Writing and compiling a CUDA code

Philip Blakely

Laboratory for Scientific Computing, University of Cambridge

Outline

1. CUDA Language
2. Multi-dimensional thread blocks
3. CUDA extensions to C++
4. Global memory performance
5. Global memory correctness
6. Shared memory and performance

The CUDA language

- If we want fast code, we (unfortunately) need to know more-or-less exactly what the hardware is doing - even more so than on CPU.
- So need a low-level programming language
- C, C++ or Fortran would be suitable
- C++ used more these days (for better or worse)
- CUDA is essentially C++ with extensions specific to CUDA hardware.
- CUDA Fortran compiler exists `https://developer.nvidia.com/cuda-fortran`
- Fortran may be more amenable to CUDA optimization, but I have not tested this.

The CUDA language

- CUDA is very similar to C++, with a few additions
- Allows access to lightweight threading model via language extensions
- Functions must be designated as CUDA functions to run on the GPU
- All the speed and potential pitfalls of C++ are available
- Segmentation faults can be more dangerous on a GPU - usually because these will not be caught and will corrupt data rather than causing execution failure.

Runtime or Driver API?

Two APIs available: Runtime and Driver

- Both APIs are capable of roughly the same things:
  - Provide information about device parameters to host
  - Provide access to device memory from host
  - Allow host to set up and execute kernels
- Driver API more geared towards pre-compiled function libraries loaded at run-time (execution setup more complex)
- Runtime API more useful for your own functions
- For more information, see Reference Manual
  We concentrate on the Runtime API here.

Vector addition

As a first example of using CUDA, we shall look at a program which

- takes two vectors on the CPU
- passes them to the GPU
- adds them on the GPU
- passes them back to the CPU
- outputs them on the CPU.

The full code is in `Examples/addVectors.cu`
Starts with

```
#include <iostream>
#include <cuda.h>
```

Vector addition - Memory

- Memory is best allocated on the GPU from the CPU
- Dynamic memory allocation is possible from the GPU, but not advisable for performance reasons.

Allocating Memory

```
float *a, *b, *c;
cudaMalloc((void **) &a, N*sizeof(float));
```

sets `a` equal to a memory address on the GPU that is the start of a block of memory of size `N*sizeof(float)` bytes.
Note that we pass a *pointer* to `a` to `cudaMalloc`, and that `a` exists on the CPU.
General form:

```
cudaError_t cudaMalloc (void **devPtr, size_t size)
```

Vector addition - Copying data

On CPU, allocate space as normal:

```
float *aHost = new float[N];
```

In order to copy data from **aHost** (a pointer to data on the CPU) to **a** (a pointer to data on the GPU):

Memory copy

```
cudaMemcpy(a, aHost, N*sizeof(float),
  cudaMemcpyHostToDevice);
```

copies `N*sizeof(float)` bytes of data from **aHost** to **a**.

General form

```
cudaError_t cudaMemcpy (void *dst, const void *src, size_t
    count, enum cudaMemcpyKind kind)
```

Vector addition - Copying data

In order to copy data back from the GPU, use

```
cudaMemcpy(cHost, c, N*sizeof(float),
  cudaMemcpyDeviceToHost);
```

It is also possible to copy between memory spaces on the device itself

GPU to GPU copy (called on the CPU)

```
cudaMemcpy(c, d, N*sizeof(float), cudaMemcpyDeviceToDevice);
```

Vector addition - freeing memory

Freeing Memory

```
cudaFree(a);
```

releases the memory pointed to by `a` for later use by other `cudaMalloc` calls.
General form:

```
cudaError_t cudaFree (void *devPtr)
```

Vector addition - kernel

```
__global__ void add(float* a, float* b, float* c, int N)
{
  int i = threadIdx.x;

  if( i < N )
  {
    c[i] = a[i] + b[i];
  }
}
```

- Kernel designated by `__global__` keyword
- Kernel must have `void` return type.
- No direct return of information possible from kernels (asynchronous execution)
- Thread number given by the struct `threadIdx` (.x, .y, .z)
- Executed on the GPU - pointers assumed to relate to GPU memory.

Vector addition - launching kernel

- Kernel launches require thread-block and grid-dimension sizes to be specified.
- Recall that threads are arranged in blocks, and blocks are arranged into a grid.
- All thread-blocks in a grid have the same size.

Calling a simple kernel

```
const int N = 1024;

add<<<1, N>>>(a,b,c,N);
```

- launches a single block of 1024 threads for the kernel with
- `threadIdx.x` $= 0, 1, \ldots, 1023$
- First parameter is grid dimension, second is thread block dimension. These can be chosen at run-time.

Error handling

- CUDA API and kernel calls are often asynchronous i.e. they return immediately, potentially before the operation has completed.
- This allows CPU and GPU execution to overlap for performance.
- Therefore, if a function causes an error, the relevant error code may be returned by a later function.
- Error detection functions are:
  - `cudaError_t cudaGetLastError (void)`

    (returns last error, but also resets last error to `cudaSuccess`)
  - `const char* cudaGetErrorString (cudaError_t error)`

    returns a message string from an error code.
- It is therefore useful to surround all CUDA function calls and kernel calls by error checking using the above functions.
- When debugging, remember that the error you see may have been produced by a different function.

Error handling

If your compute GPU also controls the display (as opposed to the more usual compute-dedicated GPU):

- Kernels are limited to 5s each (not a major restriction as you usually run many kernels lasting for a few 10ms)
- Segmentation faults/buffer overflows can corrupt your display
- In extreme cases, display may freeze - a reboot is required
- Debugging on the GPU is not possible with `cuda-gdb` unless you switch to a virtual terminal (Ctrl+Alt+F<n>)

Vector addition - Larger vectors

- A thread block has a maximum size of 1024 threads and only uses a single SM.
- To use larger arrays (and more SMs), we must use a grid of thread-blocks.
- Use `blockIdx` containing index of current block within grid

Adding larger vectors

```
__global__ void add(float* a, float* b, float* c, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if( i < N ) {
    c[i] = a[i] + b[i];
  }
}
```

and to call the kernel:

```
  dim3 blocks((int)ceil(N / 1024.0));
  add<<<blocks, 1024>>>(a,b,c,N);
```

dim3

- The variables `threadIdx`, `blockIdx`, `blockDim` are of type `dim3`
- `dim3` is a struct with 3 integer members:
  (.x, .y, .z)
- These variables are assigned values by the hardware at run-time, and can therefore be used by your code.
- A variable of type `dim3` can be constructed with one, two, or three integers; the remaining components default to 1.

Outline

1 CUDA Language

2 Multi-dimensional thread blocks

3 CUDA extensions to C++

4 Global memory performance

5 Global memory correctness

6 Shared memory and performance

General thread blocks and grids

- So far we've looked at 1D thread blocks and grids
- To launch a 2D set of threads with all combinations of
  `threadIdx.x` $= 0, \ldots, 31$
  `threadIdx.y` $= 0, \ldots, 31$
  `blockIdx.x` $= 0, \ldots, 255$,
  `blockIdx.y` $= 0, \ldots, 127$:

General kernel launch

```
dim3 dimBlock(32,32,1);
dim3 dimGrid(256,128,1);
add<<<dimGrid, dimBlock>>>(a, b, c, N)
```

which would launch $32 \times 32 \times 256 \times 128 = 33,554,432$ threads.

- Every kernel will have variables:

```
dim3 blockDim(32,32,1);
dim3 gridDim(256,128,1);
```

General thread blocks and grids

- Running a 2D set of threads is simply a change in labelling by the hardware; you could have performed the 1D $\rightarrow$ 2D mapping yourself.
- A similar approach will work for 3D set of threads.
- It is up to you to decide how to divide work between threads and blocks and what dimensions to give each.
- A useful rule-of-thumb is: one grid-cell or matrix-element or data-point per thread (at least initially).
- For best performance, blocks should have power-of-two sizes, as large as possible, for reasons that will become clear later.
- If more threads are launched than grid-cells, then use `if(i < N)` constructs to avoid out-of-bounds access.

Thread blocks ctd

- So, there are two forms of launching a kernel:

  ```
  add<<<256, 1024>>>(a, b, c, N)
  ```

  which is equivalent to the second form:

  ```
  dim3 dimBlock(1024,1,1);
  dim3 dimGrid(256,1,1);
  add<<<dimGrid, dimBlock>>>(a, b, c, N)
  ```

  or even:

  ```
  dim3 dimBlock(1024);
  dim3 dimGrid(256);
  add<<<dimGrid, dimBlock>>>(a, b, c, N)
  ```

Vector - complex function

For a more complicated function of two vectors, we may want to use a separate function:

Discontinuous function

```
__device__ float f(float a, float b){
  if( a < 0 ){
    return 2*a;
  }
  else{
    return sin(a) + b;
  }
}

__global__ void g(...){
  ...
  c[i] = f(a[i], b[i]);
  ...
}
```

Device function calling

- Functions to be run on the GPU must have a `__device__` or `__global__` attribute.
- Any depth of `__device__` functions can be called within a `__global__` function.
- A `__global__` function may be called from within other kernels, using the `kernel<<<...>>>` syntax, but this is usually not advisable.
- A `__device__` function may only be called by a `__global__` or `__device__` function (i.e. not from the CPU directly).

Thread limits

- Maximum 1024 threads per block
- Maximum x/y dimension of block (in threads): 1024
- Maximum z dimension of block: 64
- Maximum $2^{31} - 1$ blocks in x/y/z grid dimension

For full details, see Appendix K of the CUDA C Programming Guide.

Summary of Automatic variables

The following variables are available in any `__global__` and `__device__` function:

- `gridDim` (`dim3`)
  Dimension of current grid
- `blockIdx` (`uint3`)
  Index of current thread block within grid
- `blockDim` (`dim3`)
  Dimension of current block
- `threadIdx` (`uint3`)
  Index of current thread within block
- warpSize (`int`) Size of current warp (Always 32 on current hardware)

All (except the last) are structs with three members: `x, y, z`.
Their members are generated by the hardware and are read-only.

Vector addition - profiling

- Asynchronous calls are problematic with timing CUDA programs.
- `clock()` may not give fine enough timings.
- We create events, and find the time between them afterwards.

```
cudaEvent_t start, endMemcpy, endAdd, end;
cudaEventCreate(&start);

cudaEventRecord(start, 0);
add<<<1, N>>>(a,b,c,N);
cudaEventRecord(end, 0);

cudaEventSynchronize(end);

float memcpyTime, addTime, totalTime;
cudaEventElapsedTime(&totalTime, start, end);

cudaEventDestroy(start);
```

Gives time in milliseconds with resolution of 0.5 microseconds.
Imagine markers being placed in the stream of CUDA function-calls and their actual times extracted later.

Outline

1 CUDA Language

2 Multi-dimensional thread blocks

3 CUDA extensions to C++

4 Global memory performance

5 Global memory correctness

6 Shared memory and performance

CUDA extensions to C++ - Host functions

Host functions (on CPU)

- All valid C++17 code should be permitted
- Functions can be prefixed with `__host__` attribute (not required)
- Callable on and by the host only.
- For consistency, use `nvcc` for all compilation; host compiler is used for host-only code.
- Main CUDA API:

```
#include <cuda.h>
#include <driver_types.h>
#include <cuda_runtime_api.h>
```

- For extra CUDA types (`int2, float3` etc.):

```
#include <vector_types.h>
#include <vector_functions.h>
```

You only need to use the CUDA compiler when

- Defining `__device__` or `__global__` functions/variables
- Calling kernels via `kernel<<<N,M>>>` syntax.

CUDA extensions to C++ - Kernel functions

Kernel (global) functions

- Kernel functions must be prefixed with `__global__`
- Executed on device, callable from host or device.
- Parameters cannot be references.
- Parameters are passed via constant memory.
- Must have `void` return type.
- Call is asynchronous - returns before device has finished execution

Use `cudaDeviceSynchronize()` to ensure that all kernels on device attached to current CPU-thread have finished execution.

CUDA extensions to C++ - Device functions

Device functions

- Have to prefix functions explicitly with `__device__`
- All valid C++ code (except STL, exceptions, and run-time type-information)
- Most C++17 features supported.
- Device code executed on device and callable from device only

Device functions may therefore make use of:

- Function overloading
- Default parameters
- Namespaces
- Function templates (effectively passing parameters at compile-time)
- Classes

CUDA extensions to C++ - functions

- Functions can be declared as both `__device__` and `__host__` and are compiled for both CPU and GPU as necessary.
- This is particularly useful for classes that are needed on CPU and GPU, for example:

```
class Array{
public:
  __device__ __host__ float& getElement(size_t i);
};
```

In this case, `__host__` is necessary.

CUDA extensions to C++ - variable attributes

Variables defined outside functions can have the following attributes:

__device__

- Resides in global memory on device (kernels can read/write)
- Lasts for whole application
- Accessible on all device threads, and from host for read/write via `cudaMemcpyToSymbol()`

__constant__

- Resides in constant memory on device (kernels can only read)
- Lasts for whole application
- Accessible on all device threads, and from host for read/write via `cudaMemcpyToSymbol()`

CUDA extensions to C++ - variable attributes

Solver parameter

Header file:

```
__constant__ int solver;
```

On host:

```
int solver_CPU;
std::cin >> solver_CPU;
cudaMemcpyToSymbol(solver, &solver_CPU, sizeof(int), 0,
    cudaMemcpyHostToDevice);
```

On device:

```
__device__ void update(float* U, float dt){
  switch(solver){
  ...
  }
}
```

We are copying `sizeof(int)` bytes from the CPU to the GPU.

CUDA extensions to C++ - variable attributes

Variables defined in `__device__` or `__global__` functions can have the following attributes:

__shared__

- Resides in shared memory of Streaming Multiprocessor on which thread block is running
- Lasts for lifetime of thread block
- Shared/accessible between all threads in same block
- Not accessible from other thread-blocks even in the same grid.
- Need special commands to avoid race conditions on read/write.

volatile

- Applies to a variable in global or shared memory
- Forces explicit memory-read when variable is read
- Otherwise compiler will assume that value doesn't change between reads - for optimization

CUDA extensions - New vector types

- Types such as `int4`, `float3`, `double2` are available in both host and device code
- with elements `x`, `y`, `z`, `w` as appropriate
- Can be constructed with e.g. `make_int2(1,2)`
- Elementwise arithmetic operators such as `+,-,*,/` available
- Only present for convenience - they do not translate to vectorized instructions - which don't exist on NVIDIA hardware.
- (This is because all NVIDIA instructions are effectively very wide vector-instructions that apply to all threads in a warp.)

Laplace's equation and memory bandwidth

Philip Blakely

Laboratory for Scientific Computing, University of Cambridge

Simple 2D example

Suppose we want to solve Laplace's equation in 2D

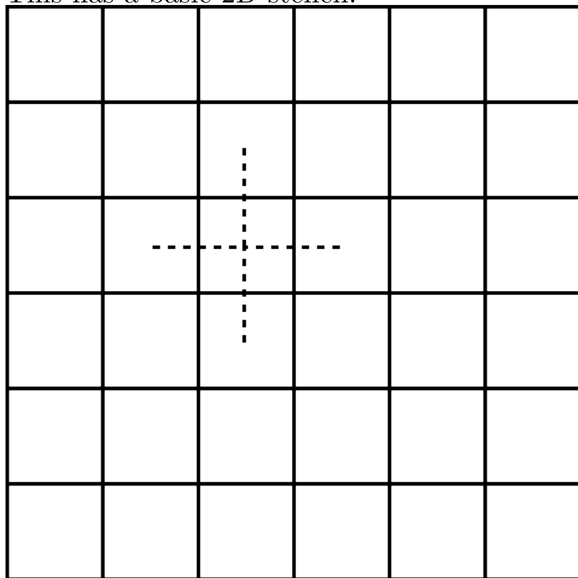
$$\frac{\partial u}{\partial t} = \nabla^2 u$$

using a forward Euler method:

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \left(\frac{\Delta t}{\Delta x^2}\right)\left(u_{i+1,j}^{n} + u_{i,j+1}^{n} - 4u_{i,j}^{n} + u_{i-1,j}^{n} + u_{i,j-1}^{n}\right)$$

Simple 2D example ctd

This has a basic 2D stencil:



Basic thread set-up

- For simplicity, we choose to update one cell per thread.
- Our thread-block size will be 32×32.
- Therefore, our grid dimension will be:

```
dim3 blockDim(32, 32, 1);
dim3 gridDim((int)ceil(cells/32.0), (int)ceil(cells/32.0), 1);
```

- We call a kernel using:

```
setInitialData<<<gridDim, blockDim>>>(dataCurr, cells);
```

- Within a kernel, the cell we update is:

```
int i = threadIdx.x + blockIdx.x * 32;
int j = threadIdx.y + blockIdx.y * 32;
```

- If we wanted to choose our block-size at run-time, we could use `blockSize.x` and `blockSize.y` in place of 32.

Out-of-bounds protection

If our grid is 100×100, for example, we launch a grid of 4×4 thread-blocks, each with 32×32 threads.
Block $(3, 3)$ corresponds to elements $(96, 96)$ to $(127, 127)$.
Therefore, we include within the kernel:

```
__global__ void setInitialData(float* data, int N)
{
  int i = threadIdx.x + blockIdx.x * 32;
  int j = threadIdx.y + blockIdx.y * 32;
  if(i <= N-1 && j <= N-1)
  {

...
  }
}
```

to stop out-of-bounds access from happening.
This is normal practice within CUDA.

Memory allocation

- The forward-Euler update requires two arrays to be kept in memory:
- One for the currrent time-step, and one for the next time-step:

```
float *dataCurr, *dataNext, *hostData;
hostData = new float[cells *cells];
cudaMalloc(&dataCurr, cells * cells * sizeof(float));
cudaMalloc(&dataNext, cells * cells * sizeof(float));
```

- We swap the pointers after each time-step to avoid an explicit copy operation.

Evolution step

We update all cells inside the boundary according to the finite-difference formula:

```
__global__ void advance(const float* dataOld, float* dataNew,
    int N, float dt)
{
  int i = threadIdx.x + blockIdx.x * 32;
  int j = threadIdx.y + blockIdx.y * 32;

  if(i > 0 && i < N-1 && j > 0 && j < N-1)
  {
    dataNew[i + j*N] = dataOld[i + j*N] +
      (dt/(dx*dx))*(dataOld[i+1 + j*N]
        + dataOld[i-1 + j*N]
        + dataOld[i + (j+1)*N]
        + dataOld[i + (j-1)*N]
        - 4*dataOld[i + j*N]);
  }
}
```

Host instructions

To call the kernel from the host:

- We compute $\Delta t = \mu \Delta x^2$:

```
float dt = mu*dxCPU*dxCPU;
```

- call the `advance` kernel:

```
advance<<<gridDim, blockDim>>>(dataCurr, dataNext, cells, d
```

- swap the data pointers:

```
std::swap(dataCurr, dataNext);
```

- and loop until `t == T`.

Summary

- We have now covered the basic additions to C++ that form CUDA
- However, CUDA is hard to optimize
- We need to look at hardware characteristics to understand performance
- Also need to ensure correctness of algorithms

Outline

1. CUDA Language
2. Multi-dimensional thread blocks
3. CUDA extensions to C++
4. Global memory performance
5. Global memory correctness
6. Shared memory and performance

Global memory access

- Global memory bandwidth is fast - 1,500-2,000 GB/s for latest cards
- But - there are many factors that can cause lower performance
- Global memory is accessed in 32-byte, 64-byte, and 128-byte chunks, aligned to start at a multiple of their size.
- Each warp of 32 threads coalesces global memory reads into as few chunks as possible.
- If not all the data in each chunk is used/needed (by the entire warp), then bandwidth has been wasted.

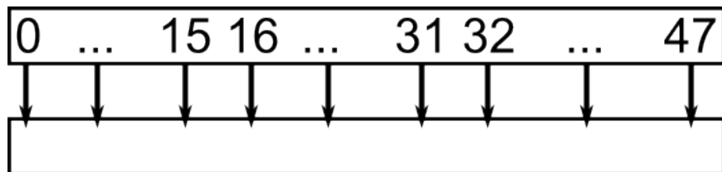
Global memory access - coalesced example

Perform an operation on each element of $N \times N$ array `a`

Take `N=16` and we concentrate on one block only.

```
__global__ void f(float* a, int N){
  int i = threadIdx.x;
  int j = threadIdx.y;
  float r = a[ j * N + i ]
  /* Do something with r */
  a[j * N + i] = r;
}
```

- Adjacent threads read adjacent 4-byte `floats` in memory - coalesced access
- Each set of sixteen threads reads 64 bytes
- One memory transaction per 16 threads



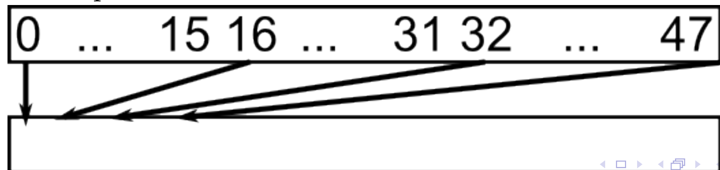
Global memory access - uncoalesced example

Perform an operation on each element of $N \times N$ array `a`

Take `N=16` and we concentrate on one block only.

```
__global__ void f(float* a, int N){
  int i = threadIdx.x;
  int j = threadIdx.y;
  float r = a[ i * N + j ]
  /* Do something with r */
  a[i * N + j] = r;
}
```

- Adjacent threads read `floats` separated by 16×4 bytes
- Each thread reads the minimum-sized chunk of 32 bytes (512 bytes for 16 threads)
- One memory transaction per thread
- Effective bandwidth reduced by factor of 8 on all compute capabilities



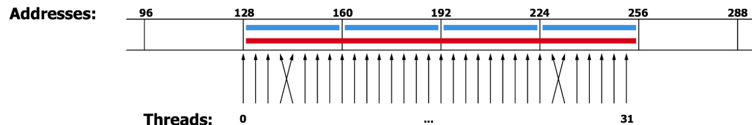
Coalescence details ctd.

- If some threads do not access any data, this does not add any further overhead to the data access.
- The order in which threads access the data within a block is not important.
- Global memory access is cached through the global L2 cache and per-multiprocessor L1 cache. Attaining the maximum memory bandwidth is therefore somewhat easier than the above might suggest.
- The method used to determine which memory reads can be coalesced is complicated. For full details, see the Programming Guide.

Coalescence illustrations

Consider: `float x = a[i];` (with minor variations)

Aligned accesses (sequential/non-sequential)



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

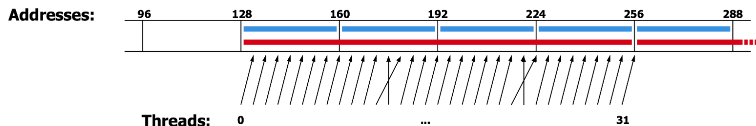
Diagram Copyright NVIDIA

When all threads access sequential elements aligned to a 128 byte boundary, we get the minimum possible amount of data transfer. Whether caching occurs or not depends partly on the compiler and partly on the run-time configuration of the L1/L2 caches.

Coalescence illustrations

Consider: `float x = a[i+1];`

Mis-aligned accesses (sequential/non-sequential)



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Diagram Copyright NVIDIA

When threads access memory in a misaligned fashion, there is always some wastage, since we only use one element of, say, the second 128-byte block.

Determining bandwidth

- Bandwidth (in GB/s) for a kernel is given by:

$$\frac{(\text{Bytes read}) + (\text{Bytes written})}{1024^3 \times (\text{Time in ms})/1000}$$

- If a kernel requires no or little computation, this should be close to the theoretical bandwidth for the device, for best performance.
- If a calculation is bandwidth-bound, then it is necessary to work on the layout of data in memory, or find some way to cache data in shared memory, or hide the latency.

Outline

1. CUDA Language
2. Multi-dimensional thread blocks
3. CUDA extensions to C++
4. Global memory performance
5. Global memory correctness
6. Shared memory and performance

Global memory access visibility

- When a thread issues a write to global memory, the updated value may not be available to other threads.
- Remember that the order in which blocks run is unknown.

In order to ensure that read/write has occured, use

- `__threadfence_block()` waits until all global and shared memory accesses made by the calling thread are visible to the current block
- `__threadfence()` waits until all global and shared memory accesses made by the calling thread are visible to all threads in the block (for shared memory) or device (for global memory)
- `__syncthreads()` waits until all threads in the block have reached this point, and also as for `__threadfence_block()`

Without these, the compiler may optimize global/shared read/write and assume that two accesses to the same global memory location return the same value. (see also `volatile` keyword)

Potential global memory access problem

Contrived example of undefined behaviour when accessing global memory:

```
__global__ void myKernel(int *array){
  int i = threadIdx.x;
  int x = array[i];
__syncthreads();
array[i+1] = i;
__syncthreads();
int y = array[i];
array[i] = x * y;
```

- If either of the `__syncthreads()` were missing, the required effect that `array[i] *= (i-1)` might not hold.

Potential global memory access problem

Contrived example of undefined behaviour when accessing global memory:

```
__global__ void myKernel(int *array){
  int i = threadIdx.x;
  int x = array[i];

array[i+1] = i;
__syncthreads();
int y = array[i];
array[i] = x * y;
```

- If either of the `__syncthreads()` were missing, the required effect that `array[i] *= (i-1)` might not hold.
- First missing: Thread i=31 could set `array[32]=31` before thread 32 reads original value into `x`

Potential global memory access problem

Contrived example of undefined behaviour when accessing global memory:

```
__global__ void myKernel(int *array){
  int i = threadIdx.x;
  int x = array[i];
__syncthreads();
array[i+1] = i;

int y = array[i];
array[i] = x * y;
```

- If either of the `__syncthreads()` were missing, the required effect that `array[i] *= (i-1)` might not hold.
- First missing: Thread i=31 could set `array[32]=31` before thread 32 reads original value into `x`
- Second missing: Thread 32 could read `array[32]` before thread 31 has set `array[32]=31`

Warp synchronization

- The use of 32 above is deliberate; threads in different warps will almost certainly not run simultaneously.
- Even for smaller values of 32 we could have encountered a problem; use `__syncwarp()` instead of `__syncthreads()`.

General advice

Easiest approach for Global memory is found by writing each global memory location from precisely one thread.
This should avoid any issues of correctness.

Outline

1 CUDA Language

2 Multi-dimensional thread blocks

3 CUDA extensions to C++

4 Global memory performance

5 Global memory correctness

6 Shared memory and performance

Shared memory

Recall: Shared memory is allocated *per thread-block* and is available to all threads in that block

Using shared memory

```
__global__ void f(float a, float b){
  __shared__ float data[16][16];
}
```

- The `data` array (1024 bytes) would be available for reading and writing by all threads in the block.
- There is one `data` array per thread block. Thread blocks cannot access each other's shared memory.
- The maximum shared memory available is around 48kB (may be higher on some high-end cards). It is very easy to exceed this limit, and get a kernel that will fail to run.
- Shared memory results only available to threads in other warps after `__syncthreads()`

Use of shared memory

Matrix multiplication

Given matrices:
$A\ (M \times 16)$, $B\ (16 \times N)$,
multiply to give:
$C = AB\ \ (M \times N)$

- We shall cover several ways of doing this, with increasing efficiency.
- Kernels taken from NVIDIA_CUDA_C_BestPractices.pdf
- Surrounding code just initializes matrices and times kernels
- Use block-size 16×16.
- Remember that warp size is 32, so block height and width are a half-warp.

Naïve approach

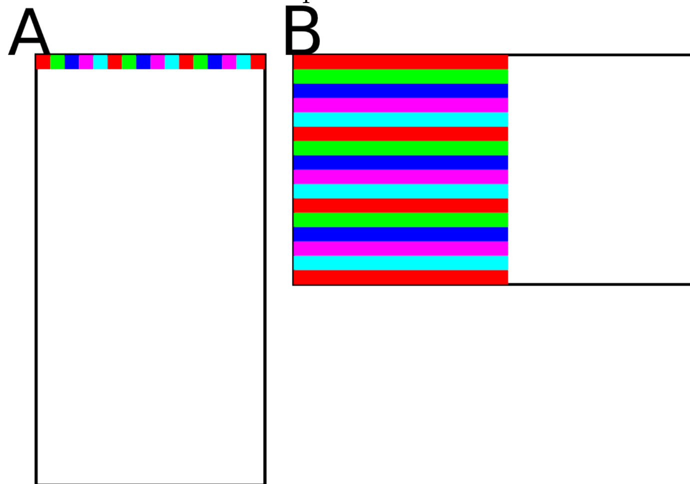
All data read directly from global memory on each thread.

```
#define TILE_DIM 16

// Taken from CUDA Best-Practices Guide Listing 3.7
__global__ void simpleMultiply(float *a, float* b, float *c,
     int N){
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for (int i = 0; i < TILE_DIM; i++) {
    sum += a[row*TILE_DIM+i] * b[i*N+col];
  }
  c[row*N+col] = sum;
}
```

- Note that every thread in the warp executes $i = 0, 1, \ldots$ in order.
- Each thread reads same element of `a` at the same time since `row*TILE_DIM` is same on all threads in half-warp.
- This is coalesced access, but wasteful.
- Each thread reads sequential elements from `b`
- This is coalesced access, and uses full bandwidth.

Naïve approach

Different colours correspond to different `i`.



- Bandwidth: 12.5 GBps (Tesla K20c - peak 147 GBps - CC 3.5)
- Bandwidth: 5.5 GBps (Quadro K620 - peak 27 GBps - CC 5.0)

Using shared memory

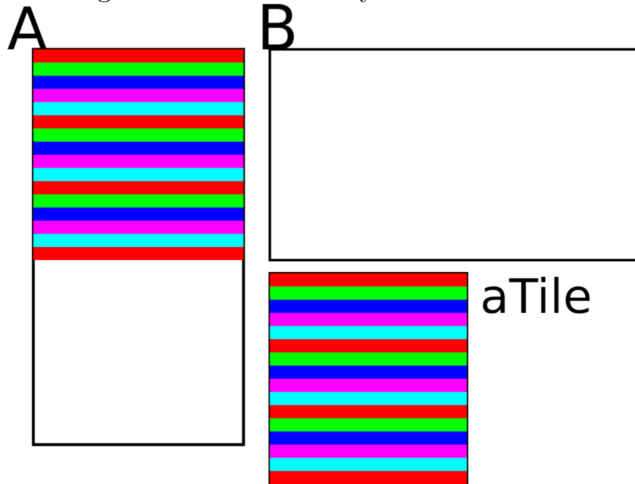
```
// Taken from CUDA Best-Practices Guide Listing 3.8
__global__ void coalescedMultiply(float *a, float* b, float
    *c, int N){
  __shared__ float aTile[TILE_DIM][TILE_DIM];

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  float sum = 0.0f;

  aTile[threadIdx.y][threadIdx.x] =
    a[row*TILE_DIM+threadIdx.x];

  for (int i = 0; i < TILE_DIM; i++) {
    sum += aTile[threadIdx.y][i]* b[i*N+col];
  }

  c[row*N+col] = sum;
}
```

Using shared memory

- A tile of `a` is read into shared memory using coalesced access
- No `__syncthreads` call needed since threads in same warp run in step.
- Threads then read from shared memory - much quicker than global

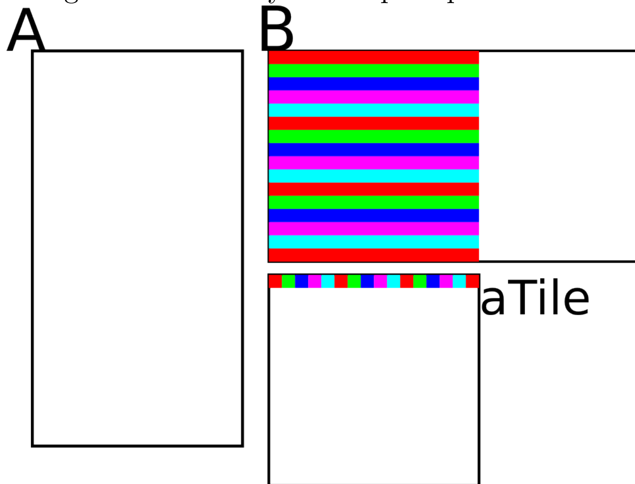
Using shared memory

Reading into shared memory.



Using shared memory

Using shared memory to compute product.



- Bandwidth: 18.89 GBps (Tesla K20c)
- Bandwidth: 11.1 GBps (Quadro K620)

Using more shared memory

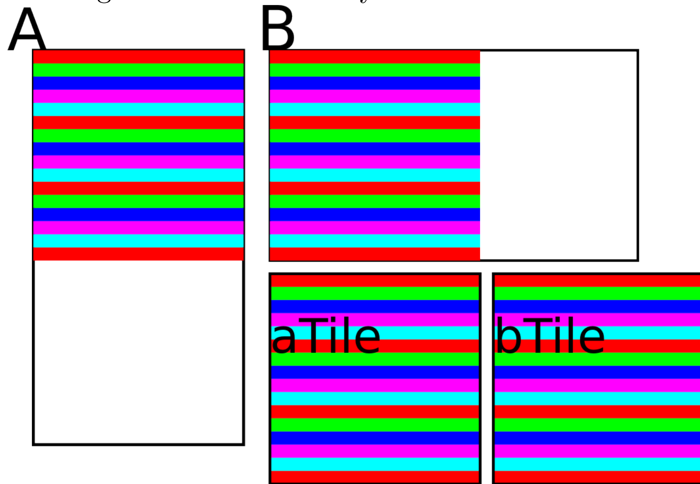
```
// Taken from CUDA Best-Practices Guide Listing 3.9
__global__ void sharedABMultiply(float *a, float* b, float *c,
                                 int N){
  __shared__ float aTile[TILE_DIM][TILE_DIM],
                   bTile[TILE_DIM][TILE_DIM];

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  float sum = 0.0f;

  aTile[threadIdx.y][threadIdx.x] =
    a[row*TILE_DIM+threadIdx.x];
  bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
  __syncthreads();

  for (int i = 0; i < TILE_DIM; i++) {
    sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
  }

  c[row*N+col] = sum;
}
```

Using more shared memory

- Also read a tile of `b` into shared memory.
- Need `__syncthreads` call since threads from different warps need access to each row of `bTile`
- Global write to `c[row*N+col]` is coalesced anyway.

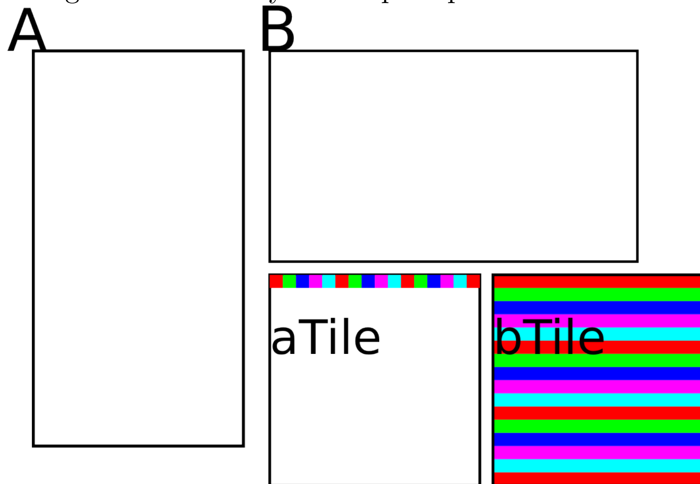
Using more shared memory ctd

Reading into shared memory



Using more shared memory ctd

Using shared memory to compute product.



- Bandwidth: 28.3 Gbps (Tesla K20c)
- Bandwidth: 12.9 GBps (Quadro K620)

Shared memory and caches

- The preceding approaches, while useful for demonstrating shared-memory usage, are not as useful as they once were.
- About 7 years ago I used to see a factor of 8 performance improvement just by using the second form of the function, and a further factor 3 for the third.
- Global memory reads are now cached on the streaming-multiprocessor, without input from the programmer.
- Use of shared-memory is potentially still useful for more complex global memory reads/writes, though.

Branching

- Where possible, threads in a warp execute in step
- If some threads branch one way and some another, then instructions are executed serially in sets.
- `if( threadIdx.x % 2 == 0 )` will cause divergent branching while some threads execute the `if` block,
- `if( blockIdx.x % 2 == 0 )` will not cause divergent branching because all threads in the warp (and block) will branch the same way.
- The first case will cause slowdown since some threads in the warp are paused while the other threads execute.
- This is the price we pay for higher FLOPS.
- It is best to avoid divergent branching if at all possible.