CUDA Programming

Johan Seland
johan.seland@sintef.no

24. January
Geilo Winter School
What is CUDA?

*Compute Unified Device Architecture*

- C programming language on GPUs
- Requires no knowledge of graphics APIs or GPU programming
- Access to native instructions and memory
- Easy to get started and to **get real performance benefits**!
- Designed and developed by NVIDIA
  - Requires an NVIDIA GPU (GeForce 8xxx/Tesla/Quadro)
- Stable, available (for free), documented and supported
- For both Windows and Linux
Why use an entire session for an API tied to specific hardware?

- The most modern API available for stream computing
  - If a market leader exists, it is CUDA
- 40 million CUDA enabled devices in December 2007
- Exposes the different types of memory available
  - Easier to get maximal performance out of the hardware
- For high performance, you target a specific processor/API
  - Traditional GPGPU programming has targeted a specific chip anyway (even with “standard” APIs)
- Future proof
  - CUDA is guaranteed to be supported on future hardware
Plan for the day

1. Programming Model

2. Application Programming Interface

3. Performance and Optimizations
   - Optimize memory access
   - Loop unrolling

4. Conclusion
GPU is viewed as a **compute device** operating as a coprocessor to the main CPU (host)

- Data-parallel, compute intensive functions should be off-loaded to the device
- Functions that are executed many times, but independently on different data, are prime candidates
  - I.e. body of `for`-loops
- A function compiled for the device is called a **kernel**
- The kernel is executed on the device as many different **threads**
- Both host (CPU) and device (GPU) manage their own memory, **host memory** and **device memory**
  - Data can be copied between them
The computational grid consists of a grid of thread blocks. Each thread executes the kernel. The application specifies the grid and block dimensions. The grid layouts can be 1, 2, or 3-dimensional. The maximal sizes are determined by GPU memory and kernel complexity. Each block has a unique block ID. Each thread has a unique thread ID (within the block).
Example – Elementwise Matrix Addition

**CPU Program**

```c
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**

```c
__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 );
}
```
Example – Elementwise Matrix Addition

### CPU Program

```c
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

```c
__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.
CUDA exposes all the different types of memory on the GPU:

- Global memory
- Constant memory
- Texture memory
- Local memory
- Registers
- Shared memory
- Thread (0, 0)
- Thread (1, 0)
- Block (0, 0)
- Block (1, 0)
Memory model II

To summarize

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Access</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Per thread</td>
<td>Read-Write</td>
<td></td>
</tr>
<tr>
<td>Local memory</td>
<td>Per thread</td>
<td>Read-Write</td>
<td></td>
</tr>
<tr>
<td>Shared memory</td>
<td>Per block</td>
<td>Read-Write</td>
<td>For sharing data within a block</td>
</tr>
<tr>
<td>Global memory</td>
<td>Per grid</td>
<td>Read-Write</td>
<td>Not cached</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Per grid</td>
<td>Read-only</td>
<td>Cached</td>
</tr>
<tr>
<td>Texture memory</td>
<td>Per grid</td>
<td>Read-only</td>
<td>Spatially cached</td>
</tr>
</tbody>
</table>

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
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
1 Programming Model

2 Application Programming Interface

3 Performance and Optimizations
   - Optimize memory access
   - Loop unrolling

4 Conclusion
Goals for this session

- Describe how CUDA integrates with existing source code
- Will only describe the most important functions
  - The rest are well documented in the programming guide
- Enough information to understand the performance issues
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

- CUDA C Compiler (nvcc)
- Machine independent assembly (PTX)
- CUDA Driver
- Debugger Profiler
- Optimized libraries: math.h, BLAS, FFT

- CPU Host Code
- Standard C compiler
- CPU
- Integrated CPU and GPU source code

Applied Mathematics
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 \( \_\_\text{sin}(x) \)
  - Detailed error bounds are available
- \( \_\_\text{syncthreads}() \)
  - 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)
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**
Common Runtime Component

- Built-in Vector types
  - I.e. float1, float2, int3, ushort4 etc.
  - Constructor type creation: int2 i = make_int2( i, j)
- Mathematical functions
  - Standard math.h on CPU, dedicated HW on GPU
- Time function for benchmarking
- Texture references
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;
}
const int N = 1024;
const int blocksize = 16;

#include <cuda_runtime.h>

#define CUDA_H "MatrixAdd.cu"

__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;
}
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);
    cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );
    cudaMemcpy( cd, c, 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];
}

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];
}

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;
}
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 );
    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;
}
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;
}
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;
}
Compileable example

```c
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!
1 Programming Model

2 Application Programming Interface

3 Performance and Optimizations
   - Optimize memory access
   - Loop unrolling

4 Conclusion
To optimize or not to optimize

Hoare said (and Knuth restated)

“Premature optimization is the root of all evil.”
To optimize or not to optimize

Hoare said (and Knuth restated)

“We should forget about small efficiencies, say about 97% of the time:
Premature optimization is the root of all evil.”

↓

3% of the time we really should worry about small efficiencies
(Every 33rd codeline)
We will go through the following:

- The CUDA execution model
- The cost of arithmetic operations
- A large section (with examples) on accessing global memory
- Using templates to unroll loops
Optimization goals

- We should strive to reach GPU performance
- We must know the GPU performance
  - Vendor specifications
  - Synthetic benchmarks
- Choose a performance metric
  - Memory bandwidth or GFLOPS?
- Use `clock()` to measure
- Experiment and profile!
How threads are executed

- A GPU consists of $N$ multiprocessors (MP)
- Each MP has $M$ scalar processors (SP)
- Each MP processes batches of blocks
  - A block is processed by only one MP
- Each block is split into SIMD groups of threads called warps
  - A warp is executed physically in parallel
- A scheduler switches between warps
- A warp contains threads of consecutive, increasing thread ID
- The warp size is 32 threads today
Consequences of execution model

- # of blocks / # of MPs > 1
  - Ensure that all MPs always have something to do
- # of blocks / # of MPs > 2
  - So multiple blocks can run concurrently on a MP
- # of blocks > 100 to scale to future devices
- Per block resources ≤ total available resources
  - Shared memory and registers
  - Multiple blocks can run concurrently on a MP
- Avoid divergent branching within one warp
  - Different execution paths should be serialized
Know the arithmetic cost of operations

- **4 clock cycles:**
  - Floating point: add, multiply, fused multiply-add
  - Integer add, bitwise operations, compare, min, max

- **16 clock cycles:**
  - Reciprocal, reciprocal square root, \( \log(x) \), 32-bit integer multiplication

- **32 clock cycles:**
  - \( \sin(x) \), \( \cos(x) \) and \( \exp(x) \)

- **36 clock cycles:**
  - Floating point division (24-bit version in 20 cycles)

- **Particularly costly:**
  - Integer division, modulo
  - **Remedy:** Replace with shifting whenever possible

- **Double precision (when available) will perform at half the speed**
1 Programming Model

2 Application Programming Interface

3 Performance and Optimizations
   - Optimize memory access
     - Loop unrolling

4 Conclusion
CUDA exposes all the different types of memory on the GPU:
Use the right kind of memory

- **Constant memory:**
  - Quite small, \( \approx 20K \)
  - As fast as register access if all threads in a warp access the same location

- **Texture memory:**
  - Spatially cached
  - Optimized for 2D locality
  - Neighboring threads should read neighboring addresses
  - No need to think about coalescing

- **Constraint:**
  - These memories can only be updated from the CPU
Accessing global memory

- 4 cycles to issue on memory fetch
- but **400-600** cycles of latency
  - The equivalent of 100 MADs
- Likely to be a performance bottleneck
- Order of magnitude speedups possible
  - **Coalesce** memory access
- Use shared memory to re-order non-coalesced addressing
How to achieve coalesced global memory access

For best performance, global memory accesses should be coalesced:

- A memory access coordinated within a warp
- A contiguous, aligned, region of global memory
  - 128 bytes – each thread reads a float or int
  - 256 bytes – each thread reads a float2 or int2
  - 512 bytes – each thread reads a float4 or int4
  - float3s are not aligned!
- Warp base address (WBA) must be a multiple of 16*sizeof(type)
- The kth thread should access the element at WBA + k
- Not all threads need to participate
- These restrictions apply to both reading and writing
  - Use shared memory to achieve this
Examples: Coalesced memory access

Coalesced memory access:
Thread $k$ accesses $WBA + k$
Examples: Coalesced memory access

Coalesced memory access:
Thread $k$ accesses $WBA + k$
Not all threads need to participate
Examples: Coalesced memory access

Non-Coalesced memory access:
Misaligned starting address
Examples: Coalesced memory access

Non-Coalesced memory access:
Non-sequential access
Examples: Coalesced memory access

Non-Coalesced memory access:
Wrong size of type
Example: Matrix Transpose

Example from the SDK

- Illustrates transpose using shared memory

\[
\begin{array}{cccc}
C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\
C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\
C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\
C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \\
\end{array}
\]

\[
\begin{array}{cccc}
C_{1,1} & C_{2,1} & C_{3,1} & C_{4,1} \\
C_{1,2} & C_{2,2} & C_{3,2} & C_{4,2} \\
C_{1,3} & C_{2,3} & C_{3,3} & C_{4,3} \\
C_{1,4} & C_{2,4} & C_{3,4} & C_{4,4} \\
\end{array}
\]
Uncoalesced transpose

```c
__global__ void
transpose_naive( float *out, float *in, int w, int h ) {
    unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;
    if ( xIdx < w && yIdx < h ) {
        unsigned int idx_in = xIdx + w * yIdx;
        unsigned int idx_out = yIdx + h * xIdx;
        out[idx_out] = in[idx_in];
    }
}
```

Reading from global mem:

```
[ ] [ ] [ ] [ ] [ ]
```

stride=1, coalesced

Writing to global mem:

```
[ ] [ ] [ ] [ ] [ ]
```

stride=16, uncoalesced
Coalesced transpose

- Assumption: Matrix is partitioned into square tiles
- Threadblock \((bx, by)\):
  - Read the \((bx, by)\) input tile, store into shared memory
  - Write the shared memory to \((by, bx)\) output tile
    - Transpose the indexing into shared memory
- Thread \((tx, ty)\):
  - Reads element \((tx, ty)\) from input tile
  - Writes element \((tx, ty)\) into output tile
- Coalescing is achieved if:
  - Block/tile dimensions are multiplies of 16
Coalesced transpose II

Read from global mem

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

Write to shared mem

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

Read “transposed” address from SMEM

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

Write to global mem

<table>
<thead>
<tr>
<th>0,0</th>
<th>1,0</th>
<th>2,0</th>
<th>15,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>1,0</td>
<td>1,1</td>
<td>1,15</td>
</tr>
<tr>
<td>0,2</td>
<td>1,2</td>
<td>1,15</td>
<td></td>
</tr>
<tr>
<td>0,15 1,15 2,15</td>
<td>15,15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
__global__ void
transpose( float *out, float *in, int w, int h ) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, index_transpose;

    if ( xIndex < width && yIndex < height ) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if ( xIndex < width && yIndex < height ) {
        out[index_out] = block[index_transpose];
    }
}
__global__ void
transpose(float *out, float *in, int w, int h) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, index_transpose;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if (xIndex < width && yIndex < height) {
        out[index_out] = block[index_transpose];
    }
}
__global__ void
transpose( float *out, float *in, int w, int h ) {
    __shared__ float block[BLOCK_DIM*BLOC
__global__ void
transpose( float *out, float *in, int w, int h ) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];
    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;
    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
    unsigned int index_out, index_transpose;
    if ( xIndex < width && yIndex < height ) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
        block[index_block] = in[index_in];
        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();
    if ( xIndex < width && yIndex < height ) {
        out[index_out] = block[index_transpose];
    }
}
__global__ void
transpose( float *out, float *in, int w, int h ) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, index_transpose;

    if ( xIndex < width && yIndex < height ) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if ( xIndex < width && yIndex < height ) {
        out[index_out] = block[index_transpose];
    }
}
```c
__global__ void
transpose( float *out, float *in, int w, int h ) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, index_transpose;

    if ( xIndex < width && yIndex < height ) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
}
```

Allocate shared memory.
Set up indexing.
Check that we are within domain, calculate more indices.
Write to shared memory.
Calculate output indices.

```
    __syncthreads();

    if ( xIndex < width && yIndex < height ) {
        out[index_out] = block[index_transpose];
    }
}
```
```
__global__ void
transpose( float *out, float *in, int w, int h ) {
  __shared__ float block[BLOCK_DIM*BLOCK_DIM];
  unsigned int xBlock = blockDim.x * blockIdx.x;
  unsigned int yBlock = blockDim.y * blockIdx.y;
  unsigned int xIndex = xBlock + threadIdx.x;
  unsigned int yIndex = yBlock + threadIdx.y;
  unsigned int index_out, index_transpose;
  if ( xIndex < width && yIndex < height ) {
    unsigned int index_in = width * yIndex + xIndex;
    unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
    block[index_block] = in[index_in];
    index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
    index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
  }
  __syncthreads();
  if ( xIndex < width && yIndex < height ) {
    out[index_out] = block[index_transpose];
  }
}
```
```c
__global__ void transpose(float *out, float *in, int w, int h) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

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

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __synchthreads();

    if ( xIndex < width && yIndex < height ) {
        out[index_out] = block[index_transpose];
    }
}
```
Was it worth the trouble?

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>Coalesced</th>
<th>Non-coalesced</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 \times 128$</td>
<td>0.011 ms</td>
<td>0.022 ms</td>
<td>2.0×</td>
</tr>
<tr>
<td>$512 \times 512$</td>
<td>0.07 ms</td>
<td>0.33 ms</td>
<td>4.5×</td>
</tr>
<tr>
<td>$1024 \times 1024$</td>
<td>0.30 ms</td>
<td>1.92 ms</td>
<td>6.4×</td>
</tr>
<tr>
<td>$1024 \times 2048$</td>
<td>0.79 ms</td>
<td>6.6 ms</td>
<td>8.4×</td>
</tr>
</tbody>
</table>
 transpose timings

Was it worth the trouble?

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>Coalesced</th>
<th>Non-coalesced</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 \times 128$</td>
<td>0.011 ms</td>
<td>0.022 ms</td>
<td>2.0×</td>
</tr>
<tr>
<td>$512 \times 512$</td>
<td>0.07 ms</td>
<td>0.33 ms</td>
<td>4.5×</td>
</tr>
<tr>
<td>$1024 \times 1024$</td>
<td>0.30 ms</td>
<td>1.92 ms</td>
<td>6.4×</td>
</tr>
<tr>
<td>$1024 \times 2048$</td>
<td>0.79 ms</td>
<td>6.6 ms</td>
<td>8.4×</td>
</tr>
</tbody>
</table>

For me, this is a clear yes.
Memory optimizations roundup

- CUDA memory handling is complex
  - And I have not covered all topics...
- Using memory correctly can lead to huge speedups
  - At least CUDA expose the memory hierarchy, unlike CPUs
- Get your algorithm up and running first, then optimize
- Use shared memory to let threads cooperate
- Be wary of “data ownership”
  - A thread does not have to read/write the data it calculates
1. Programming Model

2. Application Programming Interface

3. Performance and Optimizations
   - Optimize memory access
   - Loop unrolling

4. Conclusion
Loop unrolling

- Sometimes we know some kernel parameters at compile time:
  - # of loop iterations
  - Degrees of polynomials
  - Number of data elements
- If we could “tell” this to the compiler, it can unroll loops and optimize register usage
- We need to be generic
  - Avoid code duplication, sizes unknown at compile time
- Templates to rescue
  - The same trick can be used for regular C++ sources
Example: de Casteljau algorithm

A standard algorithm for evaluating polynomials in Bernstein form

\[ f(x) = b_{00}^d \]

Recursively defined:

\[ f(x) = b_{00}^d \]

\[ b_{i,j}^k = xb_{i+1,j}^{k-1} + (1 - x)b_{i,j+1}^{k-1} \]

\[ b_{0,j}^0 \text{ are coefficients} \]
The de Casteljau algorithm is usually implemented as nested for-loops

- Coefficients are overwritten for each iteration

```c
float deCasteljau(float* c, float x, int d)
{
    for ( uint i = 1; i <= d; ++i ) {
        for ( uint j = 0; j <= d-i; ++j )
            c[j] = (1.0f-x)*c[j] + x*c[j+1];
    }
    return c[0];
}
```

\[
f(x) = c_0^d
\]
We make \( d \) a template parameter

```cpp
template<int d>
float deCasteljau(float* c, float x, int d) {
    for ( uint i = 1; i <= d; ++i ) {
        for ( uint j = 0; j <= d - i; ++j )
            c[j] = (1.0f-x)*c[j] + x*c[j+1];
    }
    return c[0];
}
```

Kernel is called as

```cpp
switch ( d ) {
    case 1:
        deCasteljau<1><<<dimGrid, dimBlock>>> ( c, x ); break;
    case 2:
        deCasteljau<2><<<dimGrid, dimBlock>>> ( c, x ); break;
    ...
    case MAXD:
        deCasteljau<MAXD><<<dimGrid, dimBlock>>> ( c, x ); break;
}
```
Results

- For the de Castelaju algorithm we see a relatively small speedup
  - \( \approx 1.2 \times (20\%...) \)
- Very easy to implement
- Can lead to long compile times

Conclusion:
- Probably worth it near end of development cycle
1 Programming Model

2 Application Programming Interface

3 Performance and Optimizations
   - Optimize memory access
   - Loop unrolling

4 Conclusion
The CUDA libraries
- BLAS library - uses Fortran ordering
- FFT library
- CUDPP - library for parallel sum, sort and reduction

The CUDA profiler

Debugging CUDA applications (compile kernels for CPU)

Self-tuning CUDA applications

Bank conflicts when accessing shared memory
Why CUDA and not OpenGL/DirectX?

Advantages:
- No need to learn graphics API
- Better memory handling
- Better documentation
- *Scattered write*

Disadvantages:
- Tied to one vendor
- Not all transistors on GPU used through CUDA
  - Fancy tricks has been demonstrated using the rasterizer
Available resources

- NVIDIA CUDA Homepage
  - Contains downloads, documentation, examples and links
  - http://www.nvidia.com/cuda

- Programming Guide
  - Print and keep under your pillow

- CUDA Forums
  - http://forums.nvidia.com
  - The CUDA designers actually read and respond on the forum

- Supercomputing 2007 CUDA Tutorials
Conclusion

■ CUDA is here today (40M+ devices worldwide)
■ “Easy” to get started
■ Difficult to reach peak performance
  ■ Peak performance requires redesigned algorithm, not just “ports”
■ Single vs Double precision issues
Thank you for listening

- Please contact johan.seland@sintef.no with questions

The End!
Thank you for listening

- Please contact johan.seland@sintef.no with questions

Remember the quiz tonight:

- The price is an Nvidia Tesla C870