GPU Programming and CUDA

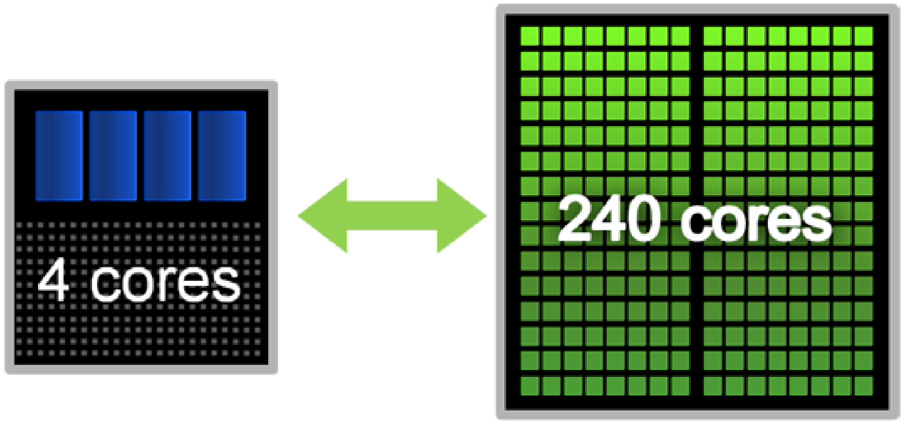
M. Reza

GPU

- Graphical Processing Unit
- A single GPU consists of large number of cores – hundreds of cores.
- Whereas a single CPU can consist of 2, 4, 8 or 12 cores
- Cores? – Processing units in a chip sharing at least the memory and L1 cache

GPU and CPU

- Typically GPU and CPU coexist in a heterogeneous setting
- “Less” computationally intensive part runs on CPU (coarse-grained parallelism), and more intensive parts run on GPU (fine-grained parallelism)
- NVIDIA’s GPU architecture is called CUDA (Compute Unified Device Architecture) architecture, accompanied by CUDA programming model, and CUDA C language



[3]

NVIDIA Kepler K40

- 2880 streaming processors/cores (SPs) organized as 15 streaming multiprocessors (SMs)
- Each SM contains 192 cores
- Memory size of the GPU system: 12 GB
- Clock speed of a core: 745 MHz

Hierarchical Parallelism

- Parallel computations arranged as grids
- One grid executes after another
- Grid consists of blocks
- Blocks assigned to SM. A single block assigned to a single SM. Multiple blocks can be assigned to a SM.
- Block consists of elements
- Elements computed by threads

Hierarchical Parallelism

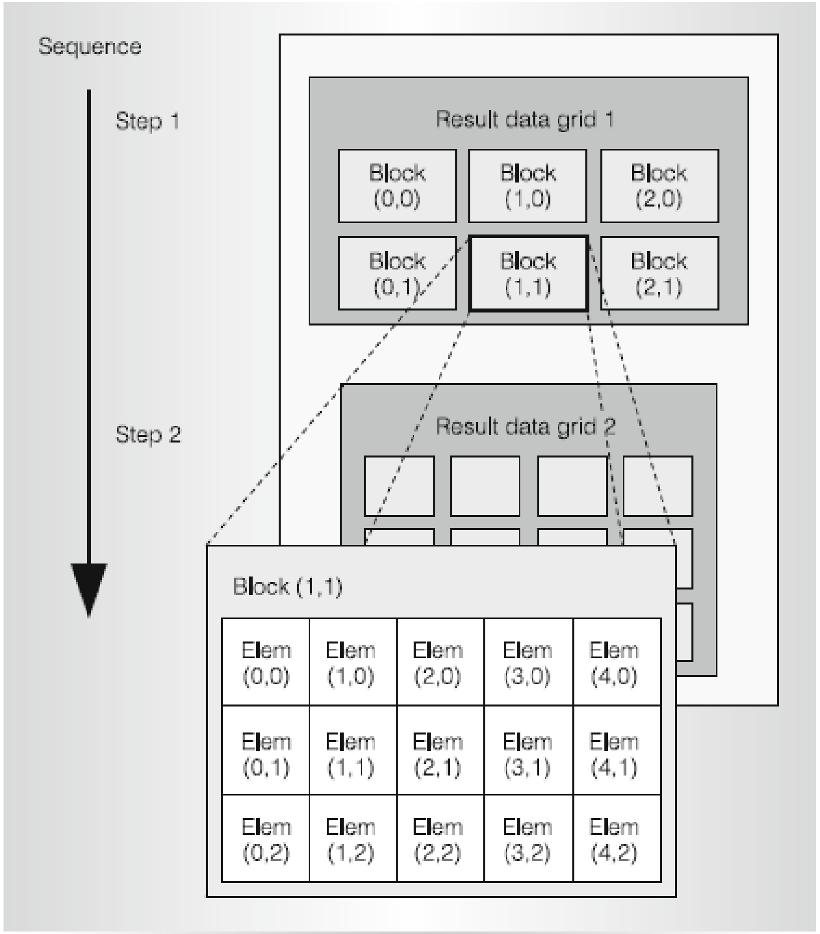


Figure 5. Decomposing result data into a grid of blocks partitioned into elements to be computed in parallel.

[6]

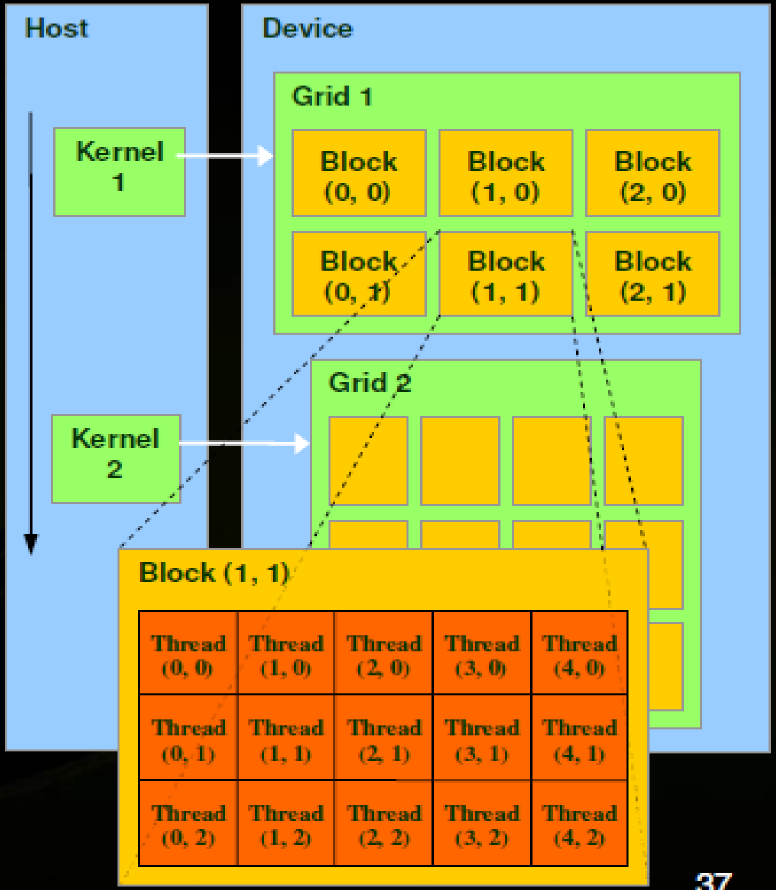
Thread Blocks

- Thread block – an array of concurrent threads that execute the same program and can cooperate to compute the result
- Consists of up to 1024 threads (for K40)
- Has shape and dimensions (1d, 2d or 3d) for threads
- A thread ID has corresponding 1,2 or 3d indices
- Each SM executes up to sixteen thread blocks concurrently (for K40)
- Threads of a thread block share memory
- Maximum threads per SM: 2048 (for K40)

CUDA Programming Model

A kernel is executed by a grid of thread blocks

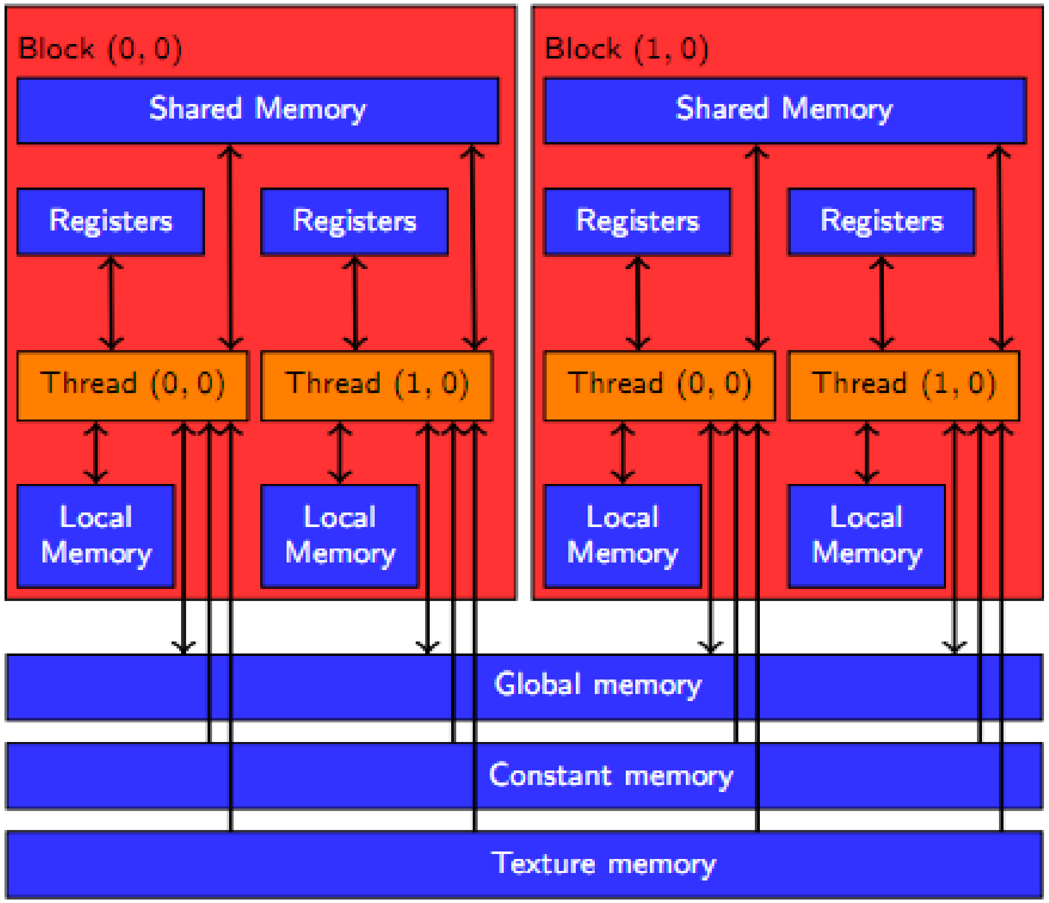
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



37

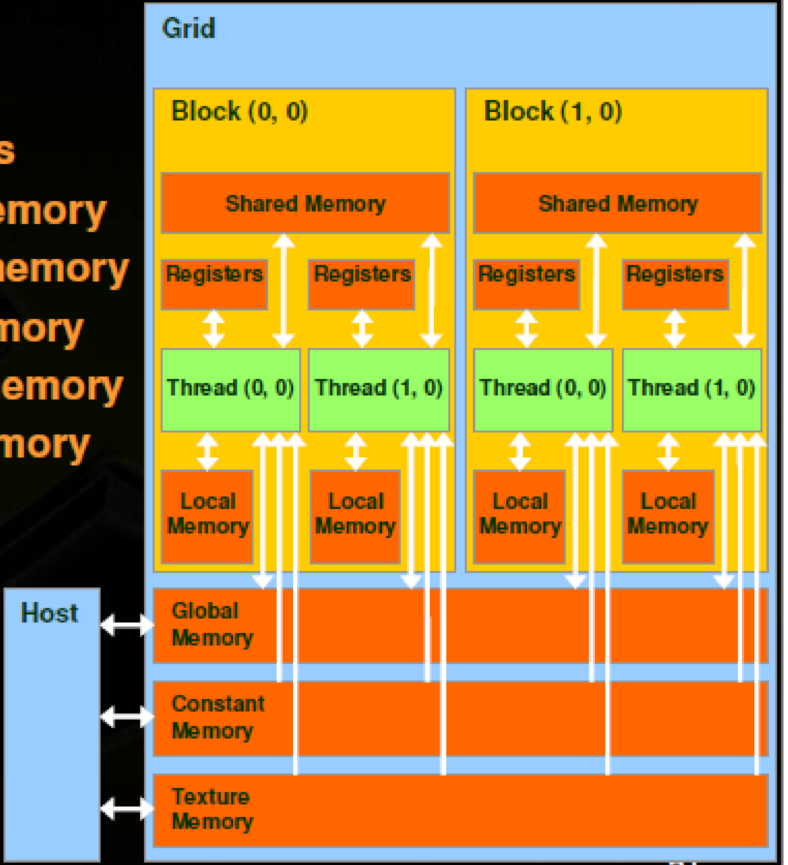
Memory model

CUDA exposes all the different types of memory on the GPU



CUDA Memory Spaces

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread **local memory**
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read only per-grid **constant memory**
  - Read only per-grid **texture memory**
- The host can read/write global, constant, and texture memory (stored in DRAM)



Memory model II

To summarize

Registers	Per thread	Read-Write	
Local memory	Per thread	Read-Write	
Shared memory	Per block	Read-Write	For sharing data within a block
Global memory	Per grid	Read-Write	Not cached
Constant memory	Per grid	Read-only	Cached
Texture memory	Per grid	Read-only	Spatially cached

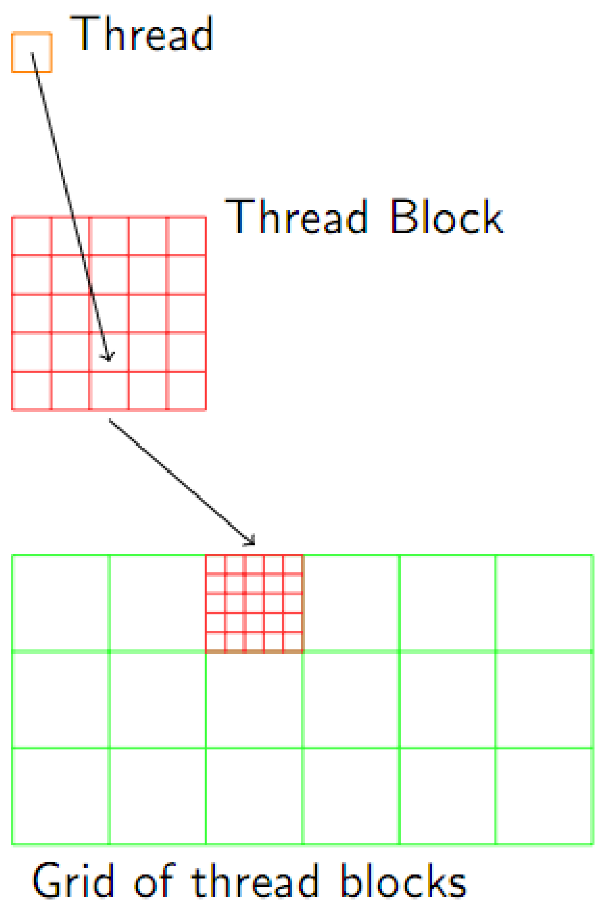
Don't panic!

- You do not need to use all of these to get started
- Start by using just global mem, then optimize
  - More about this later

Memory Management

- Explicit GPU memory allocation and deallocation
  - `cudaMalloc()` and `cudaFree()`
- Pointers to GPU memory
- Copy between CPU and GPU memory
  - A slow operation, aim to minimize this

Recap



Multiple levels of parallelism

- Thread block
  - Up to 512 threads per block
  - Communicate via shared memory
  - Threads guaranteed to be resident
  - `threadIdx, blockIdx`
  - `__syncthreads()`
- Grid of thread blocks
  - `f<<<N, T>>>( a, b, c)`
  - Communicate via global memory

GeForce 8800 Architecture Overview

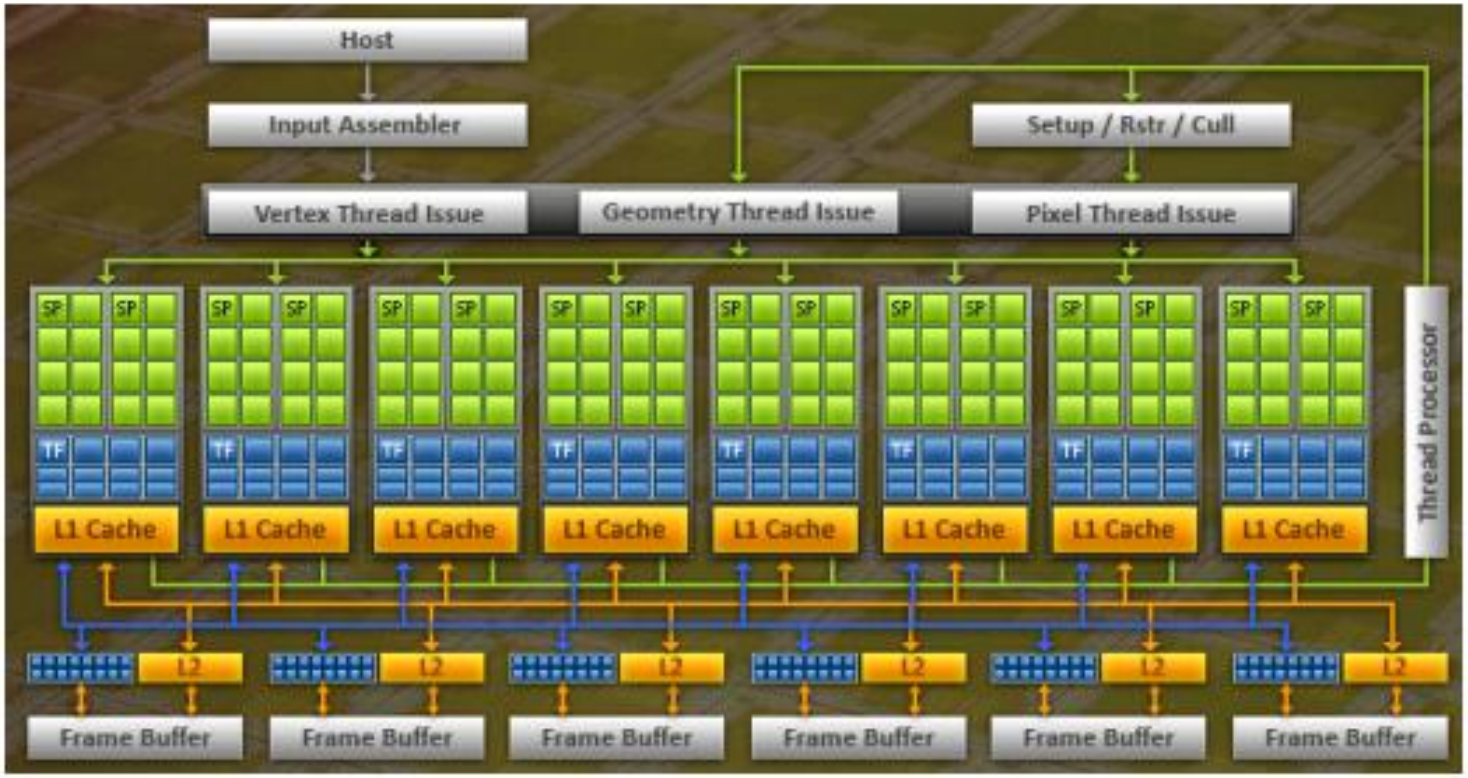


Image courtesy of rage3d.com

GeForce 8800 Architecture Overview (cont'd)

- 16 stream processors in each multiprocessor
- 128 stream processors in total.
- L1 cache shared between all stream processors in a multiprocessor

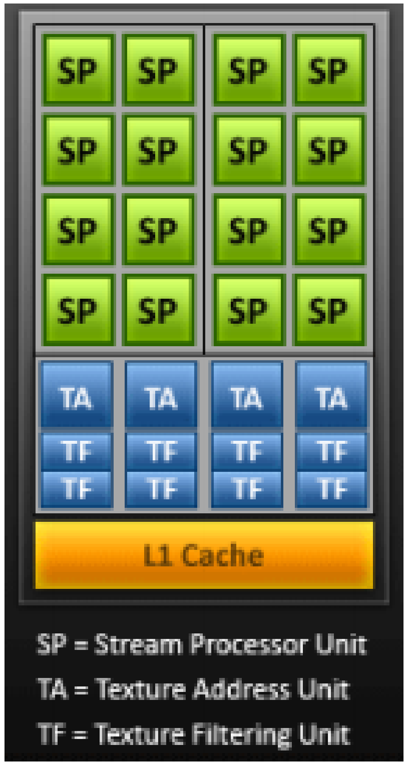

SP
TA
TF
L1 Cache
SP = Stream Processor Unit
TA = Texture Address Unit
TF = Texture Filtering Unit

Image courtesy of rage3d.com

API Design

CUDA Programming Guide

"The goal of the CUDA programming interface is to provide a relatively simple path for users familiar with the C programming language to easily write programs for execution on the device."

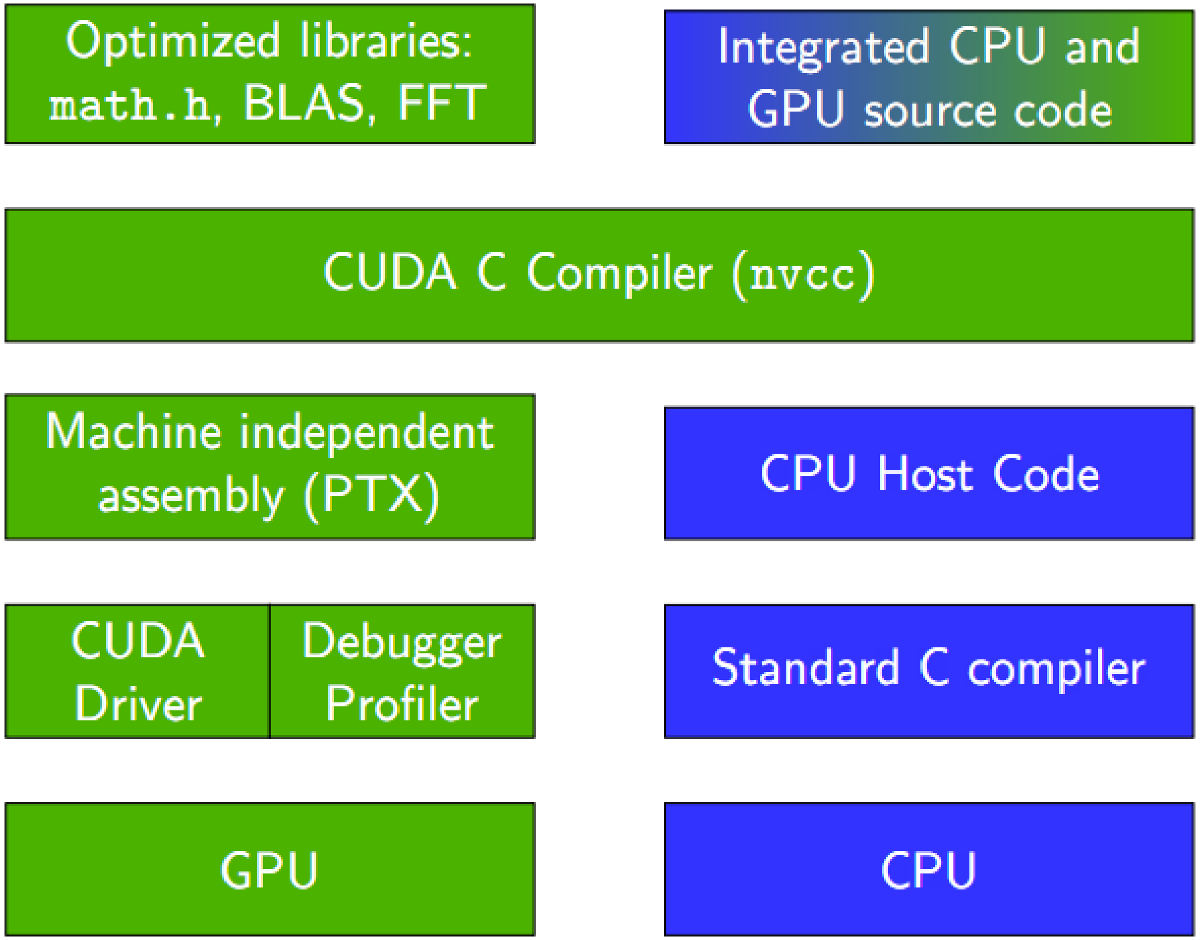
- Minimal C extensions
- A runtime library
  - A host (CPU) component to control and access GPU(s)
  - A device component
  - A common component
    - Built-in vector types, C standard library subset

Language extensions

- Function type qualifiers
  - Specify where to call and execute a function
  - `__device__`, `__global__` and `__host__`
- Variable type qualifiers
  - `__device__`, `__constant__` and `__shared__`
- Kernel execution directive
  - `foo<<<GridDim, BlockDim>>>(...)`
- Built-in variables for grid/block size and block/thread indices

Source files must be compiled with the CUDA compiler `nvcc`.

CUDA Software Development Kit


Optimized libraries: math.h, BLAS, FFT
Integrated CPU and GPU source code
CUDA C Compiler (nvcc)
Machine independent assembly (PTX)
CPU Host Code
CUDA Driver
Debugger Profiler
Standard C compiler
GPU
CPU

The NVCC compiler

- CUDA kernels are typically stored in files ending with `.cu`
- NVCC uses the host compiler (CL/G++) to compile CPU code
- NVCC automatically handles `#include`'s and linking
- Very nice for toy projects
- Does not support exceptions
  - Most STL headers (i.e. `iostream`) can not be included

Integrating CUDA into larger projects

- Write kernels+CPU caller in `.cu` files
  - Compile with `nvcc`
- Store signature of CPU caller in header file
- `#include` header file in C(++) sources
- Modify build system accordingly

Device (GPU) Runtime Component

The following extensions are only available on the GPU:

- Less accurate, faster math functions `__sin(x)`
  - Detailed error bounds are available
- `__syncthreads()`
  - Wait until all threads in the block has reached this point
- Type conversion functions, with rounding mode
- Type casting functions
- Texture functions
- Atomic Functions
  - Guarantees that operation (like add) is performed on a variable without interference from other threads
  - Only on newer GPUs (Compute capability 1.1)

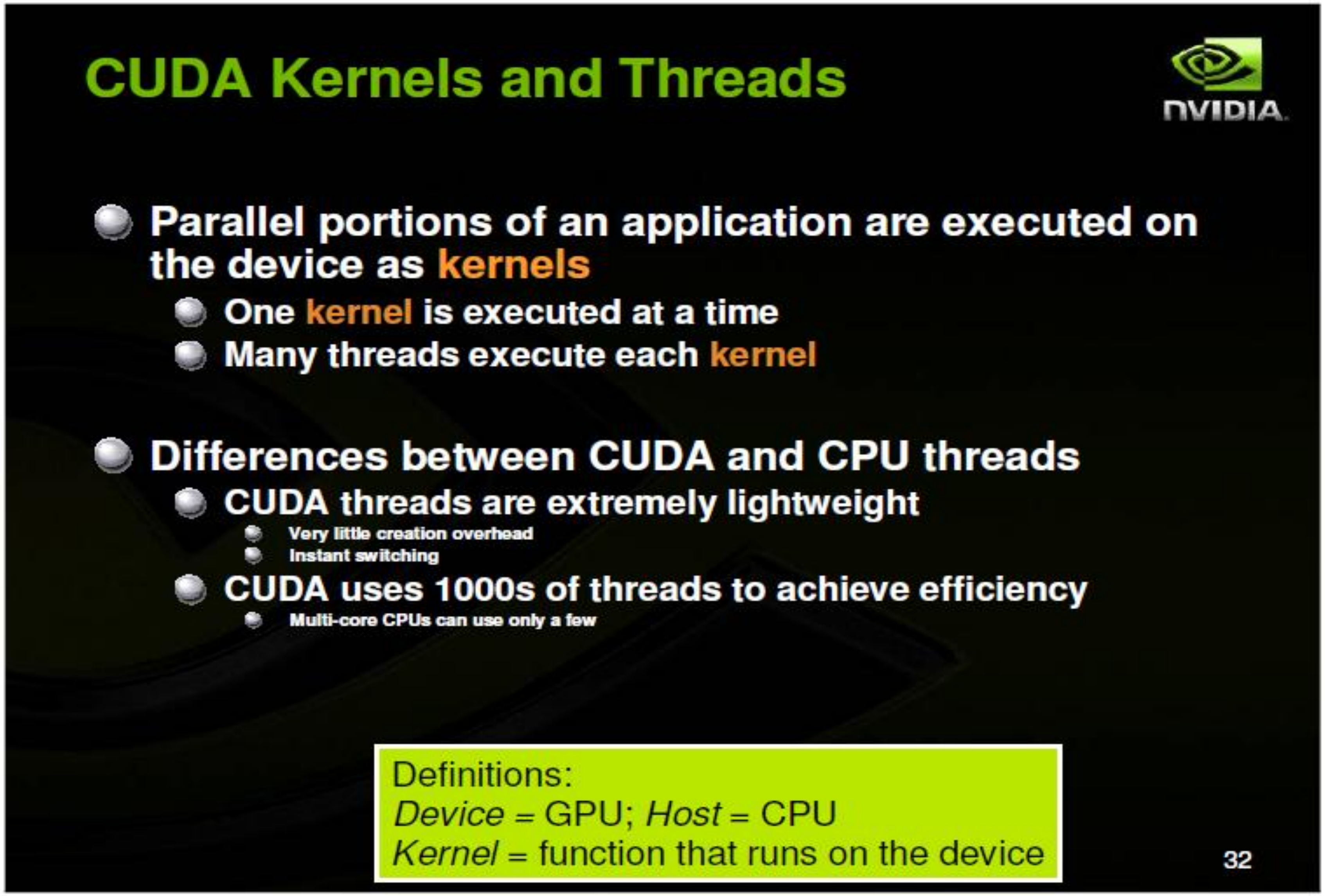
Host (CPU) Runtime Component

The following is only available from on the CPU:

- Device Management
  - Get device properties, multi-GPU control etc.
- Memory Management
  - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()` etc.
- Texture management
- OpenGL and DirectX interoperability
  - Map global memory to OpenGL buffers etc.
- Asynchronous Concurrent Execution
- Also a low-level (driver) API

CUDA Programming Language

- Programming language for threaded parallelism for GPUs
- Minimal extension of C
- A serial program that calls parallel kernels
- Serial code executes on CPU
- Parallel kernels executed across a set of parallel threads on the GPU
- Programmer organizes threads into a hierarchy of thread blocks and grids



CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:
Device = GPU; *Host* = CPU
Kernel = function that runs on the device

32

CUDA C

- Built-in variables:
  - threadIdx.{x,y,z} – thread ID within a block
  - blockIDx.{x,y,z} – block ID within a grid
  - blockDim.{x,y,z} – number of threads within a block
  - gridDim.{x,y,z} – number of blocks within a grid
- kernel<<<nBlocks,nThreads>>>(args)
  - Invokes a parallel kernel function on a grid of nBlocks where each block instantiates nThreads concurrent threads

General CUDA Steps

1. Copy data from CPU to GPU
2. Compute on GPU
3. Copy data back from GPU to CPU

- By default, execution on host doesn't wait for kernel to finish
- General rules:
  - Minimize data transfer between CPU & GPU
  - Maximize number of threads on GPU

CUDA Elements

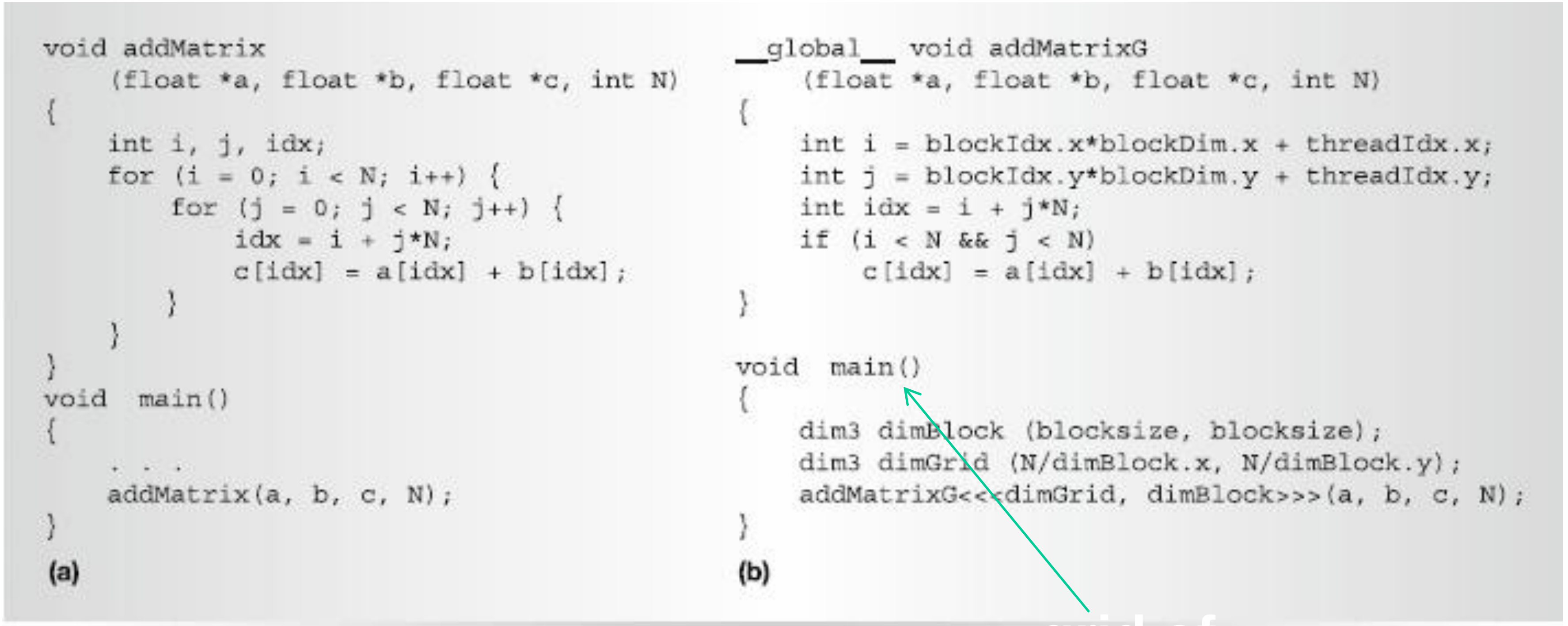
- cudaMalloc – for allocating memory in device
- cudaMemCopy – for copying data to allocated memory from host to device, and from device to host
- cudaFree – freeing allocated memory
- void syncthreads__() – synchronizing all threads in a block like barrier

Hello World Program

```
#include "../common/book.h"

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

```
#include "../common/book.h"
 __global__ void kernel( void ) {
}
 int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Example: Elementwise Matrix addition



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

(a)

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

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

(b)

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.

grid of

Example – Elementwise Matrix Addition

CPU Program

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

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

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

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

The nested `for`-loops are replaced with an implicit grid

Compileable example

```
const int N = 1024;
const int blocksize = 16;

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Compileable example

Set grid size

```
const int N = 1024;
const int blocksize = 16;

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

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

Compileable example

```
const int N = 1024;
const int blocksize = 16;
```

Compute kernel

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Compileable example

```
const int N = 1024;
const int blocksize = 16;

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

CPU Mem Allocation

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Compileable example

```
const int N = 1024;
const int blocksize = 16;

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = [illegible]
    a[i] = 1.0f;
```

GPU Mem Allocation

```
  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Compileable example

```
const int N = 1024;
const int blocksize = 16;

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&
  cudaMalloc( (void**)&
```

Copy data to GPU

```
  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Compileable example

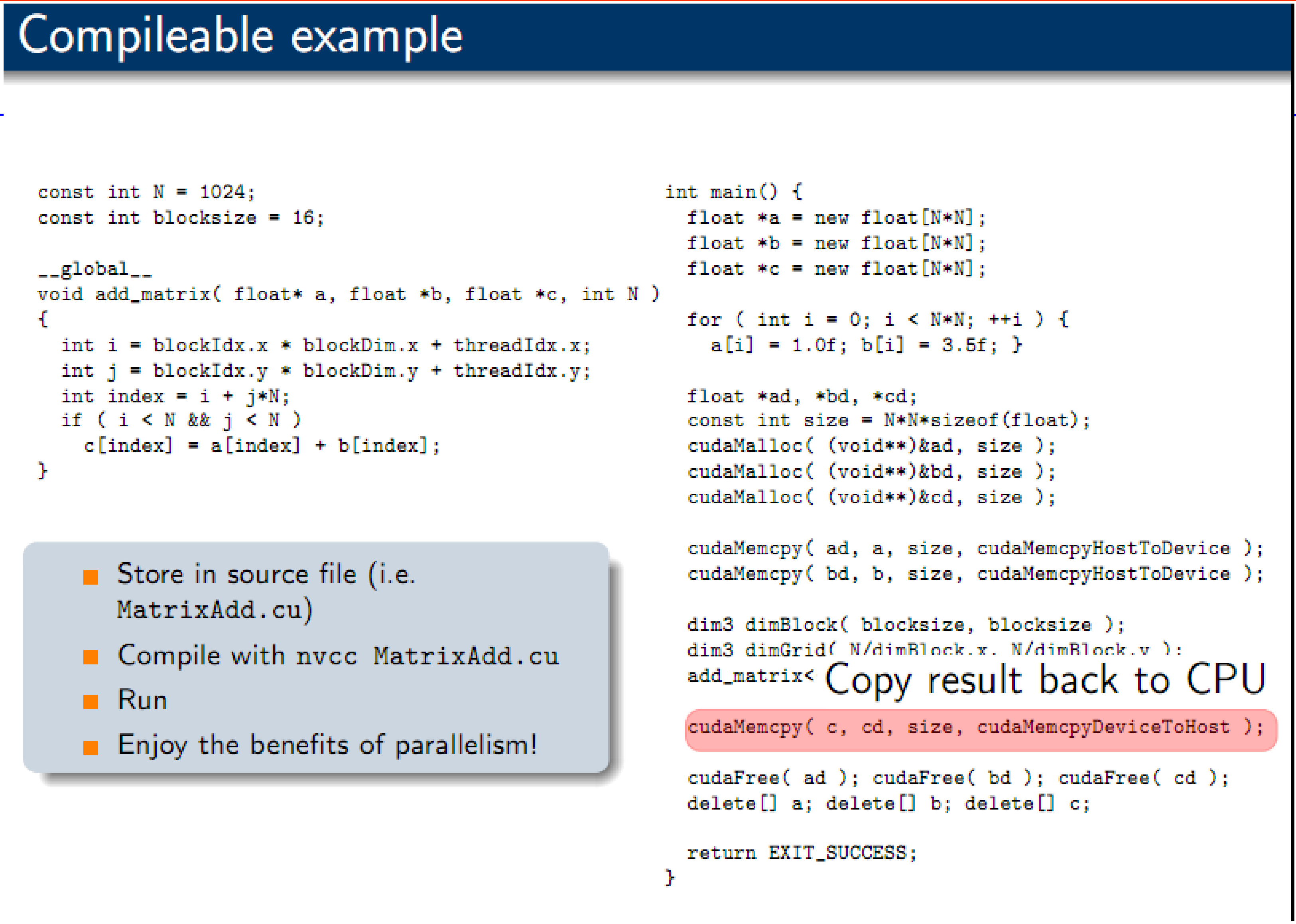
```
const int N = 1024;
const int blocksize = 16;

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cud
  cudaMemcpy( bd, b, size, cu
```

Execute kernel

```
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```



Compileable example

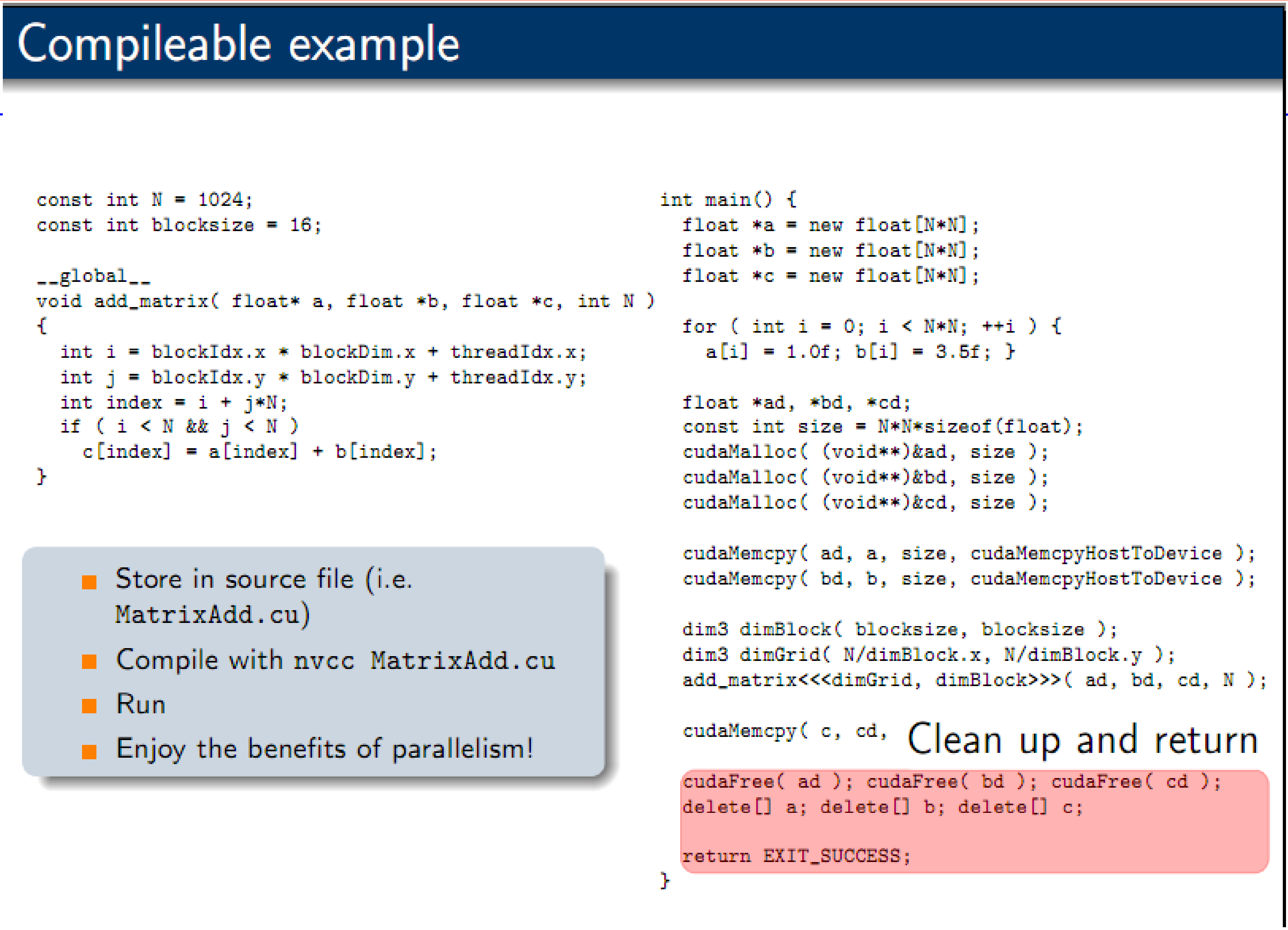
```
const int N = 1024;
const int blocksize = 16;

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<
```

Copy result back to CPU

```
  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```



Compileable example

```
const int N = 1024;
const int blocksize = 16;

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

- Store in source file (i.e. `MatrixAdd.cu`)
- Compile with `nvcc MatrixAdd.cu`
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd,
```

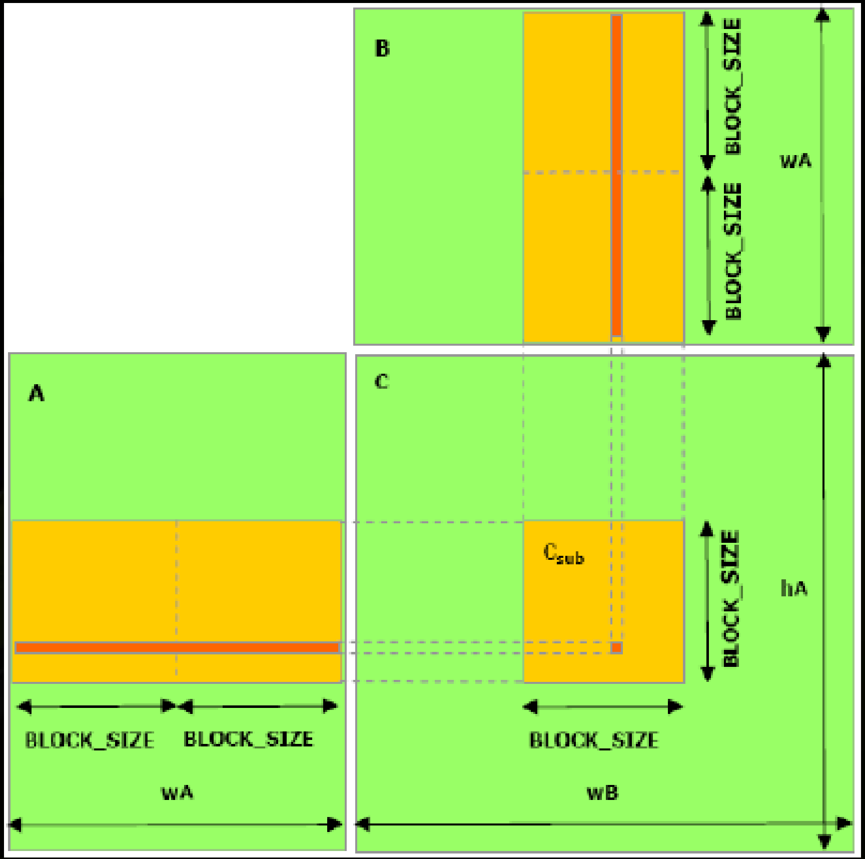
Clean up and return

```
  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

EXAMPLE 3: MATRIX MULTIPLICATION

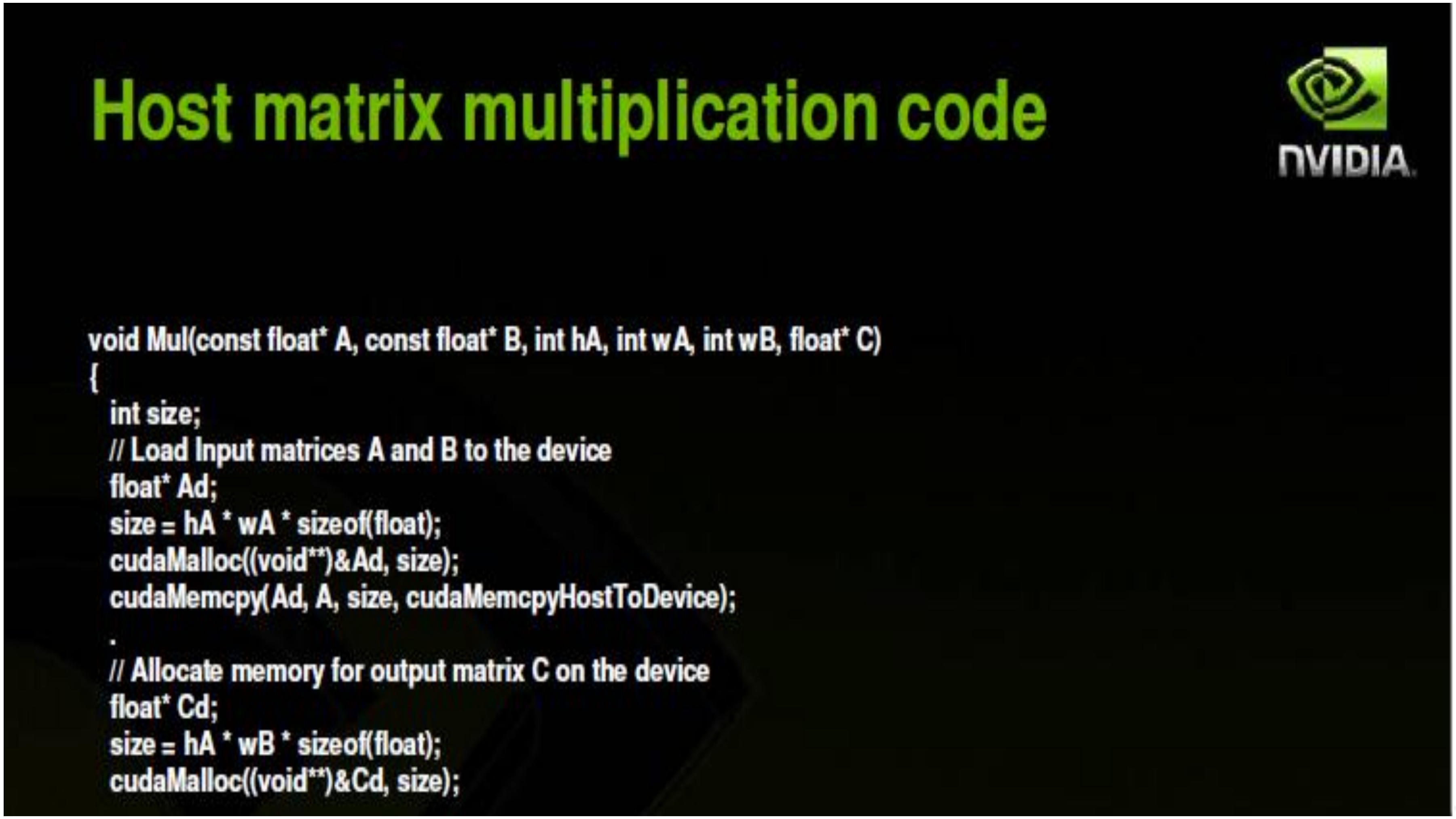
Matrix Multiplication Example

- Computing the product *C* of two matrices:
  - *A : (wA, hA)*
  - *B : (wB, wA).*
- Each thread block computes one square sub-matrix *Csub* of *C*;
- Each thread within the block computes one element of *Csub*.



National Institute of Science & Technology

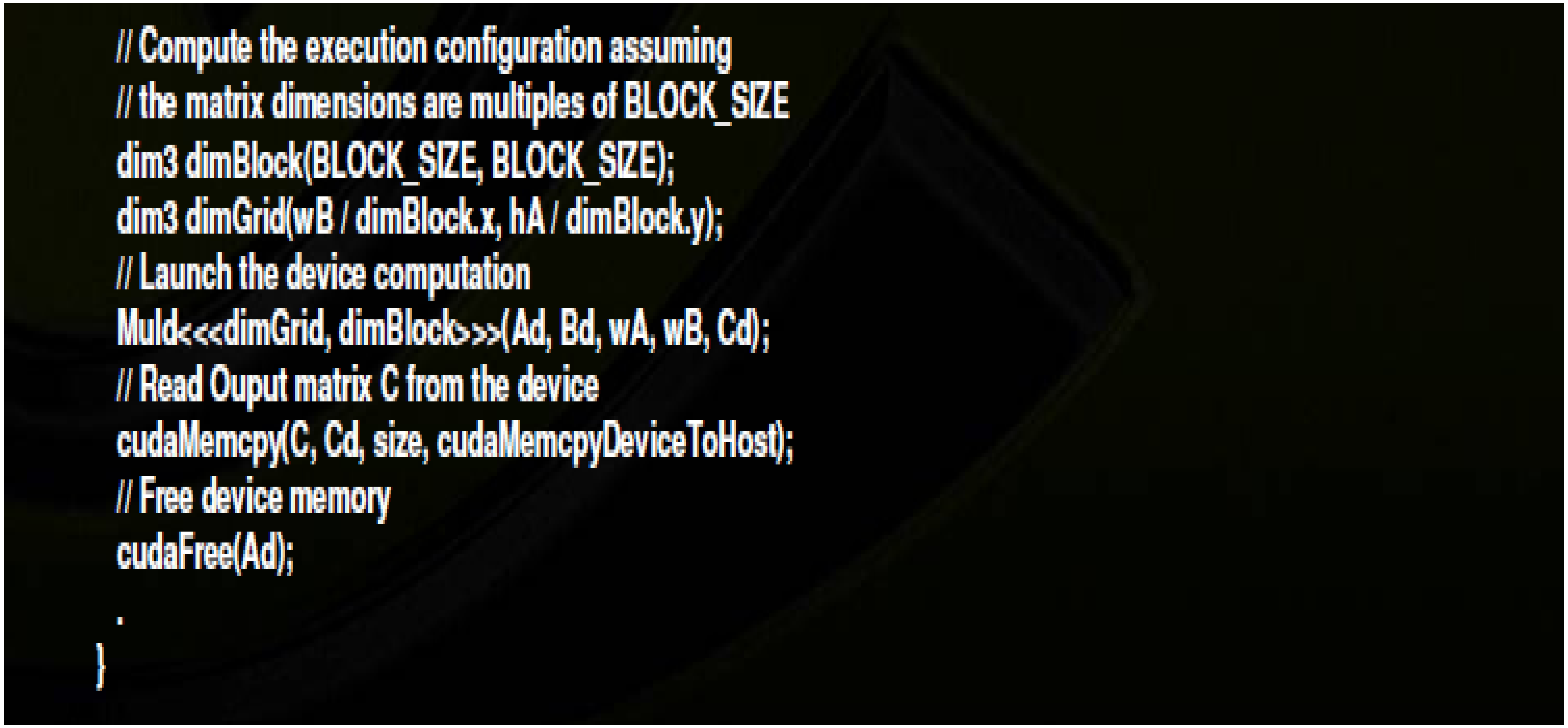
Example : Matrix Multiplication



Host matrix multiplication code

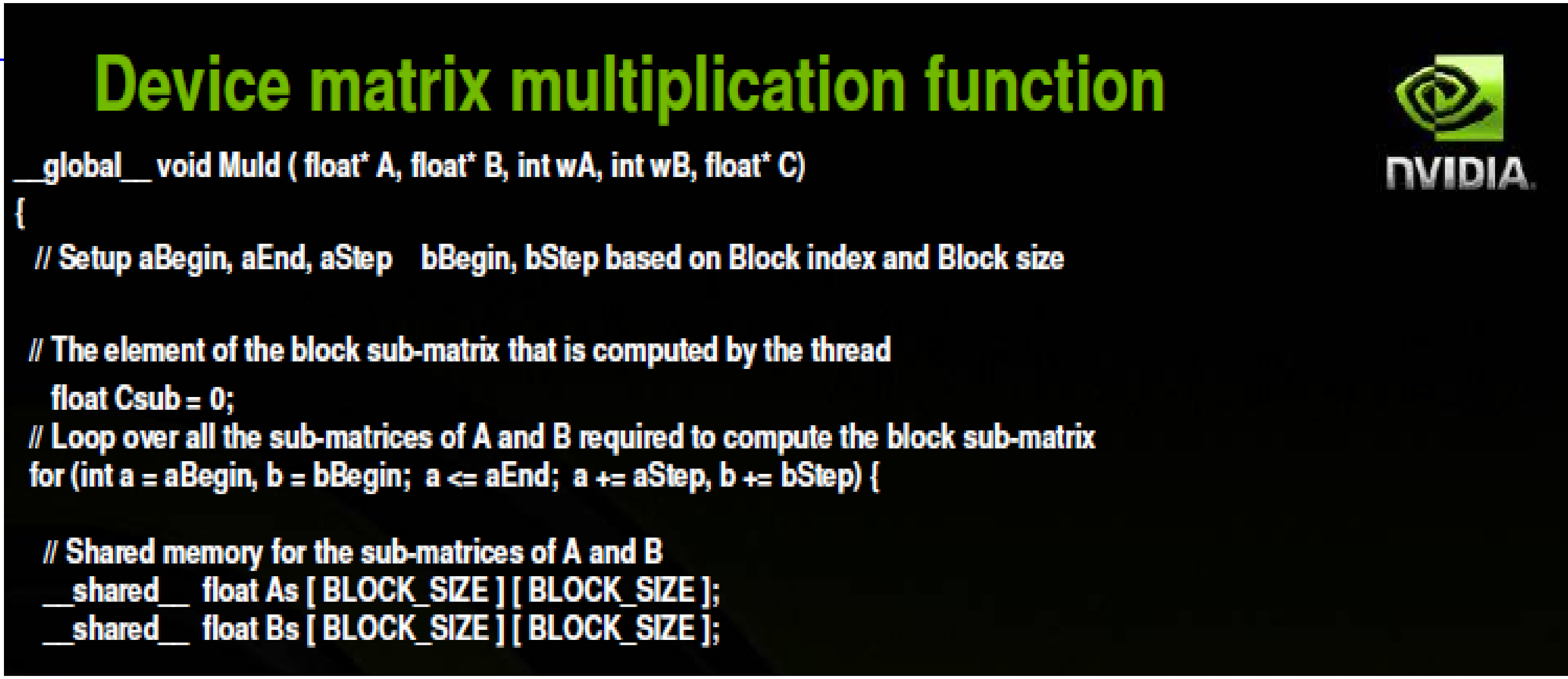
NVIDIA

```
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
   int size;
   // Load Input matrices A and B to the device
   float* Ad;
   size = hA * wA * sizeof(float);
   cudaMalloc((void**)&Ad, size);
   cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
   .
   // Allocate memory for output matrix C on the device
   float* Cd;
   size = hA * wB * sizeof(float);
   cudaMalloc((void**)&Cd, size);
```



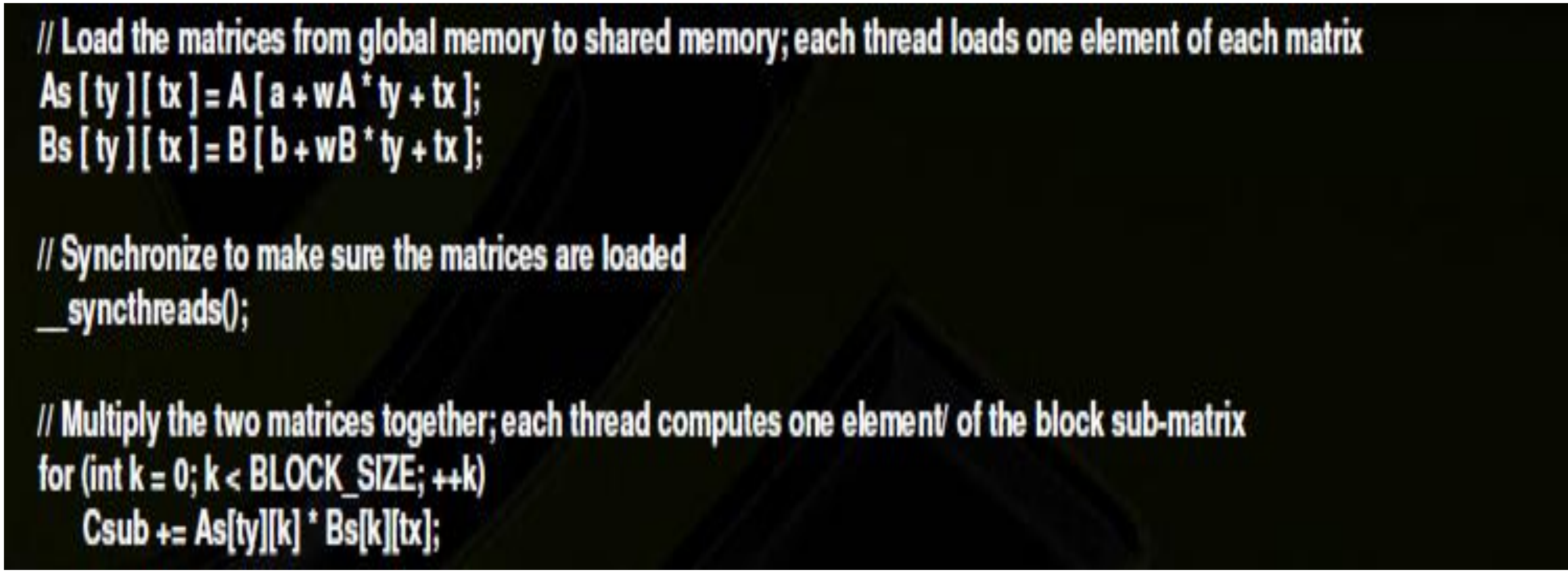
```
    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
    // Read Ouput matrix C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(Ad);

    .
}
```

Example 4



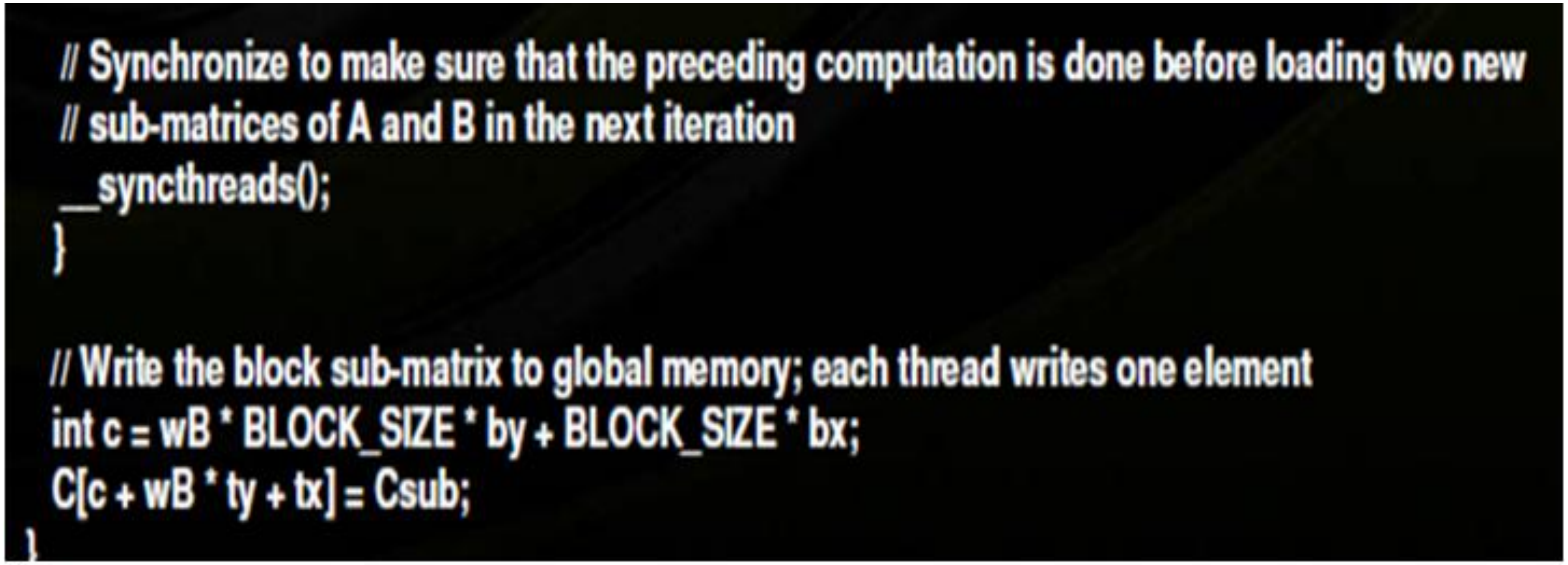
Device matrix multiplication function

```
__global__ void Muld ( float* A, float* B, int wA, int wB, float* C)
{
  // Setup aBegin, aEnd, aStep   bBegin, bStep based on Block index and Block size

 // The element of the block sub-matrix that is computed by the thread
   float Csub = 0;
 // Loop over all the sub-matrices of A and B required to compute the block sub-matrix
 for (int a = aBegin, b = bBegin;  a <= aEnd;  a += aStep, b += bStep) {

  // Shared memory for the sub-matrices of A and B
  __shared__   float As [ BLOCK_SIZE ] [ BLOCK_SIZE ];
  __shared__   float Bs [ BLOCK_SIZE ] [ BLOCK_SIZE ];
```



```
// Load the matrices from global memory to shared memory; each thread loads one element of each matrix
As [ ty ] [ tx ] = A [ a + wA * ty + tx ];
Bs [ ty ] [ tx ] = B [ b + wB * ty + tx ];

// Synchronize to make sure the matrices are loaded
__syncthreads();

// Multiply the two matrices together; each thread computes one element/ of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += As[ty][k] * Bs[k][tx];
```

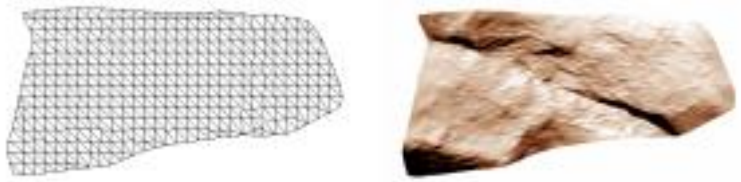
Example 4

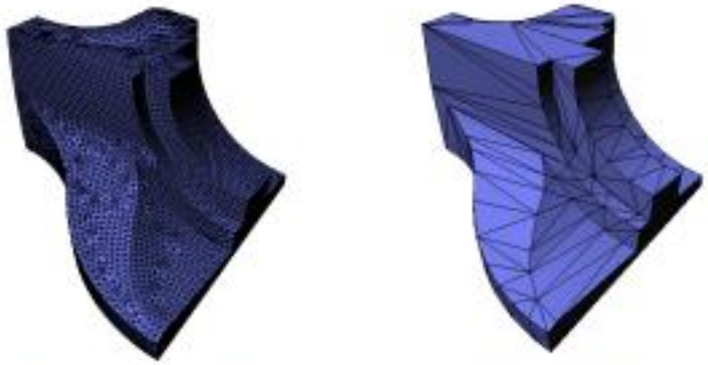


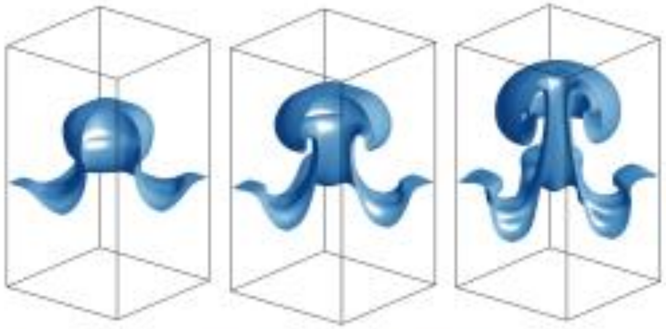
```
// Synchronize to make sure that the preceding computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}

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

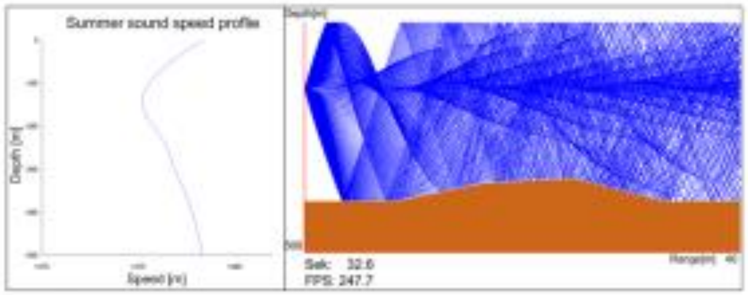
- For more information…
- CUDA SDK code samples – NVIDIA - http://www.nvidia.com/object/cuda_get_samples.html

GPU Activities at SINTEF Applied Mathematics


View-dependent tessellation


Preparation of finite element models (~5x)

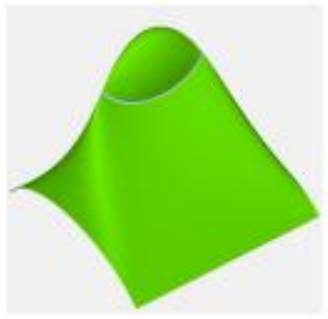

Soliving partial differential equations (~25x)



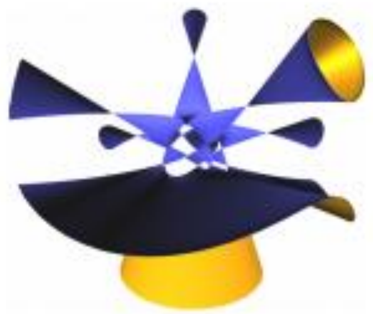
Marine aqoustics (~20x)

GPU Activities at SINTEF Applied Mathematics

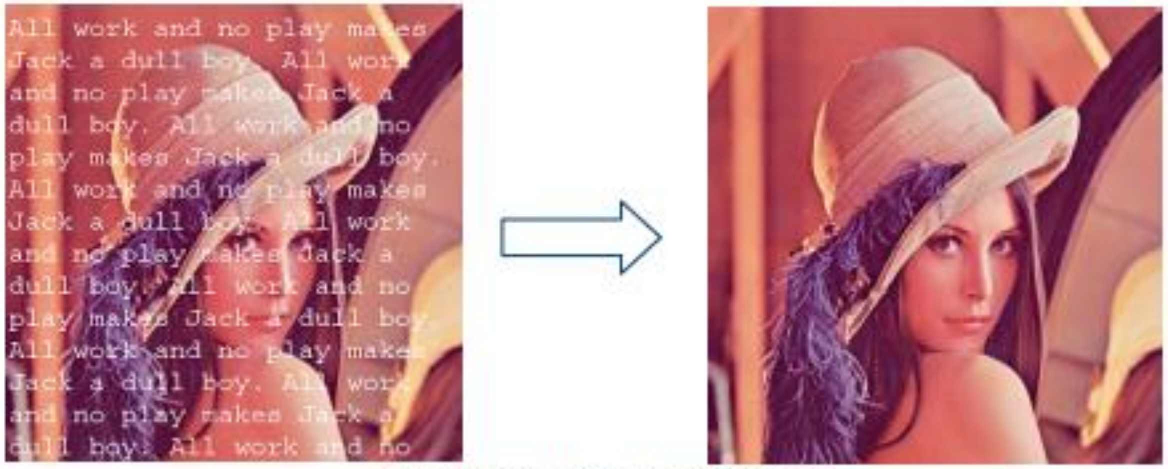
Silhouette refinement


Self-intersection detection of NURBS surfaces (~10x)

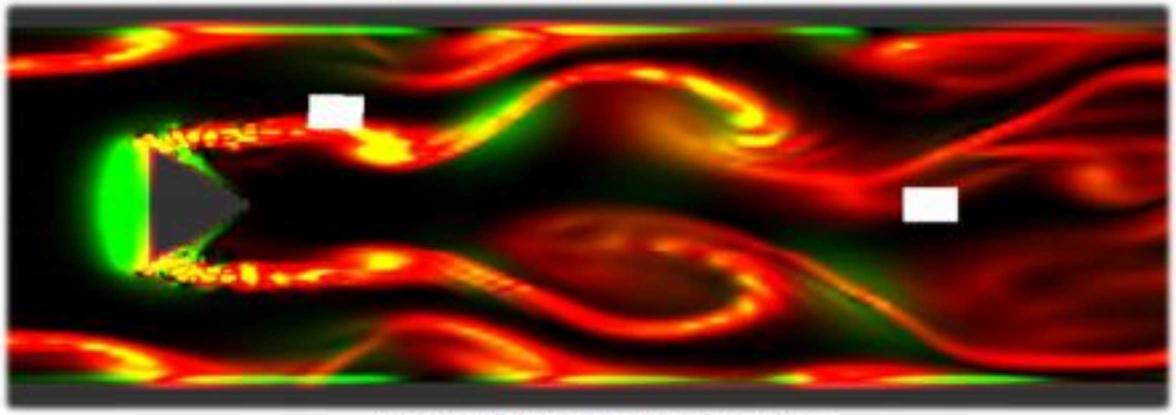
Registration of medical data (~20x)


Visualization of algebraic surfaces

GPU Activities at SINTEF Applied Mathematics

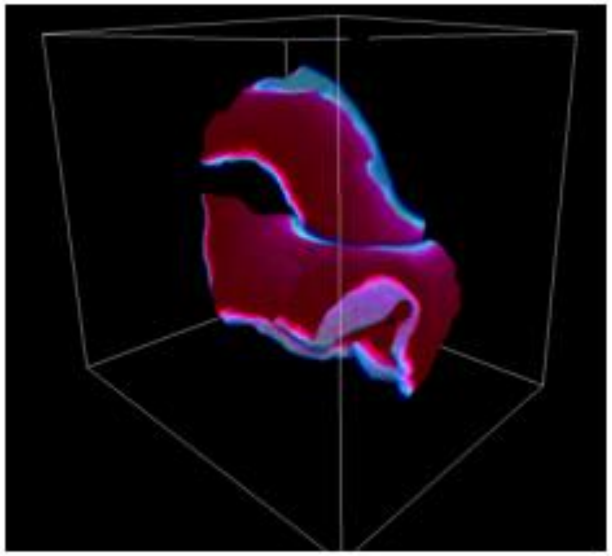

All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy.

Inpainting (~400x matlab code)



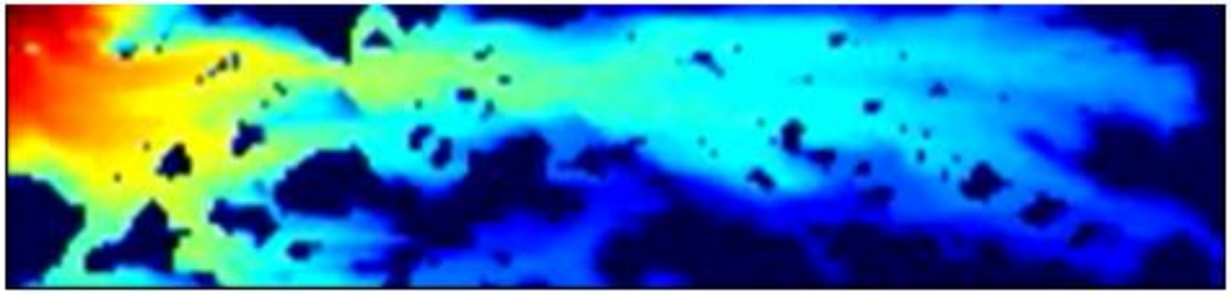
Navier-Stokes: Fluid dynamics

GPU Activities at SINTEF Applied Mathematics



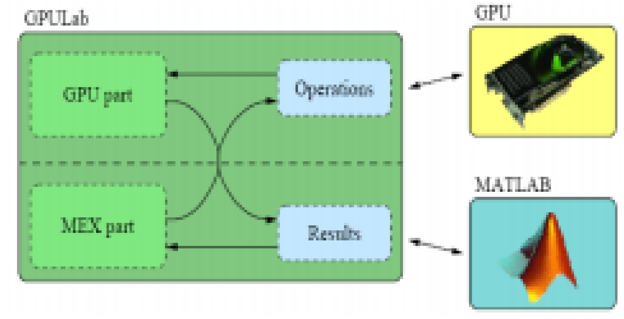
Volume visualization

Electric activity in a human heart.

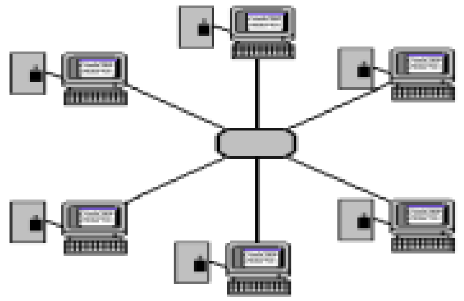


Water injection in a fluvial reservoir (20x)

GPU Activities at SINTEF Applied Mathematics



Matlab Interface to the GPU



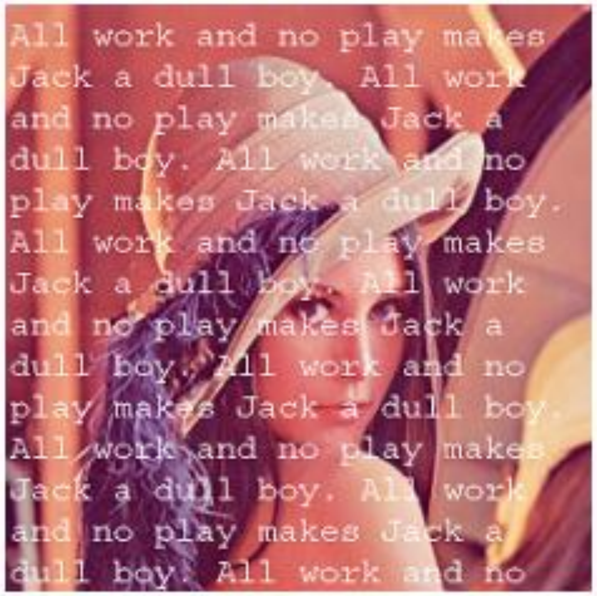
Cluster of GPU's

$$\begin{bmatrix} b_1 - a_{-\frac{1}{2}} & -a_{\frac{1}{2}} & 0 & 0 & 0 & \cdots & 0 \\ -a_{\frac{1}{2}} & b_2 & -a_{\frac{3}{2}} & 0 & 0 & \cdots & 0 \\ 0 & -a_{\frac{3}{2}} & b_3 & -a_{\frac{5}{2}} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -a_{n-\frac{5}{2}} & b_{n-2} & -a_{n-\frac{3}{2}} & 0 \\ 0 & \cdots & 0 & 0 & -a_{n-\frac{3}{2}} & b_{n-1} & -a_{n-\frac{1}{2}} \\ 0 & \cdots & 0 & 0 & 0 & a_{n-\frac{1}{2}} & b_n - a_{n+\frac{1}{2}} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n-2} \\ c_{n-1} \\ c_n \end{bmatrix}$$

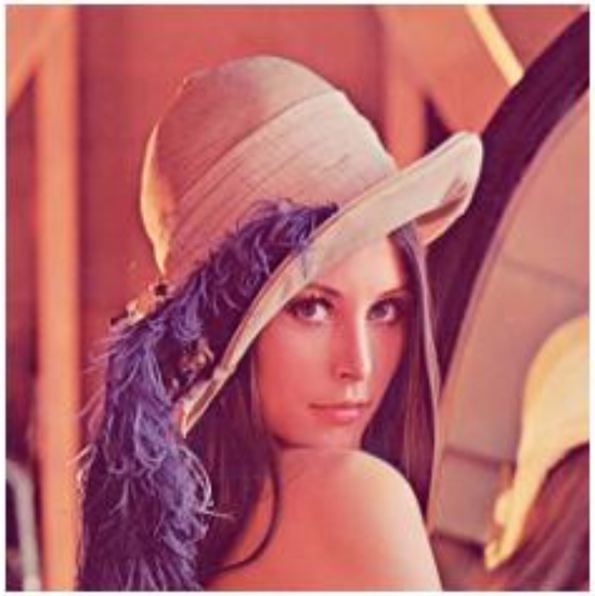
Linear algebra / load balancing CPU - GPU

TV-Stokes Inpainting

- Demo: GPU-application running on the laptop

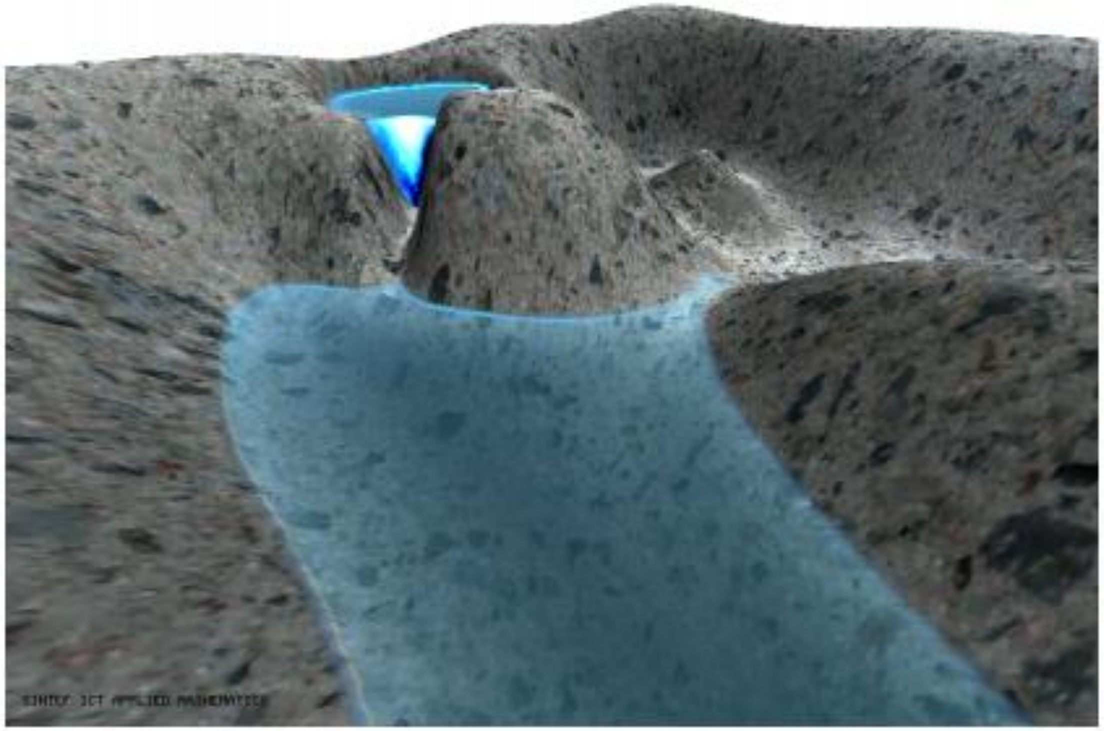

All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no

input image



result

Shallow-Water Equations (~25x)



Thank you