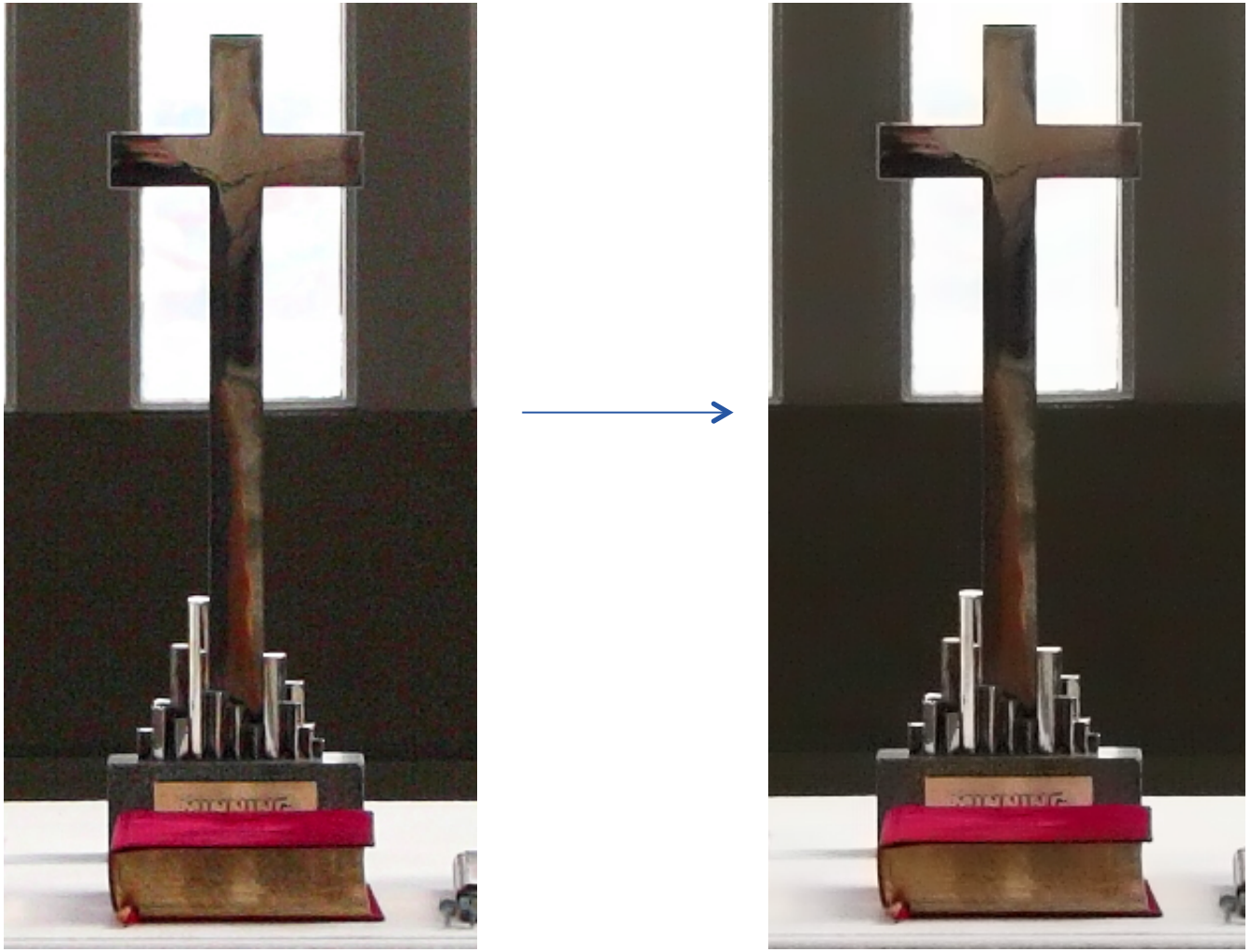
```
float centerValue = tex2D(texSrc,x,y);
float sum = 0.0F;
for(int j = -KERNEL_RADIUS; j < KERNEL_RADIUS;  j++) {
    for(int i = -KERNEL_RADIUS; i < KERNEL_RADIUS;  i++) {
        float2 offset          = c_Offset[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
        float  rangeWeight     = c_Kernel[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
        float  value           = tex2D(texSrc,  x + offset.x,  y + offset.y);
        sum+=
        value*rangeWeight*tex1D((texSignalDepWeightLUT,fabs(value-centerValue));
    }
}
```

2D texture convolution sampling



- Example: Using a neighbor distance of 1.5-2.0 samples
  - Covering a 9x9 neighborhood with just 25 samples

Example : Photo noise reduction



Branching

- Despite computational benefits, texture fetch rate is limited compared to the other computational resources, especially on newer GPUs

- Branching can be used to avoid unnecessary texture fetches

- In most real cases, image data is locally coherent

- Data dependent conditional branching is therefore often effective

Example : Shadow area image processing

```
float centerValue = tex2D(texSrc,x,y);
 float sum = 0.0F;
If (centerValue<shadowAreaLimit) { // i.e 0.2
      for(int j = -KERNEL_RADIUS; j < KERNEL_RADIUS;  j++) {
            for(int i = -KERNEL_RADIUS; i < KERNEL_RADIUS;  i++) {
               float2 offset              = c_Offset[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
               float   rangeWeight = c_Kernel[KERNEL_RADIUS – j] [KERNEL_RADIUS – i];
               float   value              = tex2D(texSrc,  x + offset.x,  y + offset.y);
               sum+= value*rangeWeight*tex1D((texSignalDepWeightLUT,fabs(value-centerValue));
            }
      }
}
else
     sum=centerValue;
```

Speedup on typical images (9x9 lookups): 3-4 x (Fermi GPUs)

Examples of using shared memory

Result of a column-wise 1D filter may be written to shared memory:

- Column-wise filtering with use of texturing, results put in shared memory
- Row-wise filtering done from shared memory
- Reduces global memory bandwidth needs.

Exercise: Change SDK Example: «convolutionTexture»

Spatially invariant filtering can benefit greatly from shared memory when coefficients can be assumed to be constant locally

- Filter coefficients read into shared memory
  or
- Filter coefficient indices read into shared memory

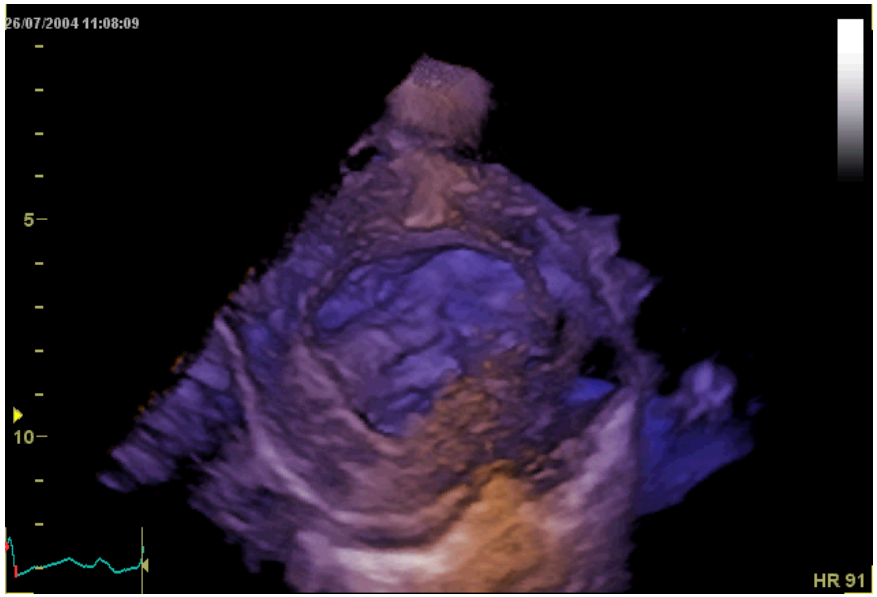
Example: Spatially variant filtering

Spatially variant filtering often useful (i.e lense-specific deblurring of photos)

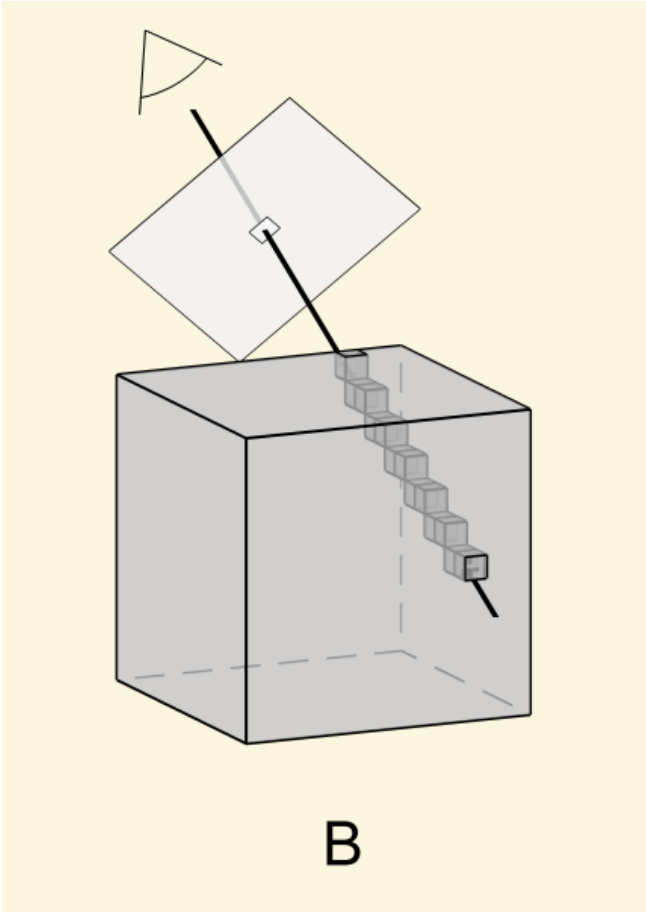
```
.....
// Assuming one index per threadBlock available in global memory and a limited number of coefficient sets in constant mem.
__shared__ uint s_ind1;
__shared__  uint s_ind2;
if (threadIdx.x==0 && threadIdx.y==0) {
    s_ind1 = globalOffsetIndexTable[blockIdx.x][blockIdx.y];
    s_ind2 = globalKernelIndexTable[blockIdx.x][blockIdx.y]
  }
  __syncthreads();

   ....
  for(int k = -KERNEL_RADIUS k <= KERNEL_RADIUS; k++)
     sum += tex2D(texSrc, x + c_Offset[s_ind1][KERNEL_RADIUS – k], y) * c_Kernel[s_ind2][KERNEL_RADIUS -
    k];
  .....
```

Case 2: Volume rendering

Use of 3D texturing and branching



3D rendering principle



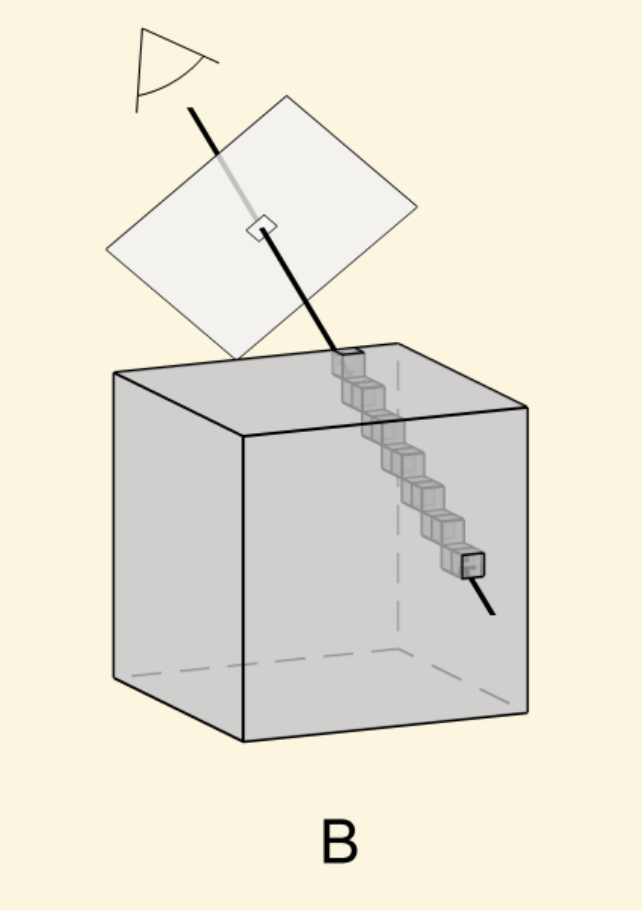
The rendered image is a type of projection of an entire volume.

In its simplest form it can be thought of as a digital x-ray image.

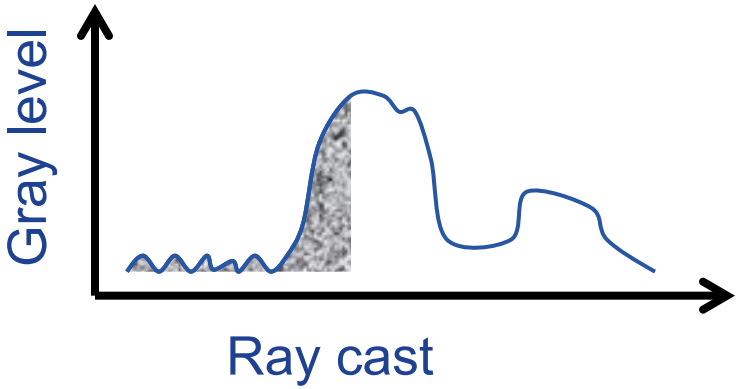
In medical imaging we are usually interested in displaying surfaces of anatomical structures.

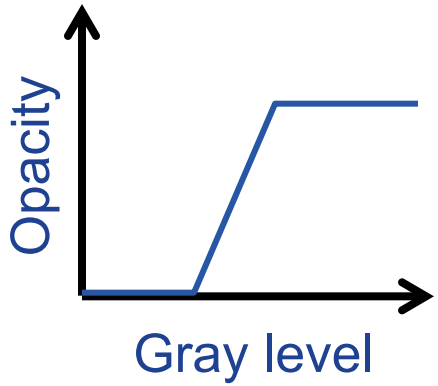
The rendering should be robust and give a good depth perception

Opacity function controls rendering appearance



Weighted integration along rays:

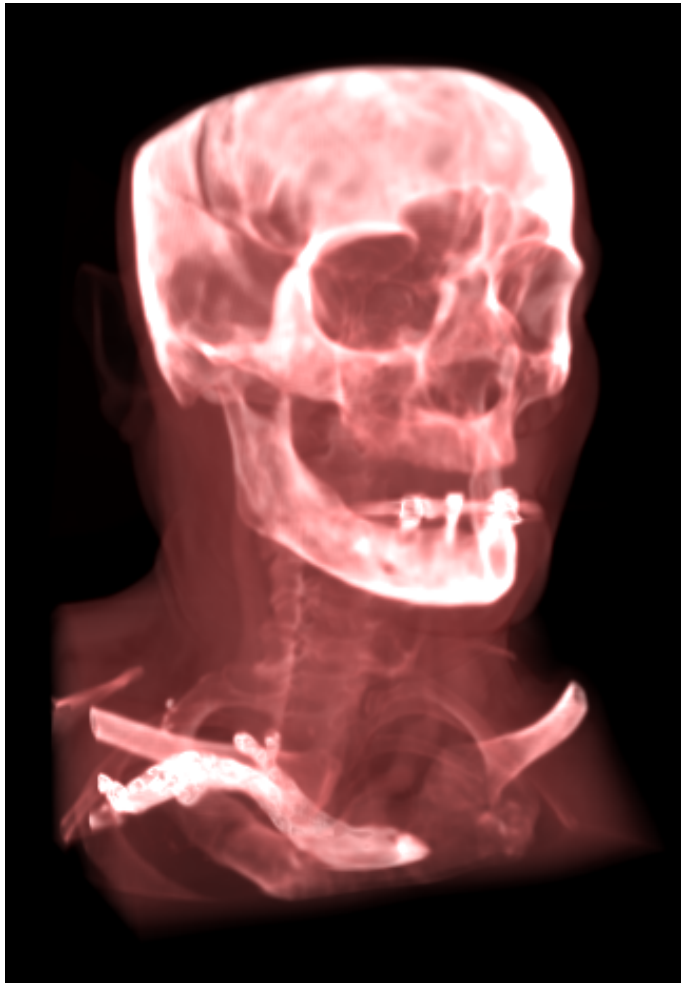




Ray casting loop (Nvidia SDK sample)

SDK Example: «volumeRender»

```
float4 sum = make_float4(0.0f);
for (i=0;i<maxSteps;i++)  {
{
        // read from 3D texture with remapped positions to [0,1] coordinates
        float3 posNorm = pos*0.5f+make_float3(0.5f,0.5f,0.5f);
        float sample = tex3D(tex, posNorm.x,posNorm.y,posNorm.z);

        // lookup in transfer function texture
        float4 col = tex1D(transferTex, (sample-transferOffset)*transferScale);

        // pre-multiply alpha
        col.x *= col.w; col.y *= col.w; col.z *= col.w;

        // "over" operator for front-to-back blending
        sum = sum + col*(1.0f - sum.w);

        // exit early if opaque
        if (sum.w >= opacityThreshold) break;

        t += tstep; pos += step;
}
```

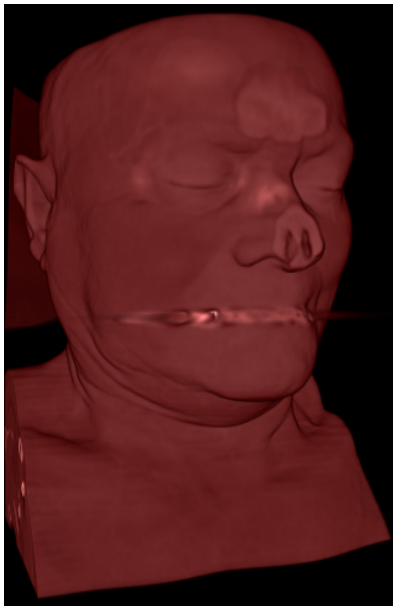


A simple gradient shading scheme

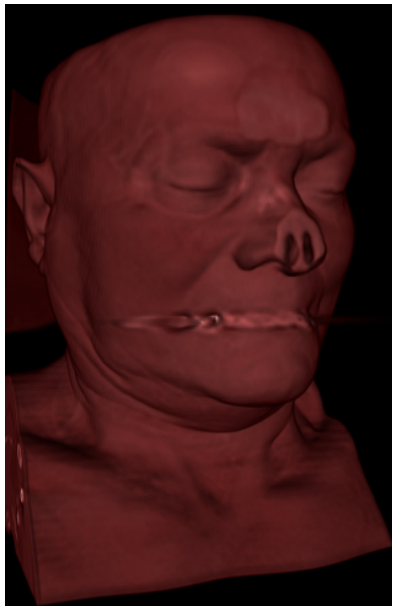
- Selectively modifying each voxel value using gradient shading
- Each gradient computation uses 6 additional texture fetches
- Gradients normalized and used in lighting calculation

Simple shading

```
__device__ inline float ComputeShadeValue(
        const float3 &pN,
       const float3 &dirX,
       const float3 &dirY,
       const float3 &dirZ,
       const float3 &lightDir)
{
         // Get 6 neighbors in directions aligned with viewing plane
          float v1,v2,v3,v4,v5,v6;
           v1=tex3D(tex,pN.x-dirX.x,pN.y-dirX.y,pN.z-dirX.z);
          v2=tex3D(tex,pN.x+dirX.x,pN.y+dirX.y,pN.z+dirX.z);
       ......

          float3 gradUnNormalized;
          gradUnNormalized.x=v2-v1;
          gradUnNormalized.y=v4-v3;
           gradUnNormalized.z=v6-v5;
       // 25 % ambient and 75 % diffuse reflection
       return 0.25f+0.75f*fabsf(dot(lightDir,normalize
       (gradUnNormalized)));
}
```



No shading



Simple Gradient based shading

Ray casting loop (w/ simple shading)

```
// read from 3D texture with remapped positions to [0,1] coordinates
float3 posNorm = pos*0.5f+make_float3(0.5f,0.5f,0.5f);

float sample      = tex3D(tex, posNorm.x,posNorm.y,posNorm.z);

// lookup in transfer function texture
float4 col        = tex1D(transferTex, (sample-transferOffset)*transferScale);

if (sample>threshold) {        // Selective shading
    float shadeVal      =  ComputeShadeValue(posNorm,eyeRay.dx,eyeRay.dy,eyeRay.dz,c_lightDir);
    col.x*=shadeVal;
    col.y*=shadeVal;
    col.z*=shadeVal;
}
```

The actual threshold value may impact performance substantially !

Questions ?