Lecture 1: an introduction to CUDA

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

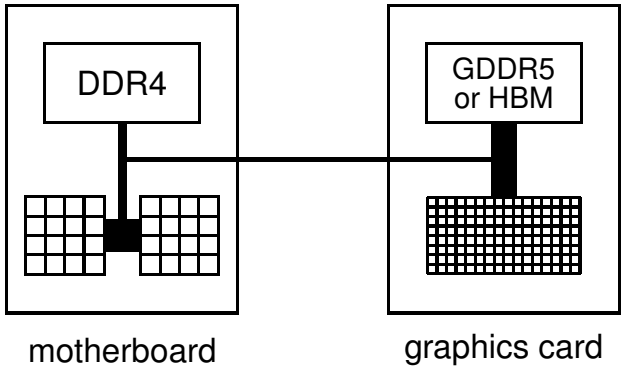
Oxford e-Research Centre

Overview

- hardware view
- software view
- CUDA programming

Hardware view

At the top-level, a PCIe graphics card with a many-core GPU and high-speed graphics "device" memory sits inside a standard PC/server with one or two multicore CPUs:



Hardware view

Currently, 4 generations of hardware cards in use:

- Kepler (compute capability 3.x):
  - first released in 2012, including HPC cards with excellent DP
  - our practicals will use K40s and K80s
- Maxwell (compute capability 5.x):
  - first released in 2014; only gaming cards, so poor DP
- Pascal (compute capability 6.x):
  - first released in 2016
  - many gaming cards and several HPC cards in Oxford
- Volta (compute capability 7.x):
  - first released in 2018; only HPC cards so far

Hardware view

The Pascal generation has cards for both gaming/VR and HPC

Consumer graphics cards (GeForce):

- GTX 1060: 1280 cores, 6GB (£230)
- GTX 1070: 1920 cores, 8GB (£380)
- GTX 1080: 2560 cores, 8GB (£480)
- GTX 1080 Ti: 3584 cores, 11GB (£650)

HPC (Tesla):

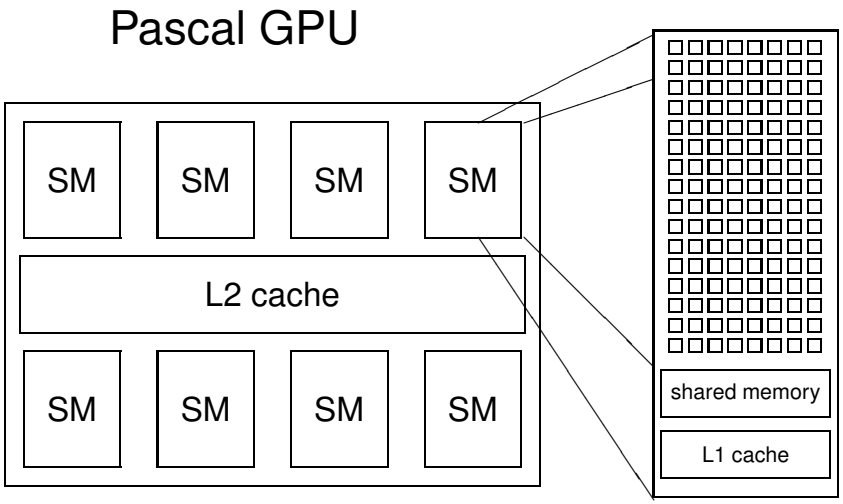
- P100 (PCIe): 3584 cores, 12GB HBM2 (£5k)
- P100 (PCIe): 3584 cores, 16GB HBM2 (£6k)
- P100 (NVlink): 3584 cores, 16GB HBM2 (£8k?)

Hardware view

- building block is a "streaming multiprocessor" (SM):
  - 128 cores (64 in P100) and 64k registers
  - 96KB (64KB in P100) of shared memory
  - 48KB (24KB in P100) L1 cache
  - 8-16KB (?) cache for constants
  - up to 2K threads per SM
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX 1060	10	192 GB/s	6 GB	120W
GTX 1070	16	256 GB/s	8 GB	150W
GTX 1080	20	320 GB/s	8 GB	180W
GTX Titan X	28	480 GB/s	12 GB	250W
P100	56	720 GB/s	16 GB HBM2	300W

Hardware View



Hardware view

There were multiple products in the Kepler generation

Consumer graphics cards (GeForce):

- GTX Titan Black: 2880 cores, 6GB
- GTX Titan Z: 2×2880 cores, 2×6GB

HPC cards (Tesla):

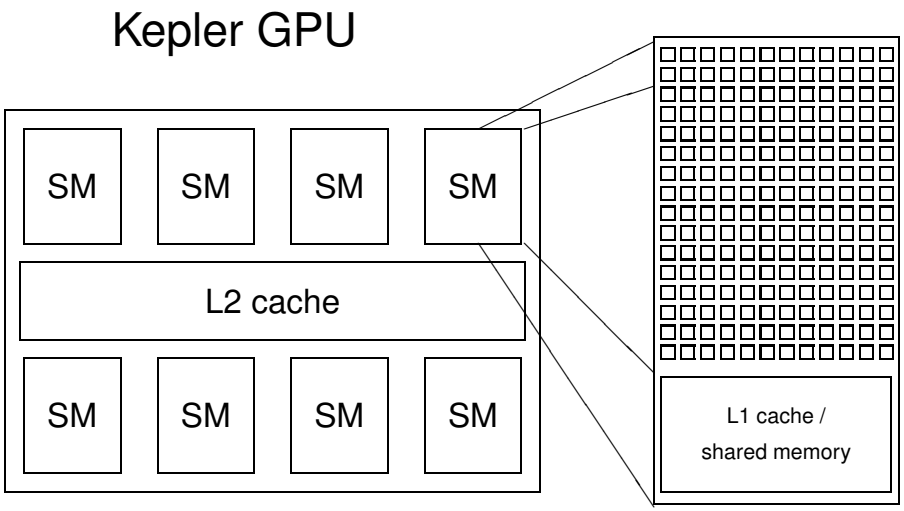
- K20: 2496 cores, 5GB
- K40: 2880 cores, 12GB
- K80: 2×2496 cores, 2×12GB

Hardware view

- building block is a “streaming multiprocessor” (SM):
  - 192 cores and 64k registers
  - 64KB of shared memory / L1 cache
  - 8KB cache for constants
  - 48KB texture cache for read-only arrays
  - up to 2K threads per SM
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX Titan Z	2×15	2×336 GB/s	2×6 GB	375W
K40	15	288 GB/s	12 GB	245W
K80	2×14	2×240 GB/s	2×12 GB	300W

Hardware View



Multithreading

Key hardware feature is that the cores in a SM are SIMT (Single Instruction Multiple Threads) cores:

- groups of 32 cores execute the same instructions simultaneously, but with different data
- similar to vector computing on CRAY supercomputers
- 32 threads all doing the same thing at the same time
- natural for graphics processing and much scientific computing
- SIMT is also a natural choice for many-core chips to simplify each core

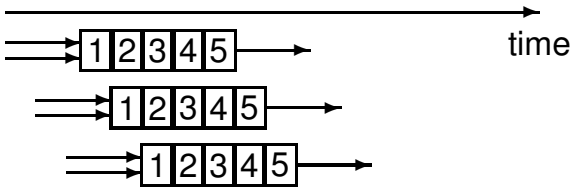
Multithreading

Lots of active threads is the key to high performance:

- no “context switching”; each thread has its own registers, which limits the number of active threads
- threads on each SM execute in groups of 32 called “warps” – execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

Multithreading

- originally, each thread completed one operation before the next started to avoid complexity of pipeline overlaps



  however, NVIDIA have now relaxed this, so each thread can have multiple independent instructions overlapping

- memory access from device memory has a delay of 200-400 cycles; with 40 active warps this is equivalent to 5-10 operations, so enough to hide the latency?

Software view

At the top level, we have a master process which runs on the CPU and performs the following steps:

1. initialises card
2. allocates memory in host and on device
3. copies data from host to device memory
4. launches multiple instances of execution “kernel” on device
5. copies data from device memory to host
6. repeats 3-5 as needed
7. de-allocates all memory and terminates

Software view

At a lower level, within the GPU:

- each instance of the execution kernel executes on a SM
- if the number of instances exceeds the number of SMs, then more than one will run at a time on each SM if there are enough registers and shared memory, and the others will wait in a queue and execute later
- all threads within one instance can access local shared memory but can’t see what the other instances are doing (even if they are on the same SM)
- there are no guarantees on the order in which the instances execute

CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA’s program development environment:

- based on C/C++ with some extensions
- FORTRAN support provided by compiler from PGI (owned by NVIDIA) and also in IBM XL compiler
- lots of example code and good documentation – fairly short learning curve for those with experience of OpenMP and MPI programming
- large user community on NVIDIA forums

CUDA Components

Installing CUDA on a system, there are 3 components:

- driver
  - low-level software that controls the graphics card
- toolkit
  - `nvcc` CUDA compiler
  - Nsight IDE plugin for Eclipse or Visual Studio
  - profiling and debugging tools
  - several libraries
- SDK
  - lots of demonstration examples
  - some error-checking utilities
  - not officially supported by NVIDIA
  - almost no documentation

CUDA programming

Already explained that a CUDA program has two pieces:

- host code on the CPU which interfaces to the GPU
- kernel code which runs on the GPU

At the host level, there is a choice of 2 APIs (Application Programming Interfaces):

- runtime
  - simpler, more convenient
- driver
  - much more verbose, more flexible (e.g. allows run-time compilation), closer to OpenCL

We will only use the runtime API in this course, and that is all I use in my own research.

CUDA programming

At the host code level, there are library routines for:

- memory allocation on graphics card
- data transfer to/from device memory
  - constants
  - ordinary data
- error-checking
- timing

There is also a special syntax for launching multiple instances of the kernel process on the GPU.

CUDA programming

In its simplest form it looks like:

```
kernel_routine<<<gridDim, blockDim>>>(args);
```

- `gridDim` is the number of instances of the kernel (the “grid” size)
- `blockDim` is the number of threads within each instance (the “block” size)
- `args` is a limited number of arguments, usually mainly pointers to arrays in graphics memory, and some constants which get copied by value

The more general form allows `gridDim` and `blockDim` to be 2D or 3D to simplify application programs

CUDA programming

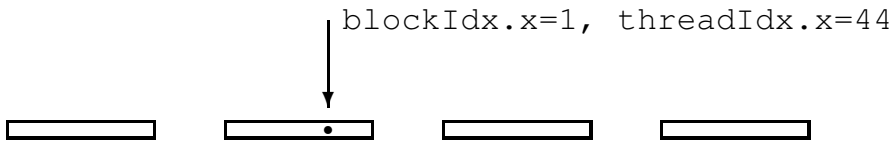
At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- some variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
  - `gridDim` size (or dimensions) of grid of blocks
  - `blockDim` size (or dimensions) of each block
  - `blockIdx` index (or 2D/3D indices) of block
  - `threadIdx` index (or 2D/3D indices) of thread
  - `warpSize` always 32 so far, but could change

CUDA programming

1D grid with 4 blocks, each with 64 threads:

- `gridDim = 4`
- `blockDim = 64`
- `blockIdx` ranges from `0` to `3`
- `threadIdx` ranges from `0` to `63`



CUDA programming

The kernel code looks fairly normal once you get used to two things:

- code is written from the point of view of a single thread
  - quite different to OpenMP multithreading
  - similar to MPI, where you use the MPI “rank” to identify the MPI process
  - all local variables are private to that thread
- need to think about where each variable lives (more on this in the next lecture)
  - any operation involving data in the device memory forces its transfer to/from registers in the GPU
  - often better to copy the value into a local register variable

Host code

```
int main(int argc, char **argv) {
 float *h_x, *d_x;          //  h=host, d=device
 int   nblocks=2, nthreads=8, nsize=2*8;

 h_x = (float *)malloc(nsize*sizeof(float));
 cudaMalloc((void **)&d_x,nsize*sizeof(float));

 my_first_kernel<<<nblocks,nthreads>>>(d_x);

 cudaMemcpy(h_x,d_x,nsize*sizeof(float),
            cudaMemcpyDeviceToHost);

 for (int n=0; n<nsize; n++)
   printf(" n,  x  =  %d  %f \n",n,h_x[n]);

 cudaFree(d_x); free(h_x);
}
```

Kernel code

```
#include <helper_cuda.h>

__global__ void my_first_kernel(float *x)
{
 int tid = threadIdx.x + blockDim.x*blockIdx.x;

 x[tid] = (float) threadIdx.x;
}
```

- `__global__` identifier says it's a kernel function
- each thread sets one element of `x` array
- within each block of threads, `threadIdx.x` ranges from 0 to `blockDim.x-1`, so each thread has a unique value for `tid`

CUDA programming

Suppose we have 1000 blocks, and each one has 128 threads – how does it get executed?

On Kepler hardware, would probably get 8-12 blocks running at the same time on each SM, and each block has 4 warps $\Longrightarrow$ 32-48 warps running on each SM

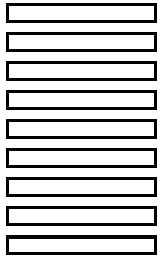
Each clock tick, SM warp scheduler decides which warps to execute next, choosing from those not waiting for

- data coming from device memory (memory latency)
- completion of earlier instructions (pipeline delay)

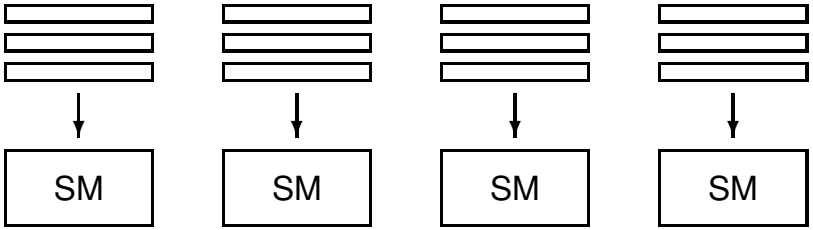
Programmer doesn't have to worry about this level of detail, just make sure there are lots of threads / warps

CUDA programming

Queue of waiting blocks:



Multiple blocks running on each SM:



CUDA programming

In this simple case, we had a 1D grid of blocks, and a 1D set of threads within each block.

If we want to use a 2D set of threads, then `blockDim.x, blockDim.y` give the dimensions, and `threadIdx.x, threadIdx.y` give the thread indices

and to launch the kernel we would use something like

```
dim3 nthreads(16,4);
my_new_kernel<<<nblocks,nthreads>>>(d_x);
```

where `dim3` is a special CUDA datatype with 3 components `.x,.y,.z` each initialised to 1.

CUDA programming

A similar approach is used for 3D threads and 2D / 3D grids; can be very useful in 2D / 3D finite difference applications.

How do 2D / 3D threads get divided into warps?

1D thread ID defined by

```
threadIdx.x +
threadIdx.y * blockDim.x +
threadIdx.z * blockDim.x * blockDim.y
```

and this is then broken up into warps of size 32.

Practical 1

- start from code shown above (but with comments)
- learn how to compile / run code within Nsight IDE (integrated into Visual Studio for Windows, or Eclipse for Linux)
- test error-checking and printing from kernel functions
- modify code to add two vectors together (including sending them over from the host to the device)
- if time permits, look at CUDA SDK examples

Practical 1

Things to note:

- memory allocation
  `cudaMalloc((void **)&d_x, nbytes);`
- data copying
  ```
  cudaMemcpy(h_x,d_x,nbytes,
             cudaMemcpyDeviceToHost);
  ```
- reminder: prefix `h_` and `d_` to distinguish between arrays on the host and on the device is not mandatory, just helpful labelling
- kernel routine is declared by `__global__` prefix, and is written from point of view of a single thread

Practical 1

Second version of the code is very similar to first, but uses an SDK header file for various safety checks – gives useful feedback in the event of errors.

- check for error return codes:
  `checkCudaErrors( ... );`
- check for kernel failure messages:
  `getLastCudaError( ... );`

Practical 1

One thing to experiment with is the use of `printf` within a CUDA kernel function:

- essentially the same as standard `printf`; minor difference in integer return code
- each thread generates its own output; use conditional code if you want output from only one thread
- output goes into an output buffer which is transferred to the host and printed later (possibly much later?)
- buffer has limited size (1MB by default), so could lose some output if there's too much
- need to use either `cudaDeviceSynchronize();` or `cudaDeviceReset();` at the end of the main code to make sure the buffer is flushed before termination

Practical 1

The practical also has a third version of the code which uses "managed memory" based on Unified Memory.

In this version

- there is only one array / pointer, not one for CPU and another for GPU
- the programmer is not responsible for moving the data to/from the GPU
- everything is handled automatically by the CUDA run-time system

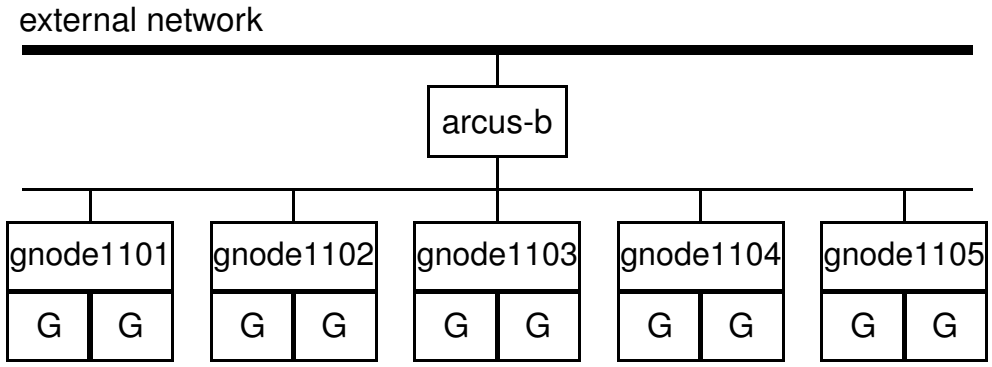
Practical 1

This leads to simpler code, but it's important to understand what is happening because it may hurt performance:

- if the CPU initialises an array x, and then a kernel uses it, this forces a copy from CPU to GPU
- if the GPU modifies x and the CPU later tries to read from it, that triggers a copy back from GPU to CPU

Personally, I prefer to keep complete control over data movement, so that I know what is happening and I can maximise performance.

ARCUS-B cluster



- `arcus-b.arc.ox.ac.uk` is the head node
- the GPU compute nodes have two K80 cards with a total of 4 GPUs, numbered 0 – 3
- read the Arcus notes before starting the practical

Key reading

CUDA Programming Guide, version 8.0:

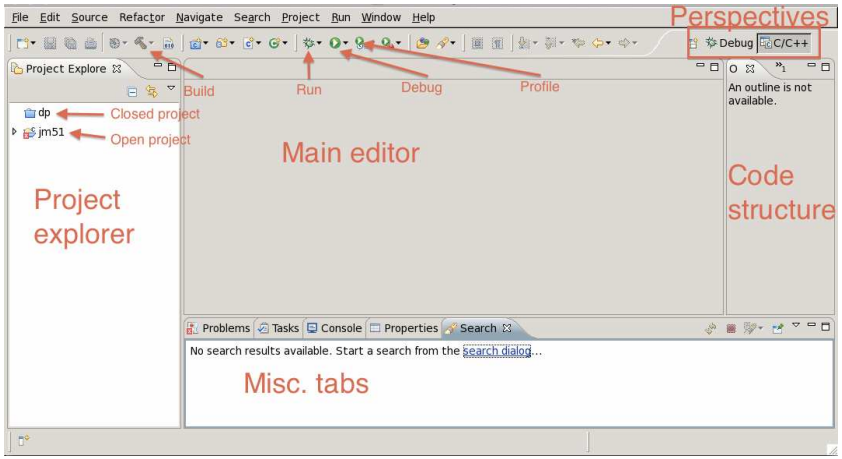
- Chapter 1: Introduction
- Chapter 2: Programming Model
- Section 5.4: performance of different GPUs
- Appendix A: CUDA-enabled GPUs
- Appendix B, sections B.1 – B.4: C language extensions
- Appendix B, section B.17: `printf` output
- Appendix G, section G.1: features of different GPUs

Wikipedia (clearest overview of NVIDIA products):

- `en.wikipedia.org/wiki/Nvidia_Tesla`
- `en.wikipedia.org/wiki/GeForce_10_series`

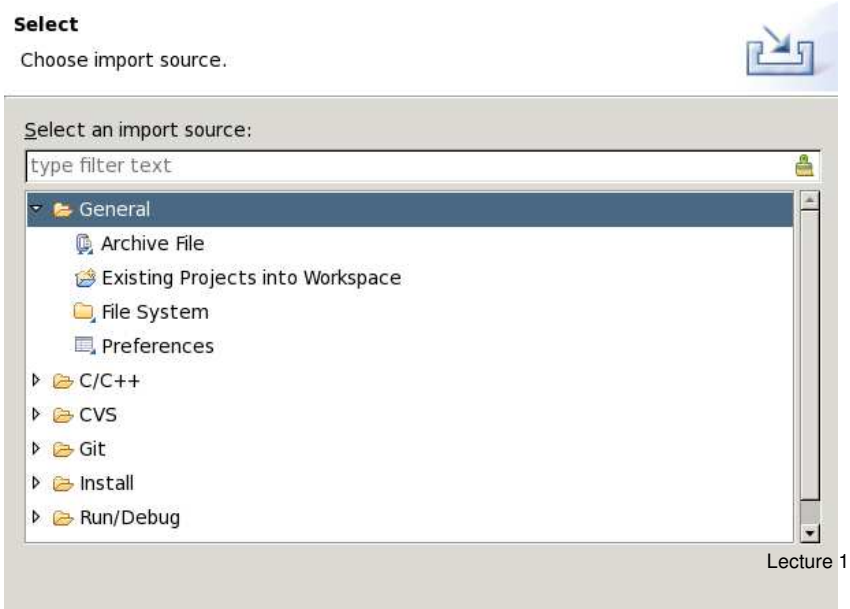
Nsight

General view:

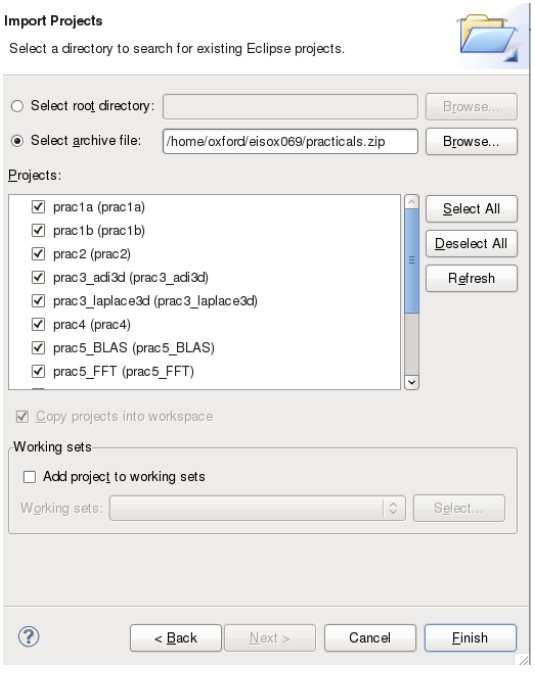


Nsight

Importing the practicals: select General – Existing Projects



Nsight



Lecture 2: different memory and variable types

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

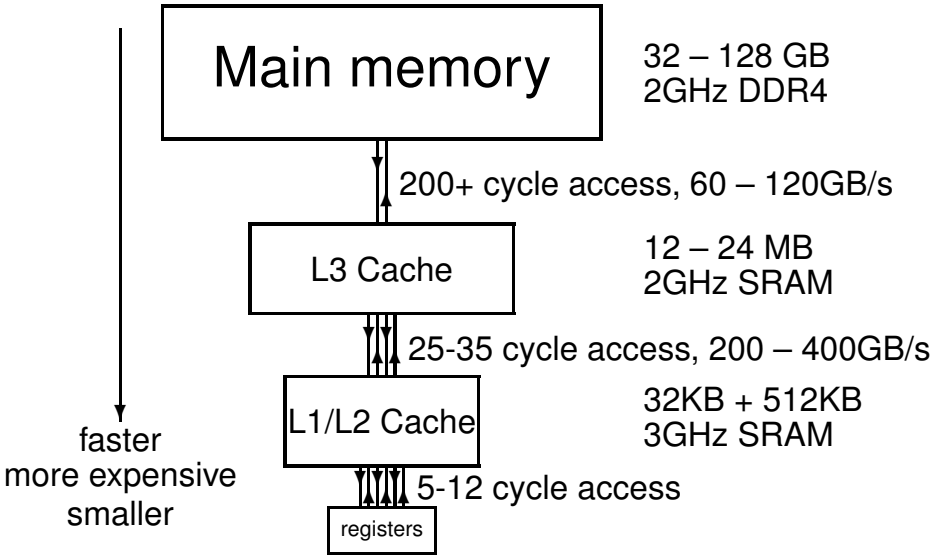
Oxford e-Research Centre

Memory

Key challenge in modern computer architecture

- no point in blindingly fast computation if data can't be moved in and out fast enough
- need lots of memory for big applications
- very fast memory is also very expensive
- end up being pushed towards a hierarchical design

CPU Memory Hierarchy



Memory Hierarchy

Execution speed relies on exploiting data *locality*

- temporal locality: a data item just accessed is likely to be used again in the near future, so keep it in the cache
- spatial locality: neighbouring data is also likely to be used soon, so load them into the cache at the same time using a 'wide' bus (like a multi-lane motorway)

This wide bus is only way to get high bandwidth to slow main memory

Caches

The cache line is the basic unit of data transfer; typical size is 64 bytes $\equiv 8\times$ 8-byte items.

With a single cache, when the CPU loads data into a register:

- it looks for line in cache
- if there (hit), it gets data
- if not (miss), it gets entire line from main memory, displacing an existing line in cache (usually least recently used)

When the CPU stores data from a register:

- same procedure

Importance of Locality

Typical workstation:

20 Gflops per core

40 GB/s L3 $\longleftrightarrow$ L2 cache bandwidth

64 bytes/line

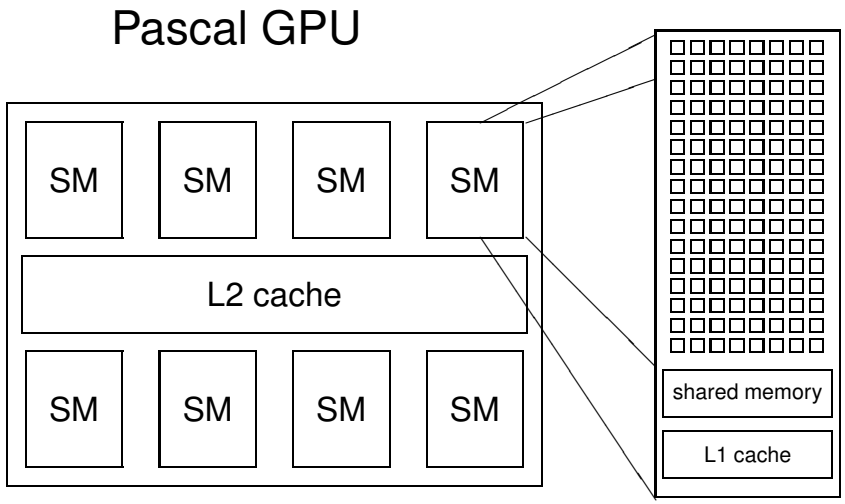
40GB/s $\equiv$ 600M line/s $\equiv$ 5G double/s

At worst, each flop requires 2 inputs and has 1 output, forcing loading of 3 lines $\Longrightarrow$ 200 Mflops

If all 8 variables/line are used, then this increases to 1.6 Gflops.

To get up to 20Gflops needs temporal locality, re-using data already in the L2 cache.

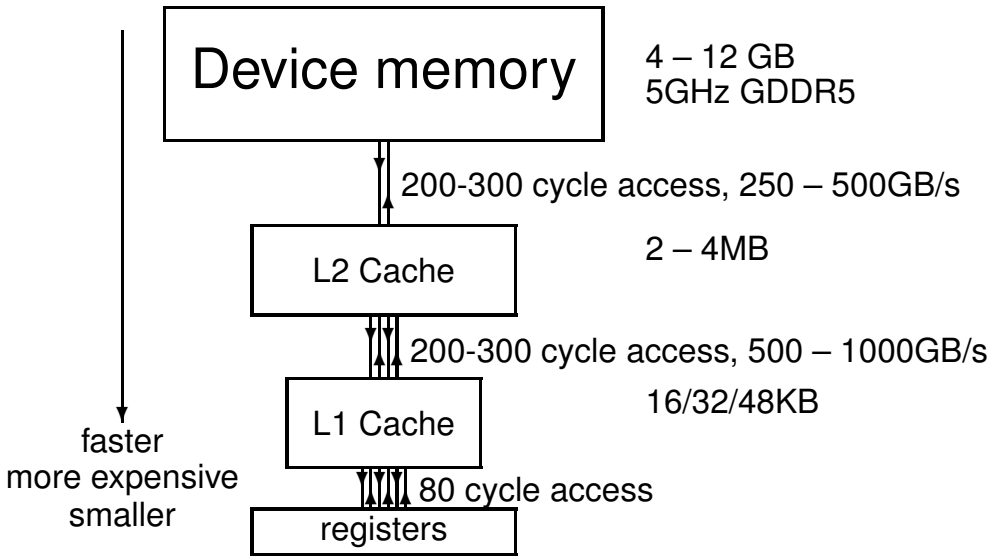
Pascal



Pascal

- usually 32 bytes cache line (8 floats or 4 doubles)
- P100: 4092-bit memory path from HBM2 device memory to L2 cache $\Longrightarrow$ up to 720 GB/s bandwidth
- GeForce cards: 384-bit memory bus from GDDR5 device memory to L2 cache $\Longrightarrow$ up to 320 GB/s
- unified 4MB L2 cache for all SM's
- each SM has 64-96kB of shared memory, and 24-48kB of L1 cache
- no global cache coherency as in CPUs, so should (almost) never have different blocks updating the same global array elements

GPU Memory Hierarchy



Importance of Locality

5Tflops GPU

320 GB/s memory $\longleftrightarrow$ L2 cache bandwidth

32 bytes/line

320GB/s $\equiv$ 10G line/s $\equiv$ 40G double/s

At worst, each flop requires 2 inputs and has 1 output, forcing loading of 3 lines $\Longrightarrow$ 3 Gflops

If all 4 doubles/line are used, increases to 13 Gflops

To get up to 2TFlops needs about 50 flops per double transferred to/from device memory

Even with careful implementation, many algorithms are bandwidth-limited not compute-bound

Practical 1 kernel

```
__global__ void my_first_kernel(float *x)
{
 int tid = threadIdx.x + blockDim.x*blockIdx.x;

 x[tid] = threadIdx.x;
}
```

- 32 threads in a warp will address neighbouring elements of array `x`
- if the data is correctly "aligned" so that `x[0]` is at the beginning of a cache line, then `x[0]` – `x[31]` will be in same cache line – a "coalesced" transfer
- hence we get perfect spatial locality

A bad kernel

```
__global__ void bad_kernel(float *x)
{
 int tid = threadIdx.x + blockDim.x*blockIdx.x;

 x[1000*tid] = threadIdx.x;
}
```

- in this case, different threads within a warp access widely spaced elements of array `x` – a "strided" array access
- each access involves a different cache line, so performance will be awful

Global arrays

So far, concentrated on global / device arrays:

- held in the large device memory
- allocated by host code
- pointers held by host code and passed into kernels
- continue to exist until freed by host code
- since blocks execute in an arbitrary order, if one block modifies an array element, no other block should read or write that same element

Global variables

Global variables can also be created by declarations with global scope within kernel code file

```
__device__ int reduction_lock=0;

__global__ void kernel_1(...) {
  ...
}

__global__ void kernel_2(...) {
  ...
}
```


Global variables

- the `__device__` prefix tells `nvcc` this is a global variable in the GPU, not the CPU.
- the variable can be read and modified by any kernel
- its lifetime is the lifetime of the whole application
- can also declare arrays of fixed size
- can read/write by host code using special routines `cudaMemcpyToSymbol, cudaMemcpyFromSymbol` or with standard `cudaMemcpy` in combination with `cudaGetSymbolAddress`
- in my own CUDA programming, I rarely use this capability but it is occasionally very useful

Constant variables

Very similar to global variables, except that they can't be modified by kernels:

- defined with global scope within the kernel file using the prefix `__constant__`
- initialised by the host code using `cudaMemcpyToSymbol, cudaMemcpyFromSymbol` or `cudaMemcpy` in combination with `cudaGetSymbolAddress`
- I use it all the time in my applications; practical 2 has an example

Constant variables

Only 64KB of constant memory, but big benefit is that each SM has a 10KB cache

- when all threads read the same constant, almost as fast as a register
- doesn't tie up a register, so very helpful in minimising the total number of registers required

Constants

A constant variable has its value set at run-time

But code also often has plain constants whose value is known at compile-time:

```
#define PI 3.1415926f

  a = b / (2.0f * PI);
```

Leave these as they are – they seem to be embedded into the executable code so they don't use up any registers

Don't forget the `f` at the end if you want single precision; in C/C++

$$\text{single} \times \text{double} = \text{double}$$

Registers

Within each kernel, by default, individual variables are assigned to registers:

```
__global__ void lap(int I, int J,
                    float *u1, float *u2) {
  int i  = threadIdx.x + blockIdx.x*blockDim.x;
  int j  = threadIdx.y + blockIdx.y*blockDim.y;
  int id = i + j*I;

  if (i==0 || i==I-1 || j==0 || j==J-1) {
    u2[id] = u1[id];   // Dirichlet b.c.'s
  }
  else {
    u2[id] = 0.25f * ( u1[id-1] + u1[id+1]
                     + u1[id-I] + u1[id+I] );
  }
}
```


Registers

- 64K 32-bit registers per SM
- up to 255 registers per thread
- up to 2048 threads (at most 1024 per thread block)
- max registers per thread $\Longrightarrow$ 256 threads
- max threads $\Longrightarrow$ 32 registers per thread
- big difference between "fat" and "thin" threads

Registers

What happens if your application needs more registers?

They "spill" over into L1 cache, and from there to device memory – precise mechanism unclear, but

either certain variables become device arrays with one element per thread

or the contents of some registers get "saved" to device memory so they can used for other purposes, then the data gets "restored" later

Either way, the application suffers from the latency and bandwidth implications of using device memory

Local arrays

What happens if your application uses a little array?

```
__global__ void lap(float *u) {

float ut[3];

int tid = threadIdx.x + blockIdx.x*blockDim.x;

for (int k=0; k<3; k++)
  ut[k] = u[tid+k*gridDim.x*blockDim.x];

for (int k=0; k<3; k++)
  u[tid+k*gridDim.x*blockDim.x] =
    A[3*k]*ut[0]+A[3*k+1]*ut[1]+A[3*k+2]*ut[2];
}
```


Local arrays

In simple cases like this (quite common) compiler converts to scalar registers:

```
__global__ void lap(float *u) {
int tid = threadIdx.x + blockIdx.x*blockDim.x;
float ut0 = u[tid+0*gridDim.x*blockDim.x];
float ut1 = u[tid+1*gridDim.x*blockDim.x];
float ut2 = u[tid+2*gridDim.x*blockDim.x];

u[tid+0*gridDim.x*blockDim.x] =
        A[0]*ut0 + A[1]*ut1 + A[2]*ut2;
u[tid+1*gridDim.x*blockDim.x] =
        A[3]*ut0 + A[4]*ut1 + A[5]*ut2;
u[tid+2*gridDim.x*blockDim.x] =
        A[6]*ut0 + A[7]*ut1 + A[8]*ut2;
}
```


Local arrays

In more complicated cases, it puts the array into device memory

- this is because registers are not dynamically addressable – compiler has to specify exactly which registers are used for each instruction
- still referred to in the documentation as a "local array" because each thread has its own private copy
- held in L1 cache by default, may never be transferred to device memory
- 48kB of L1 cache equates to 12k 32-bit variables, which is only 12 per thread when using 1024 threads
- beyond this, it will have to spill to device memory

Shared memory

In a kernel, the prefix `__shared__` as in

```
__shared__ int   x_dim;
__shared__ float x[128];
```

declares data to be shared between all of the threads in the thread block – any thread can set its value, or read it.

There can be several benefits:

- essential for operations requiring communication between threads (e.g. summation in lecture 4)
- useful for data re-use
- alternative to local arrays in device memory

Shared memory

If a thread block has more than one warp, it's not pre-determined when each warp will execute its instructions – warp 1 could be many instructions ahead of warp 2, or well behind.

Consequently, almost always need thread synchronisation to ensure correct use of shared memory.

Instruction

```
__syncthreads();
```

inserts a "barrier"; no thread/warp is allowed to proceed beyond this point until the rest have reached it (like a roll call on a school outing)

Shared memory

So far, have discussed statically-allocated shared memory – the size is known at compile-time

Can also create dynamic shared-memory arrays but this is more complex

Total size is specified by an optional third argument when launching the kernel:

```
kernel<<<blocks,threads,shared_bytes>>>(...)
```

Using this within the kernel function is complicated/tedious; see B.2.3 in Programming Guide

Read-only arrays

With "constant" variables, each thread reads the same value.

In other cases, we have arrays where the data doesn't change, but different threads read different items.

In this case, can get improved performance by telling the compiler by declaring global array with

```
const __restrict__
```

qualifiers so that the compiler knows that it is read-only

Non-blocking loads/stores

What happens with the following code?

```
__global__ void lap(float *u1, float *u2) {
  float a;

  a = u1[threadIdx.x + blockIdx.x*blockDim.x]
  ...
  ...
  c = b*a;
  u2[threadIdx.x + blockIdx.x*blockDim.x] = c;
  ...
  ...
}
```

Load doesn't block until needed; store also doesn't block

Active blocks per SM

Each block require certain resources:

- threads
- registers (registers per thread $\times$ number of threads)
- shared memory (static $+$ dynamic)

Together these determine how many blocks can be run simultaneously on each SM – up to a maximum of 32 blocks

Active blocks per SM

My general advice:

- number of active threads depends on number of registers each needs
- good to have at least 4 active blocks, each with at least 128 threads
- smaller number of blocks when each needs lots of shared memory
- larger number of blocks when they don't need shared memory

Active blocks per SM

On Pascal:

- maybe 4 big blocks (512 threads) if each needs a lot of shared memory
- maybe 12 small blocks (128 threads) if no shared memory needed
- or 4 small blocks (128 threads) if each thread needs lots of registers

Very important to experiment with different block sizes to find what gives the best performance.

Summary

- dynamic device arrays
- static device variables / arrays
- constant variables / arrays
- registers
- spilled registers
- local arrays
- shared variables / arrays

Key reading

CUDA Programming Guide, version 9.0:

- Appendix B.1-B.4 – essential
- Chapter 3, sections 3.2.1-3.2.3

Other reading:

- Wikipedia article on caches:
  `en.wikipedia.org/wiki/CPU_cache`
- web article on caches:
  `lwn.net/Articles/252125/`
- "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System":
  `portal.acm.org/citation.cfm?id=1637764`

Lecture 3: control flow and synchronisation

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Warp divergence

Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time.

What happens if different threads in a warp need to do different things?

```
if (x<0.0)
  z = x-2.0;
else
  z = sqrt(x);
```

This is called *warp divergence* – CUDA will generate correct code to handle this, but to understand the performance you need to understand what CUDA does with it

Warp divergence

This is not a new problem.

Old CRAY vector supercomputers had a logical merge vector instruction

```
z = p ? x : y;
```

which stored the relevant element of the input vectors `x,y` depending on the logical vector `p`

```
for(i=0; i<I; i++) {
  if (p[i]) z[i] = x[i];
  else      z[i] = y[i];
}
```


Warp divergence

Similarly, NVIDIA GPUs have *predicated* instructions which are carried out only if a logical flag is true.

```
 p:  a = b + c;  // computed only if p is true
```

In the previous example, all threads compute the logical predicate and two predicated instructions

```
     p = (x<0.0);
 p:  z = x-2.0;      // single instruction
!p:  z = sqrt(x);
```

Warp divergence

Note that:

- `sqrt(x)` would usually produce a `NaN` when `x<0`, but it's not really executed when `x<0` so there's no problem
- all threads execute both conditional branches, so execution cost is sum of both branches
  $\Longrightarrow$ potentially large loss of performance

Warp divergence

Another example:

```
if (n>=0)
  z = x[n];
else
  z = 0;
```

- `x[n]` is only read here if `n>=0`
- don't have to worry about illegal memory accesses when `n` is negative

Warp divergence

If the branches are big, `nvcc` compiler inserts code to check if all threads in the warp take the same branch (*warp voting*) and then branches accordingly.

```
    p = ...

    if (any(p)) {
 p:    ...
 p:    ...
    }

    if (any(!p)) {
!p:    ...
!p:    ...
    }
```


Warp divergence

Note:

- doesn't matter what is happening with other warps – each warp is treated separately
- if each warp only goes one way that's very efficient
- warp voting costs a few instructions, so for very simple branches the compiler just uses predication without voting

Warp divergence

In some cases, can determine at compile time that all threads in the warp must go the same way

e.g. if `case` is a run-time argument

```
if (case==1)
  z = x*x;
else
  z = x+2.3;
```

In this case, there's no need to vote

Warp divergence

Warp divergence can lead to a big loss of parallel efficiency – one of the first things I look out for in a new application.

In worst case, effectively lose factor 32$\times$ in performance if one thread needs expensive branch, while rest do nothing

Typical example: PDE application with boundary conditions

- if boundary conditions are cheap, loop over all nodes and branch as needed for boundary conditions
- if boundary conditions are expensive, use two kernels: first for interior points, second for boundary points

Warp divergence

Another example: processing a long list of elements where, depending on run-time values, a few require very expensive processing

GPU implementation:

- first process list to build two sub-lists of "simple" and "expensive" elements
- then process two sub-lists separately

Note: none of this is new – this is what we did more than 25 years ago on CRAY and Thinking Machines systems.

What's important is to understand hardware behaviour and design your algorithms / implementation accordingly

Synchronisation

Already introduced `__syncthreads();` which forms a barrier – all threads wait until every one has reached this point.

When writing conditional code, must be careful to make sure that all threads do reach the `__syncthreads();`

Otherwise, can end up in *deadlock*

Typical application

```
// load in data to shared memory
  ...
  ...
  ...

// synchronisation to ensure this has finished

  __syncthreads();

// now do computation using shared data
  ...
  ...
  ...
```


Synchronisation

There are other synchronisation instructions which are similar but have extra capabilities:

- `int __syncthreads_count(predicate)`

  counts how many predicates are true
- `int __syncthreads_and(predicate)`

  returns non-zero (true) if all predicates are true
- `int __syncthreads_or(predicate)`

  returns non-zero (true) if any predicate is true

I've not used these, and don't currently see a need for them

Warp voting

There are similar *warp voting* instructions which operate at the level of a warp:

- `int __all(predicate)`

  returns non-zero (true) if all predicates in warp are true
- `int __any(predicate)`

  returns non-zero (true) if any predicate is true
- `unsigned int __ballot(predicate)`

  sets n^{th} bit based on n^{th} predicate

Again, I've never used these

Atomic operations

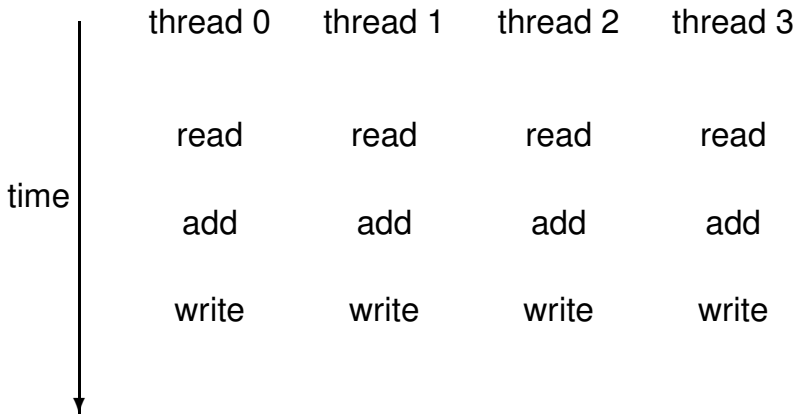
Occasionally, an application needs threads to update a counter in shared memory.

```
__shared__ int count;

...

if ( ... ) count++;
```

In this case, there is a problem if two (or more) threads try to do it at the same time

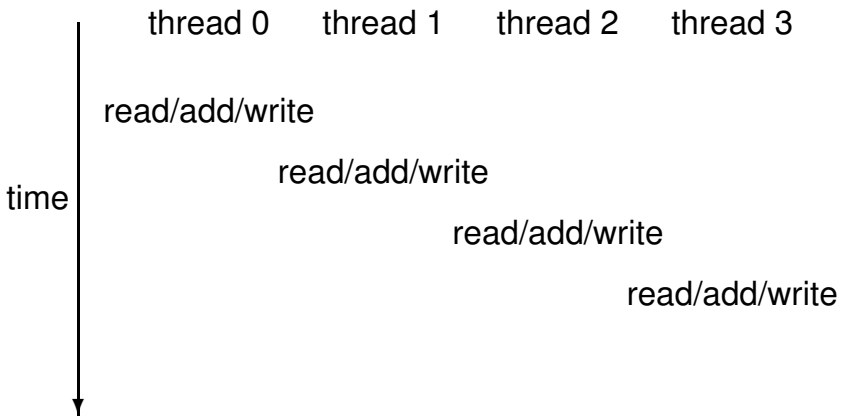
Atomic operations

Using standard instructions, multiple threads in the same warp will only update it once.



Atomic operations

With atomic instructions, the read/add/write becomes a single operation, and they happen one after the other



Atomic operations

Several different atomic operations are supported, almost all only for integers:

- addition (integers, 32-bit floats – also 64-bit in Pascal)
- minimum / maximum
- increment / decrement
- exchange / compare-and-swap
- bitwise AND / OR / XOR

These are fast for variables in shared memory, and only slightly slower for data in device global memory (operations performed in L2 cache)

Atomic operations

Compare-and-swap:

```
int atomicCAS(int* address,int compare,int val);
```

- if `compare` equals `old` value stored at `address` then `val` is stored instead
- in either case, routine returns the value of `old`
- seems a bizarre routine at first sight, but can be very useful for atomic locks
- also can be used to implement 64-bit floating point atomic addition (now available in hardware in Pascal)

Global atomic lock

```
// global variable: 0 unlocked, 1 locked
__device__ int lock=0;

__global__ void kernel(...) {
  ...

  if (threadIdx.x==0) {
    // set lock
    do {} while(atomicCAS(&lock,0,1));

    ...

    // free lock
    lock = 0;
  }
}
```


Global atomic lock

Problem: when a thread writes data to device memory the order of completion is not guaranteed, so global writes may not have completed by the time the lock is unlocked

```
__global__ void kernel(...) {
  ...

  if (threadIdx.x==0) {
    do {} while(atomicCAS(&lock,0,1));
    ...
    __threadfence(); // wait for writes to finish

    // free lock
    lock = 0;
  }
}
```


__threadfence

- `__threadfence_block();`

  wait until all global and shared memory writes are visible to
  - all threads in block

- `__threadfence();`

  wait until all global and shared memory writes are visible to
  - all threads in block
  - all threads, for global data

Atomic addition for double

```
// atomic addition from Jon Cohen at NVIDIA

static double atomicAdd(double *addr, double val)
{
  double old=*addr, assumed;

  do {
    assumed = old;
    old = __longlong_as_double(
      atomicCAS((unsigned long long int*)addr,
            __double_as_longlong(assumed),
            __double_as_longlong(val+assumed) ) );
  } while( assumed!=old );

  return old;
}
```

Summary

- lots of esoteric capabilities – don't worry about most of them
- essential to understand warp divergence – can have a very big impact on performance
- `__syncthreads()` is vital – will see another use of it in next lecture
- the rest can be ignored until you have a critical need – then read the documentation carefully and look for examples in the SDK

Key reading

CUDA Programming Guide, version 9.0:

- Section 5.4.2: control flow and predicates
- Section 5.4.3: synchronization
- Appendix B.5: `__threadfence()` and variants
- Appendix B.6: `__syncthreads()` and variants
- Appendix B.12: atomic functions
- Appendix B.13: warp voting

- Appendix C: Cooperative Groups – this is new in CUDA 9.0 and may lead to changes/updates in some of the material in this lecture

2D Laplace solver

Jacobi iteration to solve discrete Laplace equation on a uniform grid:

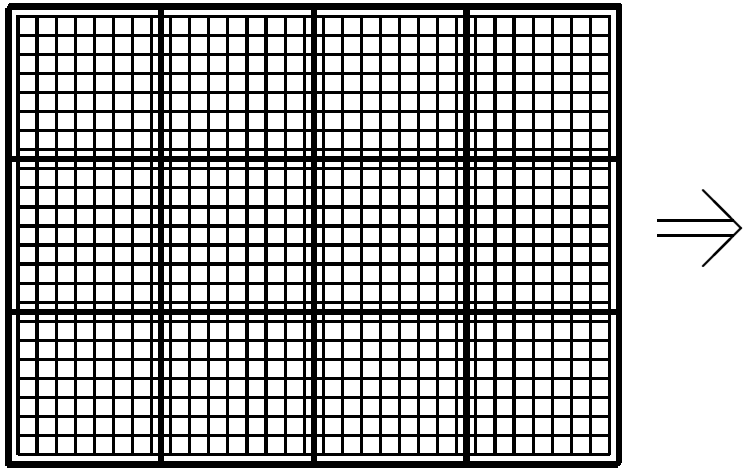
```
for (int j=0; j<J; j++) {
  for (int i=0; i<I; i++) {

    id = i + j*I;     // 1D memory location

    if (i==0 || i==I-1 || j==0 || j==J-1)
      u2[id] = u1[id];
    else
      u2[id] = 0.25*( u1[id-1] + u1[id+1]
                    + u1[id-I] + u1[id+I] );
  }
}
```


2D Laplace solver

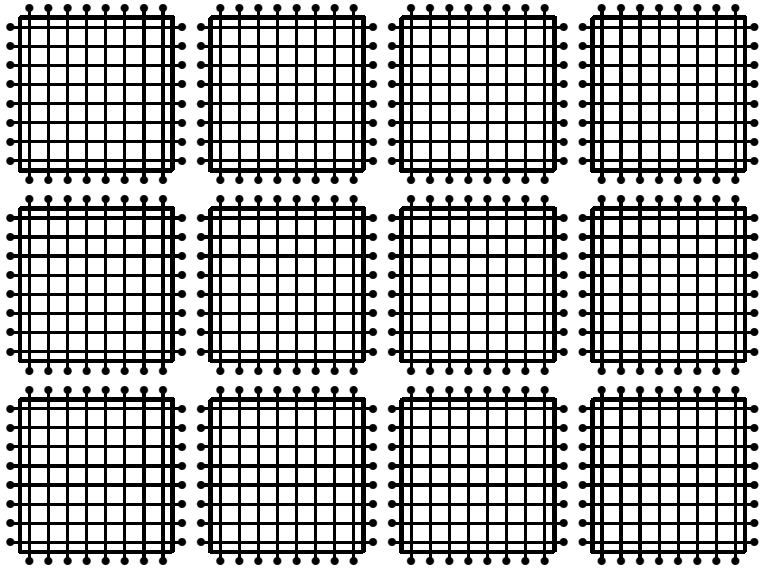
How do we tackle this with CUDA?

- each thread responsible for one grid point
- each block of threads responsible for a block of the grid
- conceptually very similar to data partitioning in MPI distributed-memory implementations, but much simpler
- (also similar to blocking techniques to squeeze the best cache performance out of CPUs)
- great example of usefulness of 2D blocks and 2D "grid"s

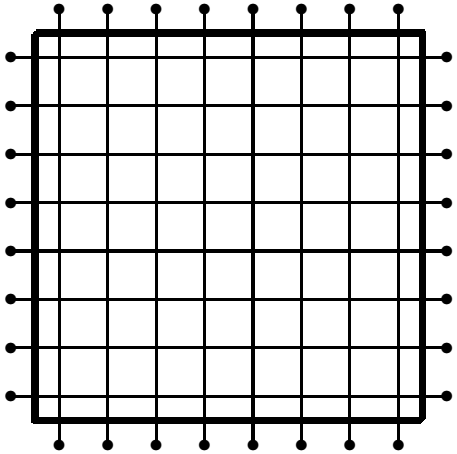
2D Laplace solver



2D Laplace solver



2D Laplace solver



Each block of threads processes one of these grid blocks, reading in old values and computing new values

2D Laplace solver

```
__global__ void lap(int I, int J,
          const float* __restrict__ u1,
                float* __restrict__ u2) {

  int i  = threadIdx.x + blockIdx.x*blockDim.x;
  int j  = threadIdx.y + blockIdx.y*blockDim.y;
  int id = i + j*I;

  if (i==0 || i==I-1 || j==0 || j==J-1) {
    u2[id] = u1[id];   // Dirichlet b.c.'s
  }
  else {
    u2[id] = 0.25 * ( u1[id-1] + u1[id+1]
                    + u1[id-I] + u1[id+I] );
  }
}
```

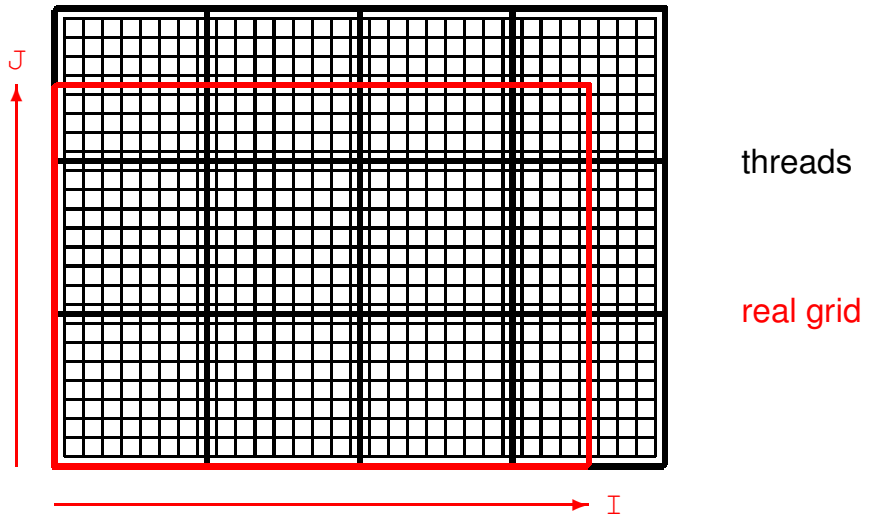
2D Laplace solver

Assumptions:

- `I` is a multiple of `blockDim.x`
- `J` is a multiple of `blockDim.y`
- hence grid breaks up perfectly into blocks

Can remove these assumptions by testing whether `i`, `j` are within grid

2D Laplace solver

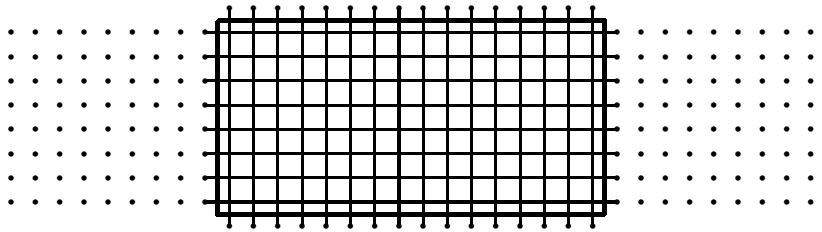


2D Laplace solver

```
__global__ void lap(int I, int J,
          const float* __restrict__ u1,
                float* __restrict__ u2) {

  int i  = threadIdx.x + blockIdx.x*blockDim.x;
  int j  = threadIdx.y + blockIdx.y*blockDim.y;
  int id = i + j*I;

  if (i==0 || i==I-1 || j==0 || j==J-1) {
    u2[id] = u1[id];   // Dirichlet b.c.'s
  }
  else if (i<I && j<J) {
    u2[id] = 0.25f * ( u1[id-1] + u1[id+1]
                     + u1[id-I] + u1[id+I] );
  }
}
```


2D Laplace solver

How does cache function in this application?



- if block size is a multiple of 32 in x-direction, then interior corresponds to set of complete cache lines
- "halo" points above and below are full cache lines too
- "halo" points on side are the problem – each one requires the loading of an entire cache line
- optimal block shape has aspect ratio of roughly 32:1 (or 8:1 if cache line is 32 bytes)

3D Laplace solver

- practical 3
- each thread does an entire line in z-direction
- x, y dimensions cut up into blocks in the same way as 2D application
- `laplace3d.cu` and `laplace3d_kernel.cu` follow same approach described above

- this used to give the fastest implementation, but a new version uses 3D thread blocks, with each thread responsible for just 1 grid point
- the new version has lots more integer operations, but is still faster (due to many more active threads?)

Lecture 4: warp shuffles, and reduction / scan operations

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Warp shuffles

Warp shuffles are a faster mechanism for moving data between threads in the same warp.

There are 4 variants:

- `__shfl_up_sync`
  copy from a lane with lower ID relative to caller
- `__shfl_down_sync`
  copy from a lane with higher ID relative to caller
- `__shfl_xor_sync`
  copy from a lane based on bitwise XOR of own lane ID
- `__shfl_sync`
  copy from indexed lane ID

Here the lane ID is the position within the warp
(`threadIdx.x%32` for 1D blocks)

Warp shuffles

```
T __shfl_up_sync(unsigned mask, T var,
unsigned int delta);
```

- `mask` controls which threads are involved — usually set to `-1` or `0xffffffff`, equivalent to all 1's
- `var` is a local register variable (int, unsigned int, long long, unsigned long long, float or double)
- `delta` is the offset within the warp – if the appropriate thread does not exist (i.e. it's off the end of the warp) then the value is taken from the current thread

```
T __shfl_down_sync(unsigned mask, T var,
unsigned int delta);
```

- defined similarly

Warp shuffles

```
T __shfl_xor_sync(unsigned mask, T var, int
laneMask);
```

- an XOR (exclusive or) operation is performed between `laneMask` and the calling thread's `laneID` to determine the lane from which to copy the value

  (`laneMask` controls which bits of `laneID` are "flipped")

- a "butterfly" type of addressing, very useful for reduction operations and FFTs

```
T __shfl_sync(unsigned mask, T var, int
srcLane);
```

- copies data from `srcLane`

Warp shuffles

Very important

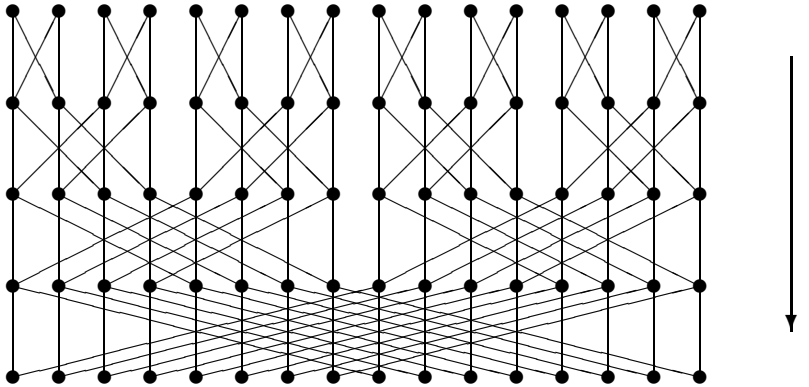
> Threads may only read data from another thread which is actively participating in the shuffle command. If the target thread is inactive, the retrieved value is undefined.

This means you must be very careful with conditional code.

Warp shuffles

Two ways to sum all the elements in a warp: method 1

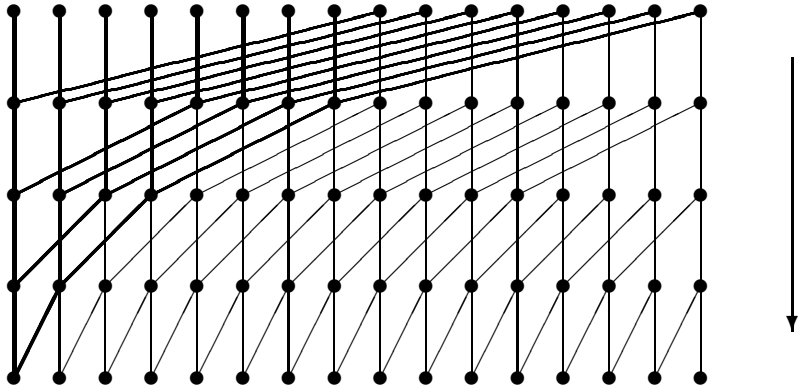
```
for (int i=1; i<32; i*=2)
  value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles

Two ways to sum all the elements in a warp: method 2

```
for (int i=16; i>0; i=i/2)
  value += __shfl_down_sync(-1, value, i);
```



Reduction

The most common reduction operation is computing the sum of a large array of values:

- averaging in Monte Carlo simulation
- computing RMS change in finite difference computation or an iterative solver
- computing a vector dot product in a CG or GMRES iteration

Reduction

Other common reduction operations are to compute a minimum or maximum.

Key requirements for a reduction operator $\circ$ are:

- commutative: $a \circ b = b \circ a$
- associative: $a \circ (b \circ c) = (a \circ b) \circ c$

Together, they mean that the elements can be re-arranged and combined in any order.

(Note: in MPI there are special routines to perform reductions over distributed arrays.)

Approach

Will describe things for a summation reduction – the extension to other reductions is obvious

Assuming each thread starts with one value, the approach is to

- first add the values within each thread block, to form a partial sum
- then add together the partial sums from all of the blocks

I'll look at each of these stages in turn

Local reduction

The first phase is contructing a partial sum of the values within a thread block.

Question 1: where is the parallelism?

"Standard" summation uses an accumulator, adding one value at a time $\Longrightarrow$ sequential

Parallel summation of N values:

- first sum them in pairs to get $N/2$ values
- repeat the procedure until we have only one value

Local reduction

Question 2: any problems with warp divergence?

Note that not all threads can be busy all of the time:

- $N/2$ operations in first phase
- $N/4$ in second
- $N/8$ in third
- etc.

For efficiency, we want to make sure that each warp is either fully active or fully inactive, as far as possible.

Local reduction

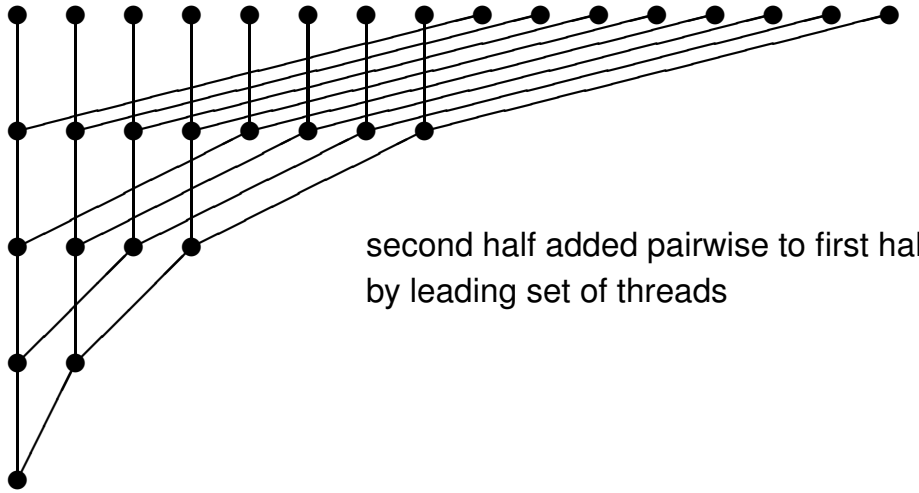
Question 3: where should data be held?

Threads need to access results produced by other threads:

- global device arrays would be too slow, so use shared memory
- need to think about synchronisation

Local reduction

Pictorial representation of the algorithm:



second half added pairwise to first half by leading set of threads

Local reduction

```
__global__ void sum(float *d_sum,float *d_data)
{
 extern __shared__  float temp[];
 int tid = threadIdx.x;

 temp[tid] = d_data[tid+blockIdx.x*blockDim.x];

 for (int d=blockDim.x>>1; d>=1; d>>=1) {
   __syncthreads();
   if (tid<d) temp[tid] += temp[tid+d];
 }

 if (tid==0) d_sum[blockIdx.x] = temp[0];
}
```


Local reduction

Note:

- use of dynamic shared memory – size has to be declared when the kernel is called
- use of `__syncthreads` to make sure previous operations have completed
- first thread outputs final partial sum into specific place for that block

- could use shuffles when only one warp still active
- alternatively, could reduce each warp, put partial sums in shared memory, and then the first warp could reduce the sums – requires only one `__syncthreads`

Global reduction: version 1

This version of the local reduction puts the partial sum for each block in a different entry in a global array

These partial sums can be transferred back to the host for the final summation – practical 4

Global reduction: version 2

Alternatively, can use the atomic add discussed in the previous lecture, and replace

```
if (tid==0) d_sum[blockIdx.x] = temp[0];
```

by

```
if (tid==0) atomicAdd(&d_sum,temp[0]);
```


Global reduction: version 2

More general reduction operations could can use the atomic lock mechanism, also discussed in the previous lecture:

```
if (tid==0) d_sum[blockIdx.x] = temp[0];
```

by

```
if (tid==0) {
 do {} while(atomicCAS(&lock,0,1)); // set lock

 *d_sum += temp[0];
 __threadfence();  // wait for write completion

 lock = 0;                          // free lock
}
```


Scan operation

Given an input vector $u_i,\ i = 0, \ldots, I-1$, the objective of a scan operation is to compute

$$v_j = \sum_{i<j} u_i \quad \text{for all } \ j < I.$$

Why is this important?

- a key part of many sorting routines
- arises also in particle filter methods in statistics
- related to solving long recurrence equations:

$$v_{n+1} = (1-\lambda_n)v_n + \lambda_n u_n$$

- a good example that looks impossible to parallelise

Scan operation

Before explaining the algorithm, here's the "punch line":

- some parallel algorithms are tricky – don't expect them all to be obvious
- check the examples in the CUDA SDK, check the literature using Google – don't put lots of effort into re-inventing the wheel
- the relevant literature may be 25–30 years old – back to the glory days of CRAY vector computing and Thinking Machines' massively-parallel CM5

Scan operation

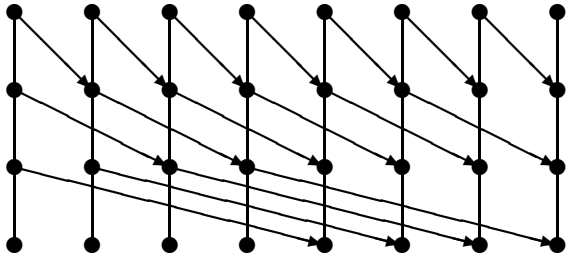
Similar to the global reduction, the top-level strategy is

- perform local scan within each block
- add on sum of all preceding blocks

Will describe two approaches to the local scan, both similar to the local reduction

- first approach:
  - very simple using shared memory, but $O(N \log N)$ operations
- second approach:
  - more efficient using warp shuffles and a recursive structure, with $O(N)$ operations

Local scan: version 1



- after n passes, each sum has local plus preceding $2^n - 1$ values
- $\log_2 N$ passes, and $O(N)$ operations per pass
  $\Longrightarrow$ $O(N \log N)$ operations in total

Local scan: version 1

```
__global__ void scan(float *d_data) {

 extern __shared__ float temp[];
 int tid   = threadIdx.x;
 temp[tid] = d_data[tid+blockIdx.x*blockDim.x];

 for (int d=1; d<blockDim.x; d<<=1) {
   __syncthreads();
   float temp2 = (tid >= d) ? temp[tid-d] : 0;
   __syncthreads();
   temp[tid] += temp2;
 }

 ...
}
```

Local scan: version 1

Notes:

- increment is set to zero if no element to the left
- both `__syncthreads();` are needed

Local scan: version 2

The second version starts by using warp shuffles to perform a scan within each warp, and store the warp sum:

```
__global__ void scan(float *d_data) {
  __shared__ float temp[32];
  float temp1, temp2;
  int tid = threadIdx.x;
  temp1   = d_data[tid+blockIdx.x*blockDim.x];

  for (int d=1; d<32; d<<=1) {
    temp2 = __shfl_up_sync(-1, temp1,d);
    if (tid%32 >= d) temp1 += temp2;
  }

  if (tid%32 == 31) temp[tid/32] = temp1;
  __syncthreads();
  ...
```


Local scan: version 2

Next we perform a scan of the warp sums (assuming no more than 32 warps):

```
if (tid < 32) {
  temp2 = 0.0f;
  if (tid < blockDim.x/32)
    temp2 = temp[tid];

  for (int d=1; d<32; d<<=1) {
    temp3 = __shfl_up_sync(-1, temp2,d);
    if (tid%32 >= d) temp2 += temp3;
  }
  if (tid < blockDim.x/32) temp[tid] = temp2;
}
```


Local scan: version 2

Finally, we add the sum of previous warps:

```
  __syncthreads();

  if (tid >= 32) temp1 += temp[tid/32 - 1];

  ...
}
```

Global scan: version 1

To complete the global scan there are two options

First alternative:

- use one kernel to do local scan and compute partial sum for each block
- use host code to perform a scan of the partial sums
- use another kernel to add sums of preceding blocks

Global scan: version 2

Second alternative – do it all in one kernel call

However, this needs the sum of all preceding blocks to add to the local scan values

Problem: blocks are not necessarily processed in order, so could end up in deadlock waiting for results from a block which doesn't get a chance to start.

Solution: use atomic increments

Global scan: version 2

Declare a global device variable

```
__device__  int my_block_count = 0;
```

and at the beginning of the kernel code use

```
__shared__ unsigned int my_blockId;
if (threadIdx.x==0) {
  my_blockId = atomicAdd( &my_block_count, 1 );
}
__syncthreads();
```

which returns the old value of `my_block_count` and increments it, all in one operation.

This gives us a way of launching blocks in strict order.

Global scan: version 2

In the second approach to the global scan, the kernel code does the following:

- get in-order block ID
- perform scan within the block
- wait until another global counter `my_block_count2` shows that preceding block has computed the sum of the blocks so far
- get the sum of blocks so far, increment the sum with the local partial sum, then increment `my_block_count2`
- add previous sum to local scan values and store the results

Global scan: version 2

```
// get global sum, and increment for next block

if (tid == 0) {
  // do-nothing atomic forces a load each time
  do {} while( atomicAdd(&my_block_count2,0)
                                < my_blockId );

  temp = sum;           // copy into register
  sum  = temp + local; // increment and put back
  __threadfence();    // wait for write completion

  atomicAdd(&my_block_count2,1);
                    // faster than plain addition
}
```


Scan operation

Conclusion: this is all quite tricky!

Advice: best to first see if you can get working code from someone else (e.g. investigate Thrust library)

Don't re-invent the wheel unless you really think you can do it better.

Recurrence equation

Given s_n, u_n, want to compute v_n defined by

$$v_n = s_n\, v_{n-1} + u_n$$

(Often have

$$v_n = (1-\lambda_n)\, v_{n-1} + \lambda_n\, u_n$$

with $0<\lambda_n<1$ so this computes a running weighted average, but that's not important here.)

Again looks naturally sequential, but in fact it can be handled in the same way as the scan.

Recurrence equation

Starting from

$$\begin{aligned} v_n &= s_n\, v_{n-1} + u_n \\ v_{n-1} &= s_{n-1}\, v_{n-2} + u_{n-1} \end{aligned}$$

then substituting the second equation into the first gives

$$v_n = (s_n s_{n-1})\, v_{n-2} + (s_n u_{n-1} + u_n)$$

so $\quad (s_{n-1}, u_{n-1}),\ (s_n, u_n) \longrightarrow (s_n s_{n-1},\ s_n u_{n-1} + u_n)$

The same at each level of the scan, eventually giving

$$v_n = s'_n v_{-1} + u'_n$$

where v_{-1} represents the last element of the previous block.

Recurrence equation

When combining the results from different blocks we have the same choices as before:

- store s', u' back to device memory, combine results for different blocks on the CPU, then for each block we have v_{-1} and can complete the computation of v_n
- use atomic trick to launch blocks in order, and then after completing first phase get v_{-1} from previous block to complete the computation.

Similarly, the calculation within a block can be performed using shuffles in a two-stage process:

1. use shuffles to compute solution within each warp
2. use shared memory and shuffles to combine results from different warps and update solution from first stage

Lecture 5: libraries and tools

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

CUDA libraries

Originally, NVIDIA planned to provide only one or two maths libraries, but over time these have steadily increased

- CUDA math library
  all of the standard math functions you would expect (i.e. very similar to what you would get from Intel)
  - various exponential and log functions
  - trigonometric functions and their inverses
  - hyperbolic functions and their inverses
  - error functions and their inverses
  - Bessel and Gamma functions
  - vector norms and reciprocals (esp. for graphics)
  - mainly single and double precision – a few in half precision

CUDA libraries

- cuBLAS
  - basic linear algebra subroutines for dense matrices
  - includes matrix-vector and matrix-matrix product
  - significant input from Vasily Volkov at UC Berkeley; one routine contributed by Jonathan Hogg from RAL
  - it is possible to call cuBLAS routines from user kernels – device API
  - some support for a single routine call to do a "batch" of smaller matrix-matrix multiplications
  - also support for using CUDA streams to do a large number of small tasks concurrently

CUDA libraries

cuBLAS is a set of routines to be called by user host code:

- helper routines:
  - memory allocation
  - data copying from CPU to GPU, and vice versa
  - error reporting
- compute routines:
  - matrix-matrix and matrix-vector product
  - Warning! Some calls are asynchronous, i.e. the call starts the operation but the host code then continues before it has completed

`simpleCUBLAS` example in SDK is a good example code

cuBLASxt extends cuBLAS to multiple GPUs

CUDA libraries

- cuFFT
  - Fast Fourier Transform
  - 1D, 2D, 3D
  - significant input from Satoshi Matsuoka and others at Tokyo Institute of Technology
  - has almost all of the variations found in FFTW and other CPU libraries?
  - nothing yet at device level?

CUDA libraries

Like cuBLAS, it is a set of routines called by user host code:

- helper routines include “plan” construction
- compute routines perform 1D, 2D, 3D FFTs
- it supports doing a “batch” of independent transforms, e.g. applying 1D transform to a 3D dataset
- `simpleCUFFT` example in SDK

CUDA libraries

- cuSPARSE
  - various routines to work with sparse matrices
  - includes sparse matrix-vector and matrix-matrix products
  - could be used for iterative solution
  - also has solution of sparse triangular system
  - note: batched tridiagonal solver is in cuBLAS not cuSPARSE
  - contribution from István Reguly (Oxford)

CUDA libraries

- cuRAND
  - random number generation
  - XORWOW, `mrg32k3a`, Mersenne Twister and `Philox_4x32_10` pseudo-random generators
  - Sobol quasi-random generator (with optimal scrambling)
  - uniform, Normal, log-Normal, Poisson outputs
  - includes device level routines for RNG within user kernels
- cuSOLVER:
  - key LAPACK dense solvers, 3 – 6x faster than MKL
  - sparse direct solvers, 2–14x faster than CPU equivalents

CUDA libraries

- CUB
  - provides a collection of basic building blocks at three levels: device, thread block, warp
  - functions include sort, scan, reduction
  - Thrust uses CUB for CUDA version of key algorithms
- AmgX (originally named NVAMG)
  - library for algebraic multigrid
  - available from `http://developer.nvidia.com/amgx`

CUDA Libraries

- cuDNN
  - library for Deep Neural Networks
  - some parts developed by Jeremy Appleyard (NVIDIA) working in Oxford
- nvGraph
  - Page Rank, Single Source Shortest Path, Single Source Widest Path
- NPP (NVIDIA Performance Primitives)
  - library for imaging and video processing
  - includes functions for filtering, JPEG decoding, etc.
- CUDA Video Decoder API

CUDA Libraries

- Thrust
  - high-level C++ template library with an interface based on the C++ Standard Template Library (STL)
  - very different philosophy to other libraries; users write standard C++ code (no CUDA) but get the benefits of GPU parallelisation
  - also supports x86 execution
  - relies on C++ object-oriented programming; certain objects exist on the GPU, and operations involving them are implicitly performed on the GPU
  - I've not used it, but for some applications it can be very powerful – e.g. lots of built-in functions for operations like sort and scan
  - also simplifies memory management and data movement

CUDA Libraries

- Kokkos
  - another high-level C++ template library
  - developed in the US DoE Labs, so considerable investment in both capabilities and on-going software maintenance
  - again I've not used it, but possibly worth investigating
  - for more information see
    `https://github.com/kokkos/kokkos/wiki`
    `https://trilinos.org/packages/kokkos/`

Useful header files

- `dbldbl.h` available from
  `https://gist.github.com/seibert/5914108`
  Header file for double-double arithmetic for quad-precision (developed by NVIDIA, but published independently under the terms of the BSD license)
- `cuComplex.h` part of the standard CUDA distribution
  Header file for complex arithmetic – defines a class and overloaded arithmetic operations.
- `helper_math.h` available in CUDA SDK
  Defines operator-overloading operations for CUDA intrinsic vector datatypes such as `float4`

Other libraries

- MAGMA
  - a new LAPACK for GPUs – higher level numerical linear algebra, layered on top of CUBLAS
  - open source – freely available
  - developed by Jack Dongarra, Jim Demmel and others

Other libraries

- ArrayFire from Accelereyes:
  - was commercial software, but now open source
  - supports both CUDA and OpenCL execution
  - C, C++ and Fortran interfaces
  - wide range of functionality including linear algebra, image and signal processing, random number generation, sorting
  - `www.accelereyes.com/products/arrayfire`

NVIDIA maintains webpages with links to a variety of CUDA libraries:
`developer.nvidia.com/gpu-accelerated-libraries`
and other tools:
`developer.nvidia.com/tools-ecosystem`

The 7 dwarfs

- Phil Colella, senior researcher at Lawrence Berkeley National Laboratory, talked about “7 dwarfs” of numerical computation in 2004
- expanded to 13 by a group of UC Berkeley professors in a 2006 report: “A View from Berkeley”
  `www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`
- key algorithmic kernels in many scientific computing applications
- very helpful to focus attention on HPC challenges and development of libraries and problem-solving environments/frameworks.

The 7 dwarfs

- dense linear algebra
- sparse linear algebra
- spectral methods
- N-body methods
- structured grids
- unstructured grids
- Monte Carlo

Dense linear algebra

- cuBLAS
- cuSOLVER
- MAGMA
- ArrayFire

Sparse linear algebra

- iterative solvers:
  - some available in PetSc
  - others can be implemented using sparse matrix-vector multiplication from cuSPARSE
  - NVIDIA has AmgX, an algebraic multigrid library
- direct solvers:
  - NVIDIA's cuSOLVER
  - SuperLU project at University of Florida (Tim Davis)
    `www.cise.ufl.edu/ davis/publications_files/qrgpu_paper.pdf`
  - project at RAL (Jennifer Scott & Jonathan Hogg)
    `https://epubs.stfc.ac.uk/work/12189719`

Spectral methods

- cuFFT
  - library provided / maintained by NVIDIA
- nothing else needed?

N-body methods

- OpenMM
  - `http://openmm.org/`
  - open source package to support molecular modelling, developed at Stanford
- Fast multipole methods:
  - ExaFMM by Yokota and Barba: `http://www.bu.edu/exafmm/`
  - FMM2D by Holm, Engblom, Goude, Holmgren: `http://user.it.uu.se/~stefane/freeware`
  - software by Takahashi, Cecka, Fong, Darve: `onlinelibrary.wiley.com/doi/10.1002/nme.3240/pdf`

Structured grids

- lots of people have developed one-off applications
- no great need for a library for single block codes (though possible improvements from “tiling”?)
- multi-block codes could benefit from a general-purpose library, mainly for MPI communication
- Oxford OPS project has developed a high-level open-source framework for multi-block codes, using GPUs for code execution and MPI for distributed-memory message-passing

  all implementation details are hidden from “users”, so they don’t have to know about GPU/MPI programming

Unstructured grids

In addition to GPU implementations of specific codes there are projects to create high-level solutions which others can use for their application codes:

- Alonso, Darve and others (Stanford)
- Oxford / Imperial College project developed OP2, a general-purpose open-source framework based on a previous framework built on MPI

May be other work I’m not aware of

Monte Carlo

- NVIDIA cuRAND library
- Accelereyes ArrayFire library
- some examples in CUDA SDK distribution
- nothing else needed except for more output distributions?

Tools

Debugging:

- `cuda-memcheck`
  detects array out-of-bounds errors, and mis-aligned device memory accesses – very useful because such errors can be tough to track down otherwise
- `cuda-memcheck --tool racecheck`
  this checks for shared memory race conditions:
  - Write-After-Write (WAW): two threads write data to the same memory location but the order is uncertain
  - Read-After-Write (RAW) and Write-After-Read (WAR): one thread writes and another reads, but the order is uncertain
- `cuda-memcheck --tool initcheck`
  detects reading of uninitialised device memory

Tools

Other languages:

- FORTRAN: PGI (Portland Group) CUDA FORTRAN compiler with natural FORTRAN equivalent to CUDA C; also IBM FORTRAN XL for new DoE systems
- MATLAB: can call kernels directly, or use OOP like Thrust to define MATLAB objects which live on the GPU
  `http://www.oerc.ox.ac.uk/projects/cuda-centre-excellence/matlab-gpus`
- Mathematica: similar to MATLAB?
- Python: `http://mathema.tician.de/software/pycuda`
  `https://store.continuum.io/cshop/accelerate/`
- R: `http://www.fuzzyl.com/products/gpu-analytics/`
  `http://cran.r-project.org/web/views/HighPerformanceComputing.html`
- Haskell: `https://hackage.haskell.org/package/cuda`
  `http://hackage.haskell.org/package/accelerate`

Tools

OpenACC (“More Science, Less Programming”):

- like Thrust, aims to hide CUDA programming by doing everything in the top-level CPU code
- programmer takes standard C/C++/Fortran code and inserts pragmas saying what can be done in parallel and where data should be located
- `https://www.openacc.org/`

OpenMP 4.0 is similar but newer:

- strongly pushed by Intel to accommodate Xeon Phi and unify things, in some sense
- `on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf`

Tools

Integrated Development Environments (IDE):

- Nsight Visual Studio edition – NVIDIA plug-in for Microsoft Visual Studio
  `developer.nvidia.com/nvidia-nsight-visual-studio-edition`
- Nsight Eclipse edition – IDE for Linux systems
  `developer.nvidia.com/nsight-eclipse-edition`
- these come with editor, debugger, profiler integration

Tools

NVIDIA Visual Profiler `nvprof`:

- standalone software for Linux and Windows systems
- uses hardware counters to collect a lot of useful information
- I think only 1 SM is instrumented – implicitly assumes the others are behaving similarly
- lots of things can be measured, but a limited number of counters, so it runs the application multiple times if necessary to get full info
- can also obtain instruction counts from command line:
  `nvprof --metrics "flops_sp,flops_dp" prac2`

  do `nvprof --help` for more info on other options

Summary

- active work on all of the dwarfs
- in most cases, significant effort to develop general purpose libraries or frameworks, to enable users to get the benefits without being CUDA experts
- too much going on for one person (e.g. me) to keep track of it all
- NVIDIA maintains a webpage with links to CUDA tools/libraries:
  `developer.nvidia.com/cuda-tools-ecosystem`
- the existence of this eco-system is part of why I think CUDA will remain more used than OpenCL for HPC

Lecture 6: odds and ends

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Overview

- synchronicity
- multiple streams and devices
- multiple GPUs
- other odds and ends

Warnings

- I haven't tried most of what I will describe
- some of these things have changed from one version of CUDA to the next – everything here is for the latest version
- overall, keep things simple unless it's really needed for performance
- if it is, proceed with extreme caution, do practical 11, and check out the examples in the SDK

Synchronicity

A computer system has lots of components:

- CPU(s)
- GPU(s)
- memory controllers
- network cards

Many of these can be doing different things at the same time – usually for different processes, but sometimes for the same process

Synchronicity

The von Neumann model of a computer program is synchronous with each computational step taking place one after another

- this is an idealisation – almost never true in practice
- compiler frequently generates code with overlapped instructions (pipelined CPUs) and does other optimisations which re-arrange execution order and avoid redundant computations
- however, it is usually true that as a programmer you can think of it as a synchronous execution when working out whether it gives the correct results
- when things become asynchronous, the programmer has to think very carefully about what is happening and in what order

Synchronicity

With GPUs we have to think even more carefully:

- host code executes on the CPU(s); kernel code executes on the GPU(s)
- . . . but when do the different bits take place?
- . . . can we get better performance by being clever?
- . . . might we get the wrong results?

Key thing is to try to get a clear idea of what is going on – then you can work out the consequences

GPU code

- for each warp, code execution is effectively synchronous
- different warps execute in an arbitrary overlapped fashion – use `__syncthreads()` if necessary to ensure correct behaviour
- different thread blocks execute in an arbitrary overlapped fashion

All of this has been described over the past 3 days – nothing new here.

The focus of these new slides is on host code and the implications for CPU and GPU execution

Host code

Simple/default behaviour:

- 1 CPU
- 1 GPU
- 1 thread on CPU (i.e. scalar code)
- 1 default “stream” on GPU

Host code

- most CUDA calls are synchronous / blocking:
- example: `cudaMemcpy`
  - host call starts the copying and waits until it has finished before the next instruction in the host code
  - why? – ensures correct execution if subsequent host code reads from, or writes to, the data being copied

Host code

- CUDA kernel launch is asynchronous / non-blocking
  - host call starts the kernel execution, but doesn't wait for it to finish before going on to next instruction
- similar for `cudaMemcpyAsync`
  - starts the copy but doesn't wait for completion
  - has to be done through a "stream" with page-locked memory (also known as pinned memory) – see documentation
- in both cases, host eventually waits when at a `cudaDeviceSynchronize()` call
- benefit? – in general, doesn't affect correct execution, and might improve performance by overlapping CPU and GPU execution

Host code

What could go wrong?

- kernel timing – need to make sure it's finished
- could be a problem if the host uses data which is read/written directly by kernel, or transferred by `cudaMemcpyAsync`
- `cudaDeviceSynchronize()` can be used to ensure correctness (similar to `__syncthreads()` for kernel code)

Multiple Streams

Quoting from section 3.2.5.5 in the CUDA Programming Guide:

> Applications manage concurrency through streams.
>
> A stream is a sequence of commands (possibly issued by different host threads) that execute in order.
>
> Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently.

Multiple Streams

Optional stream argument for

- kernel launch
- `cudaMemcpyAsync`

with streams creating using `cudaStreamCreate`

Within each stream, CUDA operations are carried out in order (i.e. FIFO – first in, first out); one finishes before the next starts

Key to getting better performance is using multiple streams to overlap things

Page-locked memory

Section 3.2.4:

- host memory is usually paged, so run-time system keeps track of where each page is located
- for higher performance, can fix some pages, but means less memory available for everything else
- CUDA uses this for better host <–> GPU bandwidth, and also to hold “device” arrays in host memory
- can provide up to 100% improvement in bandwidth
- also, it is required for `cudaMemcpyAsync`
- allocated using `cudaHostAlloc`, or registered by `cudaHostRegister`

Default stream

The way the default stream behaves in relation to others depends on a compiler flag:

- no flag, or `--default-stream legacy`

  old (bad) behaviour in which a `cudaMemcpy` or kernel launch on the default stream blocks/synchronizes with other streams
- `--default-stream per-thread`

  new (good) behaviour in which the default stream doesn’t affect the others
- note: flag label is a bit odd – it has other effects too

Practical 11

```
cudaStream_t streams[8];
float *data[8];

for (int i = 0; i < 8; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&data[i], N * sizeof(float));

  // launch one worker kernel per stream
  kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

  // do a Memcpy and launch a dummy kernel on default stream
  cudaMemcpy(d_data,h_data,sizeof(float),
             cudaMemcpyHostToDevice);
  kernel<<<1, 1>>>(d_data, 0);
}
cudaDeviceSynchronize();
```

Default stream

The second (main?) effect of the flag comes when using multiple threads (e.g. OpenMP or POSIX multithreading)

In this case the effect of the flag is to create separate independent (i.e. non-interfering) default streams for each thread

Using multiple default streams, one per thread, is a good alternative to using multiple “proper” streams

Practical 11

```
omp_set_num_threads(8);
float *data[8];

for (int i = 0; i < 8; i++)
  cudaMalloc(&data[i], N * sizeof(float));

#pragma omp parallel for
for (int i = 0; i < 8; i++) {
  printf(" thread ID = %d \n",omp_get_thread_num());

  // launch one worker kernel per thread
  kernel<<<1, 64>>>(data[i], N);
}

cudaDeviceSynchronize();
```


Stream commands

Each stream executes a sequence of kernels, but sometimes you also need to do something on the host.

There are at least two ways of coordinating this:

- use a separate thread for each stream
  - it can wait for the completion of all pending tasks, then do what’s needed on the host
- use just one thread for everything
  - for each stream, add a callback function to be executed (by a new thread) when the pending tasks are completed
  - it can do what’s needed on the host, and then launch new kernels (with a possible new callback) if wanted

Stream commands

- `cudaStreamCreate()`
  creates a stream and returns an opaque “handle”
- `cudaStreamSynchronize()`
  waits until all preceding commands have completed
- `cudaStreamQuery()`
  checks whether all preceding commands have completed
- `cudaStreamAddCallback()`
  adds a callback function to be executed on the host once all preceding commands have completed

Stream events

Useful for synchronisation and timing between streams:

- `cudaEventCreate(event)`
  creates an "event"
- `cudaEventRecord(event,stream)`
  puts an event into a stream (by default, stream 0)
- `cudaEventSynchronize(event)`
  CPU waits until event occurs
- `cudaStreamWaitEvent(stream,event)`
  stream waits until event occurs
- `cudaEventQuery(event)`
  check whether event has occured
- `cudaEventElapsedTime(time,event1,event2)`

Multiple devices

What happens if there are multiple GPUs?

CUDA devices within the system are numbered, not always in order of decreasing performance

- by default a CUDA application uses the lowest number device which is "visible" and available
- visibility controlled by environment variable `CUDA_VISIBLE_DEVICES`
- current device can be set by using `cudaSetDevice`
- `cudaGetDeviceProperties` does what it says
- each stream is associated with a particular device – current device for a kernel launch or a memory copy
- see `simpleMultiGPU` example in SDK
- see section 3.2.6 for more information

Multiple devices

If a user is running on multiple GPUs, data can go directly between GPUs (peer – peer) – doesn't have to go via CPU

- very important when using new direct NVlink interconnect – much faster than PCIe
- `cudaMemcpy` can do direct copy from one GPU's memory to another
- a kernel on one GPU can also read directly from an array in another GPU's memory, or write to it
- this even includes the ability to do atomic operations with remote GPU memory
- for more information see Section 4.11, "Peer Device Memory Access" in CUDA Runtime API documentation:
  `https://docs.nvidia.com/cuda/cuda-runtime-api/`

Multi-GPU computing

Single workstation / server:

- a big enclosure for good cooling
- up to 4 high-end cards in 16x PCIe v3 slots – up to 12GB/s interconnect
- 2 high-end CPUs
- 1.5kW power consumption – not one for the office

NVIDIA DGX-1 Deep Learning server

- 8 NVIDIA GV100 GPUs, each with 32GB HBM2
- 2 $\times$ 20-core Intel Xeons (E5-2698 v4 2.2 GHz)
- 512 GB DDR4 memory, 8TB SSD
- 150GB/s NVlink interconnect between the GPUs

Multi-GPU computing

A bigger configuration:

- NVIDIA DGX-2 Deep Learning server
  - 16 NVIDIA GV100 GPUs, each with 32GB HBM2
  - 2 $\times$ 24-core Intel Xeons (Platinum 8168)
  - 1.5 TB DDR4 memory, 32TB SSD
  - NVSwitch interconnect between the GPUs
- a distributed-memory cluster / supercomputer with multiple nodes, each with
  - 2-4 GPUs
  - 100 Gb/s Infiniband
- PCIe v3 bandwidth of 12 GB/s similar to Infiniband bandwidth

Multi-GPU computing

The biggest GPU systems in Top500 list (June 2018):

- Summit (Oak Ridge National Lab, USA)
  - 122 petaflop (#1), 9MW
  - IBM Power 9 CPUs, NVIDIA Volta GV100 GPUs
- Sierra (Lawrence Livermore National Lab, USA)
  - 76 petaflop (#3)
  - IBM Power 9 CPUs, NVIDIA Volta GV100 GPUs
- ABCI (AIST, Japan)
  - 19 petaflop (#5), 2MW
  - Intel Xeon CPUs, NVIDIA Volta V100 GPUs
- Piz Daint (CSCS Switzerland)
  - 20 petaflop (#6), 2MW
  - Cray XC50 with NVIDIA P100 GPUs

Multi-GPU computing

How does one use such machines?

Depends on hardware choice:

- for single machines, use shared-memory multithreaded host application
- for clusters / supercomputers, use distributed-memory MPI message-passing

MPI approach

In the MPI approach:

- one GPU per MPI process (nice and simple)
- distributed-memory message passing between MPI processes (tedious but not difficult)
- scales well to very large applications
- main difficulty is that the user has to partition their problem (break it up into separate large pieces for each process) and then explicitly manage the communication
- note: should investigate GPU Direct for maximum performance in message passing

Multi-user support

What if different processes try to use the same device?

Depends on system compute mode setting (section 3.4):

- in “default” mode, each process uses the fastest device
  - good when one very fast card, and one very slow
  - not good when you have 2 identical fast GPUs
- in “exclusive” mode, each process is assigned to first unused device; it’s an error if none are available
- `cudaGetDeviceProperties` reports mode setting
- mode can be changed by sys-admin using `nvidia-smi` command line utility

Odds and ends

Appendix B.21: loop unrolling

If you have a loop:

```
for (int k=0; k<4; k++) a[i] += b[i];
```

then `nvcc` will automatically unroll this to give

```
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
a[3] += b[3];
```

to avoid cost of incrementing and looping.

The pragma

```
#pragma unroll 5
```

will also force unrolling for loops without explicit limits

Odds and ends

Appendix B.2.5: `__restrict__` keyword

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
               float* __restrict__ c) {
 c[0] = a[0] * b[0];
 c[1] = a[0] * b[0];
 c[2] = a[0] * b[0] * a[1];
 c[3] = a[0] * a[1];
 c[4] = a[0] * b[0];
 c[5] = b[0];
 ...
}
```

The qualifier asserts that there is no overlap between `a,b,c` , so the compiler can perform more optimisations

Odds and ends

Appendix E.3.3.3: `volatile` keyword

Tells the compiler the variable may change at any time, so not to re-use a value which may have been loaded earlier and apparently not changed since.

This can sometimes be important when using shared memory

Odds and ends

Compiling:

- `Makefile` for first few practicals uses `nvcc` to compile both the host and the device code
  - internally it uses `gcc` for the host code, at least by default
  - device code compiler based on open source LLVM compiler
- sometimes, prefer to use other compilers (e.g. `icc`, `mpicc`) for main code that doesn't have any CUDA calls
- this is fine provided you use `-fPIC` flag for position-independent-code (don't know what this means but it ensures interoperability)
- can also produce libraries for use in the standard way

Odds and ends

Prac 6 Makefile:

```
INC    := -I$(CUDA_HOME)/include -I.
LIB    := -L$(CUDA_HOME)/lib64 -lcudart
FLAGS  := --ptxas-options=-v --use_fast_math

main.o: main.cpp
	g++ -c -fPIC -o main.o main.cpp

prac6.o: prac6.cu
	nvcc prac6.cu -c -o prac6.o $(INC) $(FLAGS)

prac6: main.o prac6.o
	g++ -fPIC -o prac6 main.o prac6.o $(LIB)
```


Odds and ends

Prac 6 Makefile to create a library:

```
INC    := -I$(CUDA)/include -I.
LIB    := -L$(CUDA)/lib64 -lcudart
FLAGS  := --ptxas-options=-v --use_fast_math

main.o: main.cpp
     g++ -c -fPIC -o main.o main.cpp

prac6.a: prac6.cu
     nvcc prac6.cu -lib -o prac6.a $(INC) $(FLAGS)

prac6a: main.o prac6.a
     g++ -fPIC -o prac6a main.o prac6.a $(LIB)
```


Odds and ends

Other compiler options:

- `-arch=sm_35`
  specifies GPU architecture
- `-maxrregcount=n`
  asks compiler to generate code using at most `n` registers; compiler may ignore this if it's not possible, but it may also increase use up to this limit

  This is much less important now since threads can have up to 255 registers

Odds and ends

Launch bounds (B.20):

- `-maxrregcount` modifies default for all kernels
- each kernel can be individually controlled by specifying launch bounds heuristics

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
                  minBlocksPerMultiprocessor)
MyKernel(...)
```


Conclusions

This lecture has discussed a number of more advanced topics

As a beginner, you can ignore almost all of them

As you get more experienced, you will probably want to start using some of them to get the very best performance

Lecture 7: tackling a new application

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Initial planning

1) Has it been done before?

- check CUDA SDK examples
- check CUDA user forums
- check `gpucomputing.net`
- check with Google

Initial planning

2) Where is the parallelism?

- efficient CUDA execution needs thousands of threads
- usually obvious, but if not
  - go back to 1)
  - talk to an expert – they love a challenge
  - go for a long walk
- may need to re-consider the mathematical algorithm being used, and instead use one which is more naturally parallel – but this should be a last resort

Initial planning

Sometimes you need to think about "the bigger picture"

Already considered 3D finite difference example:

- lots of grid nodes so lots of inherent parallelism
- even for ADI method, a grid of 128^3 has 128^2 tri-diagonal solutions to be performed in parallel so OK to assign each one to a single thread
- but what if we have a 2D or even 1D problem to solve?

Initial planning

If we only have one such problem to solve, why use a GPU?

But in practice, often have many such problems to solve:

- different initial data
- different model constants

This adds to the available parallelism

Initial planning

2D:

- 64KB of shared memory == 16K `float` so grid of 64^2 could be held within shared memory
  - one kernel for entire calculation
  - each block handles a separate 2D problem; almost certainly just one block per SM
- for bigger 2D problems, would need to split each one across more than one block
  - separate kernel for each timestep / iteration

Initial planning

1D:

- can certainly hold entire 1D problem within shared memory of one SM
- maybe best to use a separate block for each 1D problem, and have multiple blocks executing concurrently on each SM
- but for implicit time-marching need to solve single tri-diagonal system in parallel – how?

Initial planning

Parallel Cyclic Reduction (PCR): starting from

$$a_n\, x_{n-1} + x_n + c_n\, x_{n+1} = d_n, \quad n = 0, \ldots N-1$$

with $a_m \equiv 0$ for $m<0$, $m \geq N$, subtract a_n times row $n-1$, and c_n times row $n+1$ and re-normalise to get

$$a^*_n\, x_{n-2} + x_n + c^*_n\, x_{n+2} = d^*_n$$

Repeating this $\log_2 N$ times gives the value for x_n (since $x_{n-N} \equiv 0, x_{n+N} \equiv 0$) and each step can be done in parallel.

(Practical 7 implements it using shared memory, but if $N \leq 32$ so it fits in a single warp then on Kepler hardware it can be implemented using shuffles.)

Initial planning

3) Break the algorithm down into its constituent pieces

- each will probably lead to its own kernels
- do your pieces relate to the 7 dwarfs?
- re-check literature for each piece – sometimes the same algorithm component may appear in widely different applications
- check whether there are existing libraries which may be helpful

Initial planning

4) Is there a problem with warp divergence?

- GPU efficiency can be completely undermined if there are lots of divergent branches
- may need to implement carefully – lecture 3 example:

  processing a long list of elements where, depending on run-time values, a few involve expensive computation:
  - first process list to build two sub-lists of "simple" and "expensive" elements
  - then process two sub-lists separately

- . . . or again seek expert help

Initial planning

5) Is there a problem with host <–> device bandwidth?

- usually best to move whole application onto GPU, so not limited by PCIe bandwidth (12GB/s)
- occasionally, OK to keep main application on the host and just off-load compute-intensive bits
- dense linear algebra is a good off-load example; data is $O(N^2)$ but compute is $O(N^3)$ so fine if N is large enough

Heart modelling

Heart modelling is another interesting example:

- keep PDE modelling (physiology, electrical field) on the CPU
- do computationally-intensive cellular chemistry on GPU (naturally parallel)
- minimal data interchange each timestep

Initial planning

6) is the application compute-intensive or data-intensive?

- break-even point is roughly 40 operations (FP and integer) for each 32-bit device memory access (assuming full cache line utilisation)
- good to do a back-of-the-envelope estimate early on before coding $\Longrightarrow$ changes approach to implementation

Initial planning

If compute-intensive:

- don't worry (too much) about cache efficiency
- minimise integer index operations – surprisingly costly (this changes with Volta which has separate integer units)
- if using double precision, think whether it's needed

If data-intensive:

- ensure efficient cache use – may require extra coding
- may be better to re-compute some quantities rather than fetching them from device memory
- if using double precision, think whether it's needed

Initial planning

Need to think about how data will be used by threads, and therefore where it should be held:

- registers (private data)
- shared memory (for shared access)
- device memory (for big arrays)
- constant arrays (for global constants)
- "local" arrays (efficiently cached)

Initial planning

If you think you may need to use "exotic" features like atomic locks:

- look for SDK examples
- write some trivial little test problems of your own
- check you really understand how they work

Never use a new feature for the first time on a real problem!

Initial planning

Read NVIDIA documentation on performance optimisation:

- section 5 of CUDA Programming Guide
- CUDA C Best Practices Guide
- Kepler Tuning Guide
- Maxwell Tuning Guide
- Pascal Tuning Guide

Programming and debugging

Many of my comments here apply to all scientific computing

Though not specific to GPU computing, they are perhaps particularly important for GPU / parallel computing because

debugging can be hard!

Above all, you don't want to be sitting in front of a 50,000 line code, producing lots of wrong results (very quickly!) with no clue where to look for the problem

Programming and debugging

- plan carefully, and discuss with an expert if possible
- code slowly, ideally with a colleague, to avoid mistakes but still expect to make mistakes!
- code in a modular way as far as possible, thinking how to validate each module individually
- build-in self-testing, to check that things which ought to be true, really are true

  (In my current project I have a flag `OP_DIAGS`; the larger the value the more self-testing the code does)
- overall, should have a clear debugging strategy to identify existence of errors, and then find the cause
- includes a sequence of test cases of increasing difficulty, testing out more and more of the code

Programming and debugging

When working with shared memory, be careful to think about thread synchronisation.

Very important!

Forgetting a

```
__syncthreads();
```

may produce errors which are unpredictable / rare — the worst kind.

Also, make sure all threads reach the synchronisation point — otherwise could get deadlock.

Reminder: can use `cuda-memcheck --tool racecheck` to check for race condition

Programming and debugging

In developing `laplace3d`, my approach was to

- first write CPU code for validation
- next check/debug CUDA code with `printf` statements as needed, with different grid sizes:
  - grid equal to 1 block with 1 warp (to check basics)
  - grid equal to 1 block and 2 warps (to check synchronisation)
  - grid smaller than 1 block (to check correct treatment of threads outside the grid)
  - grid with 2 blocks
- then turn on all compiler optimisations

Performance improvement

The size of the thread blocks can have a big effect on performance:

- often hard to predict optimal size *a priori*
- optimal size can also vary significantly on different hardware
- optimal size for `laplace3d` with a 128^3 grid was
  - 128×2 on Fermi generation
  - 32×4 on later Kepler generation

  at the time, the size of the change was a surprise
- we're not talking about just 1-2% improvement, can easily be a factor $2\times$ by changing block size

Performance improvement

A number of numerical libraries (e.g. FFTW, ATLAS) now feature auto-tuning – optimal implementation parameters are determined when the library is installed on the specific hardware

I think this is going to be important for GPU programming:

- write parameterised code
- use optimisation (possibly brute force exhaustive search) to find the optimal parameters
- an Oxford student, Ben Spencer, developed a simple flexible automated system to do this – can try it in one of the mini-projects

Performance improvement

Use profiling to understand the application performance:

- where is the application spending most time?
- how much data is being transferred?
- are there lots of cache misses?
- there are a number of on-chip counters can provide this kind of information

The CUDA profiler is great

- provides lots of information (a bit daunting at first)
- gives hints on improving performance

Going further

In some cases, a single GPU is not sufficient

Shared-memory option:

- single system with up to 16 GPUs
- single process with a separate host thread for each GPU, or use just one thread and switch between GPUs
- can also transfer data directly between GPUs

Distributed-memory option:

- a cluster, with each node having 1 or 2 GPUs
- MPI message-passing, with separate process for each GPU

Going further

Keep an eye on what is happening with new GPUs:

- Pascal came out in 2016:
  - P100 for HPC with great double precision
  - 16GB HBM2 memory $\rightarrow$ more memory bandwidth
  - NVlink $\rightarrow$ 4$\times$20GB/s links per GPU
- Volta came out in 2017/18:
  - V100 for HPC
  - 32GB HBM2 memory
  - roughly 50% faster than P100 in compute, memory bandwidth, and 80% faster with NVlink2
  - special "tensor cores" for machine learning (16-bit multiplication + 32-bit addition for matrix-matrix multiplication) – much faster for TensorFlow

Going further

Two GPU systems:

- NVIDIA DGX-1 Deep Learning server
  - 8 NVIDIA GV100 GPUs, each with 32GB HBM2
  - 2 $\times$ 20-core Intel Xeons (E5-2698 v4 2.2 GHz)
  - 512 GB DDR4 memory, 8TB SSD
  - 150GB/s NVlink interconnect between the GPUs
- NVIDIA DGX-2 Deep Learning server
  - 16 NVIDIA GV100 GPUs, each with 32GB HBM2
  - 2 $\times$ 24-core Intel Xeons (Platinum 8168)
  - 1.5 TB DDR4 memory, 32TB SSD
  - NVSwitch interconnect between the GPUs

JADE

Joint Academic Data science Endeavour

- funded by EPSRC under national Tier 2 initiative
- 22 DGX-1 systems (with older Pascal P100s)
- 50 / 30 / 20 split in intended use between machine learning / molecular dynamics / other
- Oxford led the consortium bid, but system sited at STFC Daresbury and run by STFC / Atos
- in operation for a year now

There is also a GPU system at Cambridge.

Going further

Intel:

- latest “Skylake” CPU architectures
  - some chips have built-in GPU, purely for graphics
  - 4−22 cores, each with a 256-bit AVX vector unit
  - one or two 512-bit AVX-512 vector units per core on new high-end Xeons
- Xeon Phi architecture
  - Intel’s competitor to GPUs, but now abandoned

ARM:

- already designed OpenCL GPUs for smart-phones
- new 64-bit Cavium Thunder-X2 has up to 54 cores, being used in Bristol’s new “Isambard” Cray supercomputer

Going further

My current software assessment:

- CUDA is dominant in HPC, because of
  - ease-of-use
  - NVIDIA dominance of hardware, with big sales in games/VR, machine learning, supercomputing
  - extensive library support
  - support for many different languages (FORTRAN, Python, R, MATLAB, etc.)
  - extensive eco-system of tools
- OpenCL is the multi-platform standard, but currently only used for low-end mass-market applications
  - computer games
  - HD video codecs

Final words

- it continues to be an exciting time for HPC
- the fun will wear off, and the challenging coding will remain – computer science objective should be to simplify this for application developers through
  - libraries
  - domain-specific high-level languages
  - code transformation
  - better auto-vectorising compilers
- confident prediction: GPUs and other accelerators / vector units will be dominant in HPC for next 5-10 years, so it’s worth your effort to re-design and re-implement your algorithms