

Lecture 1: an introduction to CUDA

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

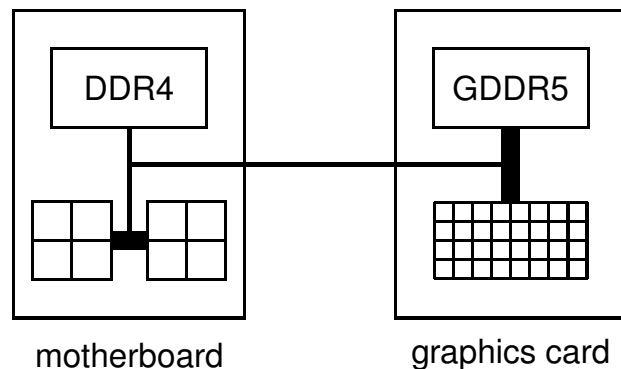
- hardware view
- software view
- CUDA programming

Lecture 1 – p. 1

Lecture 1 – p. 2

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:



Lecture 1 – p. 3

Hardware view

Currently, 3 generations of hardware cards in use:

- Kepler:
 - first released in 2012
 - includes HPC cards with excellent double precision
 - our practicals will use K40s and K80s
- Maxwell:
 - first released in 2014
 - only gaming cards, not HPC, so poor DP
- Pascal:
 - just released in 2016
 - just a few gaming and HPC cards in Oxford

Lecture 1 – p. 4

Hardware view

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

Consumer graphics cards (GeForce):

- GTX 1060: 1280 cores, 6GB (£240)
- GTX 1070: 1920 cores, 8GB (£400)
- GTX 1080: 2560 cores, 8GB (£600)
- GTX Titan X: 3584 cores, 12GB (£1200)

HPC (Tesla):

- P100: 3584 cores, 16GB HBM2 (£4-8k?)

Lecture 1 – p. 5

Hardware view

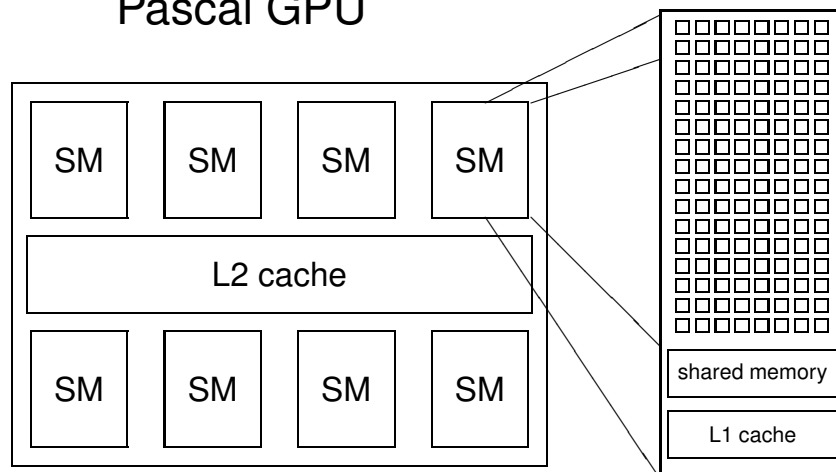
- building block is a “streaming multiprocessor” (SM):
 - 128 cores (64 in P100) and 64k registers
 - 96KB (64KB in P100) of shared memory
 - 48KB (24KB in P100) L1 cache
 - 8-16KB (?) cache for constants
 - up to 2K threads per SM
- different chips have different numbers of these SMs:

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

Lecture 1 – p. 6

Hardware View

Pascal GPU



Lecture 1 – p. 7

Hardware view

There are multiple products in the Maxwell generation, but none with good double precision performance for HPC

Consumer graphics cards (GeForce):

- GTX 960: 1024 cores, 2GB (£170 → £155)
- GTX 980: 2048 cores, 4GB (£450 → £400)
- (old) GTX Titan X: 3076 cores, 12GB (£1000)

Lecture 1 – p. 8

Hardware view

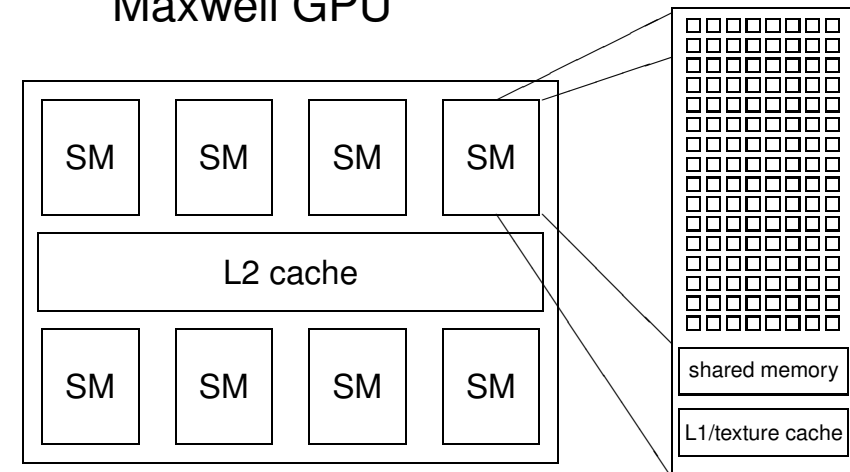
- building block is a “streaming multiprocessor” (SM):
 - 128 cores and 64k registers
 - 64-96KB of shared memory
 - 12-48KB L1 cache / read-only texture cache
 - 10KB cache for constants
 - up to 2K threads per SM
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX 960	8	112 GB/s	2 GB	120W
GTX 980	16	224 GB/s	4 GB	165W
GTX Titan X	24	336 GB/s	12 GB	250W

Lecture 1 – p. 9

Hardware View

Maxwell GPU



Lecture 1 – p. 10

Hardware view

There are multiple products in the Kepler generation

Consumer graphics cards (GeForce):

- GTX Titan Black: 2880 cores, 6GB (originally £600)
- GTX Titan Z: 2×2880 cores, 2×6GB (originally £1200)

HPC cards (Tesla):

- K80: 2×2496 cores, 2×12GB (originally £3.5k)

Lecture 1 – p. 11

Hardware view

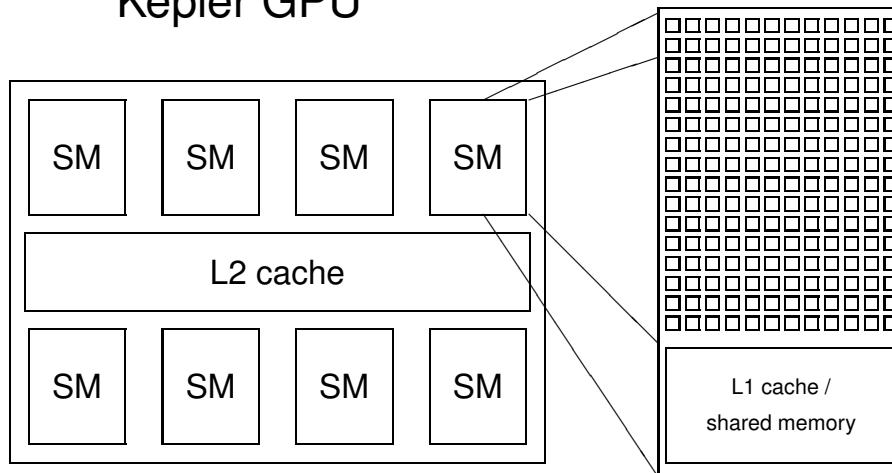
- building block is a “streaming multiprocessor” (SM):
 - 192 cores and 64k registers
 - 64KB of shared memory / L1 cache
 - 8KB cache for constants
 - 48KB texture cache for read-only arrays
 - up to 2K threads per SM
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX Titan Black	15	336 GB/s	6 GB	250W
GTX Titan Z	2×15	2×336 GB/s	2×6 GB	375W
K80	2×14	2×240 GB/s	2×12 GB	300W

Lecture 1 – p. 12

Hardware View

Kepler GPU



Lecture 1 – p. 13

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

Lecture 1 – p. 14

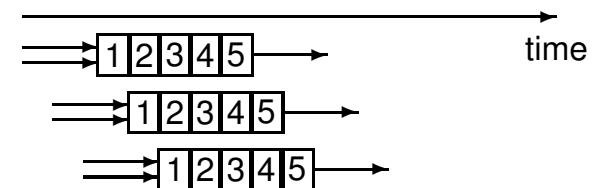
Multithreading

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?

Lecture 1 – p. 15

Lecture 1 – p. 16

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

Lecture 1 – p. 17

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 (now 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

Lecture 1 – p. 19

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

Lecture 1 – p. 18

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

Lecture 1 – p. 20

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 OpenGL

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

Lecture 1 – p. 21

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.

Lecture 1 – p. 22

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

Lecture 1 – p. 23

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:

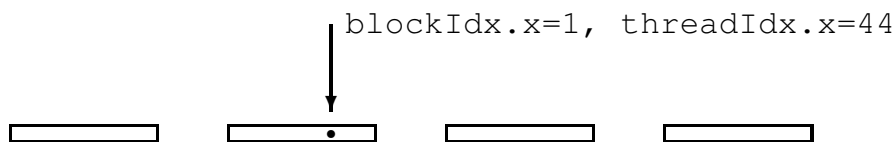
- some variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
 - `gridDim` size (or dimensions) of grid of blocks
 - `blockDim` size (or dimensions) of each 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

Lecture 1 – p. 24

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



Lecture 1 – p. 25

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);
}
```

Lecture 1 – p. 27

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

Lecture 1 – p. 26

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
```

Lecture 1 – p. 28

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 $\Rightarrow$ 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

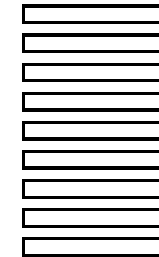
- data coming from device memory (memory latency)
- completion of earlier instructions (pipeline delay)

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

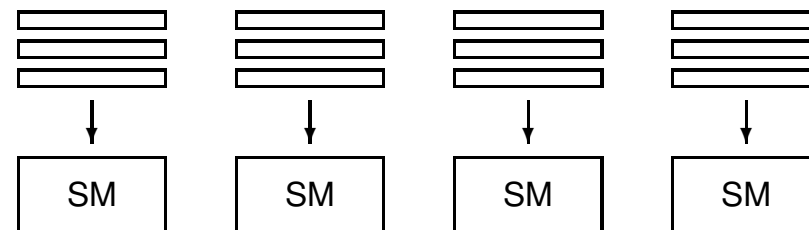
Lecture 1 – p. 29

CUDA programming

Queue of waiting blocks:



Multiple blocks running on each SM:



Lecture 1 – p. 30

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.

Lecture 1 – p. 31

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.

Lecture 1 – p. 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

Lecture 1 – p. 33

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( ... );`

Lecture 1 – p. 35

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

Lecture 1 – p. 34

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

Lecture 1 – p. 36

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

Lecture 1 – p. 37

Practical 1

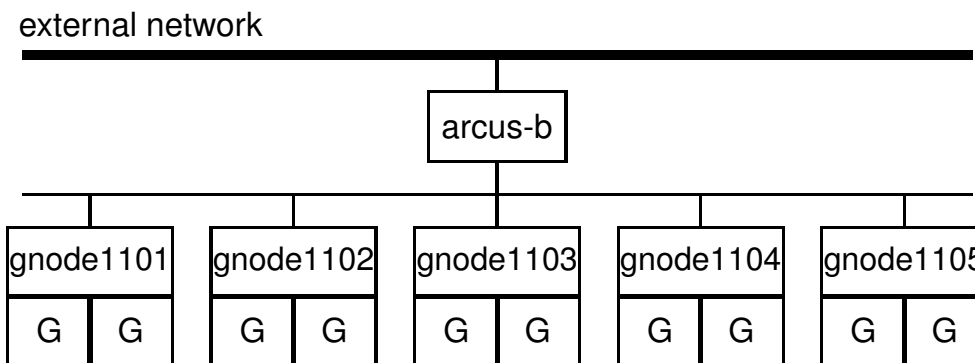
This leads to simpler code, but it’s important to understand what is happening because it may hurt performance:

- if a kernel uses an array x , then this probably forces a copy from CPU to GPU, even if the data hasn’t changed since the previous transfer
- when there is a `cudaDeviceSynchronize();` synchronisation, sometimes managed data is transferred back to the CPU, even if it hasn’t changed and/or is not needed

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

Lecture 1 – p. 38

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

Lecture 1 – p. 39

Key reading

CUDA Programming Guide, version 7.5:

- Chapter 1: Introduction
- Chapter 2: Programming Model
- 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

Lecture 1 – p. 40

Lecture 2: different memory and variable types

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

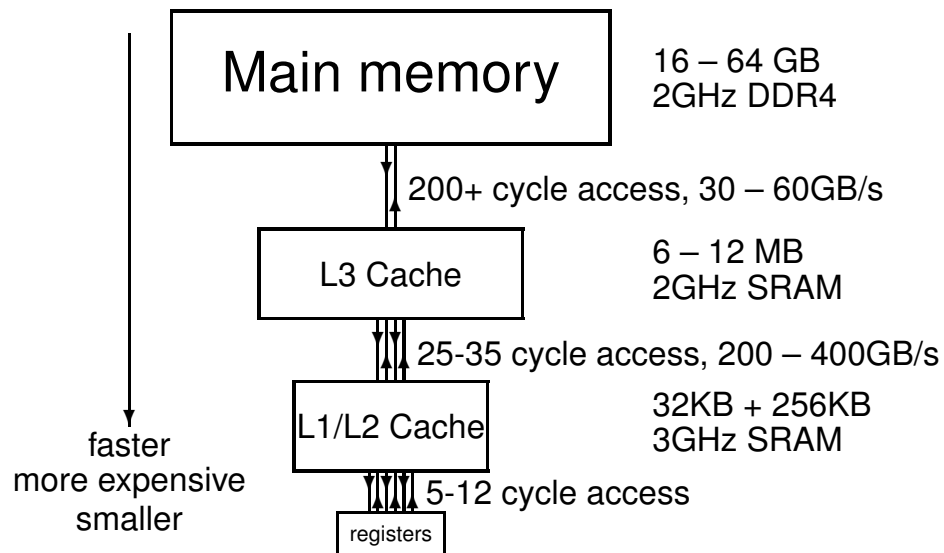
Key challenge in modern computer architecture

- 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

Lecture 2 – p. 1

Lecture 2 – p. 2

CPU Memory Hierarchy



Lecture 2 – p. 3

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

Lecture 2 – p. 4

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

Lecture 2 – p. 5

Importance of Locality

Typical workstation:

20 Gflops CPU

40 GB/s memory $\longleftrightarrow$ 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 $\implies$ 200 Mflops

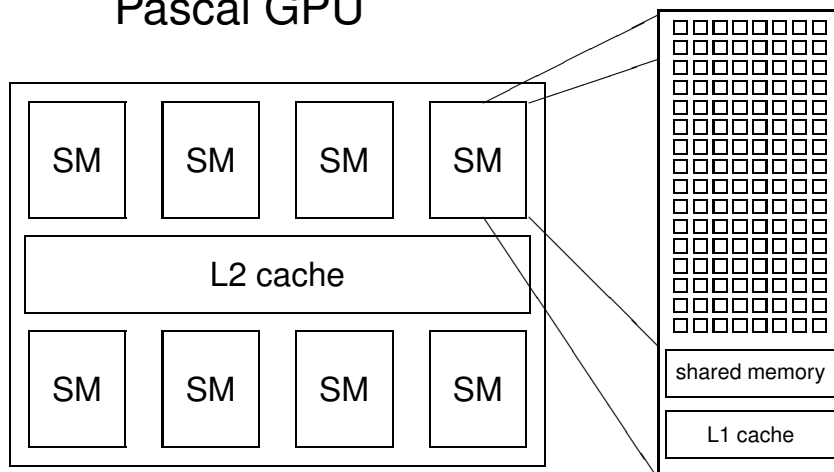
If all 8 variables/line are used, then this increases to 1.6 Gflops.

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

Lecture 2 – p. 6

Pascal

Pascal GPU



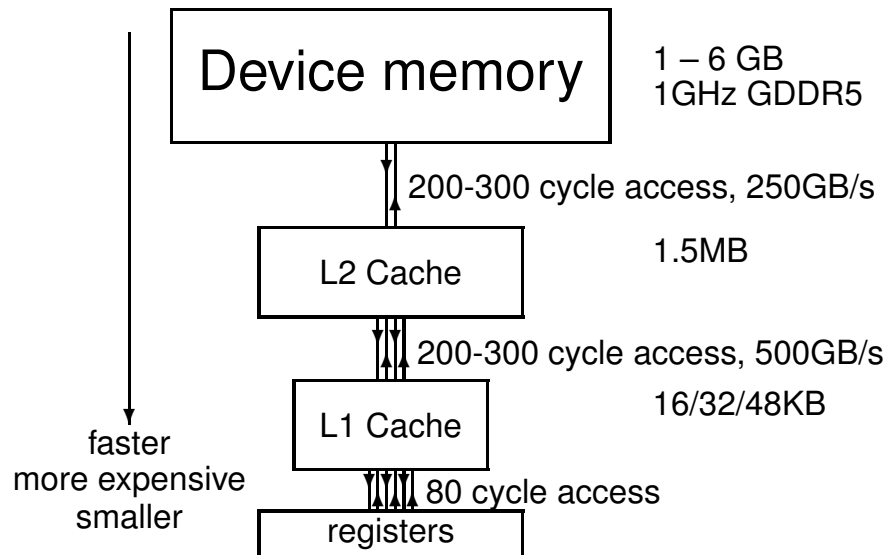
Lecture 2 – p. 7

Pascal

- usually 32 bytes cache line (8 floats or 4 doubles)
- P100: 4092-bit memory path from HBM2 device memory to L2 cache $\implies$ up to 720 GB/s bandwidth
- GeForce cards: 384-bit memory bus from GDDR5 device memory to L2 cache $\implies$ 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

Lecture 2 – p. 8

GPU Memory Hierarchy



Lecture 2 – p. 9

Importance of Locality

1Tflops GPU

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

32 bytes/line

$250\text{GB/s} \equiv 8\text{G line/s} \equiv 32\text{G double/s}$

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

If all 4 doubles/line are used, increases to 11 Gflops

To get up to 500Gflops needs about 15 flops per double transferred to/from device memory

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

Lecture 2 – p. 10

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

Lecture 2 – p. 11

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

Lecture 2 – p. 12

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

Lecture 2 – p. 13

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(...) {
    ...
}
```

Lecture 2 – p. 14

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

Lecture 2 – p. 15

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

Lecture 2 – p. 16

Constant variables

Only 64KB of constant memory, but big benefit is that each SM has a 8-10KB 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

Lecture 2 – p. 17

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++

single $\times$ double = double

Lecture 2 – p. 18

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] );
    }
}
```

Lecture 2 – p. 19

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 $\implies$ 256 threads
- max threads $\implies$ 32 registers per thread
- big difference between “fat” and “thin” threads

Lecture 2 – p. 20

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 be 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

Lecture 2 – p. 21

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];  
}
```

Lecture 2 – p. 22

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;  
}
```

Lecture 2 – p. 23

Local arrays

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

- 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

Lecture 2 – p. 24

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

Lecture 2 – p. 25

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

Lecture 2 – p. 27

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)

Lecture 2 – p. 26

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

At the hardware level, uses special read instructions which give better performance.

Lecture 2 – p. 28

Non-blocking loads/stores

What happens with the following code?

```
__kernel 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 doesn't block unless,
or until, danger of modification

Lecture 2 – p. 29

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

Lecture 2 – p. 30

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

Lecture 2 – p. 31

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.

Lecture 2 – p. 32

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 7.5:

- 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

Lecture 3 – p. 1

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];  
}
```

Lecture 3 – p. 3

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

Lecture 3 – p. 2

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);
```

Lecture 3 – p. 4

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
⇒ potentially large loss of performance

Lecture 3 – p. 5

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

Lecture 3 – p. 6

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:   ...
}
```

Lecture 3 – p. 7

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

Lecture 3 – p. 8

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

Lecture 3 – p. 9

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

Lecture 3 – p. 11

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

Lecture 3 – p. 10

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*

Lecture 3 – p. 12

Typical application

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

// synchronisation to ensure this has finished

__syncthreads();

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

Lecture 3 – p. 13

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

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

Lecture 3 – p. 14

Atomic operations

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

```
__shared__ int count;

...

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

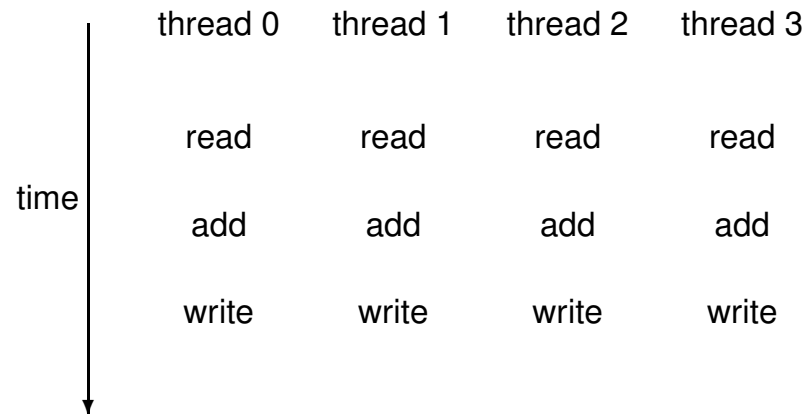
In this case, there is a problem if two (or more) threads try to do it at the same time

Lecture 3 – p. 15

Lecture 3 – p. 16

Atomic operations

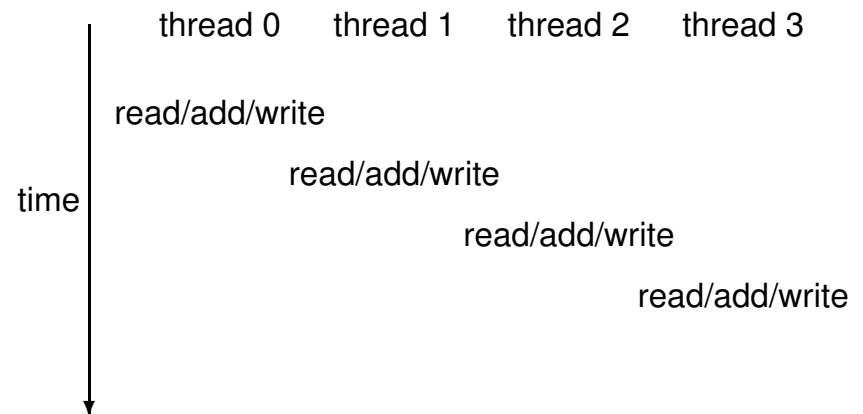
Using standard instructions, multiple threads in the same warp will only update it once.



Lecture 3 – p. 17

Atomic operations

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



Lecture 3 – p. 18

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

These are

- not very fast for data in Kepler shared memory, better in Maxwell and Pascal
- only slightly slower for data in device global memory (operations performed in L2 cache)

Lecture 3 – p. 19

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)

Lecture 3 – p. 20

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;
    }
}
```

Lecture 3 – p. 21

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;
    }
}
```

Lecture 3 – p. 22

__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

Lecture 3 – p. 23

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;
}
```

Lecture 3 – p. 24

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

Lecture 3 – p. 25

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] );

    }
}
```

Lecture 3 – p. 27

Key reading

CUDA Programming Guide, version 7.5:

- 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

Lecture 3 – p. 26

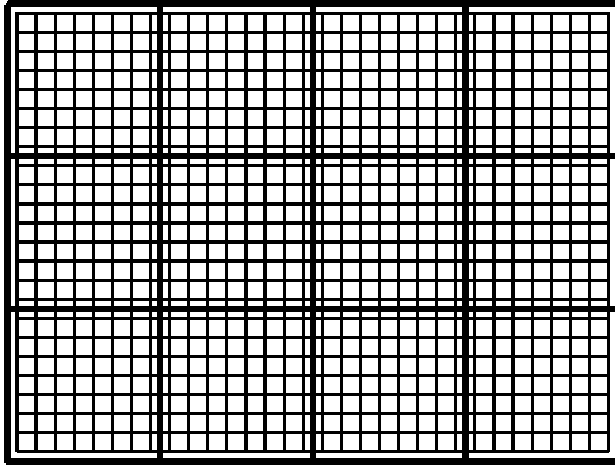
2D Laplace solver

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

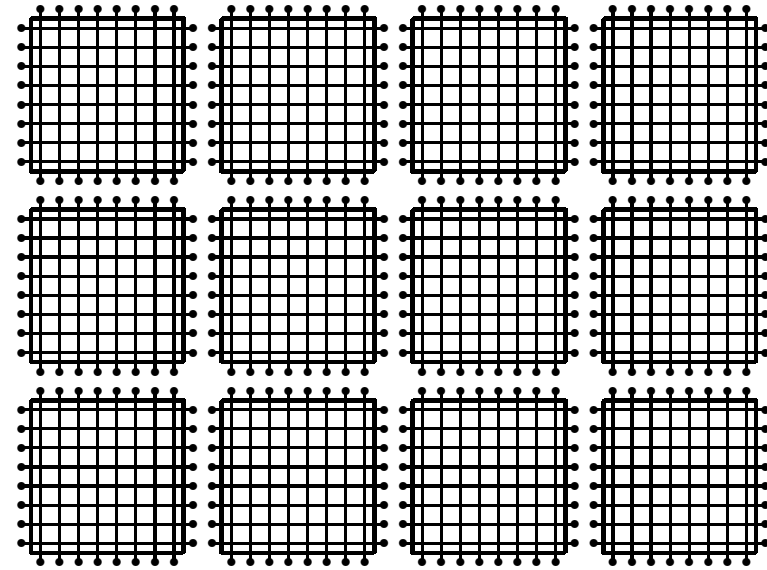
Lecture 3 – p. 28

2D Laplace solver



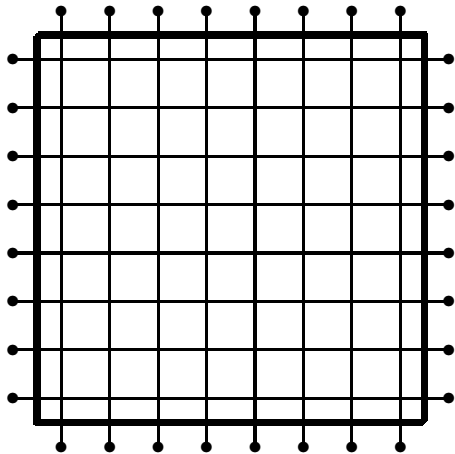
Lecture 3 – p. 29

2D Laplace solver



Lecture 3 – p. 30

2D Laplace solver



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

Lecture 3 – p. 31

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] );
    }
}
```

Lecture 3 – p. 32

2D Laplace solver

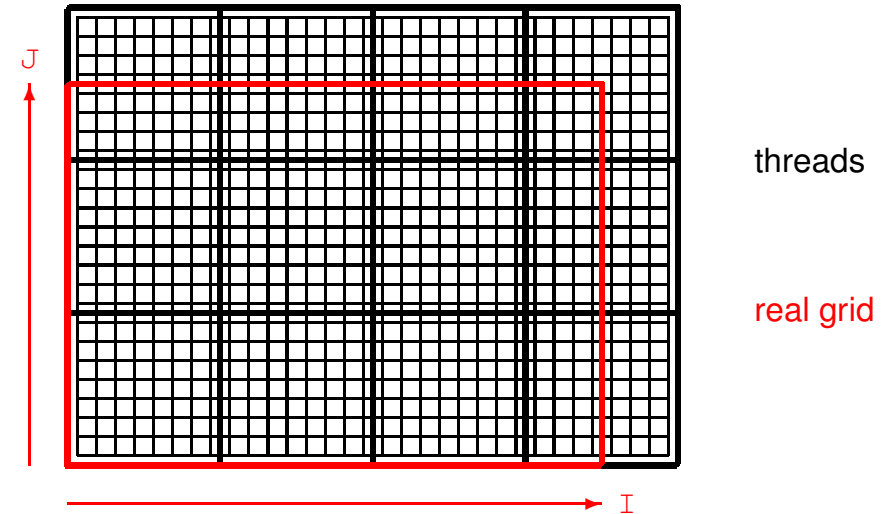
Assumptions:

- 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

Lecture 3 – p. 33

2D Laplace solver



Lecture 3 – p. 34

2D Laplace solver

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

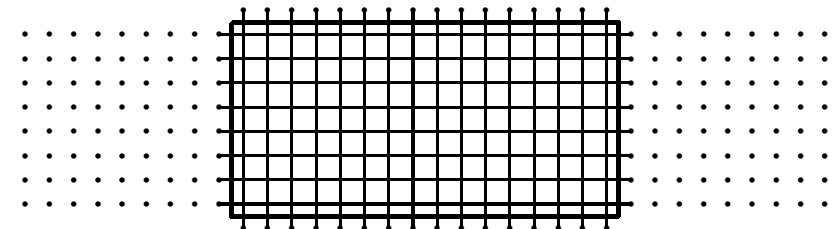
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int id = i + j*I;

    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] );
    }
}
```

Lecture 3 – p. 35

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)

Lecture 3 – p. 36

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

The Kepler architecture introduced a new machine instruction: a warp shuffle

This gives a mechanism for moving data between threads in the same warp, without using any shared memory.

At present it is only for 32-bit data, but 64-bit data can be handled (in software) as a pair of 32-bit shuffles.

Lecture 4 – p. 1

Lecture 4 – p. 2

Warp shuffles

There are 4 variants:

- `_shfl_up`
copy from a lane with lower ID relative to caller
- `_shfl_down`
copy from a lane with higher ID relative to caller
- `_shfl_xor`
copy from a lane based on bitwise XOR of own lane ID
- `_shfl`
copy from indexed lane ID

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

Lecture 4 – p. 3

Warp shuffles

```
int __shfl_up(int var, unsigned int delta);
```

- `var` is a local register variable
- `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

```
int __shfl_down(int var, unsigned int delta);
```

- defined similarly

Lecture 4 – p. 4

Warp shuffles

```
int __shfl_xor(int 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

```
int __shfl(int var, int srcLane);
```

- copies data from `srcLane`

Lecture 4 – p. 5

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.

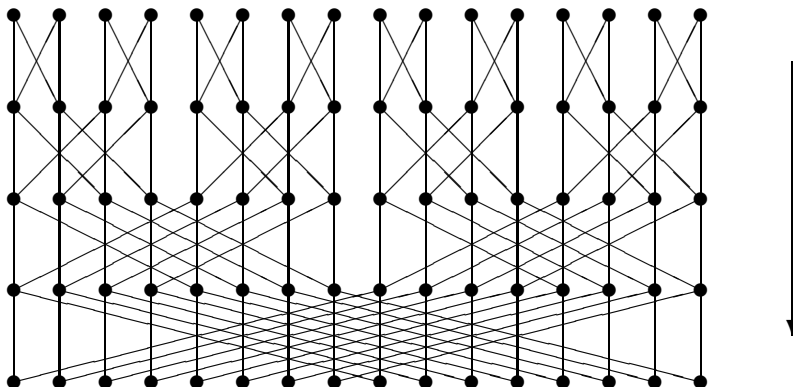
This means you must be very careful with conditional code.

Lecture 4 – p. 6

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(value, i);
```

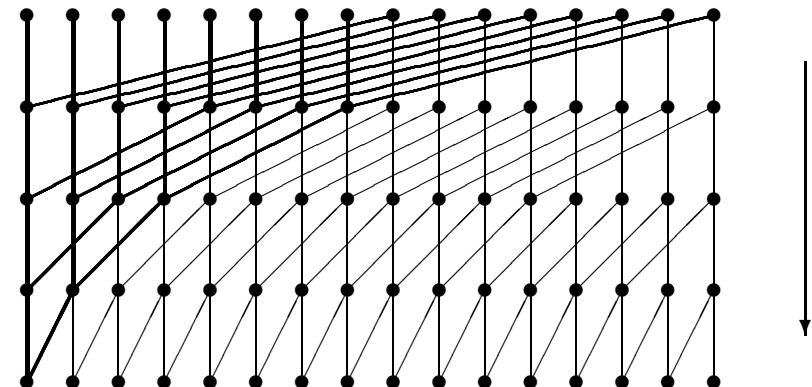


Lecture 4 – p. 7

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(value, i);
```



Lecture 4 – p. 8

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

Lecture 4 – p. 9

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

Lecture 4 – p. 11

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.)

Lecture 4 – p. 10

Local reduction

The first phase is constructing 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 $\implies$ 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

Lecture 4 – p. 12

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.

Lecture 4 – p. 13

Local reduction

Question 3: where should data be held?

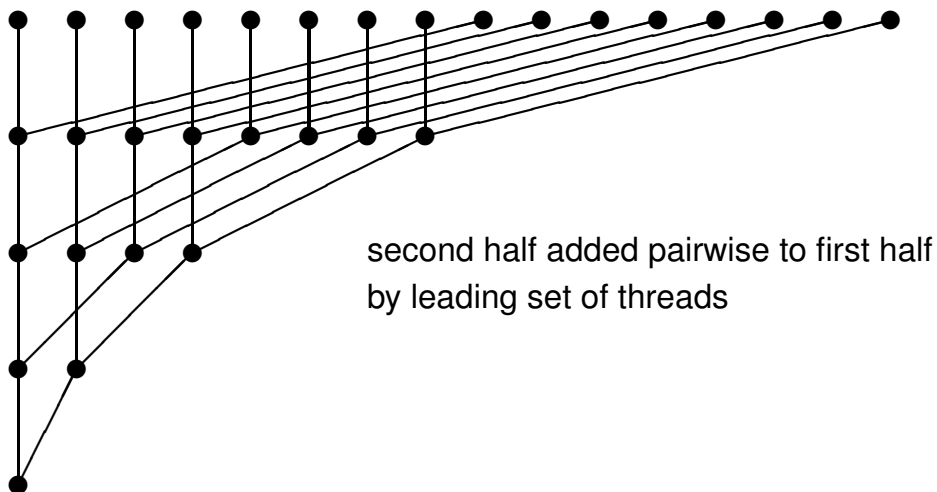
Threads need to access results produced by other threads:

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

Lecture 4 – p. 14

Local reduction

Pictorial representation of the algorithm:



Lecture 4 – p. 15

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];
}
```

Lecture 4 – p. 16

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`

Lecture 4 – p. 17

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

Lecture 4 – p. 18

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]);
```

Lecture 4 – p. 19

Global reduction: version 2

More general reduction operations could 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  
}
```

Lecture 4 – p. 20

Scan operation

Given an input vector u_i , $i = 0, \dots, I-1$, 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

Lecture 4 – p. 21

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

Lecture 4 – p. 23

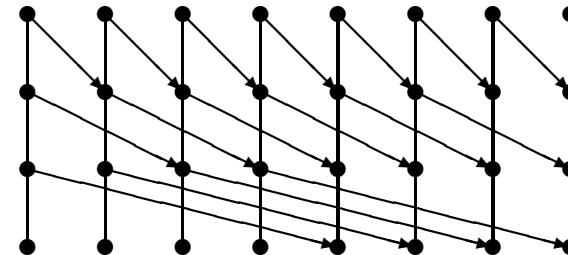
Scan operation

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

Lecture 4 – p. 22

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
 $\Rightarrow O(N \log N)$ operations in total

Lecture 4 – p. 24

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;
    }

    ...
}
```

Lecture 4 – p. 25

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(temp1,d);
        if (tid%32 >= d) temp1 += temp2;
    }

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

Lecture 4 – p. 27

Local scan: version 2

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

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

    for (int d=1; d<32; d<=1) {
        temp3 = __shfl_up(temp2,d);
        if (tid%32 >= d) temp2 += temp3;
    }

    if (tid < blockDim.x/32) temp[tid] = temp2;
}

__syncthreads();
```

Lecture 4 – p. 28

Local scan: version 2

Finally, we add the sum of previous warps:

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

Lecture 4 – p. 29

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

Lecture 4 – p. 30

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

Lecture 4 – p. 31

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.

Lecture 4 – p. 32

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

Lecture 4 – p. 33

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.

Lecture 4 – p. 35

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 p.
}
```

Lecture 4 – p. 34

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.

Lecture 4 – p. 36

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

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.

Lecture 4 – p. 37

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 4 – p. 38

Lecture 5: libraries and tools

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Lecture 5 – p. 1

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 product
 - 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

Lecture 5 – p. 3

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

• 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
- 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

Lecture 5 – p. 2

CUDA libraries

• cuFFT

- Fast Fourier Transform
- 1D, 2D, 3D
- significant input from Satoshi Matsuoka and others at Tokyo Institute of Technology
www.voltaire.com/assets/files/Case_studies/titech_case_study_final_for_SC08.pdf
- has almost all of the variations found in FFTW and other CPU libraries?

Lecture 5 – p. 4

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

Lecture 5 – p. 5

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)

Lecture 5 – p. 6

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

• cuSOLVER: brand new in CUDA 7.0

- Key LAPACK dense solvers, 3 – 6x faster than MKL
- Sparse direct solvers, 2–14x faster than CPU equivalents

Lecture 5 – p. 7

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
- see presentation given at NVIDIA's 2013 GTC conference:

<http://on-demand.gputechconf.com/gtc/2013/presentations/S3579-High-Performance-Algebraic-Multigrid-Commercial-Apps.pdf>

- available from

<http://developer.nvidia.com/amgx>

Lecture 5 – p. 8

CUDA Libraries

- **cuDNN**
 - library for Deep Neural Networks
 - some parts developed by Jeremy Appleyard (NVIDIA) working in Oxford
- **CUDA math library**
 - lots of special functions
- **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**

Lecture 5 – p. 9

Kokkos

- new C++ library developed by US Sandia National Lab
- aims for platform-independent application code with performance portability (i.e. good performance on all many-core platforms: GPU, Xeon Phi, others)
- uses “lambda” expressions, and abstracts data layouts
- to learn more see the 2015 GTC presentation

<http://on-demand.gputechconf.com/gtc/2015/presentation/S5166-H-Carter-Edwards.pdf>

Lecture 5 – p. 11

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

Lecture 5 – p. 10

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`

Lecture 5 – p. 12

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

Lecture 5 – p. 13

Other libraries

- ArrayFire from Acclereyes:
 - 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.acclereyes.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

Lecture 5 – p. 14

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.

Lecture 5 – p. 15

The 7 dwarfs

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

Lecture 5 – p. 16

Dense linear algebra

- cuBLAS
- cuSOLVER
- MAGMA
- ArrayFire

Lecture 5 – p. 17

Spectral methods

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

Lecture 5 – p. 19

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

Lecture 5 – p. 18

N-body methods

- OpenMM
 - 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

Lecture 5 – p. 20

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

Lecture 5 – p. 21

Monte Carlo

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

Lecture 5 – p. 23

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

Lecture 5 – p. 22

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

Lecture 5 – p. 24

Tools

Other languages:

- **FORTRAN:** PGI (Portland Group) CUDA FORTRAN compiler with natural FORTRAN equivalent to CUDA C
- **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://mathematician.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>
<http://chimera.labs.oreilly.com/books/1230000000929/ch06.html>

Lecture 5 – p. 25

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

Lecture 5 – p. 27

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

Lecture 5 – p. 26

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 5 – p. 28

Lecture 6: odds and ends

Prof. Mike Giles

`mike.giles@maths.ox.ac.uk`

Oxford University Mathematical Institute

Oxford e-Research Centre

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

Lecture 6 – p. 1

Lecture 6 – p. 2

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 CUDA 7.5
- 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

Lecture 6 – p. 3

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

Lecture 6 – p. 4

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

Lecture 6 – p. 5

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

Lecture 6 – p. 7

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

Lecture 6 – p. 6

Host code

Simple/default behaviour:

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

Lecture 6 – p. 8

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

Lecture 6 – p. 9

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

Lecture 6 – p. 10

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)

Lecture 6 – p. 11

Multiple Streams

Quoting from section 3.2.5.5 in the CUDA 7.5 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.

Lecture 6 – p. 12

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

Lecture 6 – p. 13

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

Lecture 6 – p. 15

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`

Lecture 6 – p. 14

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 the default stream
    cudaMemcpy(d_data, h_data, sizeof(float), cudaMemcpyHostToDevice);
    kernel<<<1, 1>>>(d_data, 0);
}

cudaDeviceSynchronize();
```

Lecture 6 – p. 16

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

Lecture 6 – p. 17

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();
```

Lecture 6 – p. 18

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

Lecture 6 – p. 19

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

Lecture 6 – p. 20

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 occurred
- `cudaEventElapsedTime(time, event1, event2)`

Lecture 6 – p. 21

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
- current device can be set by using `cudaSetDevice`
- `cudaGetDeviceProperties` does what it says
- each stream is associated with a particular device
– must also be the current device when doing a kernel launch or a memory copy
- see `simpleMultiGPU` example in SDK
- see section 3.2.6 for more information

Lecture 6 – p. 22

Multi-GPU computing

- single workstation in a big enclosure
- up to 4 high-end cards in PCIe slots
- 1kW power consumption – not one for the office

A bigger configuration:

- a distributed-memory cluster with multiple nodes, each with
 - 2-4 GPUs
 - 100 Gb/s Infiniband
- PCIe v3 bandwidth of 12 GB/s similar to Infiniband bandwidth

Lecture 6 – p. 23

Multi-GPU computing

The biggest GPU systems currently:

- Tianhe-2 (China)
 - 34 petaflop (#2)
 - Intel Xeon Phi accelerators
- Titan (ORNL)
 - 17 petaflop (#3)
 - Cray XK7 with NVIDIA K20X GPUs
- Blue Waters (UIUC)
 - 11 petaflop
 - Cray XK6 with NVIDIA K20X GPUs
- Piz Daint (CSCS Switzerland)
 - 6.3 petaflop (#8)
 - Cray XC30 with NVIDIA K20X GPUs

Lecture 6 – p. 24

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

Lecture 6 – p. 25

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

Lecture 6 – p. 27

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

Lecture 6 – p. 26

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

Lecture 6 – p. 28

Odds and ends

Appendix B.2.4: `__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

Lecture 6 – p. 29

Odds and ends

Appendix E.2.2.3: `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

Lecture 6 – p. 30

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

Lecture 6 – p. 31

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)
```

Lecture 6 – p. 32

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)
```

Lecture 6 – p. 33

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(...)
```

Lecture 6 – p. 35

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

Lecture 6 – p. 34

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 6 – p. 36

Lecture 7: tackling a new application

Prof. Mike Giles

`mike.giles@maths.ox.ac.uk`

Oxford University Mathematical Institute
Oxford e-Research Centre

1) Has it been done before?

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

Lecture 7 – p. 1

Lecture 7 – p. 2

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

Lecture 7 – p. 3

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?

Lecture 7 – p. 4

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

Lecture 7 – p. 5

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?

Lecture 7 – p. 7

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

Lecture 7 – p. 6

Initial planning

Parallel Cyclic Reduction (PCR): starting from

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

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.)

Lecture 7 – p. 8

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

Lecture 7 – p. 9

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 (5GB/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

Lecture 7 – p. 11

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

Lecture 7 – p. 10

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

Lecture 7 – p. 12

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 $\implies$ changes approach to implementation

Lecture 7 – p. 13

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)

Lecture 7 – p. 15

Initial planning

If compute-intensive:

- don't worry (too much) about cache efficiency
- minimise integer index operations – surprisingly costly
- 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

Lecture 7 – p. 14

Initial planning

With complex applications, I increasingly worry about “register pressure”, i.e. coping with a maximum of 63 registers per thread (255 registers on K20/K40):

- split big kernels into two – may increase bandwidth requirements but probably reduces register count
(Example: in Monte Carlo simulations, pre-compute random numbers or do them on-the-fly?)
- if any variables have same value for all threads, put them into shared memory, set by thread 0
- sometimes hard to predict what will work best, may need to experiment later

Lecture 7 – p. 16

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!

Lecture 7 – p. 17

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

Lecture 7 – p. 19

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

Lecture 7 – p. 18

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

Lecture 7 – p. 20

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

Lecture 7 – p. 21

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 generationat 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

Lecture 7 – p. 23

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

Lecture 7 – p. 22

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, has developed a simple flexible automated system to do this – can try it in one of the mini-projects

Lecture 7 – p. 24

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 7.0 profiler is great

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

Lecture 7 – p. 25

Going further

In some cases, a single GPU is not sufficient

Shared-memory option:

- single system with up to 8 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

Lecture 7 – p. 26

Going further

Keep an eye on what is happening with new GPUs:

- Maxwell came out in 2014:
 - consumer cards for PCs and laptops
 - 3× improvement in performance per watt
- Pascal out now:
 - P100 for HPC with great double precision
 - HBM2 memory → improved memory bandwidth
 - NVlink → 4×20GB/s links per GPU for greatly improved GPU-GPU & CPU-GPU bandwidth
 - guest lecture on Friday
- Volta due out in 2017/18

Lecture 7 – p. 27

Going further

Two systems to keep an eye on:

- NVIDIA DGX-1 Deep Learning server – out now
 - 8 NVIDIA GP100 GPUs, each with 16GB HBM2
 - 2 × 20-core Intel Xeons (E5-2698 v4 2.2 GHz)
 - 512 GB DDR4 memory, 8TB SSD
 - 80GB/s NVlink interconnect between the GPUs
- IBM “Minsky” server – out this year
 - 4 NVIDIA GP100 GPUs, each with 16GB HBM2
 - 2 × 12-core IBM Power8+ CPUs, with up to 230GB/s memory bandwidth
 - 80GB/s NVlink interconnect between the GPUs, CPUs and large system memory

Lecture 7 – p. 28

Going further

Intel:

- latest “Broadwell” / “Skylake” CPU architectures
 - some chips have built-in GPU, purely for graphics
 - 4–22 cores, each with a 256-bit AVX vector unit
 - 512-bit vector unit on future high-end Skylake Xeons
- Xeon Phi architecture
 - “Knights Corner”: 50+ cores with a 512-bit vector unit
 - “Knights Landing (KNL)”: up to 72 cores, out now
 - performance comparable to a GPU – 300 watts

ARM:

- already designed OpenCL GPUs for smart-phones
- also 64-bit CPUs with up to 48 cores

Lecture 7 – p. 29

Going further

- Intel is promoting a confusing variety of alternatives for Xeon Phi and multicore CPUs with vector units
 - low-level vector intrinsics
 - OpenCL
 - OpenMP 4.0 directives
 - TBB (thread building blocks)
 - Cilk Plus directives
 - auto-vectorising compiler

Eventually, I think the auto-vectorising compiler with OpenMP 4.0 will be the winner.

Lecture 7 – p. 31

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

Lecture 7 – p. 30

Final words

- exciting times 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

Lecture 7 – p. 32

GPU implementation of explicit and implicit finite difference methods in Finance

Mike Giles

Mathematical Institute, Oxford University
Oxford-Man Institute of Quantitative Finance
Oxford e-Research Centre

Endre László, István Reguly (Oxford)
Julien Demouth, Jeremy Appleyard (NVIDIA)

expanded version of presentation at GTC 2014

July 25th, 2014

GPUs

In the last 6 years, GPUs have emerged as a major new technology in computational finance, as well as other areas in HPC:

- over 1000 GPUs at JP Morgan, and also used at a number of other Tier 1 banks and financial institutions
- use is driven by both energy efficiency and price/performance, with main concern the level of programming effort required
- Monte Carlo simulations are naturally parallel, so ideally suited to GPU execution:
 - ▶ averaging of path payoff values using binary tree reduction
 - ▶ key requirement is parallel random number generation, and that is addressed by libraries such as CURAND

Finite Difference calculations

Focus of this work is finite difference methods for approximating Black-Scholes and other related multi-factor PDEs

- explicit time-marching methods are naturally parallel – again a good target for GPU acceleration
 - implicit time-marching methods usually require the solution of lots of tridiagonal systems of equations – not so clear how to parallelise this
 - key observation is that cost of moving lots of data to/from the main graphics memory can exceed cost of floating point computations
 - ▶ 288 GB/s bandwidth
 - ▶ 4.3 TFlops (single precision) / 1.4 TFlops (double precision)
- ⇒ should try to avoid this data movement

1D Finite Difference calculations

In 1D, a simple explicit finite difference equation takes the form

$$u_j^{n+1} = a_j u_{j-1}^n + b_j u_j^n + c_j u_{j+1}^n$$

while an implicit finite difference equation takes the form

$$a_j u_{j-1}^{n+1} + b_j u_j^{n+1} + c_j u_{j+1}^{n+1} = u_j^n$$

requiring the solution of a tridiagonal set of equations.

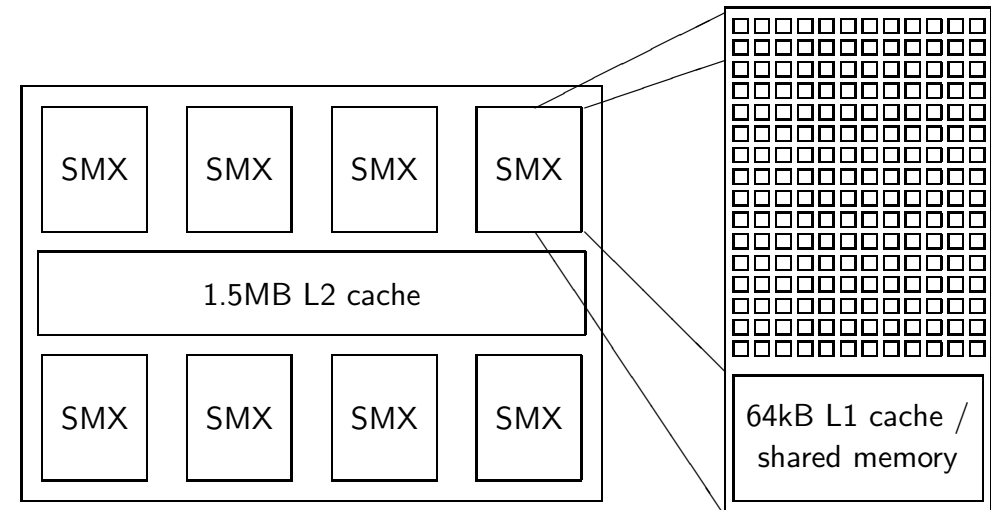
What performance can be achieved?

1D Finite Difference calculations

- grid size: 256 points
- number of options: 2048
- number of timesteps: 50000 (explicit), 2500 (implicit)
- K20 capable of 3.5 TFlops (single prec.), 1.2 TFlops (double prec.)

	single prec.			double prec.		
	msec	GFinsts	GFlops	msec	GFinsts	GFlops
explicit1	347	227	454	412	191	382
explicit2	89	882	1763	160	490	980
implicit1	28	892	1308	80	401	637
implicit2	33	948	1377	88	441	685
implicit3	14	643	1103	30	294	505

NVIDIA Kepler GPU



How is this performance achieved?

Navigation icons: back, forward, search, etc.

1D Finite Difference calculations

Approach for explicit time-marching:

- each thread block (256 threads) does one or more options
- 3 FMA (fused multiply-add) operations per grid point per timestep
- doing an option calculation within one thread block means no need to transfer data to/from graphics memory – can hold all data in SMX

1D Finite Difference calculations

- explicit1 holds data in shared memory
- each thread handles one grid point
- performance is limited by speed of shared memory access, and cost of synchronisation

```
__shared__ REAL u[258];  
...  
utmp = u[i];  
  
for (int n=0; n<N; n++) {  
    utmp = utmp + a*u[i-1] + b*utmp + c*u[i+1];  
    __syncthreads();  
    u[i] = utmp;  
    __syncthreads();  
}
```

Navigation icons: back, forward, search, etc.

1D Finite Difference calculations

explicit2 holds all data in registers

- each thread handles 8 grid points, so each warp (32 threads which act in unison) handles one option
- no block synchronisation required
- data exchange with neighbouring threads uses shuffle instructions (special hardware feature for data exchange within a warp)
- 64-bit shuffles performed using in software (Julian Demouth, GTC 2013)

1D Finite Difference calculations

```
for (int n=0; n<N; n++) {
    um = __shfl_up(u[7], 1);
    up = __shfl_down(u[0], 1);

    for (int i=0; i<7; i++) {
        u0 = u[i];
        u[i] = u[i] + a[i]*um + b[i]*u0 + c[i]*u[i+1];
        um = u0;
    }

    u[7] = u[7] + a[7]*um + b[7]*u[7] + c[7]*up;
}
```

1D Finite Difference calculations

Bigger challenge is how to solve tridiagonal systems for implicit solvers.

- want to keep computation within an SMX and avoid data transfer to/from graphics memory
- prepared to do more floating point operations if necessary to avoid the data transfer
- need lots of parallelism to achieve good performance

Solving Tridiagonal Systems

On a CPU, the tridiagonal equations

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, \dots, N-1$$

would usually be solved using the Thomas algorithm – essentially just standard Gaussian elimination exploiting all of the zeros.

- inherently sequential algorithm, with a forward sweep and then a backward sweep
- would require each thread to handle separate option
- threads don't have enough registers to store the required data – would require data transfer to/from graphics memory to hold / recover data from forward sweep
- not a good choice – want an alternative with reduced data transfer, even if it requires more floating point ops.

Solving Tridiagonal Systems

PCR (parallel cyclic reduction) is a highly parallel algorithm.

Starting with

$$a_i u_{i-1} + u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, \dots, N-1,$$

where $u_j = 0$ for $j < 0, j \geq N$, can subtract multiples of rows $i \pm 1$, and re-normalise, to get

$$a'_i u_{i-2} + u_i + c'_i u_{i+2} = d'_i, \quad i = 0, 1, \dots, N-1,$$

Repeating with rows $i \pm 2$ gives

$$a''_i u_{i-4} + u_i + c''_i u_{i+4} = d''_i, \quad i = 0, 1, \dots, N-1,$$

and after $\log_2 N$ repetitions end up with solution because $u_{i \pm N} = 0$.

Solving Tridiagonal Systems

```
template <typename REAL> __forceinline__ __device__
REAL trid1_warp(REAL a, REAL c, REAL d){
    REAL b;
    uint s=1;
    #pragma unroll
    for (int n=0; n<5; n++) {
        b = __rcp( 1.0f - a*__shfl_up(c,s)
                  - c*__shfl_down(a,s) );
        d = ( d - a*__shfl_up(d,s)
              - c*__shfl_down(d,s) ) * b;
        a = - a*__shfl_up(a,s) * b;
        c = - c*__shfl_down(c,s) * b;
        s = s<<1;
    }
    return d;
}
```

Navigation icons

1D Finite Difference calculations

Using a naive implementation of PCR we would have:

- 1 grid point per thread
- multiple warps for each option, so data exchange via shared memory, and synchronisation required – not ideal
- $O(N \log_2 N)$ floating point operations – quite a bit more than Thomas algorithm

Navigation icons

1D Finite Difference calculations

This leads us to a hybrid algorithm: `implicit1`.

- follows data layout of `explicit2` with each thread handling 8 grid points – means data exchanges can be performed by shuffles
- each thread uses Thomas algorithm to obtain middle values as a linear function of two (not yet known) “end” values

$$u_{J+j} = A_{J+j} + B_{J+j} u_J + C_{J+j} u_{J+7}, \quad 0 < j < 7$$

- the reduced tridiagonal system of size 2×32 for the “end” values is solved using PCR
- total number of floating point operations is approximately double what would be needed on a CPU using the Thomas algorithm (but CPU division is more expensive, so similar Flop count overall?)

Navigation icons

1D Finite Difference calculations

`implicit2` is very similar to `implicit1`, but instead of solving

$$a_j u_{j-1}^{n+1} + b_j u_j^{n+1} + c_j u_{j+1}^{n+1} = u_j^n$$

it instead computes the change $\Delta u_j \equiv u_j^{n+1} - u_j^n$ by solving

$$a_j \Delta u_{j-1} + b_j \Delta u_j + c_j \Delta u_{j+1} = d_j^n$$

and then updates u_j .

This gives better accuracy, which might be important if working in single precision.

1D Finite Difference calculations

Errors:

- `explicit1`, `explicit2` SP errors: 1e-5

could improve a little by changing to

$$u_j^{n+1} = u_j^n + (a_j u_{j-1}^n + b_j u_j^n + c_j u_{j+1}^n)$$

- `implicit1` SP errors: 5e-5
- `implicit2` SP errors: 1e-6
- discretisation errors: 1e-4
- model errors (wrong PDE, wrong coefficients): MUCH larger

Personally, I think single precision is perfectly sufficient, but the banks still prefer double precision.

1D Finite Difference calculations

If the matrices do not change each timestep, then some parts of the tridiagonal solution do not need to be repeated each time.

Impressively, the compiler noticed this in the original version of `implicit1`, and pre-computed as much as it could, at the cost of some additional registers.

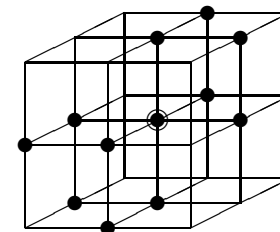
For meaningful performance results for real time-dependent matrices, I stopped this by adding a (zero) time-dependent term on the main diagonal.

However, for applications with fixed matrices, `implicit3` exploits this to pre-compute as much as possible.

3D Finite Difference calculations

What about a 3D extension on a 256^3 grid?

- memory requirements imply one kernel with multiple thread blocks to handle a single option
- kernel will need to be called for each timestep, to ensure that the entire grid is updated before the next timestep starts
- 13-point stencil for explicit time-marching



- implementation uses a separate thread for each grid point in 2D x-y plane, then marches in z-direction

3D Finite Difference calculations

- grid size: 256^3 points
- number of timesteps: 500 (explicit), 100 (implicit)
- K40 capable of 4.3 TFlops (single prec.), 1.4 TFlops (double prec.) and 288 GB/s

	single prec.			double prec.		
	msec	GFlops	GB/s	msec	GFlops	GB/s
explicit1	747	597	100	1200	367	127
explicit2	600	760	132	923	487	144
implicit1	505	360	130	921	235	139

Performance as reported by nvprof, the NVIDIA Visual Profiler

3D Finite Difference calculations

explicit1 relies on L1/L2 caches for data reuse – compiler does an excellent job of optimising loop invariant operations

```

u2[indg] = t23
            + t13
            + (c1_3*S3*S3 - c2_3*S3 - t13 - t23) * u1[indg-KOFF]
            + t12
            + (c1_2*S2*S2 - c2_2*S2 - t12 - t23) * u1[indg-JOFF]
            + (c1_1*S1*S1 - c2_1*S1 - t12 - t13) * u1[indg-IOFF]
            + (1.0f - c3 - 2.0f*( c1_1*S1*S1 + c1_2*S2*S2 + c1_3*S3*S3
                                - t12 - t13 - t23 ) ) * u1[indg]
            + (c1_1*S1*S1 + c2_1*S1 - t12 - t13) * u1[indg+IOFF]
            + (c1_2*S2*S2 + c2_2*S2 - t12 - t23) * u1[indg+JOFF]
            + t12
            + (c1_3*S3*S3 + c2_3*S3 - t13 - t23) * u1[indg+KOFF]
            + t13
            + t23
            * u1[indg+KOFF+IOFF]
            * u1[indg+KOFF+JOFF];
    
```

Mike Giles (Oxford University) PDEs on GPUs July 25th, 2014 21 / 33

3D Finite Difference calculations

explicit2 uses extra registers to hold values which will be needed again

```

u =
    t23
    + t13
    + (c1_3*S3*S3 - c2_3*S3 - t13 - t23) * u1_mm;

u1_mm = u1[indg-JOFF-IOFF];
u1_om = u1[indg-JOFF];
u1_mo = u1[indg-IOFF];
u1_pp = u1[indg+IOFF+JOFF];

u = u
    + t12
    + (c1_2*S2*S2 - c2_2*S2 - t12 - t23) * u1_mm
    + (c1_1*S1*S1 - c2_1*S1 - t12 - t13) * u1_om
    + (1.0f - c3 - 2.0f*( c1_1*S1*S1 + c1_2*S2*S2 + c1_3*S3*S3
                        - t12 - t13 - t23 ) ) * u1_mo
    + (c1_1*S1*S1 + c2_1*S1 - t12 - t13) * u1_pp
    + (c1_2*S2*S2 + c2_2*S2 - t12 - t23) * u1_po
    + t12
    * u1_pp;

indg += KOFF;
u1_m = u1_oo;
u1_oo = u1[indg];
u1_po = u1[indg+IOFF];
u1_op = u1[indg+JOFF];

u = u
    + (c1_3*S3*S3 + c2_3*S3 - t13 - t23) * u1_oo
    + t13
    + t23
    * u1_po;
    
```

Mike Giles (Oxford University) PDEs on GPUs July 25th, 2014 23 / 33

3D Finite Difference calculations

For implicit time-marching, the ADI discretisation requires the solution of a tridiagonal equations along each line in the x-direction, and then the same in the y- and z-directions.

implicit1 is based on library software being written by Endre László, István Reguly and Jeremy Appleyard (NVIDIA), based on the 1D hybrid PCR code - - better than the Thomas method because it involves much less data transfer to/from graphics memory.

The clever part of the implementation is in the data transpositions required to maximise bandwidth – a bit like transposing a matrix.

Mike Giles (Oxford University) PDEs on GPUs July 25th, 2014 24 / 33

3D Finite Difference calculations

The `implicit1` code has the following structure:

- kernel similar to explicit kernel to produce r.h.s.
- separate kernel for tridiagonal solution in each coordinate direction

Fairly balanced between computation and communication, provided attention is paid to maximising data coalescence.

In x-direction:

- each warp handles one tri-diagonal system
- data is contiguous in global memory
- data is loaded in coalesced way, then transposed in shared memory so each thread gets 8 contiguous elements
- care is taken to avoid shared memory bank conflicts
- process is reversed when storing data

Data transposition

What is the problem?

In this application, a warp of 32 threads wants to load in an array of 256 elements:

- natural coalesced load means warp reads in first 32, then next 32, and so on
- thread 0 ends up with elements 0, 32, 64, 96, 128, ...
- but, for hybrid PCR algorithm, thread 0 needs elements 0, 1, 2, 3, 4, 5, 6, 7
- so, how does data get re-arranged?
- more generally, how do we handle it with I elements per thread?
- same problem arises in applications where 32 threads want to load 32 objects (structs) each consisting of I contiguous elements

Navigation icons

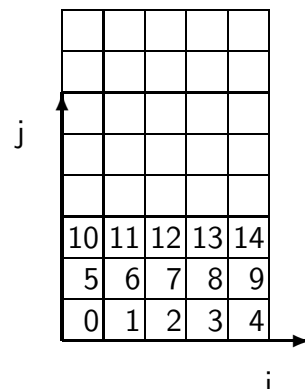
Shared memory

If I is odd, it can be done very simply using shared memory.

If j is the thread index, then

```
for i=0, I-1 do
  load global array element  $j + 32 * i$ 
  write into shared memory  $j + 32 * I$ 
end for

for i=0, I-1 do
  read from shared memory  $i + j * I$ 
end for
```



Key point is that in final read, each thread in the warp is reading from a different shared memory bank – there are 32 of these.

Navigation icons

Navigation icons

- physically, the shared memory hardware is organised into 32 memory banks
- data for shared memory location k is stored in bank $k \bmod 32$
- if two threads in the warp read different data from the same memory bank then it takes longer (the requests are handled sequentially)
- generally not a major concern, but in a worst case it can perform very poorly
- no bank conflict for odd I because $j=32$ is first positive integer with $j \bmod 32 = 0$ producing a bank conflict with $j=0$.
- when I is a power of 2, there is a problem since $j=0, 32/I, 64/I \dots$ all access the same bank – particularly bad for large I

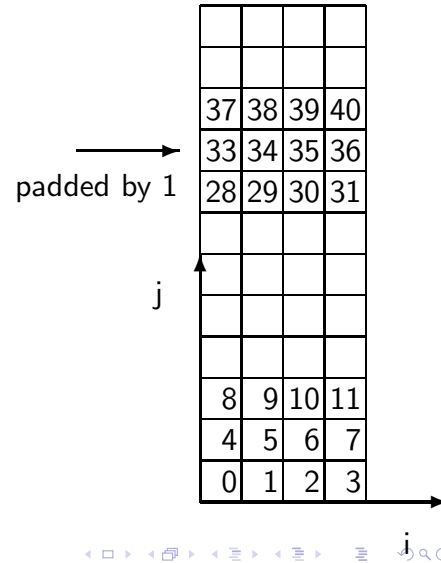
Navigation icons

Data transposition

When I is a power of 2, we avoid bank conflicts by padding the shared memory storage

```
for i=0, I-1 do
  k = j + 32 * i
  load global array element k
  write into shared memory k+k/32
end for

for i=0, I-1 do
  k = i + j * I
  read from shared memory k+k/32
end for
```



Data transposition

- this can be extended to general I (neither odd nor a power of 2)
- definitely confusing the first time you learn about it
- not worth worrying about in most applications
- can be important when developing library software to achieve the ultimate in performance

3D Finite Difference calculations

In y and z -directions:

- thread block with 8 warps to handle 8 tri-diagonal systems
- data for 8 systems is loaded simultaneously to maximise coalescence
- each thread gets 8 elements to work on
- data transposition in shared memory so that each warp handles PCR for one tridiagonal system
- then data transposition back to complete the solution and finally store the result
- quite a tricky implementation but it performs very well

Bottom line – distinctly non-trivial, so check out the code on my webpage!

Finite Difference calculations

Other dimensions?

2D:

- if the grid is small (128^2 ?) one option could fit within a single SMX
 - ▶ in this case, could adapt the 1D hybrid PCR method for the 2D ADI solver
 - ▶ main complication would be transposing the data between the x -solve and y -solve so that each tridiagonal solution is within a single warp
- otherwise, will have to use the 3D approach, but with solution of multiple 2D problems to provide more parallelism

4D:

- same as 3D, provided data can fit into graphics memory (buy a K40 with 12GB graphics memory!)

Conclusions

- GPUs can deliver excellent performance for financial finite difference calculations, as well as for Monte Carlo
- some parts of the implementation are straightforward, but others require a good understanding of the hardware and parallel algorithms to achieve the best performance
- some of this work will be built into future NVIDIA libraries (CUSPARSE, CUB?)
- we are now working to develop a program generator to generate code for arbitrary financial PDEs, based on an XML specification

For further info, see software and other details at

http://people.maths.ox.ac.uk/gilesm/codes/BS_1D/

http://people.maths.ox.ac.uk/gilesm/codes/BS_3D/

http://people.maths.ox.ac.uk/gilesm/cuda_slides.html

Outline

Memory optimisations

CUDA Course
István Reguly

- Approaches to optimisation
- How the hardware does it
- Loads in Flight
- Iterative optimisation of a transpose example
 - occupancy
 - coalescing
 - shared memory
 - memory level parallelism

Approaches to optimisation

- Optimising the algorithm: to reduce the amount of work done to compute the answer
 - Or to improve access patterns, expose more parallelism, etc...
 - Most interesting kind of optimisation...
- Execution optimisation
 - Maximising the utilisation of computational resources, given the algorithm we have

Algorithm optimisation

- Reducing complexity ($O(..)$)
 - Always good to take a step back and think about the algorithm
 - First thing to do is figure out on paper which is going to be better
- Cons:
 - Ignores implementation details
 - Constants in big O notation are tricky...
 - Data locality!

Example: matrix operations

	MATRIX-VECTOR	MATRIX-MATRIX
Flop count	$3N^2 \cdot N$	$3N^3 - N^2$
O(.) runtime	$O(N^2)$	$O(N^3)$
Memory ops	$2N^2 + N$	$3N^2$
Op per Access	1	$O(N)$

- Compare matrix-vector (GEMV) and matrix-matrix multiplication (GEMM)

Peak ratios

	PEAK GFLOP/S	PEAK GB/S	OPS/BYTE	OPS/WORD
K80 single	4368	240	~18	~72
K80 double	1456	240	~6	~48
Xeon 2690 single	416	68	~6*	~24*
Xeon 2690 double	208	68	~3*	~24*

- Without a lot of operations per data element, it's going to be bound by bandwidth!

*ignoring caches...

Memory operations

- To have computations and communications in balance, one needs a huge amount of ops/word
 - Rarely the case...
- We need to saturate the device bandwidth to get the best performance
 - Maximise the number of loads in flight

Loads in flight

- One memory load is 32 or 128 bytes (depends on cache config, texture loads are always 32)
- The latency of one transaction is ~400-600 clock cycles
 - We need a whole lot of independent loads in flight: $288 \cdot 1024^3 / 32$ per second...

Matrix transpose

- NxN matrix, rows <-> columns
- Algorithmic analysis:
 - $O(N^2)$
 - No computations



Example in SDK samples: 6_Advanced/transpose
Or transpose.cu on the website

Naive transpose

```
void reference_transpose(int rows, int cols, float *in, float *out) {  
    for (int i = 0; i < rows; ++i) {  
        for (int j = 0; j < cols; ++j) {  
            out[i*rows+j] = in[j*cols+i];  
        }  
    }  
}
```

i loop's iterations are independent!
Why not parallelise it on the GPU?

Naive CUDA implementation

```
__global__ void transpose1(int rows, int cols, float *in, float *out) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int j = 0; j < cols; ++j) {  
        out[i*rows+j] = in[j*cols+i];  
    }  
}
```

Launch 1D grid of 1D blocks, with 256 threads/block
Matrix size $2048^3 \rightarrow 8$ blocks

Runs in 2.1ms on a K80. Is that good?

Performance

KERNEL	FLOAT	DOUBLE
transpose1	15.25 GB/s	29.61 GB/s

Tesla K80

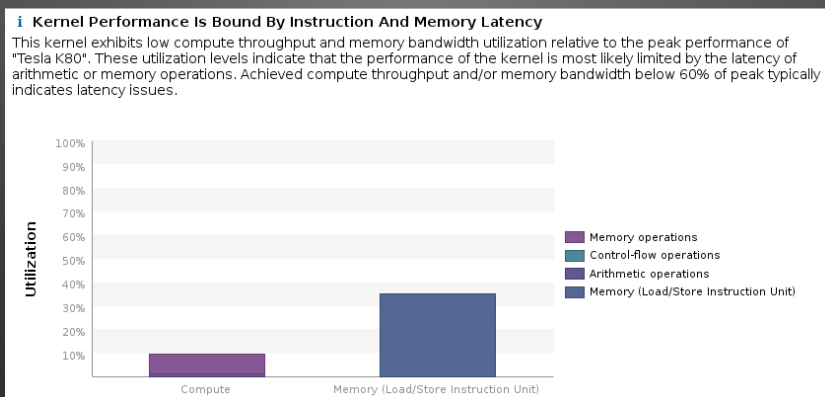
Telling how close we are

- When we know the theoretical bounds:
 - How many loads would be required by the algorithm
 - How many operations would be required by the algorithm
 - bytes/sec or flops/s
- Compare to theoretical

Get info out of the profiler

- Gives metrics for throughput from L2 and DRAM
 - dram_read_throughput
 - l2_l1_read_throughput
 - l2_tex_read_throughput
- These provide numbers from the L2/DRAM perspective
 - Includes cache misses, that could have been hits
 - Includes bandwidth used moving bytes not used by the algorithm
 - Includes ECC
- Not really indicative of real performance...

Profiling naive version



Latency analysis

- We have 8 blocks of 256 threads, each doing only one load or store at the same time
- But we have 13 SMX units available
- Far too few loads in flight!

⚠ Grid Size Too Small To Hide Compute And Memory Latency

The kernel does not execute enough blocks to hide memory and operation latency. Typically the kernel grid size must be large enough to fill the GPU with multiple "waves" of blocks. Based on theoretical occupancy, device "Tesla K80" can simultaneously execute 8 blocks on each of the 13 SMs, so the kernel may need to execute a multiple of 104 blocks to hide the compute and memory latency. If the kernel is executing concurrently with other kernels then fewer blocks will be required because the kernel is sharing the SMs with those kernels.

Optimization: Increase the number of blocks executed by the kernel.

[More...](#)

Occupancy and bandwidth

- We have 8 blocks of 256 threads each, so we can store at the same time
- But we have 13 SMX units
- Far too few loads in flight

transpose1(int, int, float*, float*)	
Start	319.104 ms (319,104, ...)
End	321.101 ms (321,100, ...)
Duration	1.997 ms (1,996,711 r ...)
Grid Size	[8,1,1]
Block Size	[256,1,1]
Registers/Thread	17
Shared Memory/Block	0 B
Occupancy	
Achieved	12.5%
Theoretical	100%
Shared Memory Configu...	
Shared Memory Requ...	112 KiB
Shared Memory Exec...	112 KiB
Shared Memory Bank...	4 B

- The memory controllers can handle X number of transactions per second
- Golden rule: always keep the “bottleneck” resource busy
 - Should be purely bandwidth bound
- More independent pieces of work, more threads, more parallelism, more simultaneous loads -> more loads in flight
- GPU is good at throughput, not latency (unlike CPUs)
 - Only way to hide latency is through parallelism - some threads wait, others have work to do
 - ->Occupancy is the only way to hide latency (well...)

Find more parallelism

- Profiler says we need a bigger computational grid
 - so we need more parallelism
- We need to find more parallelism
- Have each thread do less, distribute it across more threads
 - Let's parallelise both loops and use a 2D grid!

2D transpose

```
__global__ void transpose2(int rows, int cols, float *in, float *out) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    out[i*rows+j] = in[j*cols+i];
}
```

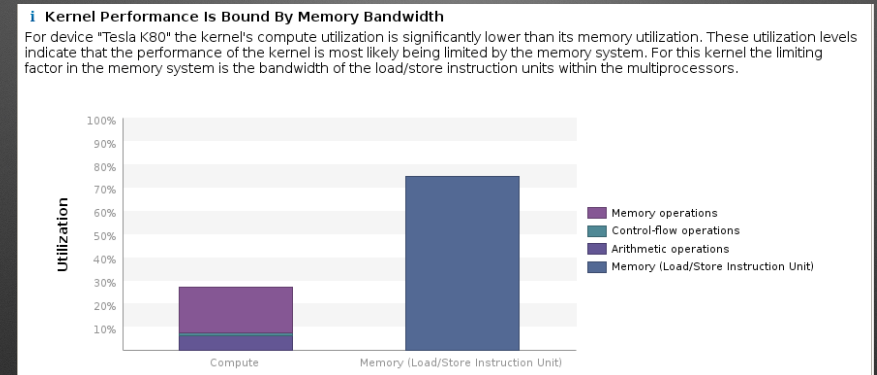
- Launch a 2D grid, with 2048 total threads in each dimension

Performance

KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
Improvement	3.5x	3.2x

Tesla K80

Profiling 2D version



Bandwidth analysis

Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

[More.](#)

Line / File | transpose.cu - /home/ireguly/projects/cuda_course

24 | Global Store L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [4194304 L2 transactions for 131072 total executions]

Line 24: `out[i*rows+j] = in[j*cols+i];`

Only complains about store access pattern!

Mapping programming model to hardware

- You launch a thread block, with each thread executing the same code
- Each block gets assigned to an SM
- An SM has 192/128 little CUDA cores - but these are not independent
- Threads are bundled together into groups of 32 called warps
 - Threads in a 2D thread block are mapped to warps in a row-contiguous way
 - All threads in a warp are executing the same instruction, just with different data (lockstep)

Warp execution

- 1 program counter per warp (there are four on an SMM/SMX)
- The next instruction is issued to 32 parallel CUDA cores
 - Each has its own set of registers
 - So same instruction, just with different operands
 - **Note: even the register indexes are the same for a given instruction, just their contents are different**
- Repeat until done

Executing a load instruction

- When encountering a load instructions, all 32 threads will load from individual addresses in memory
- LD.E R2,[R6]

THR	0	1	2	3	4	5	6	...
R6	0X0FD3A0	0X0FD3A4	0X0FD3A8	0X0FD3AB	0X0FD3B0	0X0FD3B4	0X0FD3B8	

Executing a load instruction

- From the point of view of the thread
 - Asks for a small chunk of memory (4-8 bytes)
 - `f=array[offset+threadIdx.x]`
 - Called a **request** or **access**
- From the SM's perspective
 - **Issues** a load for the cache line that contains `array[offset+threadIdx.x]`
 - May need to issue multiple loads to satisfy all requests from all threads in the warp
 - After the first, each further cache line load is a **replay**
- From the L2/DRAM perspective
 - If the segment is valid in L2, return it from there
 - Otherwise fill L2 from DRAM and send to value to SM

Views of bandwidth

- Thread's view
 - How much data the thread receives per unit time
 - Usually very low number - but there are a lot of threads!
- SM's view
 - How much data is delivered to all the threads active on the SM
 - Receives cache lines
- L2/DRAM view
 - How much data is moving between DRAM and L2 cache

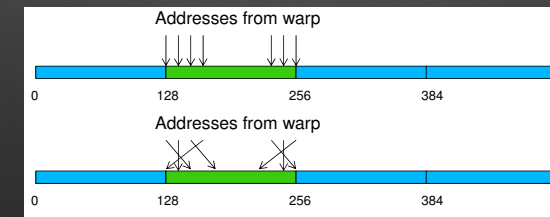
Aggregate bandwidth from thread perspective
is not the as from L2/DRAM perspective!

Memory transactions and coalescing

- Access to global memory triggers transactions
 - size of 32 or 128 bytes
 - aligned to line length
 - always fully R/W
- Degree of coalescing: #of bytes requested/#of bytes used
 - more memory read than used -> performance penalty

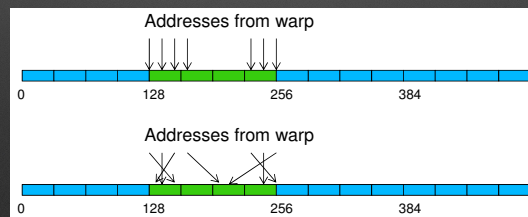
L1-Cached Thread Index Access

- 32 adjacent threads requesting 32 adjacent words
 - aligned but may be permuted
 - all fall in one cache line (128B)
 - 1 transaction



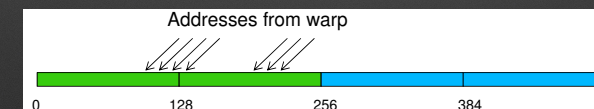
L2-Cached Thread Index Access

- 32 adjacent threads requesting 32 adjacent words
 - all addresses fall within 4 segments
 - 4 transactions



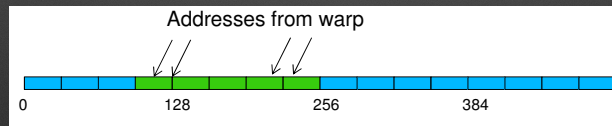
L1-Cache Shifted Access

- 32 adjacent threads requesting 32 adjacent words
 - misaligned
 - 2 transactions
 - Cache line utilisation 50%
- 2D stencil operations
 - In a 2D setting where leading dimension is not a multiple of cache line length - use padding



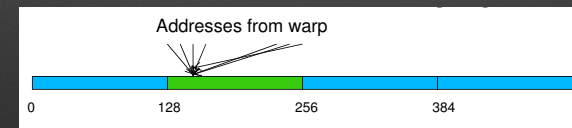
L2-Cached Shifter Access

- 32 adjacent threads requesting 32 adjacent words
 - all addresses fall within 5 segments
 - 5 transactions - 80% utilisation



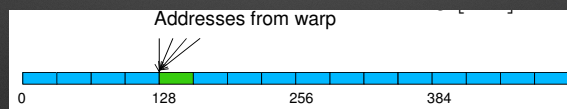
L1-Cached Single Access

- All 32 threads access the same word in memory
- Full 128B cache line is transferred - 3.125% utilisation



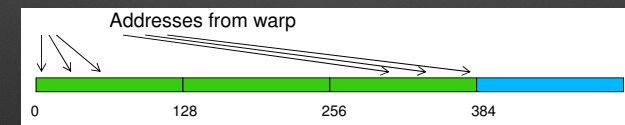
L2-Cached Single Access

- All 32 threads access the same word in memory - same segment
- Full 32B cache line is transferred - 12.5% utilisation



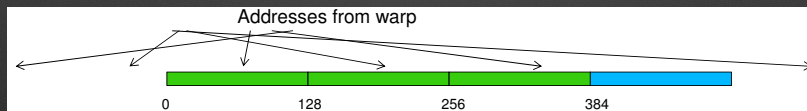
L1-Cache Strided Access

- 32 adjacent threads requesting 32 words with stride 3
 - addresses fall in 3 cache lines
 - 3 transactions
 - Cache line utilisation 33%
- Typical of 3D coordinate or RGB accesses -> use SoA layout!



L1-Cache Fully Random Access

- 32 adjacent threads requesting 32 words with random addresses
 - addresses fall in 32 different cache lines
 - 32 transactions
 - Cache line utilisation 3.125%
- Pointer chasing, trees, etc.



Coalescing

- The way to minimise the number of operations required to satisfy all requests from a single warp's one load instruction
- If multiple addresses are in the same cache line, it only gets moved once
 - The more in the same cache line the better
- Best: 32 addresses, each 4 bytes - a single 128B load
 - One L1-L2 load and 4 L2-DRAM loads
- Worst: separate transactions for each - 31 replays
 - 32 L1-L2 loads and 128 L2-DRAM loads

Good access patterns

- On CPUs, the advice is to do stride-1 accesses

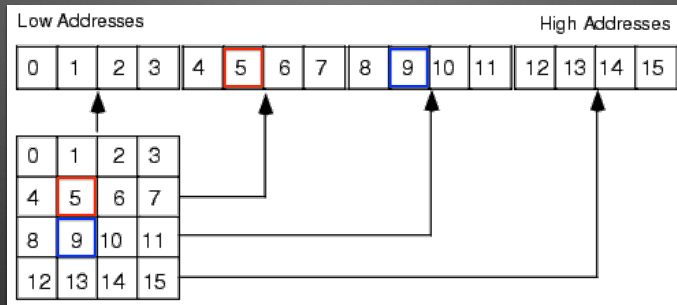
```
for (int i = 0; i < n; ++i) {
    x += data[i];
}
```
- Increases cache hits, easy to compute, prefetching work well, etc...

Good access patterns

- Basically “transposed” to threads in GPU code:

```
for (int i = threadIdx.x; i < n; i+= blockDim.x) {
    x += data[i];
}
```
- Same idea of accessing the same cache line, except we do 32 accesses at the same time
- You need locality across threads for one instruction instead of locality across subsequent instructions for one thread

Back to transpose

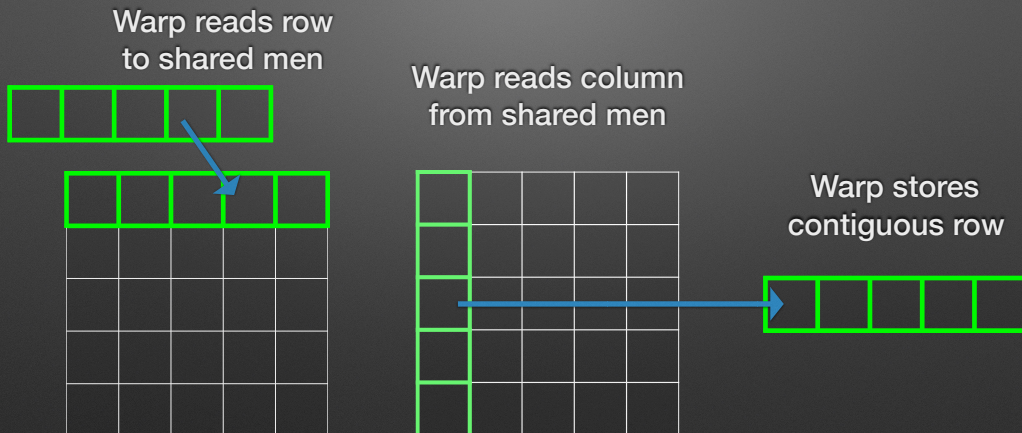


Stride-1 column accesses are stride-N accesses!

Back to transpose

- So what can we do with our transpose code?
- Reads are fine, but writes are bad
- Row read and row write is the most effective - how could we achieve that?
- Transpose in shared memory

Shared memory transpose



Shared memory transpose

```
#define TILE_SIZE 16
__global__ void transpose3(int rows, int cols, float *in, float *out) {
    __shared__ float tile[TILE_SIZE][TILE_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    tile[threadIdx.y][threadIdx.x] = in[j*cols+i];

    __syncthreads();

    i = blockIdx.y * blockDim.y + threadIdx.x;
    j = blockIdx.x * blockDim.x + threadIdx.y;
    out[j*rows+i] = tile[threadIdx.x][threadIdx.y];
}
```

Syncthreads

- Producer-consumer thinking
 - The write consumes the read value
 - With previous examples, it was the same thread
 - With shared memory transpose, one thread produces the value and another consumes it
- We have to make sure it was produced before we try to consume it
 - Great tool if you are running into problems: `cuda-memcheck --tool racecheck`

Performance

KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
transpose3 (coalesced)	80.54 GB/s	102.3 GB/s
Improvement	1.32x	0.93x

Tesla K80

Bandwidth analysis

 **Shared Memory Alignment and Access Pattern**

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.
Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.

Line / File	transpose.cu - /home/ireguly/projects/cuda_course
39	Shared Load Transactions/Access = 4, Ideal Transactions/Access = 1 [524288 transactions for 131072 total executions]

Line 39: `out[j*rows+i] = tile[threadIdx.x][threadIdx.y];`

Shared memory bank conflicts

- Shared memory is organised into banks
 - 32 banks, new bank every 4 bytes (or possibly 8 in Kepler)
 - $\text{Bank} = (\text{address}/4) \% 32$
 - Can read one 32-bit word per bank per clock
 - Warp access: reads 32 words from shared memory per instruction

Shared memory bank conflicts

BANK 1	BANK 2	BANK 3	BANK 4
0	1	2	3
32	33	34	35
64	65	66	67
96	97	98	99

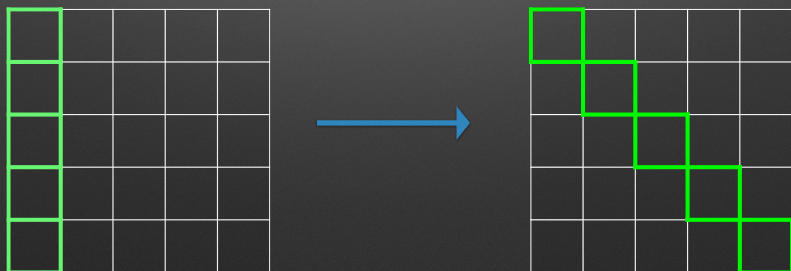
- What happens when we write
 - $\text{idx} = \text{threadIdx.y} * 32 + \text{threadIdx.x}$
 - adjacent banks...
- ...
- What happens when we read
 - $\text{idx} = \text{threadIdx.x} * 32 + \text{threadIdx.y}$
 - same bank...
- Requires replays - 32x the cost!

Impact of replays

- If a warp has to replay instructions, it cannot proceed until all replays are completed - a load may take up to 32x the number of instructions
- Occupies the warp scheduler - no useful operations in the meanwhile
- Decreases number of loads in flight...
- Note: we saw replays for both shared memory bank conflicts and non-coalesced global accesses - different reasons but the same effect

Avoiding bank conflicts

- Problem is with the index computation:
 - $\text{bank} = (\text{threadIdx.x} * 32 + \text{threadIdx.y}) \% 32$, where threadIdx.y is the same of all thread in a warp
- So let's pad our array to size 33:
 - $\text{bank} = (\text{threadIdx.x} * 33 + \text{threadIdx.y}) \% 32$



Padded shared memory

```
#define TILE_SIZE
__global__ void transpose3(int rows, int cols, float *in, float *out) {
    __shared__ float tile[TILE_SIZE][TILE_SIZE+1];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    tile[threadIdx.y][threadIdx.x] = in[j*cols+i];

    __syncthreads();

    i = blockIdx.y * blockDim.y + threadIdx.x;
    j = blockIdx.x * blockDim.x + threadIdx.y;
    out[j*rows+i] = tile[threadIdx.x][threadIdx.y];
}
```

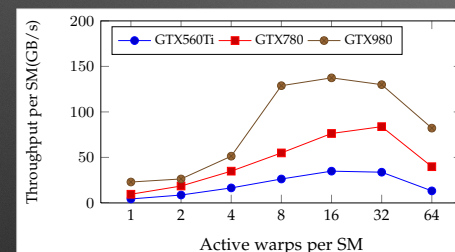
Performance

KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
transpose3 (coalesced)	80.54 GB/s	102.3 GB/s
transpose4 (no bank conf)	127 GB/s	165.7 GB/s
Improvement	1.58x	1.61x

Note: on Maxwell, cost of bank conflicts is 2-5x less

Tesla K80

Shared memory performance



Shared Memory Access Latency with Bank Conflicts					
Bank conflict	2-way	4-way	8-way	16-way	32-way
GTX980	30	34	42	58	90
GTX780	82	96	158	257	484
GTX560Ti	87	162	311	611	1209

XinXin Mei et. al. “Dissecting GPU Memory Hierarchy through Microbenchmarking”

Why is double precision better?

- 127 GB/s vs. 165 GB/s
- Single precision:
 - 1 warp, 32 loads each, 4 bytes each, fully coalesced
 - 1 cache line; 4 DRAM transactions
- Double precision:
 - 1 warp, 32 loads each, 8 bytes each, fully coalesced
 - 2 cache lines; 8 DRAM transactions
- More loads in flight!

Even more loads in flight

- We can create a little loop that transposes multiple values per thread - more independent loads!
- Total number of loads is still the same, but so is the number of active threads!

KERNEL	FLOAT	DOUBLE
transpose 4 (1/thread)	127.8 GB/s	165.7 GB/s
transpose 5 (2/thread)	161.1 GB/s	173.1 GB/s

We can keep increasing it until we run into occupancy problems

Memory level parallelism

- Multiple independent memory load operations in one thread
 - Issue all loads before the value of any is consumed
 - Makes better use of loads in flight
- Related to Instruction Level Parallelism
 - Techniques for achieving it are quite similar
 - Requires independent operations within the thread
 - MLP usually creates some ILP

MLP & ILP

```
LD.E R0, [R8];
IMUL.U32.U32 R6,R5,0x84;
ISCADD R6,R3,R6,0x2;
STS [R6], R0;
BAR.SYNC 0x0;
```

Original

With ILP & MLP

```
LD.E R5, [R14];
IMAD.HI.X R13,R8,0x4,R15;
STS [R9+0x420],R7;
IMAD R14.CC,R8,0x4,R12;
LD.E R7,[R12];
IMAD.HI.X R15,R8,0x4,R13;
STS [R9+0x630],R6;
IMAD R10.CC,R8,0x4,R13;
LD.E R6,[R14];
IMAD.HI.X R11,R4,0x4,R15;
STS [R9+0x840],R5;
...
BAR.SYNC 0x0;
```

Overall performance

KERNEL	FLOAT	DOUBLE
transpose1	16.96 GB/s	33.94 GB/s
transpose2 (2D)	59.8 GB/s	109.5 GB/s
transpose3 (coalesced)	80.54 GB/s	102.3 GB/s
transpose4 (no bank conf)	127 GB/s	165.7 GB/s
transpose5 (multiple elem)	161.1 GB/s	173.1 GB/s
Improvement	9.5x	5.24x

Summary

- Without large amount of data-reuse (ops/byte), codes will be bound by operand delivery
 - Bandwidth and/or Latency
- Bandwidth saturation requires many loads in flight
- Best practices for GPU memory utilisation
 - Address Coalescing: Efficient use of memory system
 - Use shared memory to restructure loads/stores into coalesced patterns
 - Latency hiding
 - Occupancy, Instruction level parallelism

Local memory

- Due to the way the hardware works, for a given instruction, you always need to use the same registers across all threads in a warp
 - Registers cannot be dynamically indexed
- If you have local arrays (per thread arrays)
 - Make sure their size is known at compile time
 - Make sure all threads access the same element of the array at the same time
 - Make sure the index is known at compile time: unroll loops!

Spilling

- If you use too big arrays, or indexing cannot be determined at compile-time, or you artificially restricted the number of registers:
 - The compiler will “spill” registers
- Basically puts them in global memory in a way that will result in coalesced accesses
- On Kepler, by default it is the only thing cached in L1
- On Maxwell by default only cached in L2

Static indexing

```
__global__ void kernel1(float * buf)
{
    float a[2];
    ...
    float sum = a[0] + a[1];
    ...
}

__global__ void kernel2(float * buf)
{
    float a[5];
    ...
    float sum = 0.0f;
    #pragma unroll
    for(int i = 0; i < 5; ++i)
        sum += a[i];
    ...
}
```

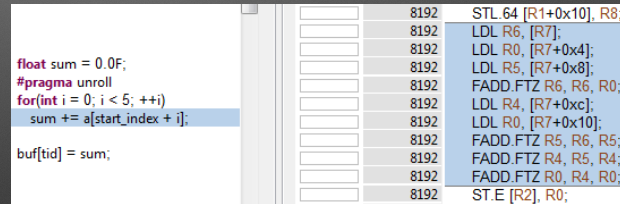
Static indexing - unrolled

float sum = 0.0f;				8192	LD.E R2, [R6+0xc];
#pragma unroll				8192	FADD.FTZ R4, R4, R0;
for(int i = 0; i < 5; ++i)				8192	LD.E R0, [R6+0x10];
sum += a[i];				8192	FADD.FTZ R3, R4, R3;
				8192	FADD.FTZ R3, R3, R2;
buf[tid] = sum;				8192	FADD.FTZ R0, R3, R0;
				8192	ST.E [R6], R0;

Assembly shows, that it was unrolled
Uses a total of 4(!) operations

Dynamic indexing

```
__global__ void kernel3(float * buf, int start_index)
{
    float a[6];
    ...
    float sum = 0.0f;
    #pragma unroll
    for(int i = 0; i < 5; ++i)
        sum += a[start_index + i];
    ...
}
```



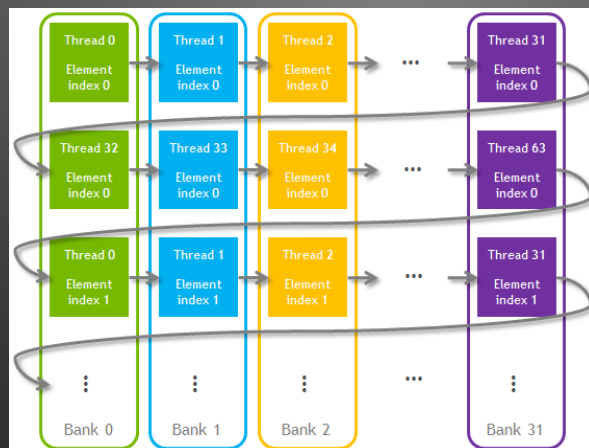
But at least we get nice coalesced accesses...

Dynamic, non-uniform indexing

```
#define ARRAY_SIZE 32
__global__ void kernel4(float * buf, int * indexbuf)
{
    float a[ARRAY_SIZE];
    ...
    int index = indexbuf[threadIdx.x + blockIdx.x * blockDim.x];
    float val = a[index];
    ...
}
```

How many replays?
Based on how many different indexes...

Put it in shared memory



One bank for
each thread

No bank conflicts
guaranteed!

Shared memory code

```
// Should be multiple of 32
#define THREADBLOCK_SIZE 64
// Could be any number, but the whole array should fit into shared memory
#define ARRAY_SIZE 32
```

```
__device__ __forceinline__ int no_bank_conflict_index(int thread_id,
int logical_index)
{
    return logical_index * THREADBLOCK_SIZE + thread_id;
}
```

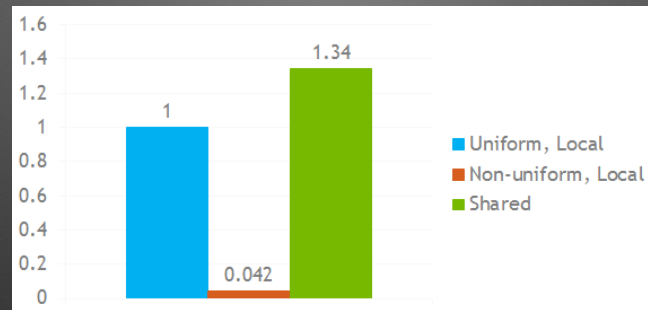
```
__global__ void kernel5(float * buf, int * index_buf)
{
    // Declare shared memory array A which will hold virtual
    // private arrays of size ARRAY_SIZE elements for all
    // THREADBLOCK_SIZE threads of a threadblock
    __shared__ float A[ARRAY_SIZE * THREADBLOCK_SIZE];
    ...
    int index = index_buf[threadIdx.x + blockIdx.x * blockDim.x];

    // Here we assume thread block is 1D so threadIdx.x
    // enumerates all threads in the thread block
    float val = A[no_bank_conflict_index(threadIdx.x, index)];
    ...
}
```

```
float x = A[no_bank_conflict_index(threadIdx.x, index)];
float y = A[no_bank_conflict_index(threadIdx.x, index + 1)];
float z = A[no_bank_conflict_index(threadIdx.x, index + 2)];
float w = A[no_bank_conflict_index(threadIdx.x, index + 3)];
sum = x + y + z + w;
A[no_bank_conflict_index(threadIdx.x, index)] = sum;
```

```
STS [R9], R8;
SHF.L.W R10, RZ, 0x2
LDS R6, [R10+0x100];
LDS R7, [R10];
LDS R4, [R10+0x200];
FADD.FTZ R7, R7, R6;
LDS R3, [R10+0x300];
```

Performance



Sources

XinXin Mei et. al. "Dissecting GPU Memory Hierarchy through Microbenchmarking"

Tony Scuderio, Memory Bandwidth Bootcamp: Best Practices

Tony Scuderio, Memory Bandwidth Bootcamp: Beyond Best Practices

Maxim Milakov: Fast Dynamic Indexing of Private Arrays in CUDA

Jiri Kraus, CUDA Performance Optimisation

Profiling & Tuning Applications

CUDA Course

István Reguly

Introduction

- Why is my application running slow?
- Work it out on paper
- Instrument code
- Profile it
 - NVIDIA Visual Profiler
 - Works with CUDA, needs some tweaks to work with OpenCL
 - nvprof – command line tool, can be used with MPI applications

Identifying Performance Limiters

- CPU: Setup, data movement
- GPU: Bandwidth, compute or latency limited
- Number of instructions for every byte moved
- Algorithmic analysis gives a good estimate
- Actual code is likely different
 - Instructions for loop control, pointer math, etc.
 - Memory access patterns
 - How to find out?
 - Use the profiler (quick, but approximate)
 - Use source code modification (takes more work)

Analysis with Source Code Modification

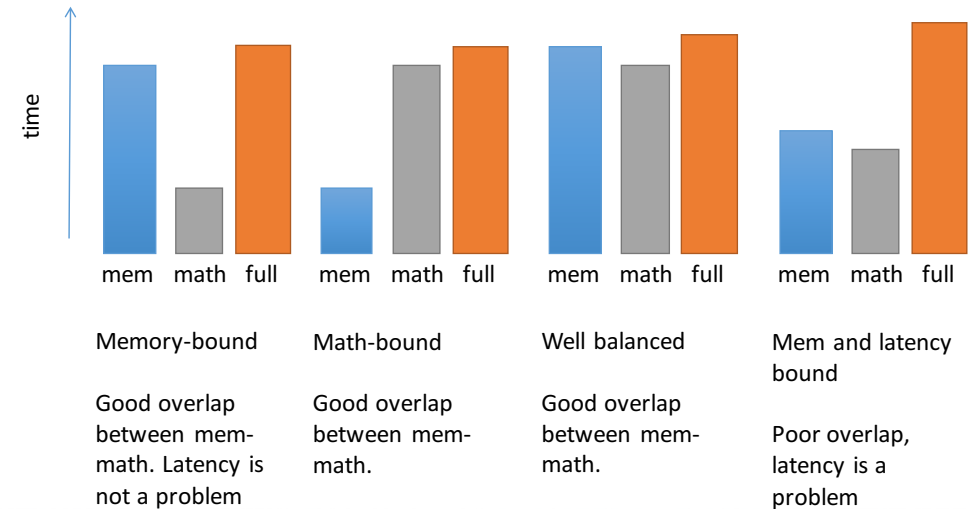
- Time memory-only and math-only versions
 - Not so easy for kernels with data-dependent control flow
 - Good to estimate time spent on accessing memory or executing instructions
- Shows whether kernel is memory or compute bound
- Put an “if” statement depending on kernel argument around math/mem instructions
 - Use dynamic shared memory to get the same occupancy

Analysis with Source Code Modification

```
__global__ void kernel(float *a) {
    int idx = threadIdx.x + blockDim.x*blockIdx.x;
    float my_a;
    my_a = a[idx];
    for (int i=0; i < 100; i++) my_a = sinf(my_a+i*3.14f);
    a[idx] = my_a;
}
```

```
__global__ void kernel(float *a, int prof) {
    int idx = threadIdx.x + blockDim.x*blockIdx.x;
    float my_a;
    if (prof & 1) my_a = a[idx];
    if (prof & 2)
        for (int i=0; i < 100; i++) my_a =
            sinf(my_a+i*3.14f);
    if (prof & 1) a[idx] = my_a;
}
```

Example scenarios



NVIDIA Visual Profiler

- Collects metrics and events during execution
 - Calls to the CUDA API
 - Overall application:
 - Memory transfers
 - Kernel launches
 - Kernels
 - Occupancy
 - Computation efficiency
 - Memory bandwidth efficiency
 - Source-level profiling
- Requires deterministic execution!

Meet the test setup

- 2D gaussian blur with a 5x5 stencil $\frac{1}{273}$
- 4096² grid

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

```
__global__ void stencil_v0(float *input, float *output,
                           int sizex, int sizey) {
```

```
    const int x = blockIdx.x*blockDim.x + threadIdx.x + 2;
    const int y = blockIdx.y*blockDim.y + threadIdx.y + 2;
    if ((x >= sizex-2) || (y >= sizey-2)) return;
    float accum = 0.0f;
    for (int i = -2; i < 2; i++) {
        for (int j = -2; j < 2; j++) {
            accum += filter[i+2][j+2]*input[sizex*(y+j) +
            (x+i)];
        }
    }
    output[sizex*y+x] = accum/273.0f;
}
```

Meet the test setup

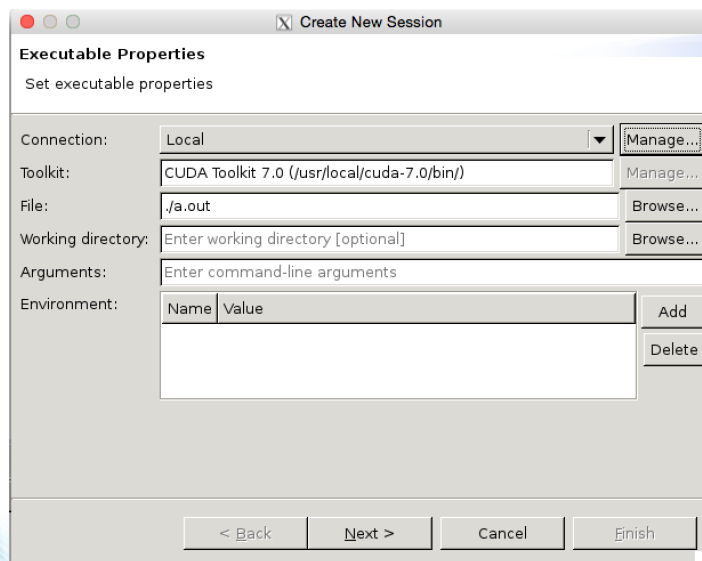
- NVIDIA K40
 - GK110B
 - SM 3.5
 - ECC on
 - Graphics clocks at 745MHz, Memory clocks at 3004MHz

- CUDA 7.0

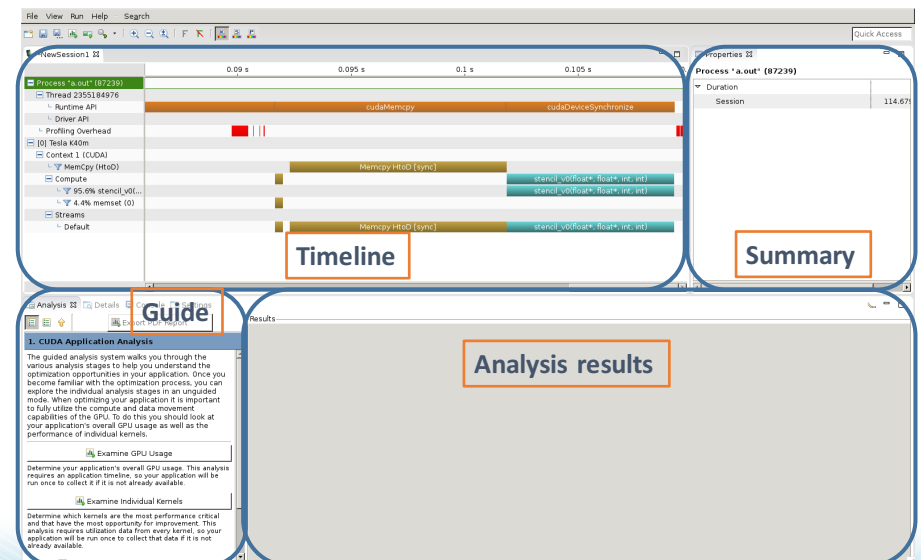
```
nvcc profiling_lecture.cu -O2 -arch=sm_35 -I. -lineinfo -DIT=0
```

Interactive demo of tuning process

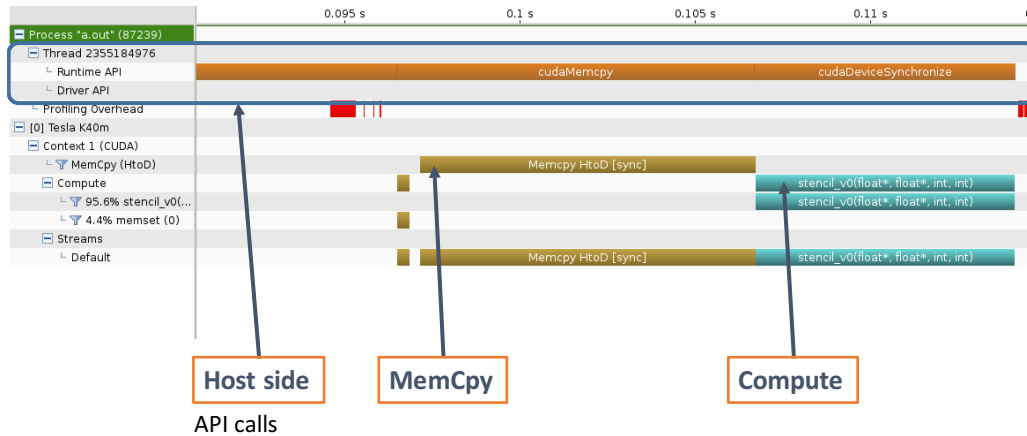
Launch a profiling session



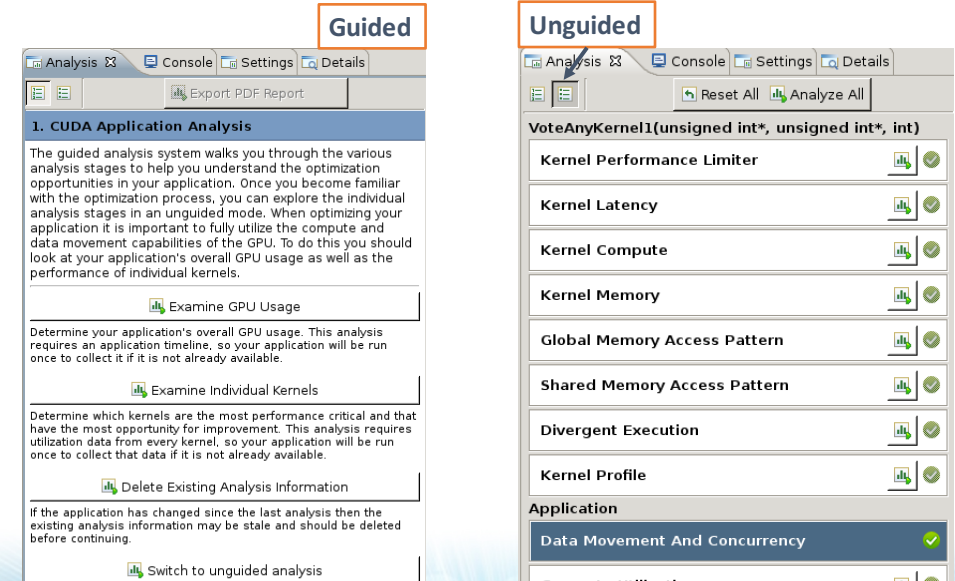
First look



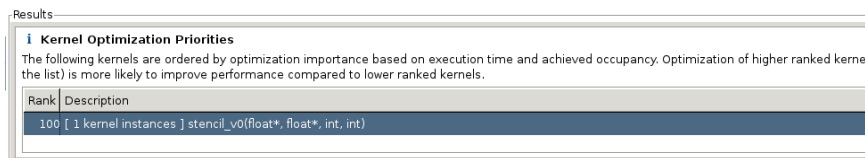
The Timeline



Analysis



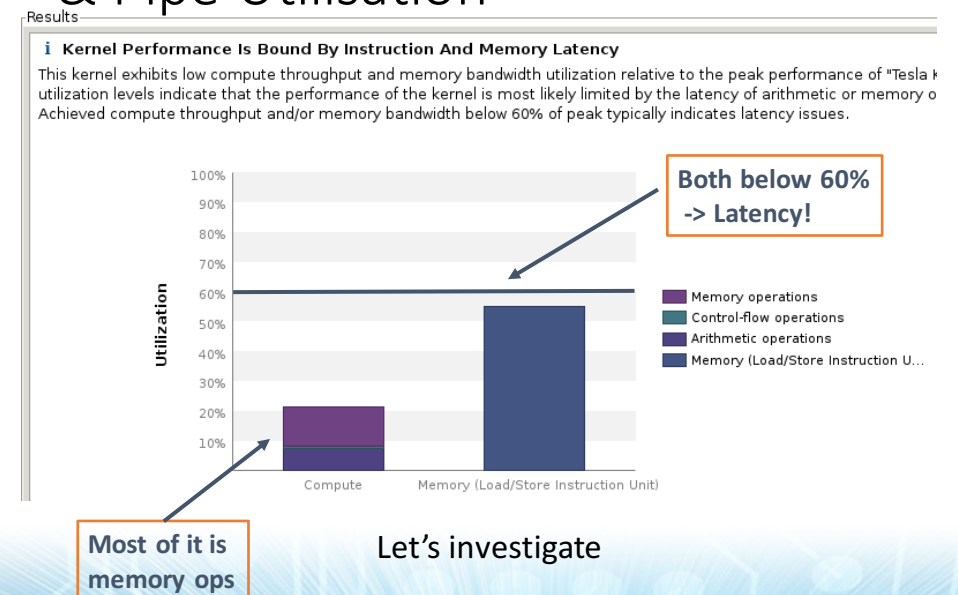
Examine Individual Kernels



Lists all kernels sorted by total execution time: the higher the rank the higher the impact of optimisation on overall performance

Initial unoptimised (v0) 8.122ms

Utilisation – Warp Issue Efficiency & Pipe Utilisation



Latency analysis

Analysis Details Console Settings

Export PDF Report

1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Band...or Latency Bound

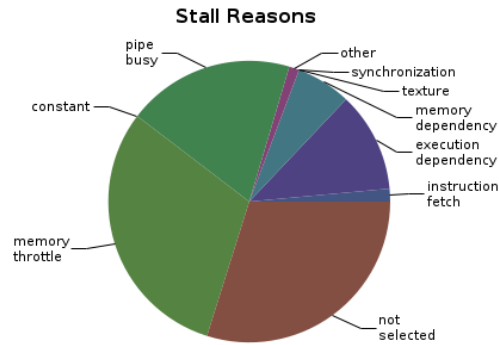
The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "stencil_v0" is most likely limited by instruction and memory latency.

Perform Latency Analysis

The most likely bottleneck to performance for this kernel is instruction and memory latency so you should first perform instruction and memory latency analysis to determine how it is limiting performance.

Perform Compute Analysis

Perform Memory Bandwidth Analysis



Memory throttle -> perform BW analysis

Memory Bandwidth analysis

Results

Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. [More...](#)

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	40894464	248.782 GB/s	
Global Stores	2621440	16.585 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	43515904	265.367 GB/s	
L2 Cache			
L1 Reads	62914560	248.782 GB/s	
L1 Writes	4194304	16.585 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Total	67108864	265.367 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	3756909	14.856 GB/s	
Writes	2904475	11.485 GB/s	
Total	6661384	26.341 GB/s	
ECC Overhead	2451525	9.694 GB/s	

L1 cache not used...

Investigate further...

Unguided

Analysis Details Console Settings

Reset All Analyze All

stencil_v0(float*, float*, int, int)

Kernel Performance Limiter ✓

Kernel Latency ✓

Kernel Compute ✓

Kernel Memory ✓

Global Memory Access Pattern ✓

Shared Memory Access Pattern ✓

Divergent Execution ✓

Global Memory Alignment and Access Pattern

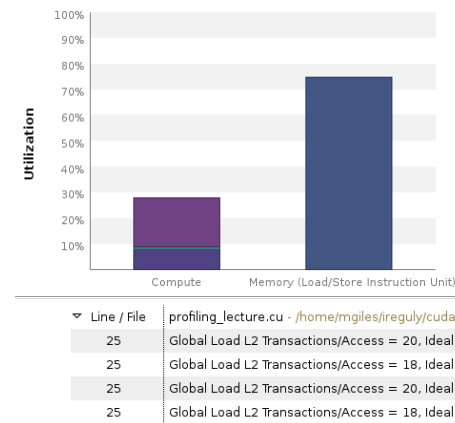
Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern. Optimization: Select each entry below to open the source code to a global load or store within the kernel with access pattern. For each load or store improve the alignment and access pattern of the memory access.

Line / File	profiling_lecture.cu - /home/mgiles/ireguly/cuda_course
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 6, Ideal Transactions/Access = 4 [3145728 L2 transacti
25	Global Load L2 Transactions/Access = 6, Ideal Transactions/Access = 4 [3145728 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti

6-8 transactions per access – something is wrong with how we access memory

Global memory load efficiency 53.3%
L2 hit rate 96.7%

Iteration 1 – turn on L1



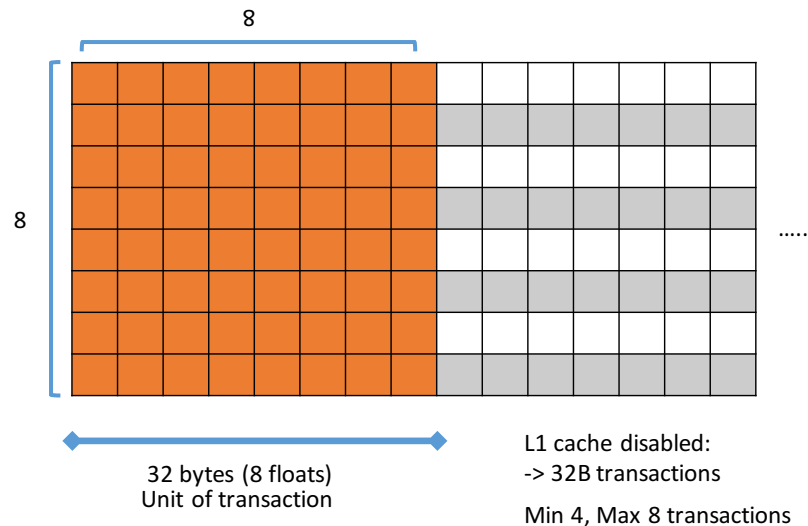
Quick & easy step:
Turn on L1 cache by using
-Xptxas -dlcm=ca

Memory unit is utilized, but Global Load efficiency became even worse: 20.5%

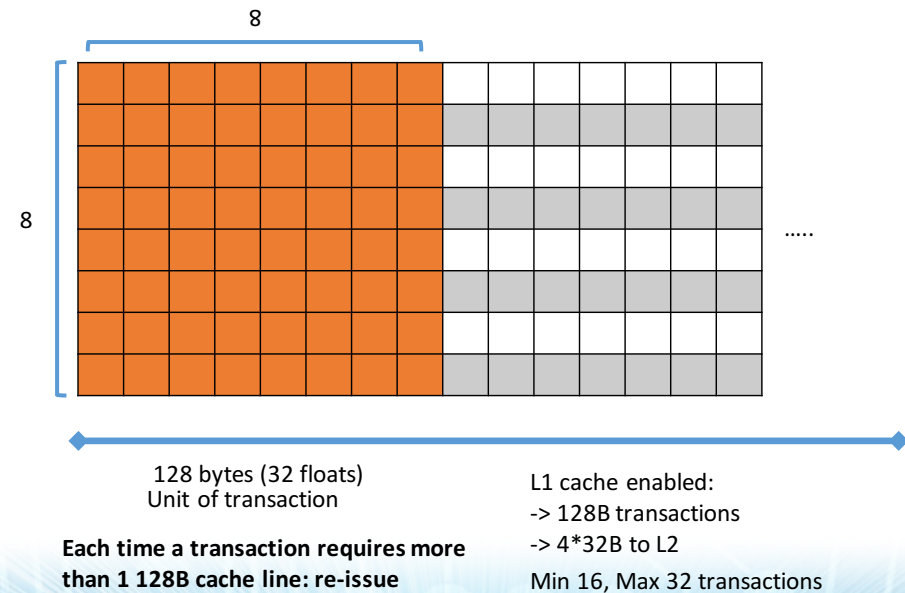
Initial unoptimised (v0) **8.122ms**

Enable L1 **6.57ms**

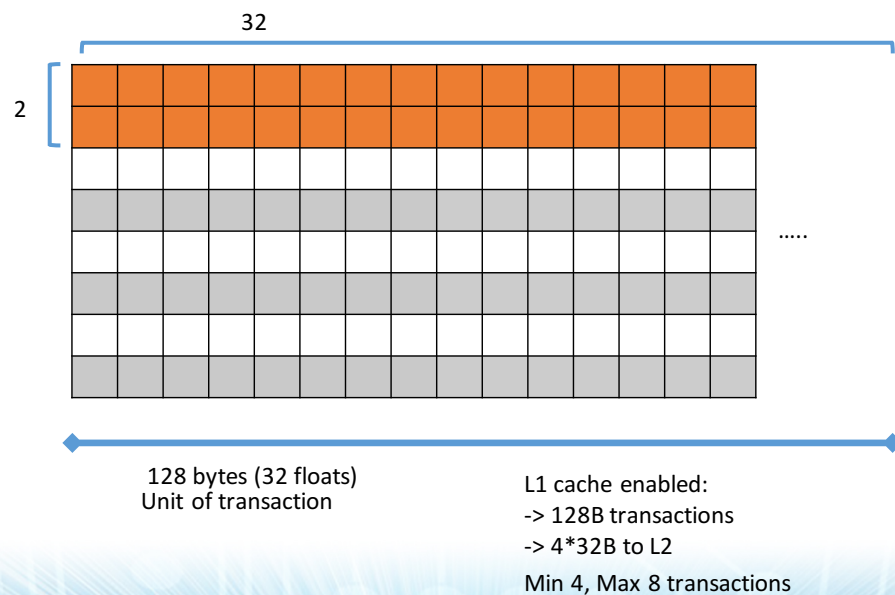
Cache line utilization



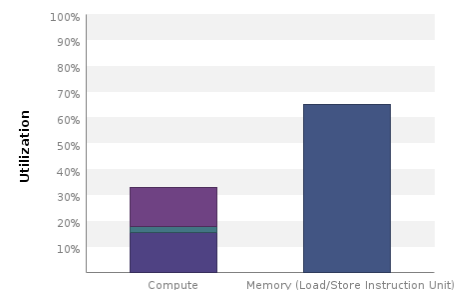
Cache line utilization



Cache line utilization



Iteration 2 – 32x2 blocks



Memory utilization decreased 10%
Performance almost doubles
Global Load Efficiency 50.8%

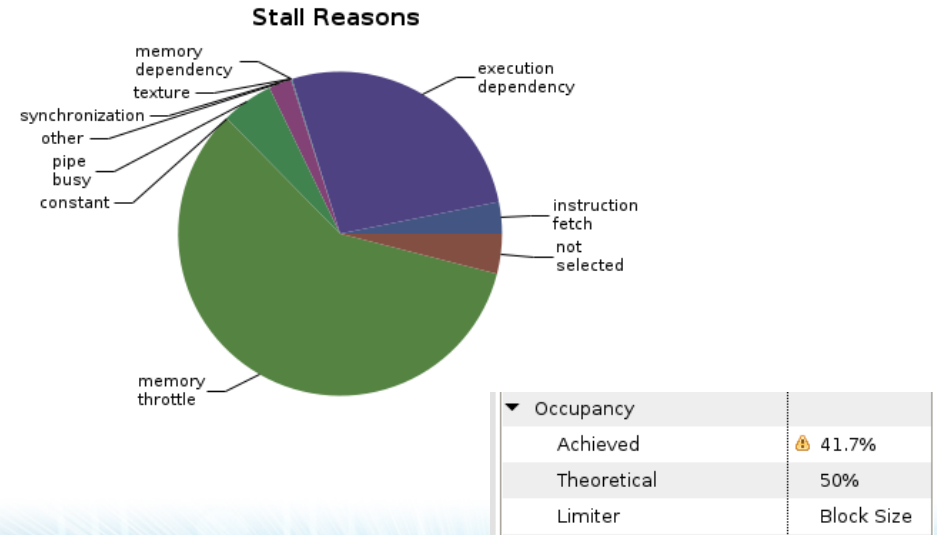
Line / File	profiling_lecture.cu - /home/mgiles/ireguly/cuda_course
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transactions for 524288 total exec
25	Global Load L2 Transactions/Access = 7.5, Ideal Transactions/Access = 4 [3932160 L2 transactions for 524288 total ex
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transactions for 524288 total exec

Initial unoptimised (v0)	8.122ms
Enable L1	6.57ms
Blocksize	3.4ms

Key takeaway

- **Latency/Bandwidth bound**
- Inefficient use of memory system and bandwidth
- Symptoms:
 - Lots of transactions per request (low load efficiency)
- Goal:
 - Use the whole cache line
 - Improve memory access patterns (coalescing)
- What to do:
 - Align data, change block size, change data layout
 - Use shared memory/shuffles to load efficiently

Latency analysis

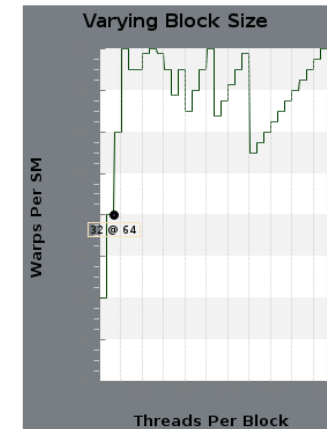


Latency analysis

Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM. [More...](#)

Variable	Achieved	Theoretical	Device Limit	Grid Size: [128,2048,1] (262144 blocks)Block Size: [3,
Occupancy Per SM				
Active Blocks		16	16	
Active Warps	26.67	32	64	
Active Threads		1024	2048	
Occupancy	41.7%	50%	100%	
Warps				
Threads/Block		64	1024	
Warps/Block		2	32	
Block Limit		32	16	

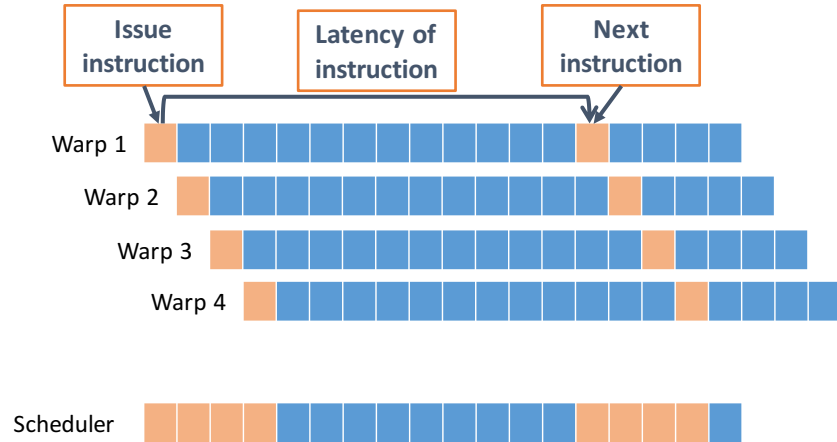
Latency analysis



Increase the block size so more warps can be active at the same time.

Kepler:
Max 16 blocks per SM
Max 2048 threads per SM

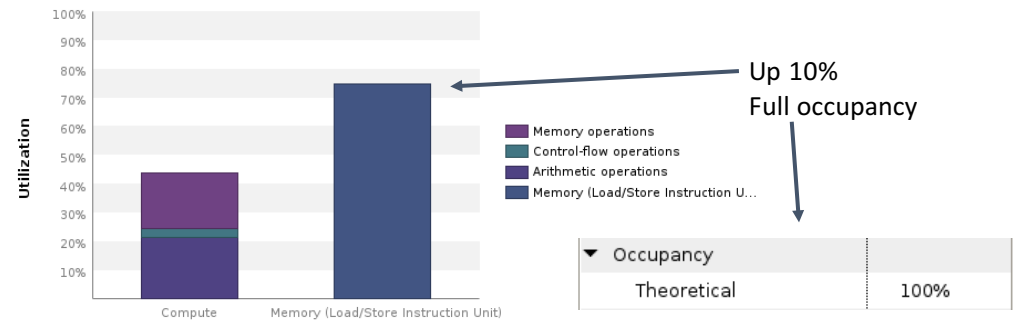
Occupancy – using all “slots”



Increase block size to 32x4

Illustrative only, reality is a bit more complex...

Iteration 3 – 32x4 blocks



Initial unoptimised (v0) 8.122ms

Enable L1 6.57ms

Blocksize 3.4ms

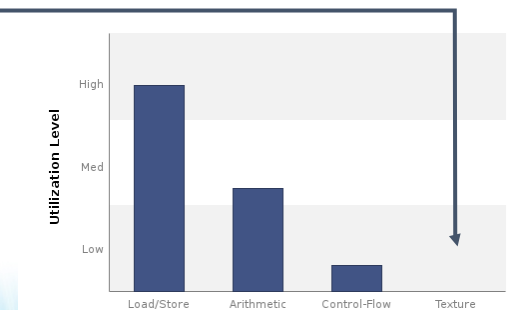
Blocksize 2 2.36ms

Key takeaway

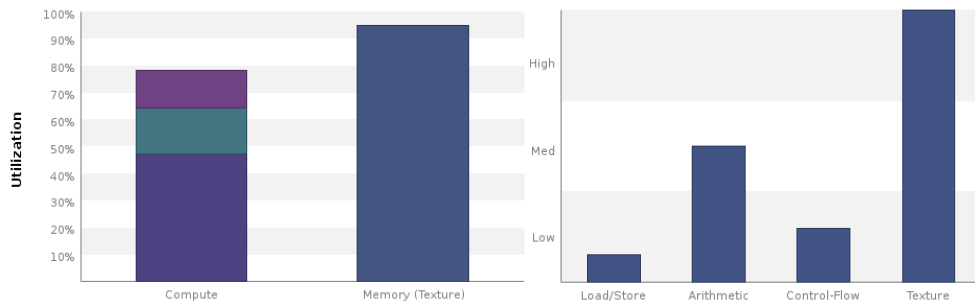
- **Latency bound – low occupancy**
- Unused cycles, exposed latency
- Symptoms:
 - High execution/memory dependency, low occupancy
- Goal:
 - Better utilise cycles by: having more warps
- What to do:
 - Determine occupancy limiter (registers, block size, shared memory) and vary it

Improving memory bandwidth

- L1 is fast, but a bit wasteful (128B loads)
 - 8 transactions on average (minimum would be 4)
- Load/Store pipe stressed
 - Any way to reduce the load?
- Texture cache
 - Dedicated pipeline
 - 32 byte loads
 - `const __restrict__ *`
 - `__ldg()`



Iteration 4 – texture cache



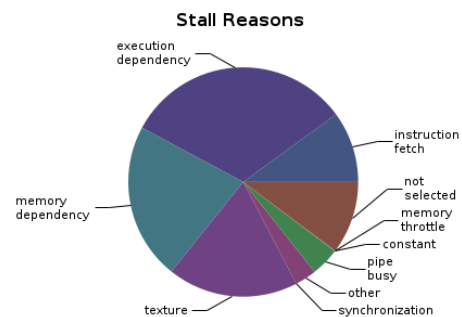
Texture Cache

Reads	65536000	1,382.851 GB/s
-------	----------	----------------

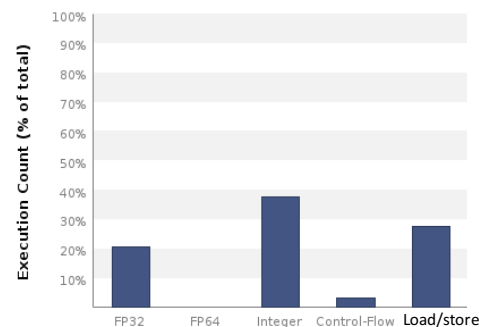
Device Memory

Initial unoptimised (v0)	8.122ms
Blocksize 2	2.36ms
Texture cache	1.53ms

Compute analysis



Instruction mix

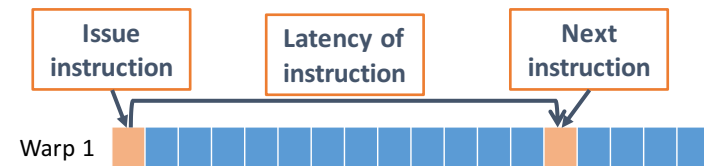


Compute utilization could be higher (~78%)
 Lots of Integer & memory instructions, fewer FP
 Integer ops have lower throughput than FP
 Try to amortize the cost: increase compute per byte

Key takeaway

- **Bandwidth bound – Load/Store Unit**
- LSU overutilised
- Symptoms:
 - LSU pipe utilisation high, others low
- Goal:
 - Better spread the load between other pipes: use TEX
- What to do:
 - Read read-only data through the texture cache
 - `const __restrict__` or `__ldg()`

Instruction Level Parallelism



- Remember, GPU is in-order:

$a=b+c$ $a=b+c$
 $d=a+e$ $d=e+f$
- Second instruction cannot be issued before first
 - But it can be issued before the first finishes– if there is no dependency
- Applies to memory instructions too – latency much higher (counts towards stall reasons)

Instruction Level Parallelism

```
for (j=0;j<2;j++)  
    acc+=filter[j]*input[x+j];
```

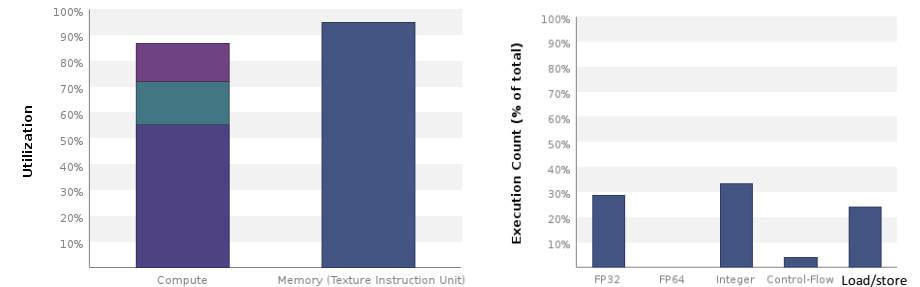
```
tmp=input[x+0]  
↓  
acc += filter[0]*tmp  
↓  
tmp=input[x+1]  
↓  
acc += filter[1]*tmp
```

#pragma unroll can help ILP
Create two accumulators
Or...

```
for (j=0;j<2;j++) {  
    acc0+=filter[j]*input[x+j];  
    acc1+=filter[j]*input[x+j+1];  
}  
tmp=input[x+0]  
↓  
acc0 += filter[0]*tmp  
↓  
acc1 += filter[0]*tmp  
↓  
tmp=input[x+1]  
↓  
acc0 += filter[1]*tmp  
↓  
acc1 += filter[1]*tmp
```

Process 2 points per thread
Bonus data re-use (register caching)

Iteration 5 – 2 points per thread

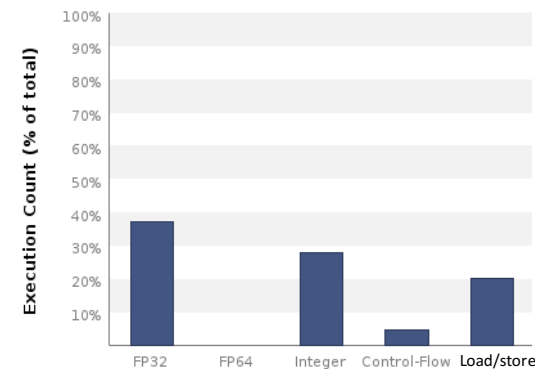


Initial unoptimised (v0)	8.122ms
Texture cache	1.53ms
2 points	1.07ms

Key takeaway

- **Latency bound – low instruction level parallelism**
- Unused cycles, exposed latency
- Symptoms:
 - High execution dependency, one “pipe” saturated
- Goal:
 - Better utilise cycles by: increasing parallel work per thread
- What to do:
 - Increase ILP by having more independent work, e.g. more than 1 output value per thread
 - #pragma unroll

Iteration 6 – 4 points per thread



168 GB/s device BW

Initial unoptimised (v0)	8.122ms
2 points	1.07ms
4 points	0.95ms

Checklist

- `cudaDeviceSynchronize()`
 - Most API calls (e.g. kernel launch) are asynchronous
 - Overhead when launching kernels
 - Get rid of `cudaDeviceSynchronize()` to hide this latency
 - Timing: events or callbacks CUDA 5.0+
- Cache config 16/48 or 48/16 kB L1/shared (default is 48k shared!) on Kepler
 - `cudaSetDeviceCacheConfig`
 - `cudaFuncSetCacheConfig`
 - Check if shared memory usage is a limiting factor

Checklist

- Occupancy
 - Max 1536 threads or 8 blocks per SM on Fermi (2048/16 for Kepler, 2048/32 for Maxwell)
 - Limited amount of registers and shared memory
 - Max 255 registers/thread, rest is spilled to global memory
 - You can explicitly limit it (`-maxregcount=xx`)
 - 48kB/16kB shared/L1: don't forget to set it
 - Visual Profiler tells you what is the limiting factor
 - In some cases though, it is faster if you don't maximise it (see Volkov paper) -> Autotuning!

Verbose compile

- Add `-Xptxas=-v`

```
ptxas info  : Compiling entry function '_Z10fem_kernelPiS_' for 'sm_20'
ptxas info  : Function properties for _Z10fem_kernelPiS_
    856 bytes stack frame, 980 bytes spill stores, 1040 bytes spill loads
ptxas info  : Used 63 registers, 96 bytes cmem[0]
```

- Check profiler figures for best occupancy

Checklist

- Precision mix (e.g. 1.0 vs 1.0f) – `cuobjdump`
 - F2F.F64.F32 (6* the cost of a multiply)
 - IEEE standard: always convert to higher precision
 - Integer multiplications are now expensive (6*)
- `cudaMemcpy`
 - Introduces explicit synchronisation, high latency
 - Is it necessary?
 - May be cheaper to launch a kernel which immediately exits
 - Could it be asynchronous? (Pin the memory!)

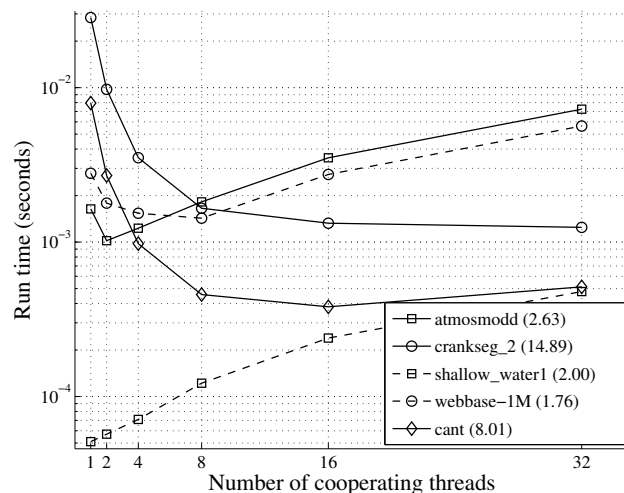
Auto-tuning

- Several parameters that affect performance
 - Block size
 - Amount of work per block
 - Application specific
- Which combination performs the best?
- Auto-tuning with Flamingo
 - #define/read the sizes, recompile/rerun combinations

Auto-tuning Case Study

- Thread cooperation on sparse matrix-vector product
 - Multiple threads doing partial dot product on the row
 - Reduction in shared memory
- Auto-tune for different matrices
 - Difficult to predict caching behavior
 - Develop a heuristic for cooperation vs. average row length

Autotuning Case Study



Conclusions

- Iterative approach to improving a code's performance
 - Identify hotspot
 - Find performance limiter, understand why it's an issue
 - Improve your code
 - Repeat
- Managed to achieve a 8.5x speedup
- Shown how NVVP guides us and helps understand what the code does
- There is more it can show...

Rapid code development with Thrust

Thrust

- Open High-Level Parallel Algorithms Library
- Parallel Analog of the C++ Standard Template Library (STL)
 - Vector containers
 - Algorithms
- Comes with the toolkit
- Productive way to use CUDA

Example

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

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

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

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    |
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Productivity

- Containers
 - host_vector
 - device_vector
- Memory management
 - Allocation, deallocation
 - Transfers
- Algorithm selection
 - Location is implicit

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);
```

```
// copy host data to device memory
thrust::device_vector<int> d_vec = h_vec;
```

```
// write device values from the host
d_vec[0] = 27;
d_vec[1] = 13;
```

```
// read device values from the host
int sum = d_vec[0] + d_vec[1];
// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());
```

Productivity

- Large set of algorithms

- ~100 functions
- CPU, GPU

Algorithm	Description
reduce	Sum of a sequence
find	First position of a value in a sequence
mismatch	First position where two sequences differ
count	Number of instances of a value
inner_product	Dot product of two sequences
merge	Merge two sorted sequences

- Flexible

- C++ templates
- User-defined types
- User-defined operators

Portability

- Implementations

- CUDA C/C++
- Threading Building Blocks
- OpenMP
- Interoperable with anything CUDA based

- Recompile

- Mix backends

`nvcc -DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP`

```
thrust::omp::vector<float> my_omp_vec(100);  
thrust::cuda::vector<float> my_cuda_vec(100);
```

Interoperability

- Thrust containers and raw pointers

- Use container in CUDA kernel

```
thrust::device_vector<int> d_vec(...);  
cuda_kernel<<<N, 128>>>(some_argument_d,
```

- Use a device pointer in thrust algorithms (not a vector though, no begin(), end(), resize() etc.)

```
int *dev_ptr;  
cudaMalloc((void**)&dev_ptr, 100*sizeof(int));  
  
thrust::device_ptr<int> dev_ptr_thrust(dev_ptr);  
thrust::fill(dev_ptr_thrust, dev_ptr_thrust+100, 0);
```

Thrust

- Constantly evolving
- Reliable – comes with the toolkit, tested every day with unit tests
- Performance – specialised implementations for different hardware
- Extensible – allocators, back-ends, etc.



Thrust documentation

<http://thrust.github.io/doc/modules.html>

