CUDA Programming on NVIDIA GPUs

Mike Giles, Mathematical Institute, Oxford University

Wes Armour, Engineering Science, Oxford University

Lecture 1: an introduction to CUDA

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Overview

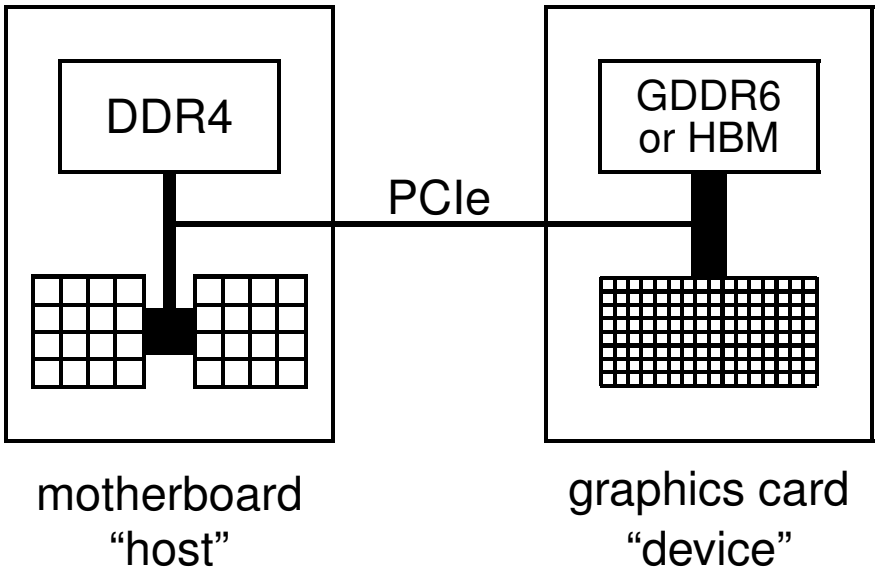
- hardware view
- software view
- CUDA programming
- first practical

Course materials are available at:

https://people.maths.ox.ac.uk/gilesm/cuda/

Hardware view

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



Hardware view

Currently, 3 generations of professional/HPC cards with excellent double precision (DP) capabilities, and special “tensor cores” for AI/ML:

- Volta (compute capability 7.0):
  - V100 released in 2018 **(we will use for practicals)**
- Ampere (compute capability 8.0):
  - A100 released in 2020, smaller A30 later
  - A2, A10, A16, A40 (all compute capability 8.6) for inference and Virtual Desktop Infrastructure
- Hopper (compute capability 9.0):
  - H100 released in 2023
  - L4, L40 (compute capability 8.9) for inference / VDI

Hardware view

In addition there are consumer/gaming cards with excellent single precision (SP) capabilities, ray tracing support, and “tensor cores” for AI/ML, but much poorer on DP

- Ampere (compute capability 8.6):
  - GeForce RTX 3060 / 3060 Ti
  - GeForce RTX 3070 / 3070 Ti
  - GeForce RTX 3080 / 3080 Ti
  - GeForce RTX 3090 / 3090 Ti
- Ada Lovelace (compute capability 8.9):
  - GeForce RTX 4060 / 4060 Ti
  - GeForce RTX 4070 / 4070 Ti
  - GeForce RTX 4080
  - GeForce RTX 4090

Hardware view

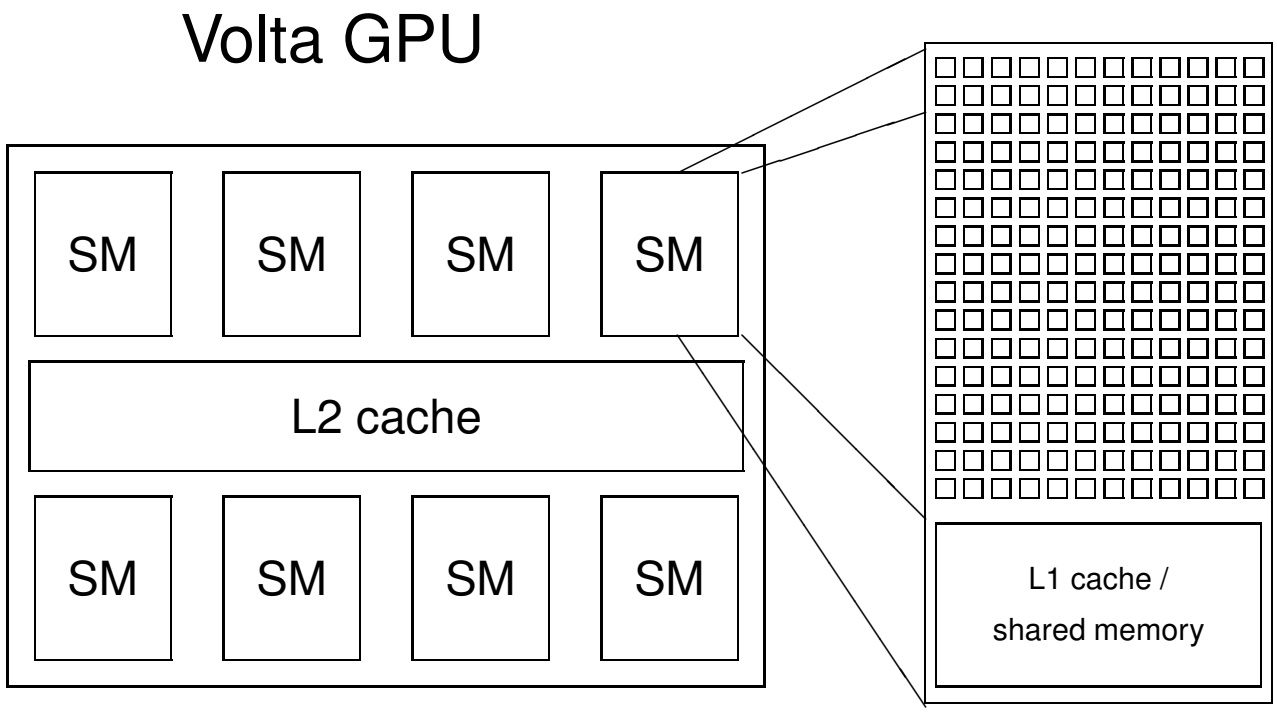
The key building block in an NVIDIA GPUs is a “streaming multiprocessor” (SM) – the V100 has 80 of them each with:

- 32 FP64 cores + 64 FP32 cores + 64 INT32 cores
- 64k registers
- 96KB of shared memory/L1 cache
- up to 2K threads per SM

In addition the V100 has:

- 6MB of L2 cache
- bandwidth of 900GB/s to 32GB external HBMe memory

Hardware View



Remember it really has 80 SMs!

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 AVX vectorisation on Intel Xeons
- 32 threads all doing the same thing at the same time
- natural for graphics processing and much scientific computing
- SIMT is also a natural choice for many-core chips to simplify each core

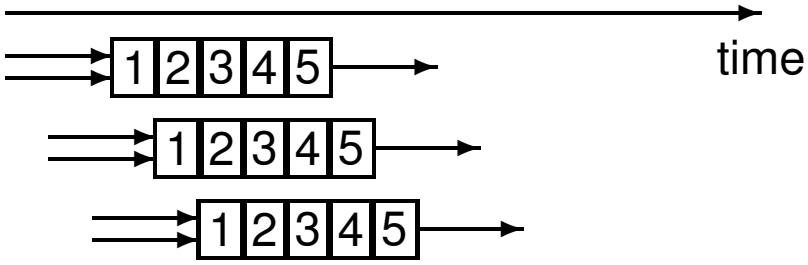
Multithreading

Lots of active threads is the key to high performance:

- no “context switching”; each thread has its own registers (up to 255 of them), which limits the number of active threads
- threads on each SM execute in groups of 32 called “warps” – execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

Multithreading

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



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

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

Software view

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

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

Software view

At a lower level, within the GPU:

- each instance (or copy) of the 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 (on the GPU) and run later
- all threads within one instance can access local shared memory but can’t see what the other instances are doing (even if they are on the same SM)
- there are no guarantees on the order in which the instances execute

CUDA

CUDA is NVIDIA's program development environment:

- based on C/C++ with some extensions
- Fortran support also available
- lots of sample codes and good documentation
  – fairly short learning curve for those with experience of OpenMP and MPI programming
- large user community on NVIDIA forums

CUDA Components

Installing CUDA on a system, there are 2 components:

- Driver
  - low-level software that controls the graphics card
- Toolkit (currently on version 12.2)
  - `nvcc` CUDA compiler
  - Nsight plugin for Eclipse or Visual Studio
  - profiling and debugging tools
  - lots of libraries

In addition NVIDIA makes available lots of sample codes in a GitHub repository

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):

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

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

CUDA programming

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

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

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

CUDA programming

In its simplest form it looks like:

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

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

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

CUDA programming

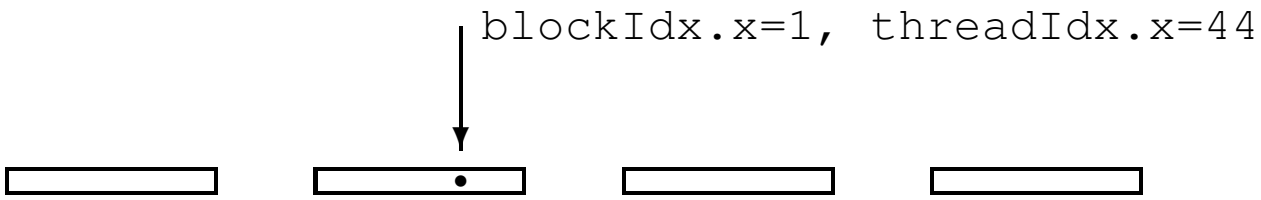
At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

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

CUDA programming

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

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



CUDA programming

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

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

Host code

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

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

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

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

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

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


Kernel code

```
#include <helper_cuda.h>

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

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

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

CUDA programming

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

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

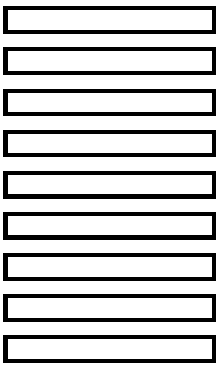
Each clock tick, SM warp scheduler decides which warps to execute next, choosing from those not waiting for

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

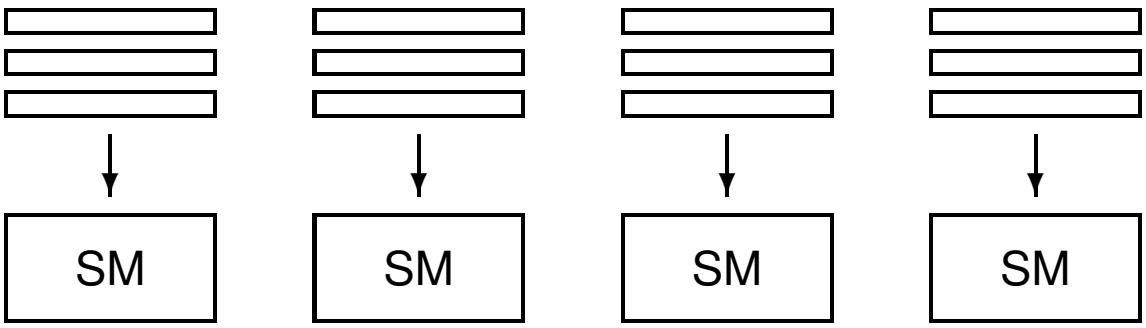
Programmer doesn't have to worry about this level of detail, just make sure there are lots of threads / warps

CUDA programming

Queue of waiting blocks:



Multiple blocks running on each SM:



CUDA programming

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

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

and to launch the kernel we would use something like

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

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

CUDA programming

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

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

1D thread ID defined by

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

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

Practical 1

- start from code shown above (but with comments)
- 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 samples

Practical 1

Things to note:

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

Practical 1

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

- check for error return codes:
  `checkCudaErrors( ... );`

- check for kernel failure messages:
  `getLastCudaError( ... );`

Practical 1

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

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

Practical 1

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

In this version

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

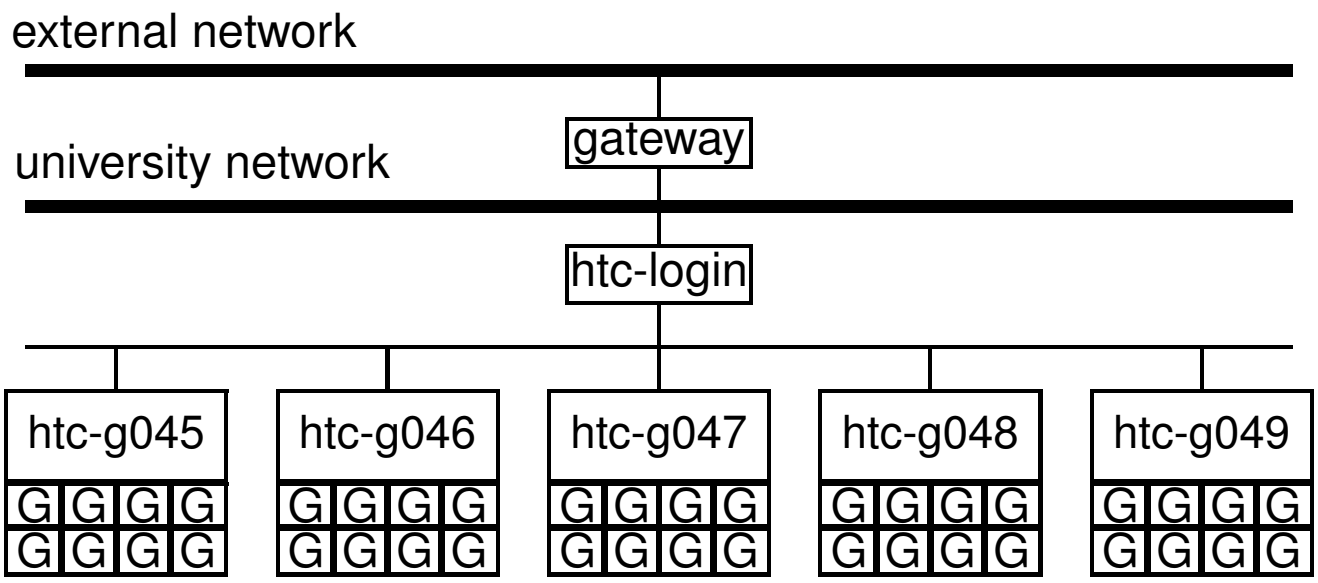
Practical 1

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

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

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

ARC “htc” cluster



- `gateway.arc.ox.ac.uk` is for external access
- `htc-login.arc.ox.ac.uk` is the head/login node
- the DGX compute nodes each have 8 Volta V100 GPUs
- read the ARC notes before starting the practical

Key reading

CUDA C++ Programming Guide:

- Section 1: Introduction
- Section 2: Programming Model
- Section 5.4: performance of different GPUs
- Section 6: CUDA-enabled GPUs
- Sections 7.1 – 7.4: C language extensions
- Section 7.33: `printf` output
- Section 16: features of different GPUs

Lecture 2: different memory and variable types

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

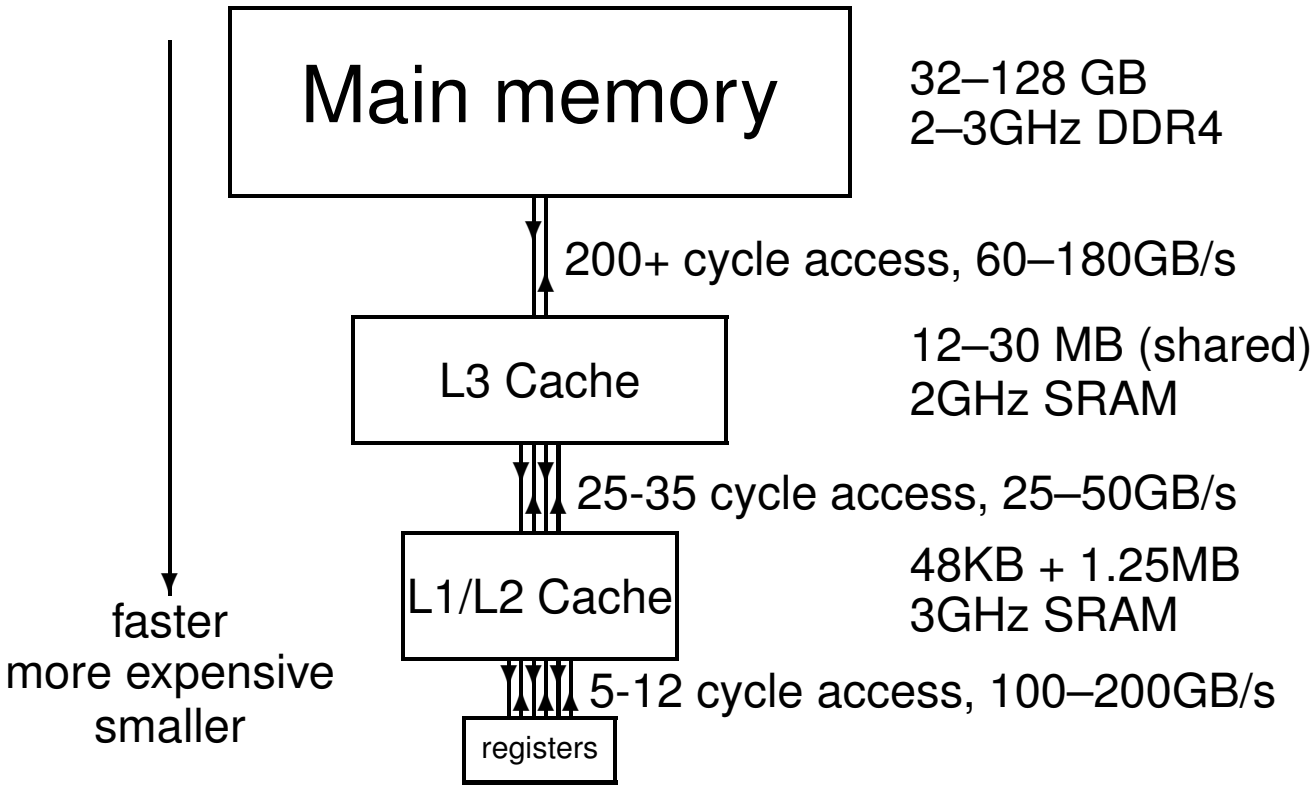
Oxford University Mathematical Institute

Memory

Key challenge in modern computer architecture

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

CPU Memory Hierarchy



Memory Hierarchy

Execution speed relies on exploiting data *locality*

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

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

Caches

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

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

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

When the CPU stores data from a register:

- same procedure

Importance of Locality

Typical workstation:

20 Gflops per core

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

64 bytes/line

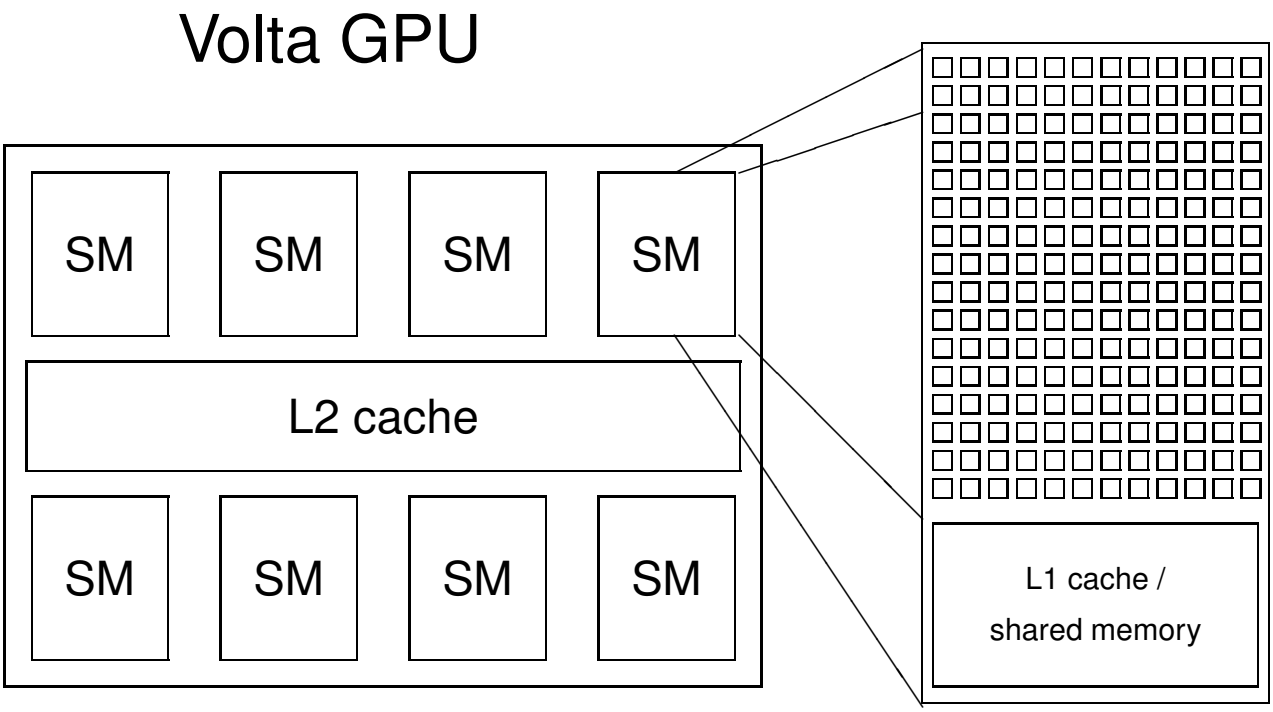
40GB/s $\equiv$ 600M line/s $\equiv$ 5G double/s

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

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

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

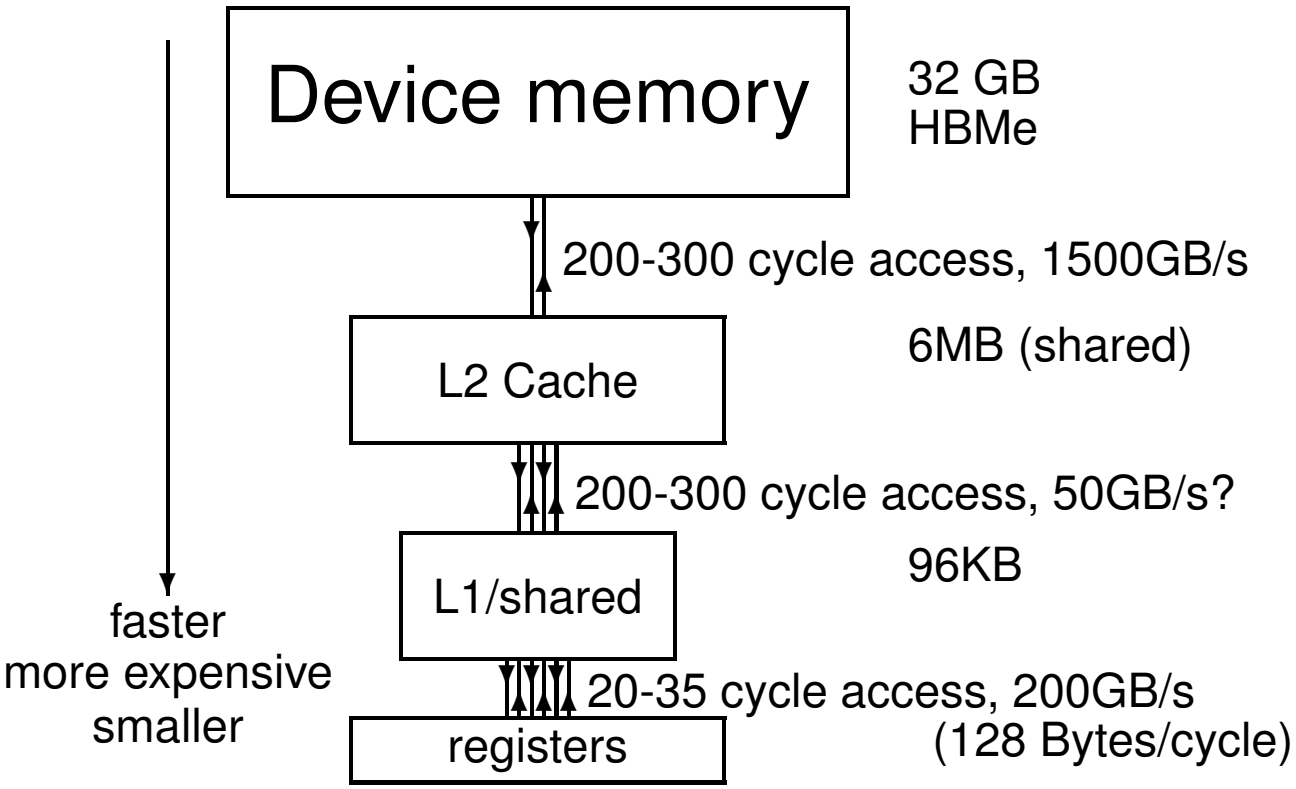
GPU Architecture



Volta

- usually 32 bytes cache line (8 floats or 4 doubles)
- V100: 4096-bit memory path from HBM2e device memory to L2 cache $\Longrightarrow$ up to 900 GB/s bandwidth
- unified 6MB L2 cache for all SM's
- each SM has 96kB of shared memory / L1 cache
- no global cache coherency as in CPUs, so should (almost) never have different blocks updating the same global array elements

GPU Memory Hierarchy



Importance of Locality

20Tflops GPU

1280 GB/s memory $\longleftrightarrow$ L2 cache bandwidth

32 bytes/line

1280 GB/s $\equiv$ 40G line/s $\equiv$ 160G double/s

At worst, each flop requires 2 inputs and has 1 output, forcing loading of 3 lines $\Longrightarrow$ 13 Gflops

If all 4 doubles/line are used, increases to 50 Gflops

To get up to 8 TFlops needs about 50 flops per double transferred to/from device memory

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

Practical 1 kernel

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

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

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

A bad kernel

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

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

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

Global arrays

So far, concentrated on global / device arrays:

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

Global variables

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

```
__device__ int reduction_lock=0;

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

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

Global variables

- the `__device__` prefix tells `nvcc` this is a global variable in the GPU, not the CPU.
- the variable can be read and modified by any kernel
- its lifetime is the lifetime of the whole application
- can also declare arrays of fixed size
- can read/write by host code using special routines `cudaMemcpyToSymbol, cudaMemcpyFromSymbol` or with standard `cudaMemcpy` in combination with `cudaGetSymbolAddress`

- in my own CUDA programming, I rarely use this capability but it is occasionally very useful

Constant variables

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

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

Constant variables

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

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

Constants

A constant variable has its value set at run-time

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

```
#define PI 3.1415926f

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

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

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

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

Registers

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

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

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


Registers

- 64K 32-bit registers per SM
- up to 255 registers per thread
- up to 2048 threads per SM (at most 1024 per thread block)
- max registers per thread $\Longrightarrow$ 256 threads
- max threads $\Longrightarrow$ 32 registers per thread
- 8$\times$ difference between “fat” and “thin” threads

Registers

What happens if your application needs more registers?

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

either certain variables become device arrays with one element per thread

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

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

Local arrays

What happens if your application uses a little array?

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

float ut[3];

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

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

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

Local arrays

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

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

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


Local arrays

In more complicated cases, array is put into device memory

- this is because registers are not dynamically addressable – compiler has to specify exactly which registers are used for each instruction
- still referred to in the documentation as a “local array” because each thread has its own private copy
- held in L1 cache by default, may never be transferred to device memory
- 96kB of L1 cache equates to 24k 32-bit variables, which is 24 per thread when using 1024 threads
- beyond this, it will have to spill to device memory

Shared memory

In a kernel, the prefix `__shared__` as in

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

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

There can be several benefits:

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

Shared memory

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

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

Instruction

```
__syncthreads();
```

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

Shared memory

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

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

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

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

Using this within the kernel function is complicated/tedious; see Section 7.2.3 in CUDA C++ Programming Guide

Read-only arrays

With “constant” variables, each thread reads the same value.

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

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

```
const __restrict__
```

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

Vector variables / 16-bit floats

Section 7.3 of CUDA C++ Programming Guide: CUDA defines small vectors

- `double2,double3,double4`: 2, 3, or 4 doubles
- `float2,float3,float4`: 2, 3, or 4 floats
- similar for ints, uints, etc.

Individual components are labelled `.x, .y, .z, .w`

Also, CUDA defines two kinds of 16-bit floats

- `half,half2`: IEEE fp16 variables
  (very limited range: 6×10^{-5} – 6×10^{4})
- `bfloat16,bfloat162`: bfloat16 variables
  (same range as `float` but much lower precision)

Built-in variables

Section 7.4 of CUDA C++ Programming Guide:

- `gridDim`: type `dim3` (like `uint3` but all three components `.x, .y, .z` initialised to 1 by default)
- `blockIdx`: type `uint3`
- `blockDim`: type `dim3`
- `threadIdx`: type `uint3`
- `warpSize`: type `int`
  (always 32 so far, but might change in future?)

Non-blocking loads/stores

What happens with the following code?

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

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

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

Active blocks per SM

Each block require certain resources:

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

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

Active blocks per SM

My general advice:

- number of active threads depends on number of registers each needs
- good to have at least 4 active blocks per SM, 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 any shared memory

Active blocks per SM

On Volta:

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

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

Summary

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

Key reading

CUDA C++ Programming Guide:

- Sections 7.1-7.4 – essential
- Sections 3.2.2, 3.2.4

Other reading:

- Wikipedia article on caches:
  `en.wikipedia.org/wiki/CPU_cache`

Lecture 3: control flow and synchronisation

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Warp divergence

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

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

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

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

Warp divergence

This is not a new problem.

Old CRAY vector supercomputers had a logical merge vector instruction

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

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

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


Warp divergence

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

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

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

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

Warp divergence

Note that:

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

Warp divergence

Another example:

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

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

Warp divergence

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

```
    p = ...

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

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


Warp divergence

Note:

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

Warp divergence

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

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

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

In this case, there's no need to vote

Warp divergence

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

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

Typical example: PDE application with boundary conditions

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

Warp divergence

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

GPU implementation:

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

Note: none of this is new – this is what we did 35 years ago on CRAY and Thinking Machines systems.

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

Synchronisation

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

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

Otherwise, can end up in *deadlock*

Typical application

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

// synchronisation to ensure this has finished

   __syncthreads();

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


Synchronisation

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

- `int __syncthreads_count(predicate)`

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

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

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

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

Warp voting

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

- `int __all(predicate)`

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

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

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

Again, I've never used these

Atomic operations

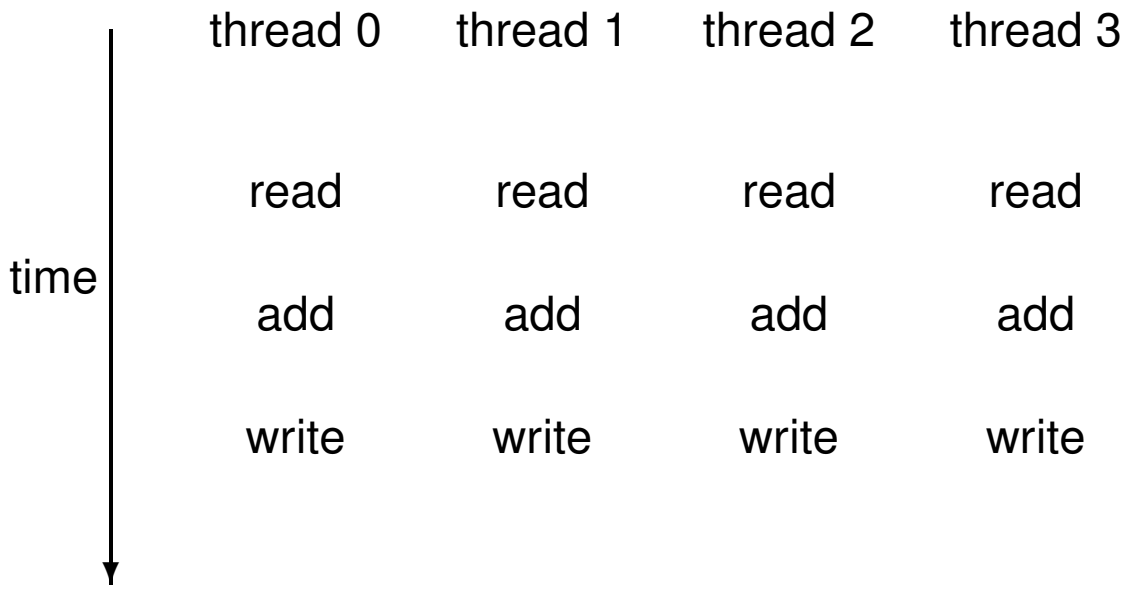
Occasionally, an application needs threads to update a counter in shared memory.

```
__shared__ int count;

...

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

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

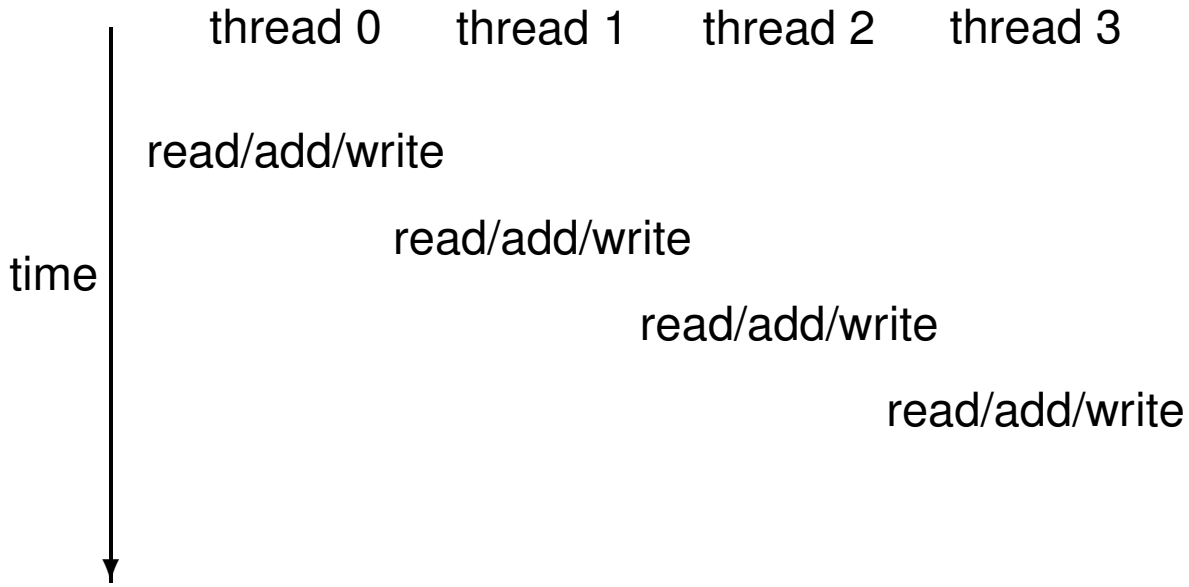
Atomic operations

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



Atomic operations

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



Atomic operations

Several different atomic operations are supported:

- addition / subtraction
  `atomicAdd, atomicSub`
- minimum / maximum
  `atomicMin, atomicMax`
- increment / decrement
  `atomicInc, atomicDec`
- exchange / compare-and-swap
  `atomicExch, atomicCAS`
- bitwise AND / OR / XOR
  `atomicAnd, atomicOr, atomicXor`

Fast for variables in shared memory, only slightly slower for variables in device global memory (operations performed in L2 cache)

Atomic operations

Compare-and-swap:

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

- if `compare` equals `old` value stored at `address` then `val` is stored instead
- in either case, routine returns the value of `old`
- seems a bizarre routine at first sight, but can be very useful for atomic locks

Global atomic lock

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

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

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

    ...

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


Global atomic lock

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

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

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

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

__threadfence

- `__threadfence_block();`

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

- `__threadfence();`

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

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 relevant NVIDIA sample codes

Key reading

CUDA C++ Programming Guide:

- Section 5.4.2: control flow and predicates
- Section 5.4.3: synchronization
- Section 7.5: `__threadfence()` and variants
- Section 7.6: `__syncthreads()` and variants
- Section 7.14: atomic functions
- Section 7.19: warp voting

2D Laplace solver

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

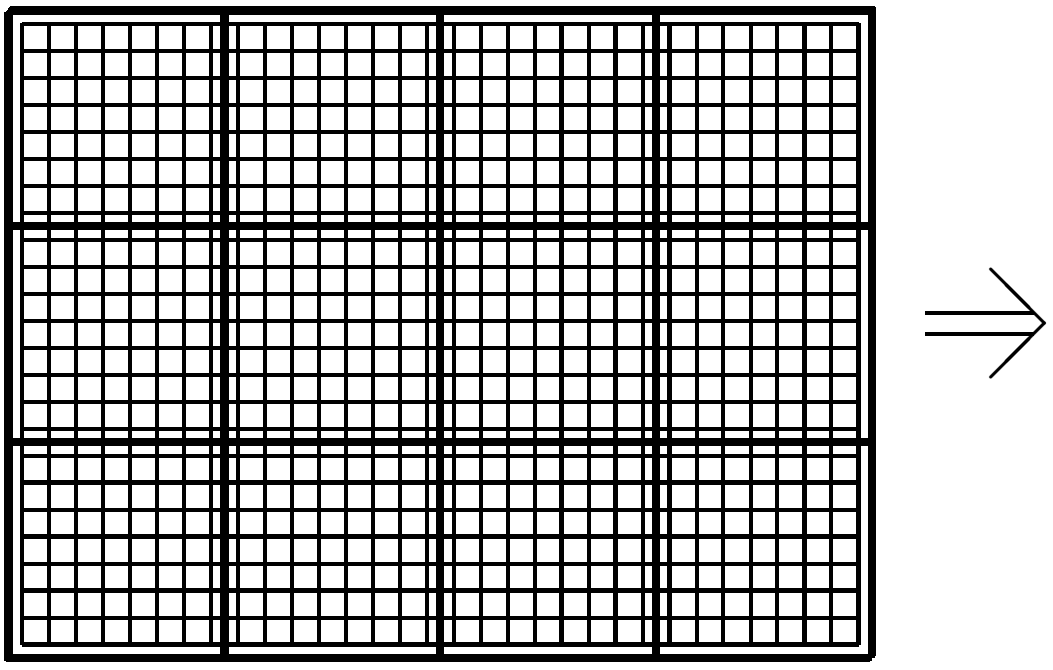
```
for (int j=0; j<J; j++) {
  for (int i=0; i<I; i++) {

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

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

2D Laplace solver

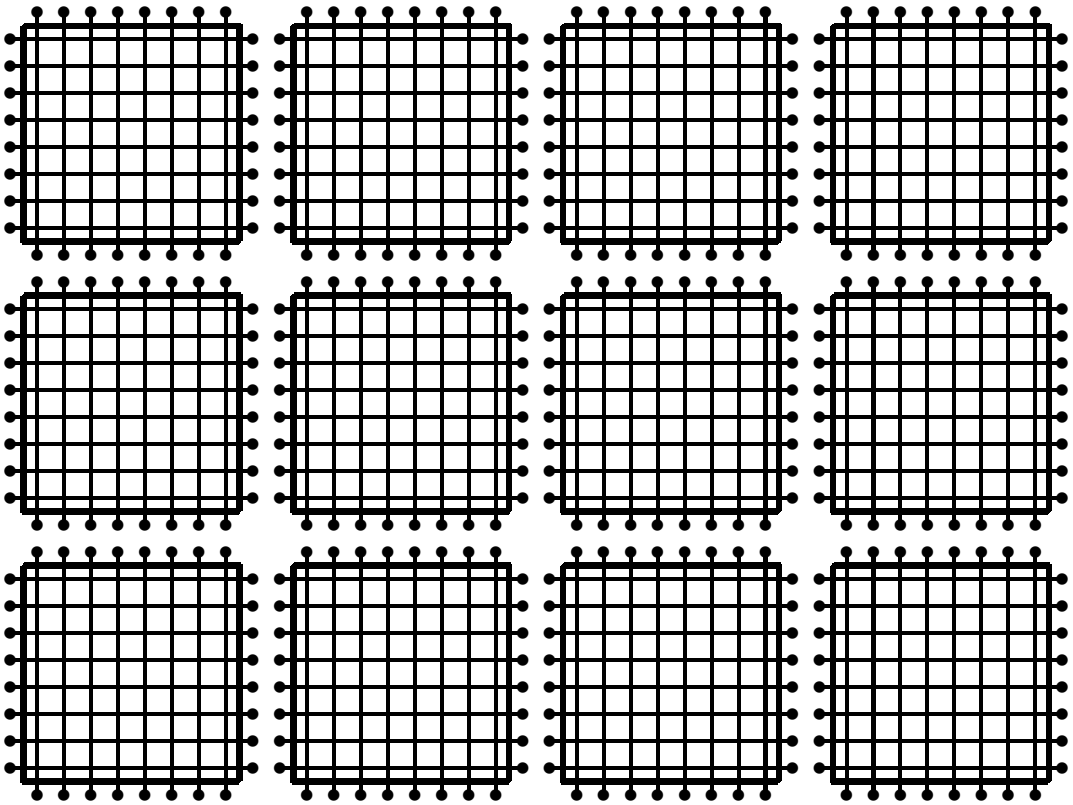
How do we tackle this with CUDA?

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

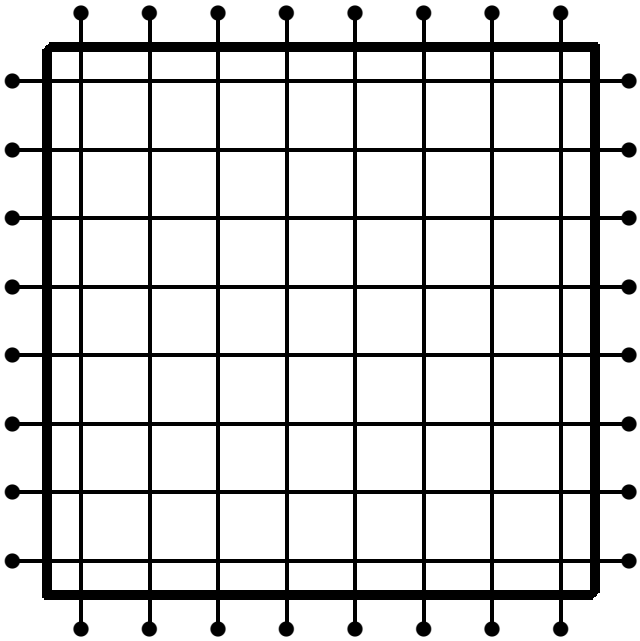
2D Laplace solver



2D Laplace solver



2D Laplace solver



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

2D Laplace solver

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

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

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

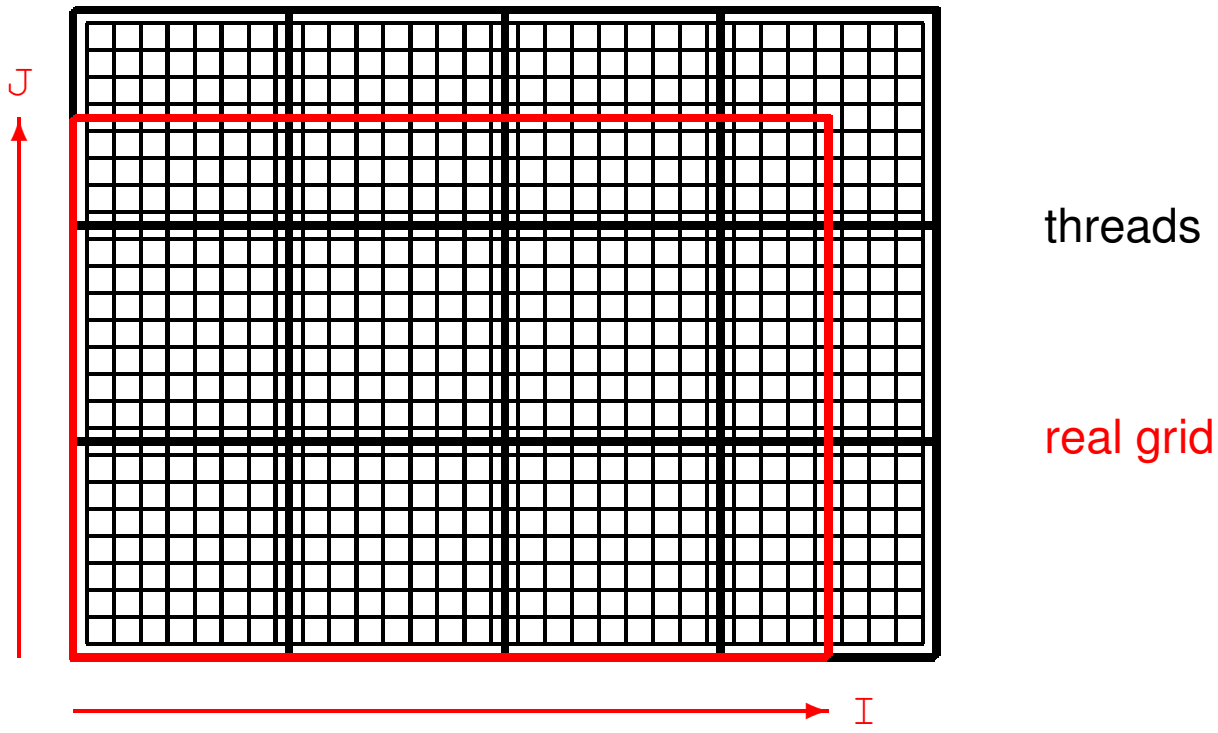

2D Laplace solver

Assumptions:

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

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

2D Laplace solver

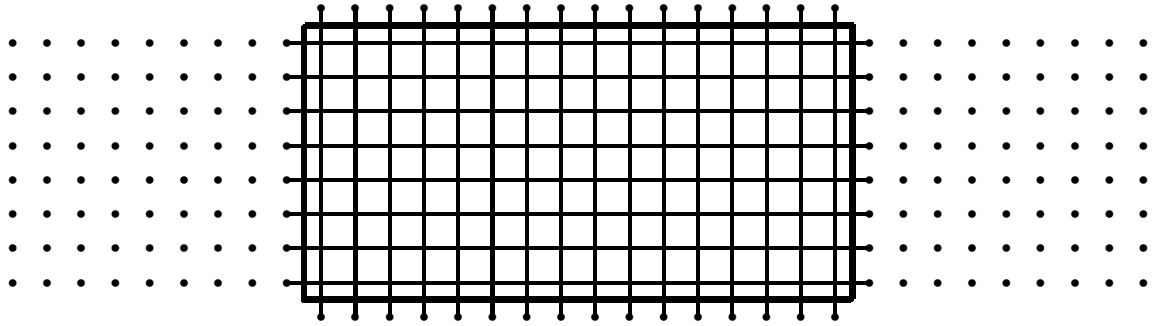


2D Laplace solver

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

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

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

2D Laplace solver

How does cache function in this application?



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

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, perhaps due to many more active threads – in either case the application is probably bandwidth-limited

Lecture 4: warp shuffles, and reduction / scan operations

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Warp shuffles

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

There are 4 variants:

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

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

Warp shuffles

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

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

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

- defined similarly

Warp shuffles

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

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

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

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

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

- copies data from `srcLane`

Warp shuffles

Very important

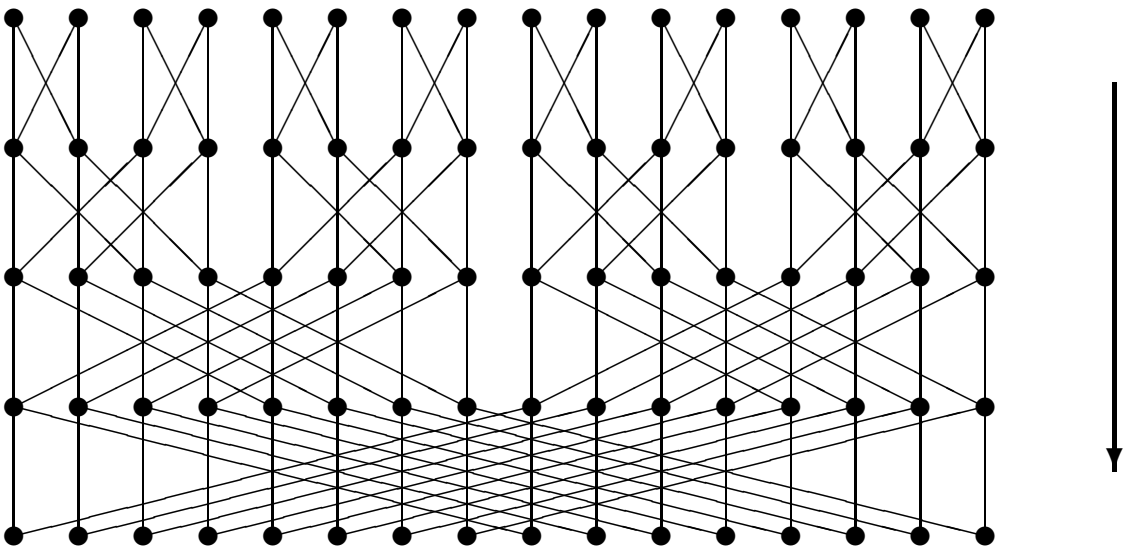
> Threads may only read data from another thread which is actively participating in the shuffle command. If the target thread is inactive, the retrieved value is undefined.

This means you must be very careful with conditional code.

Warp shuffles

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

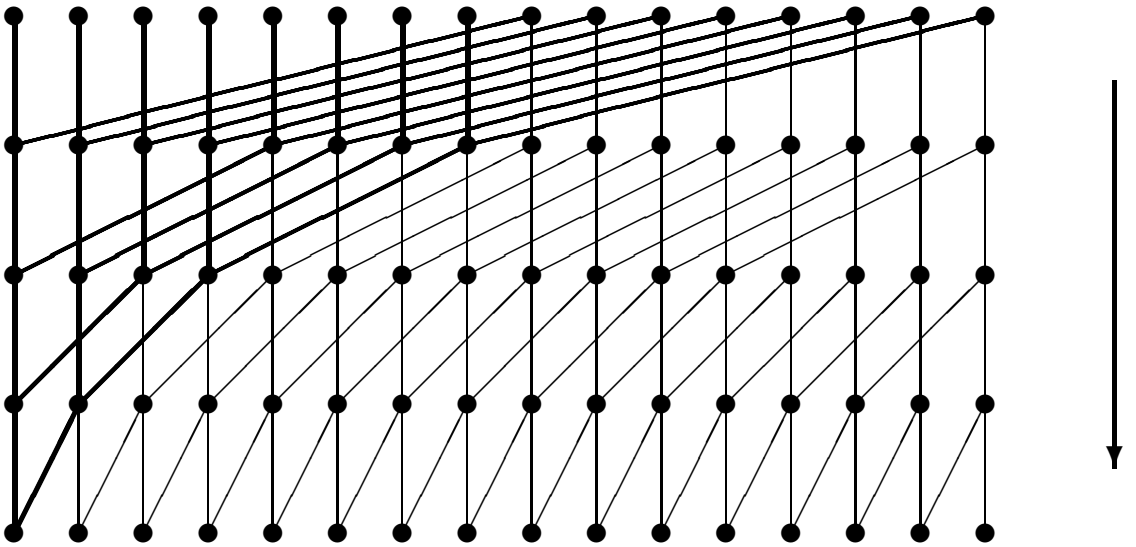
```
for (int i=1; i<warpSize; i*=2)
  value += __shfl_xor_sync(-1, value, i);
```



Warp shuffles

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

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



Reduction

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

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

Reduction

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

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

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

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

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

Approach

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

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

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

I'll look at each of these stages in turn

Local reduction

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

Question 1: where is the parallelism?

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

Parallel summation of N values:

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

Local reduction

Question 2: any problems with warp divergence?

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

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

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

Local reduction

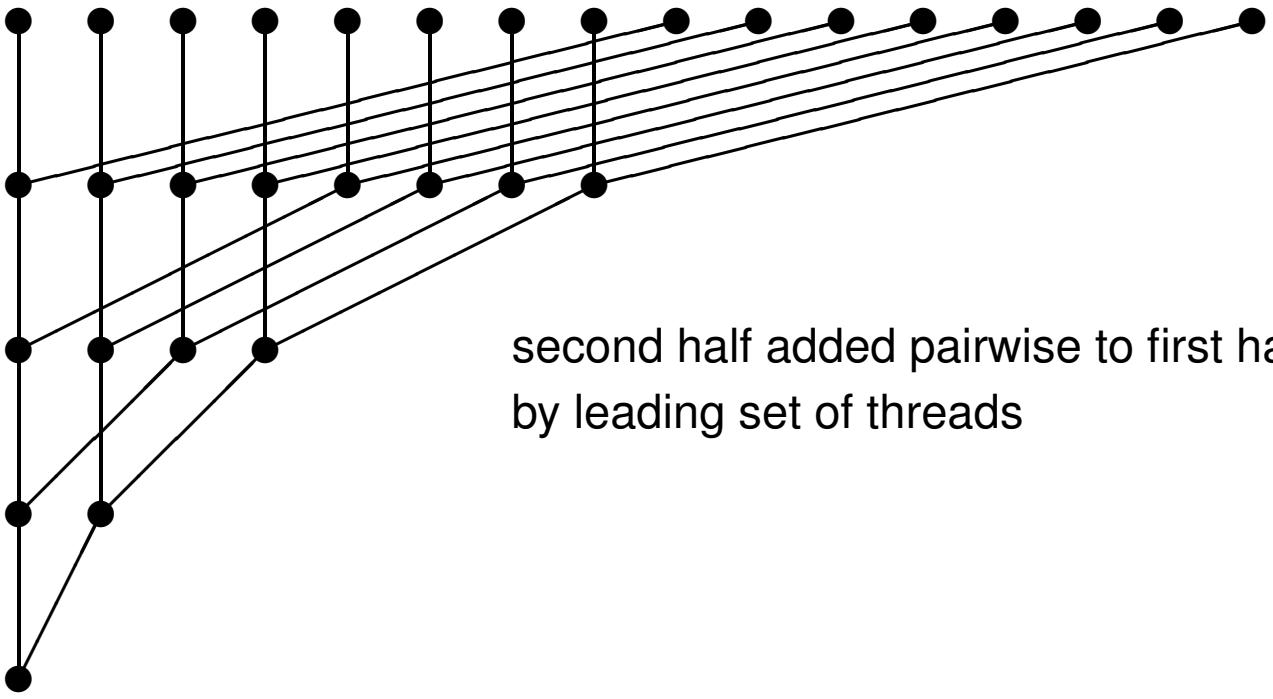
Question 3: where should data be held?

Threads need to access results produced by other threads:

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

Local reduction

Pictorial representation of the algorithm:



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

Local reduction

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

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

 for (int d=blockDim.x/2; d>=1; d=d/2) {
   __syncthreads();
   if (tid<d) temp[tid] += temp[tid+d];
 }

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


Local reduction

Note:

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

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

Global reduction: version 1

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

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

Global reduction: version 2

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

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

by

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

Global reduction: version 2

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

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

by

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

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

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


Scan operation

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

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

Why is this important?

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

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

- a good example that looks impossible to parallelise

Scan operation

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

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

Scan operation

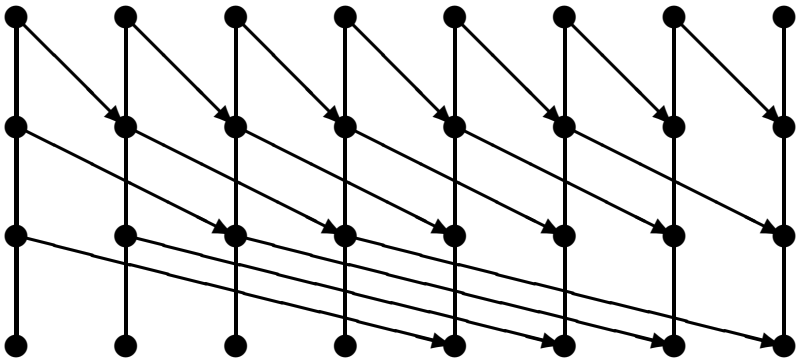
Similar to the global reduction, the top-level strategy is

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

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

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

Local scan: version 1



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

Local scan: version 1

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

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

 for (int d=1; d<blockDim.x; d=2*d) {
   __syncthreads();
   float temp2 = (tid >= d) ? temp[tid-d] : 0;
   __syncthreads();
   temp[tid] += temp2;
 }

 ...
}
```

Local scan: version 1

Notes:

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

Confession: my most common CUDA programming error is failing to use a `__syncthreads();` when 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=2*d) {
    temp2 = __shfl_up_sync(-1, temp1,d);
    if (tid%32 >= d) temp1 += temp2;
  }

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

Local scan: version 2

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

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

    for (int d=1; d<32; d=2*d) {
      temp3 = __shfl_up_sync(-1, temp2,d);
      if (tid%32 >= d) temp2 += temp3;
    }
    if (tid < blockDim.x/32) temp[tid] = temp2;
  }
```


Local scan: version 2

Finally, we add the sum of previous warps:

```
  __syncthreads();

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

  ...
}
```

Global scan: version 1

To complete the global scan there are two options

First alternative:

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

Global scan: version 2

Second alternative – do it all in one kernel call

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

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

Solution: use atomic increments to create an in-order block ID

Global scan: version 2

Declare a global device variable

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

and at the beginning of the kernel code use

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

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

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

Global scan: version 2

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

- get in-order block ID
- perform scan within the block
- wait until another global counter

  ```
  __device__ volatile int my_block_count2 = 0;
  ```

  shows that preceding block has computed the sum of the blocks so far
- get the sum of blocks so far, increment the sum with the local partial sum, then increment `my_block_count2`
- add previous sum to local scan values and store the results

Global scan: version 2

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

if (tid == 0) {
  // volatile qualifier critical here
  do {} while( my_block_count2 < my_blockId );

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

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


Scan operation

Conclusion: this is all quite tricky!

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

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

Recurrence equation

Given s_n, u_n, want to compute v_n defined by

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

(Often have

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

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

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

Recurrence equation

Starting from

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

then substituting the second equation into the first gives

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

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

Repeat at each level of the scan, eventually getting

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

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

Recurrence equation

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

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

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

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

Key reading

CUDA C++ Programming Guide:

- Section 7.22: warp shuffle instructions
- Section 7.21: new warp reduction instruction – this is only for integers currently, and I have not experimented with it

Lecture 5: Libraries and tools

Prof Wes Armour
wes.armour@eng.ox.ac.uk

Prof Mike Giles
mike.giles@maths.ox.ac.uk

Oxford e-Research Centre
Department of Engineering Science

Learning outcomes

In this fifth lecture we will learn about GPU libraries and tools for GPU programming.

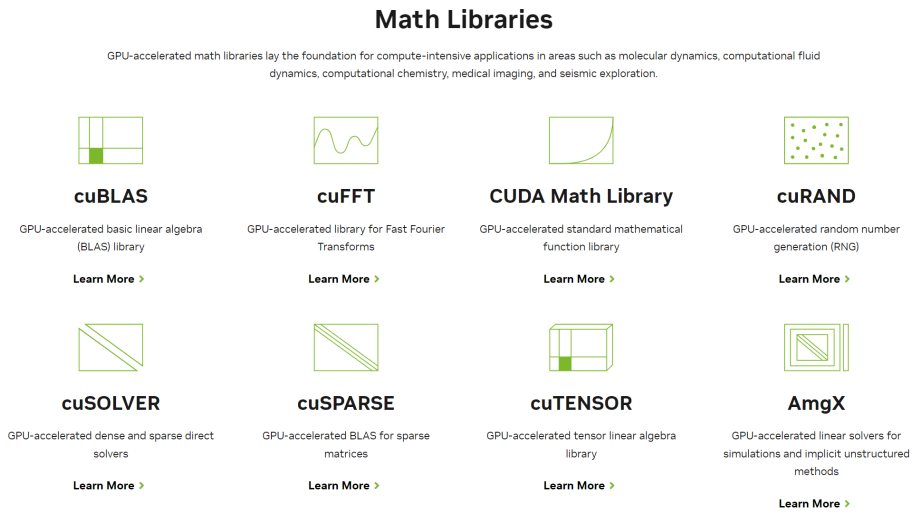
You will learn about:

- NVIDIA GPU libraries and there usefulness in scientific computing.
- Third party libraries.
- Directives based approaches to GPU computation.
- Tools for GPU programming.

Software overview

NVIDIA provides a **rich ecosystem of software tools** that allow you to easily utilise GPUs in your project.

During this lecture we will focus on a range of **libraries and software tools that will make your life easier** when either writing CUDA code or when utilising GPUs in your projects.



Dependencies – Advantages / Disadvantages

Some advantages:

- Simple to use – you don't need to write your own complex code to perform a specific task.
- Well maintained – always benefit from the latest optimisations and improvements.
- Easier(?) to move from CPU code to GPU code (for example see cuFFTW).

Some disadvantages:

- Can make installing your code on another system harder (the user also needs to have the dependency installed).
- If the dependency isn't maintained it could break your code as other things (e.g. compiler) are updated.
- If your code is very dependent on it and the developers stop supporting it – you become the owner (not a great position to be in).

CUDA math library

The CUDA Math Library contains all of the typical mathematical functions that you will need for your projects. It is very similar to Intel's MKL library.

- various exponential and log functions
- trigonometric functions and their inverses
- hyperbolic functions and their inverses
- error functions and their inverses
- Bessel and Gamma functions
- vector norms and reciprocals (esp. for graphics)
- To use - "#include math.h"

CUDA Math Library

GPU-accelerated standard mathematical function library

The library supports standard int, float and double types, but in recent years has also added support for fp8, fp16 and bfloat16.

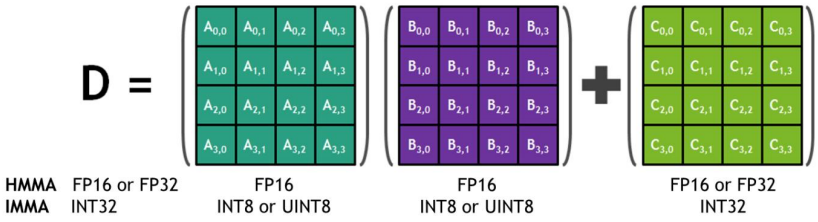
Typecasting and SIMD intrinsics are also included in this library.

CUDA Math Library | NVIDIA Developer

cuTENSOR

Tensor cores originally introduced on Volta in 2017 provided hardware enabled acceleration for matrix-matrix multiplies.

- Originally performed a 4x4 matrix multiply-accumulate (think FMA for matrices) using wmma:: instruction.
- Matrices A and B would be lower precision and the accumulators would be the same or higher precision
- With Ampere and Hopper some of these restrictions have been relaxed.

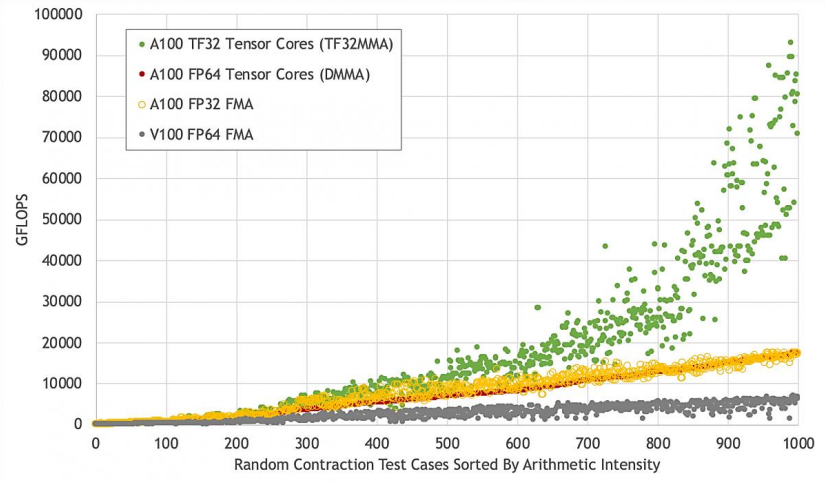


https://arxiv.org/pdf/2206.02874.pdf
https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/
https://developer.nvidia.com/blog/nvidia-automatic-mixed-precision-tensorflow/
https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9593-cutensor-high-performance-tensor-operations-in-cuda-v2.pdf

cuTENSOR

Current tensor cores (Hopper) are able to use various precisions and exploit sparsity to gain further acceleration.

- FP64 inputs with FP32 compute (DMMA).
- FP32 inputs with FP16, BF16, or TF32 compute.
- Complex-times-real operations.
- Conjugate (without transpose) support.
- Support for up to 64-dimensional tensors.



https://developer.nvidia.com/cutensor

https://docs.nvidia.com/cuda/cutensor/index.html

https://github.com/NVIDIA/CUDALibrarySamples/blob/master/cuTENSOR/reduction.cu

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

cuBLAS Library

The cuBLAS (CUDA Basic Linear Algebra Subprograms) library provides CUDA accelerated standard BLAS APIs (for 152 different routines) for dense matrices.

- includes matrix-vector and matrix-matrix product.
- it is possible to call cuBLAS routines from user kernels (via a device API).
- some support for a single routine call to do a “batch” of smaller matrix-matrix multiplications.
- also support for using CUDA streams to do a large number of small tasks concurrently.
- has support for multi-GPU operation (cuBLASTxt or cuBLASMg).
- has mixed / low precision implementations.

cuBLAS

GPU-accelerated basic linear algebra (BLAS) library

cuBLAS Library

To use cuBLAS in your codes, a set of routines are called from your host code. These come in two forms, helper routines and compute routines.

Helper routines for:

- memory allocation
- data copying from CPU to GPU, and vice versa
- error reporting

Compute routines for:

- e.g. matrix-matrix and matrix-vector product
- **Warning!** Some calls are asynchronous, i.e. the call starts the operation but the host code then continues before it has completed!!

cuBLAS

GPU-accelerated basic linear algebra (BLAS) library

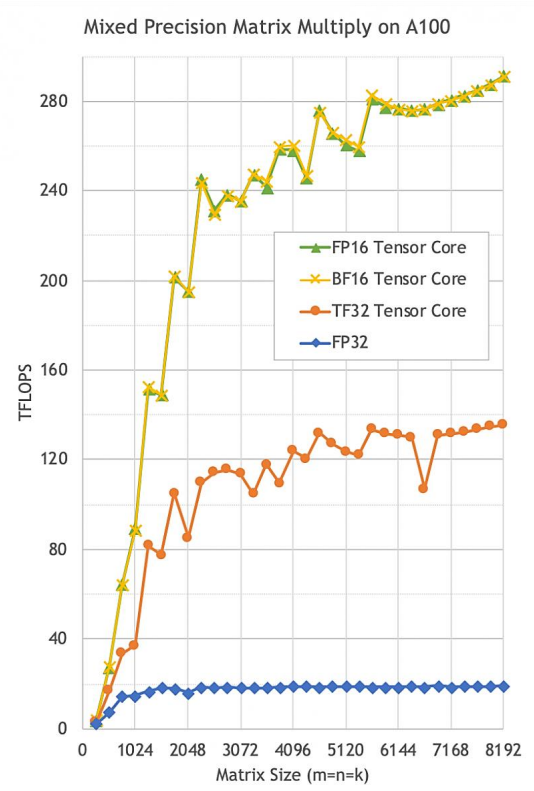
cuBLAS | NVIDIA Developer

cuBLAS Library

cuBLAS is one of three libraries that use "tensor cores". Tensor cores are different to a standard processing core, the are designed to perform very specific operations and these operations are executed on mixed precision data.

If you are able to reduce the precision of your matrix / vector operations in cuBLAS you can gain significant acceleration.

https://developer.nvidia.com/cublas



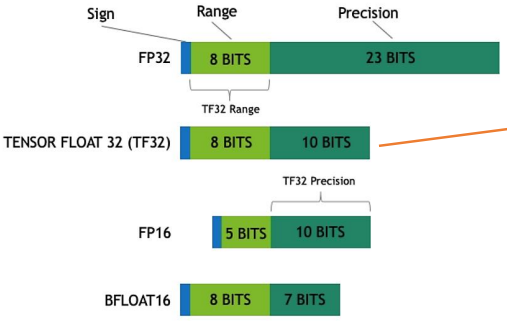
cuBLAS Library

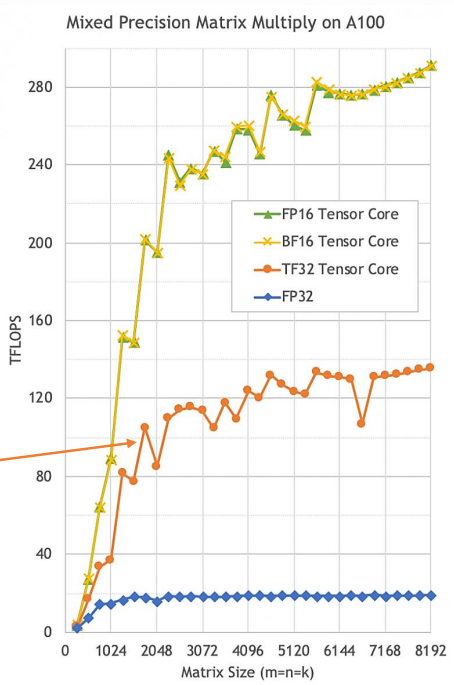
Since Ampere, there has been a "TF32" variable that is a compromise between FP32 and BFLOAT16. It allows for a compromise between lower precision and FP32 matrix-matrix operations.

Also DMMA64 for double precision – uses AI magic?

Useful for:

- AI Training
- Liner solvers



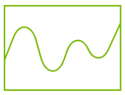


https://developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores/#:~:text=TF32%20mode%20is%20the%20default,any%20changes%20to%20model%20scripts.

cuFFT Library

The cuFFT library is a GPU accelerated library that provides Fast Fourier Transforms.

- Provides 1D, 2D and 3D FFTs.
- Has almost all of the variations found in FFTW and other CPU libraries.
- Includes the cuFFTW library, a porting tool, to enable users of FFTW to start using GPUs with minimal effort.
- Provides some device level functionality. *If this is something of interest ask Karel – he has already produced shared memory device level FFTs for our projects.*



cuFFT

GPU-accelerated library for Fast Fourier Transforms

cuFFT Library

cuFFT is used exactly like cuBLAS - it has a set of routines called by host code:

Helper routines include “plan” construction.

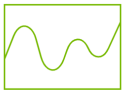
Compute routines perform 1D, 2D, 3D FFTs:

- `cufftExecC2C()` - complex-to-complex.
- `cufftExecR2C()` - real-to-complex.
- `cufftExecC2R()` - complex-to-real.

(double precision routines have different function calls, e.g. `cufftExecZ2Z()`)

It supports doing a “batch” of independent transforms, e.g. applying 1D transform to a 3D dataset

The `simpleCUFFT` example in SDK is a good starting point.



cuFFT

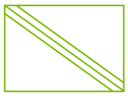
GPU-accelerated library for Fast Fourier Transforms

https://docs.nvidia.com/cuda/cufft/index.html#introduction

cuSPARSE Library

cuSPARSE is a GPU accelerated library that provides various routines to work with sparse matrices.

- Includes sparse matrix-vector and matrix-matrix products.
- Can be used for iterative solution (but see cuSOLVER for an easy life).
- Also has solution of sparse triangular system
- Note: batched tridiagonal solver is in cuBLAS not cuSPARSE



cuSPARSE

GPU-accelerated BLAS for sparse matrices

cuRAND Library

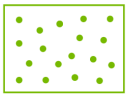
The cuRAND library is a GPU accelerated library for random number generation.

It has many different algorithms for pseudorandom and quasi-random number generation.

Pseudo: XORWOW, mrg32k3a, Mersenne Twister and Philox 4x32_10
Quasi: SOBOL and Scrambled SOBOL

Uniform, Normal, log-Normal and Poisson outputs

This library also includes device level routines for RNG within user kernels.



cuRAND

GPU-accelerated random number generation (RNG)

cuSOLVER

cuSOLVER brings together cuSPARSE and cuBLAS.

Has solvers for dense and sparse systems.

Key LAPACK dense solvers, 3 – 6x faster than MKL.

Sparse direct solvers, 2–14x faster than CPU equivalents.

cuSOLVER

GPU-accelerated dense and sparse direct solvers

Other notable libraries

CUB (CUDA Unbound): https://nvlabs.github.io/cub/

- Provides a collection of basic building blocks at three levels: device, thread block, warp.
- Functions include sort, scan and reduction.
- Thrust uses CUB for CUDA versions of key algorithms.
- Last update over a year ago…

http://on-demand.gputechconf.com/gtc/2014/presentations/S4566-cub-collective-software-primitives.pdf

AmgX (originally named NVAMG): http://developer.nvidia.com/amgx

- Library for algebraic multigrid

AmgX

GPU-accelerated linear solvers for simulations and implicit unstructured methods

Other notable libraries

cuDNN

- Library for Deep Neural Networks
- Some parts developed by Jeremy Appleyard (NVIDIA) working in Oxford

nvGraph

- Page Rank, Single Source Shortest Path, Single Source Widest Path

NPP (NVIDIA Performance Primitives)

- Library for imaging and video processing
- Includes functions for filtering, JPEG decoding, etc.

CUDA Video Decoder API…

GPU-accelerated library of primitives for deep neural networks

nvGRAPH

GPU-accelerated library for graph analytics

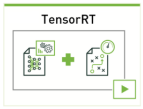
NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing

Other notable libraries

Deep Learning Libraries

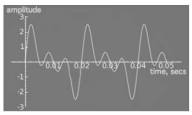
GPU-accelerated library of primitives for deep neural networks



GPU-accelerated neural network inference library for building deep learning applications

Advanced GPU-accelerated video inference library

Signal, Image and Video Libraries



cuFFT

GPU-accelerated library for Fast Fourier Transforms

NVIDIA Performance Primitives

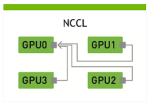
GPU-accelerated library for image and signal processing

NVIDIA Codec SDK

High-performance APIs and tools for hardware accelerated video encode and decode

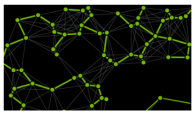
Other notable libraries

Parallel Algorithm Libraries



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



nvGRAPH

GPU-accelerated library for graph analytics

Thrust

GPU-accelerated library of parallel algorithms and data structures

MAGMA

MAGMA (Matrix Algebra on GPU and Multicore Architectures) has been available for a few years (See nice SC17 handout: http://www.icl.utk.edu/files/print/2017/magma-sc17.pdf)

- LAPACK for GPUs – higher level numerical linear algebra, layered on top of cuBLAS.
- Open source – freely available.
- Last updated February 2023.

https://icl.utk.edu/magma/
https://developer.nvidia.com/magma

ArrayFire

Originally a commercial software (from Accelereyes), but is now open source.

- Supports both CUDA and OpenCL execution.
- C, C++ and Python interfaces.
- Working to incorporate OneAPI (So should support most Intel hardware).
- Supports NVIDA and AMD GPUs/APUs, Intel processors and mobile devices from ARM, Qualcomm…
- Wide range of functionality including linear algebra, image and signal processing, random number generation, sorting…
- Actively developed.

https://arrayfire.com/
https://github.com/arrayfire/arrayfire

Thrust

Thrust is a high-level C++ template library with an interface based on the C++ Standard Template Library (STL).

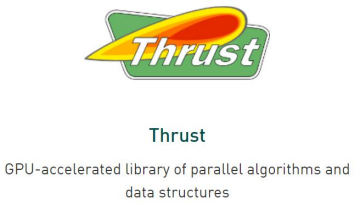
Thrust has a very different philosophy to other libraries - users write standard C++ code (*no CUDA*) but get the benefits of GPU acceleration.

Thrust relies on C++ object-oriented programming – certain objects exist on the GPU, and operations involving them are implicitly performed on the GPU.

It has lots of built-in functions for operations like sort and scan.

It also simplifies memory management and data movement.

https://thrust.github.io/



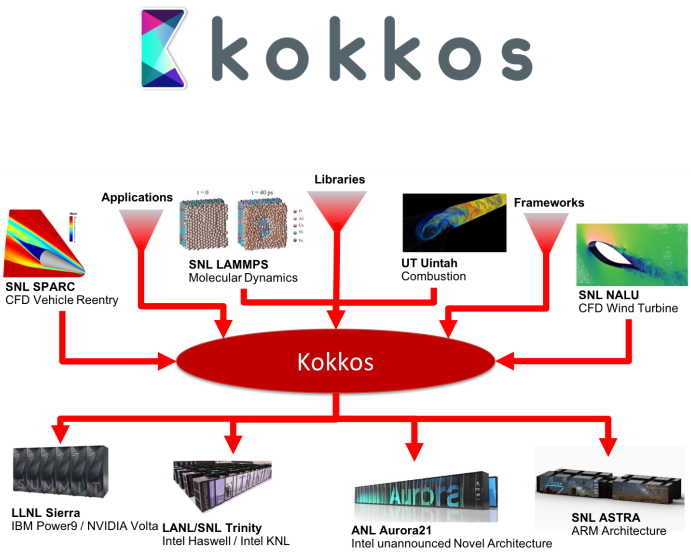
Kokkos

Kokkos is another high-level C++ template library, similar to Thrust.

It has been developed in the US DoE Labs, so there is considerable investment in both capabilities and on-going software maintenance.

Could be worth investigating if you are considering using Thrust in your projects.

For more information see
https://kokkos.github.io/kokkos-core-wiki/
https://kokkos.org/about/



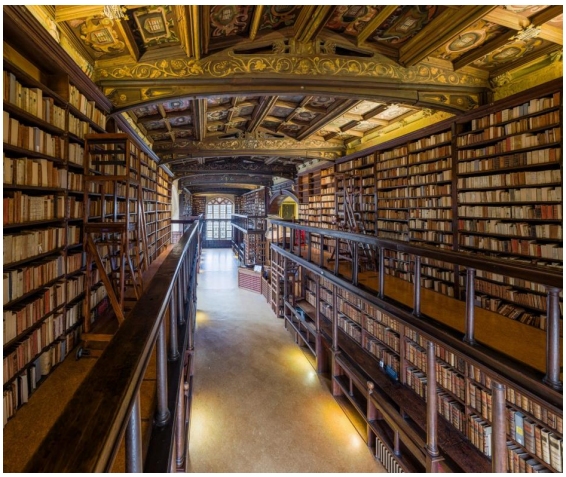
A final word on libraries

NVIDIA maintains webpages with links to a variety of CUDA libraries:

www.developer.nvidia.com/gpu-accelerated-libraries

and other tools:

www.developer.nvidia.com/tools-ecosystem



https://en.wikipedia.org/wiki/Duke_Humfrey%27s_Library

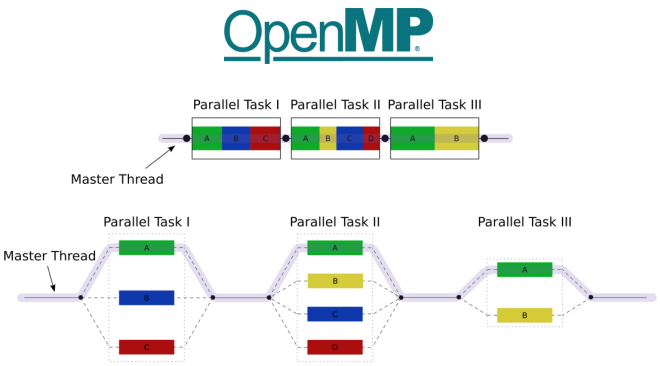
A note on directive based approaches and other languages

OpenMP

OpenMP 5.0 is a directive based approach to parallelisation.

- It uses a fork-join model.
- Can be used in C / C++ and FORTRAN codes.
- It supports both CPU and GPU hardware.

Is now becoming the industry standard for in node CPU parallelisation.



SYCL

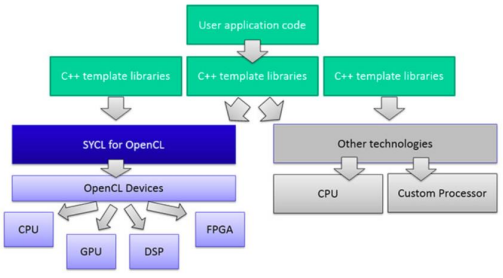
SYCL (“sickle”) - C++ Single-source Heterogeneous Programming for OpenCL.

From KHRONOS Group (responsible for OpenCL).

Provides an abstraction layer that builds on OpenCL.

It enables **code for heterogeneous processors** to be written in a **“single-source” style** using completely standard C++.

Supported by Intel, NVIDIA and AMD.



Other Languages

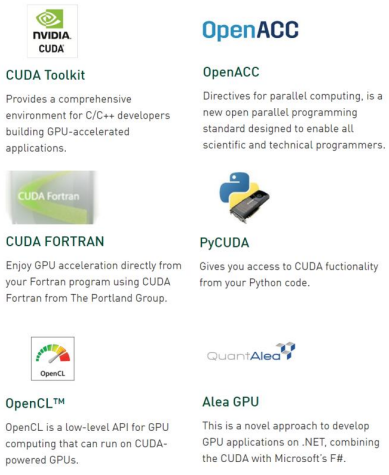
FORTRAN: CUDA FORTRAN compiler with natural FORTRAN equivalent to CUDA C.

MATLAB: can call kernels directly, or use OOP like Thrust to define MATLAB objects which live on the GPU
https://uk.mathworks.com/help/parallel-computing/run-matlab-functions-on-a-gpu.html

Mathematica: similar to MATLAB?

Python: CuPy (compatible with NumPy – acceleration for array computations), Numba and CUDA python
http://mathema.tician.de/software/pycuda
https://store.continuum.io/cshop/accelerate/
https://developer.nvidia.com/cuda-python
https://developer.nvidia.com/how-to-cuda-python
https://nvidia.github.io/cuda-python/overview.html

Other useful things…



https://developer.nvidia.com/tools-ecosystem
https://developer.nvidia.com/language-solutions
https://developer.nvidia.com/hpc-sdk
https://developer.nvidia.com/hpc-compilers

Which library should I use for my problem?

The seven dwarfs

Phil Colella a 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

The seven dwarfs

These 13 dwarfs define key algorithmic kernels in many scientific computing applications.

They have been **very helpful to focus attention on HPC challenges and development of libraries and problem-solving environments/frameworks**.

The seven dwarfs

1. Dense linear algebra
2. Sparse linear algebra
3. Spectral methods
4. N-body methods
5. Structured grids
6. Unstructured grids
7. Monte Carlo

1. Dense Linear Algebra

Many tools available, some from NVIDIA, some third party:

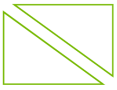
- cuBLAS
- cuSOLVER
- MAGMA
- ArrayFire

CUTLASS, an NVIDIA tool for Fast Linear Algebra in CUDA C++ might also be worth a look if you can't use the above libraries for any reason.

https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/

cuBLAS

GPU-accelerated basic linear algebra (BLAS) library



cuSOLVER

GPU-accelerated dense and sparse direct solvers

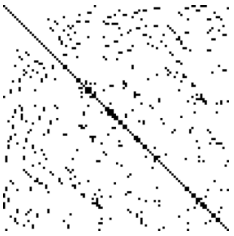
2. Sparse Linear Algebra

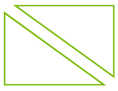
Iterative solvers

- Some available in PetSc (Portable, Extensible Toolkit for Scientific Computation, for solving PDEs) - https://petsc.org/release/overview/nutshell/
- Others can be implemented using sparse matrix-vector multiplication from cuSPARSE (is also now in PETSc).
- NVIDIA has AmgX, an algebraic multigrid library.

Direct solvers

- NVIDIA's cuSOLVER.
- SuperLU project (Gaussian elimination with partial pivoting) https://portal.nersc.gov/project/sparse/superlu/
- STRUMPACK (ask Mike) https://portal.nersc.gov/project/sparse/strumpack//





cuSOLVER

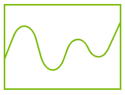
GPU-accelerated dense and sparse direct solvers

3. Spectral methods

cuFFT /cuFFTW

Library provided / maintained by NVIDIA

For those interested in FFTs on GPUs – ask karel…



cuFFT

GPU-accelerated library for Fast Fourier Transforms

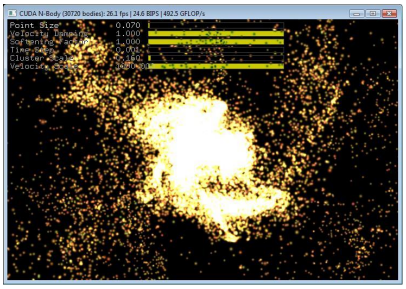
4. N-Body methods

OpenMM:

- http://openmm.org/
  open source package to support molecular modelling, developed at Stanford.

Fast multipole methods:

- ExaFMM by Yokota and Barba: http://www.bu.edu/exafmm/
- FMM2D by Holm, Engblom, Goude, Holmgren: http://user.it.uu.se/~stefane/freeware https://lorenabarba.com/figshare/exafmm-10-years-7-re-writes-the-tortuous-progress-of-computational-research/
- Software by Takahashi, Cecka, Fong, Darve: http://onlinelibrary.wiley.com/doi/10.1002/nme.3240/pdf



https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-n-body-simulation

https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/nbody/doc/nbody_gems3_ch31.pdf

5. Structured grids

Lots of people have developed one-off applications.

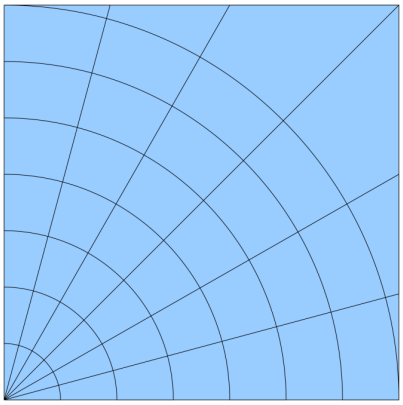
No great need for a library for single block codes (though possible improvements from “tiling”?).

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.

For those interested – ask Mike…

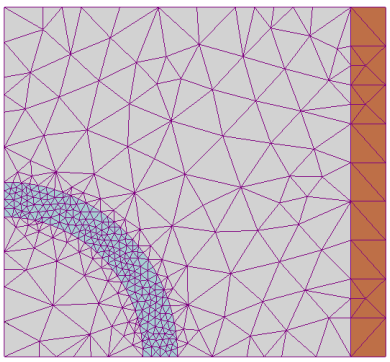


6. Unstructured grids

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

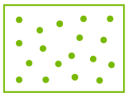
- Alonso, Darve and others (Stanford).
- Oxford / Imperial College / Warwick project developed OP2, a general-purpose open-source framework based on a previous framework built on MPI.
- If there’s interest Mike could talk about OP2 and OPS in lecture 8/9.

See https://op-dsl.github.io/ for both OPS and OP2



7. Monte Carlo methods

- NVIDIA cuRAND library.
- ArrayFire library.
- Some examples in CUDA SDK distribution.
- Nothing else needed except for more output distributions?



cuRAND

GPU-accelerated random number generation (RNG)

{ } ARRAYFIRE

Useful tools

Tools - Debugging

`compute-sanitizer –tool memcheck`
A command line tool that detects array out-of-bounds errors, and mis-aligned device memory accesses – very useful because such errors can be tough to track down otherwise.

`compute-sanitizer --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.

`compute-sanitizer --tool initcheck`
This detects the reading of uninitialised device memory.

`compute-sanitizer --tool synccheck`
This detects incorrect use of __syncthreads() and related intrinsics.

https://developer.nvidia.com/compute-sanitizer

Tools – CUDA-GDB

For those familiar with the GNU debugger – GDB, this is an extension of GDB that allows users to debug both GPU and CPU code.

All existing GDB debugging features are included for debugging host code and then further functionality allows the user to debug device code.

Supports C/C++ and Fortran (that includes CUDA code).

CUDA-GDB

Delivers a seamless debugging experience that allows you to debug both the CPU and GPU portions of your application simultaneously. Use CUDA-GDB on Linux or MacOS, from the command line, DDD or EMACS.

Tools - IDEs

Integrated Development Environments (IDE):

Nsight Systems – Unified IDE for Windows/Linux/Mac/Jetson:
https://developer.nvidia.com/nsight-systems

Nsight Visual Studio edition – NVIDIA plugin for Microsoft Visual Studio
http://developer.nvidia.com/nvidia-nsight-visual-studio-edition

Nsight Eclipse plugins
https://docs.nvidia.com/cuda/nsight-eclipse-plugins-guide/index.html

these come with editor, debugger, profiler integration

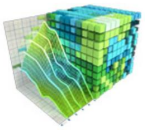
NVIDIA® Nsight™

The ultimate development platform for heterogeneous computing. Work with powerful debugging and profiling tools, optimize the performance of your CPU and GPU code. Find out about the Ecilipse Edition and the graphics debugging enabled Visual Studio Edition.

Tools - Profiling

NVIDIA Profiler `ncu` or `ncu-ui` for a graphical interface.

- This is a standalone piece of software for Linux and Windows systems.
- It uses hardware counters to collect a lot of useful information.
- Lots of things can be measured, but the limited number of counters means that, for some larger applications, it runs the application multiple to gather necessary information.
- The `ncu` CLI can be useful if you want to profile on a machine that you don't have a graphical interface to.
- Do `ncu --help` for more info on different options.



NVIDIA Visual Profiler

This is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. First introduced in 2008, Visual Profiler supports all CUDA capable NVIDIA GPUs shipped since 2006 on Linux, Mac OS X, and Windows.

https://docs.nvidia.com/nsight-compute/NsightCompute/index.html
https://docs.nvidia.com/cuda/profiler-users-guide/index.html

What have we learnt?

In this lecture we've looked at the **wide software ecosystem that now surrounds GPU computing** and how that can be used to make your life as a programmer easier.

We've looked at **directives based approaches** and how these are useful.

Finally we've looked at **tools** that allow us to develop CUDA code in an easy and maintainable way.

Lecture 6
Using multiple GPUs and loose ends

Prof Wes Armour

wes.armour@eng.ox.ac.uk

Prof Mike Giles

mike.giles@maths.ox.ac.uk

Oxford e-Research Centre

Department of Engineering Science

Learning outcomes

In this sixth lecture we will look at CUDA streams and how they can be used to increase performance in GPU computing. We will also look at some other useful odds and ends.

You will learn about:

- Synchronicity between host and device.
- Multiple streams and devices.
- How to use multiple GPUs.
- Some other odds and ends.

Setting the scene

Modern computers are typically comprised of many different components.

- Central Processing Unit (CPU).
- Random Access Memory (RAM).
- Graphics Processing Unit (GPUs).
- Hard Disk Drive (HDD) / Solid State Drive (SSD).
- Network Interface Controller (NIC)…

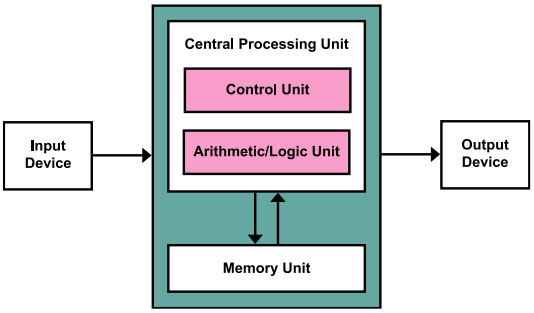
Typically, each of these different components will be performing a different task, maybe for different processes, at the same time.

Synchronicity

Synchronicity

The von Neumann model of a computer program is synchronous with each computational step taking place one after another (because instruction fetch and data movement share the same communication bus).

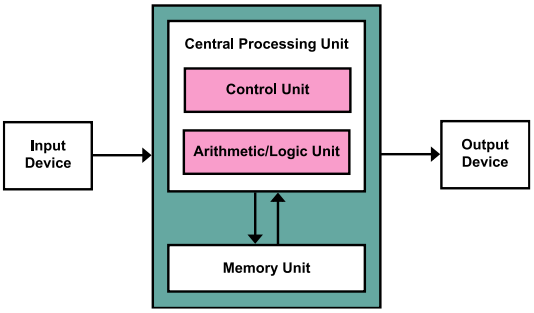
This is an idealisation, and is almost never true in practice.



Synchronicity

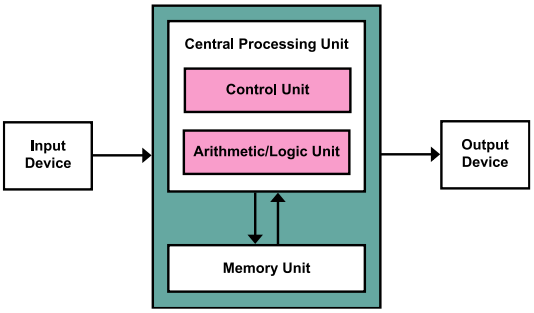
Compilers will generate code with overlapped instructions (pipelining – see lecture one), re-arrange execution order and avoid redundant computations to produce more optimal code.

As a programmer we don't normally worry about this and think of execution sequentially when working out whether a program gives the correct result.



Synchronicity

However, when things become asynchronous, the programmer has to think very carefully about what is happening and in what order!



Synchronicity - GPUs

When writing code for GPUs we have to think even more carefully, because:

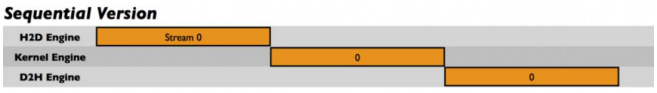
Our host code executes on the CPU(s);

Our 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?



The most important thing is to try to get a clear idea of what is going on, and when – then you can work out the consequences…

Simple host code

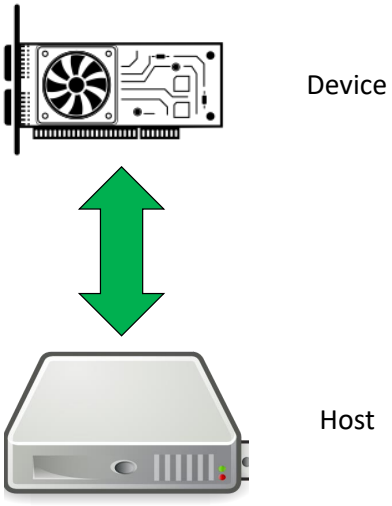
The basic / simple / default behaviour in CUDA is that we have:

1x CPU.

1x GPU.

1x thread on CPU (i.e. scalar code).

1x "**stream**" on GPU (called the "**default stream**").



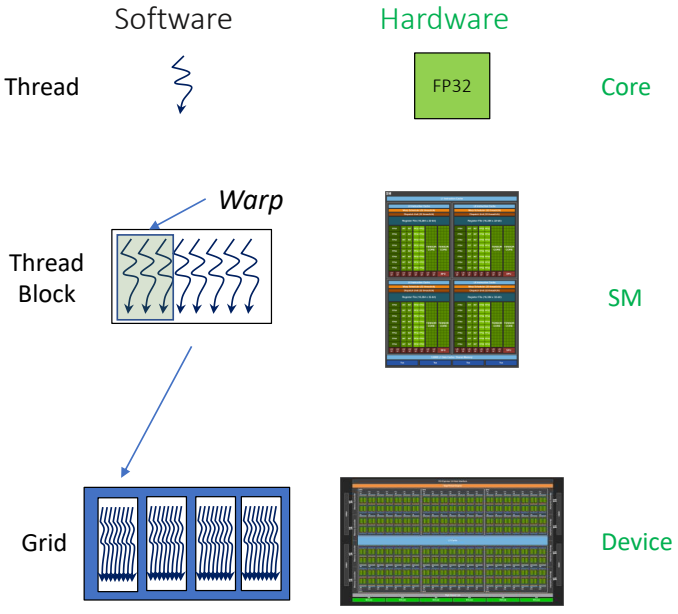
Recap – GPU Execution

We've looked at how code executes on GPUs, lets have a quick recap:

- For each warp, code execution is effectively synchronous within the warp.
- 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.

So nothing new here.

*Over the next few slides we will discuss streams – **asynchronous execution** and the implications for CPU and GPU execution.*



Blocking and non-blocking calls

Host code – blocking calls

Most CUDA calls are synchronous (often called “blocking”).

An example of a blocking call is `cudaMemcpy()`.

1. Host call starts the copy (HostToDevice / DeviceToHost).
2. Host **waits** until it the copy has finished.
3. Host continues with the next instruction in the host code once the copy has completed.

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

Host code – blocking calls

Why do this???

This mode of operation ensures correct execution.

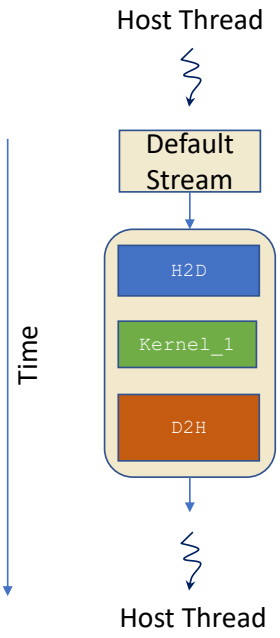
For example it ensures that data is present if the next instruction needs to read from the data that has been copied…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```


Host code – blocking calls

So the control flow for our code looks something like…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

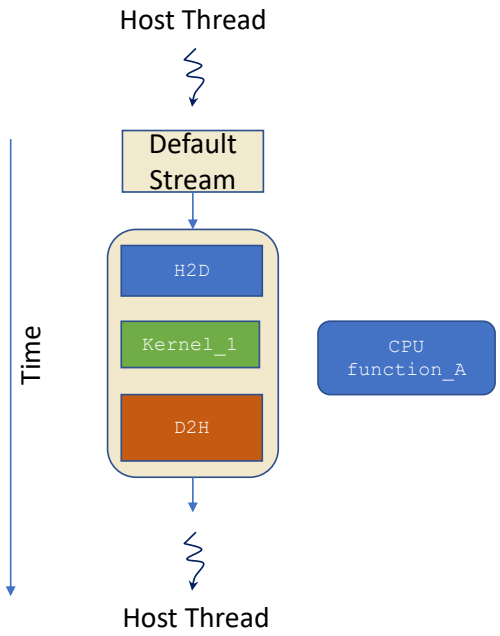


Host code – non-blocking calls

In CUDA, kernel launches are asynchronous (often called “non-blocking”).

An example of kernel execution from host perspective:

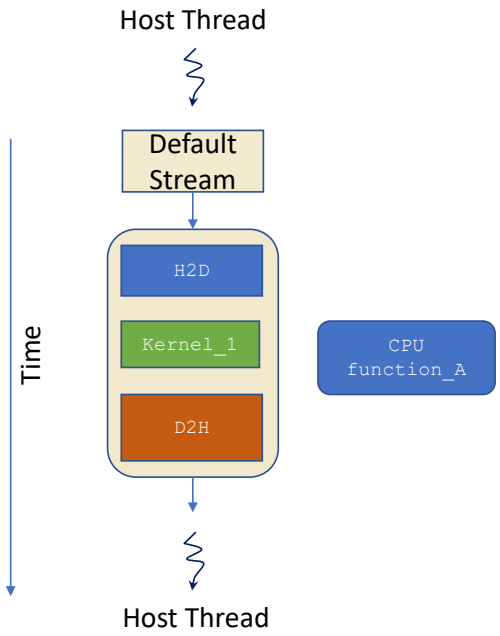
1. Host call starts the kernel execution.
2. Host does not wait for kernel execution to finish.
3. Host moves onto the next instruction.



Host code – non-blocking calls

The “crazy code” for our last control flow diagram might look like…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
CPU_function_A( … );
cudaMemcpy ( …, D2H );

…
```



Host code – non-blocking calls

Another example of a non-blocking call is `cudaMemcpyAsync()`.

This function starts the copy but doesn't wait for completion.

Synchronisation is performed through a "stream".

You must use page-locked memory (also known as pinned memory) – see Documentation.

In both of our examples, the host eventually waits when at (for example) a `cudaDeviceSynchronize()` call.

The benefit of using streams is that you can improve performance (in some cases, not all) by overlapping communication and compute, or CPU and GPU execution.

Asynchronous host code

When using asynchronous calls, things to watch out for, and things that can go wrong are:

- 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).

CUDA Streams

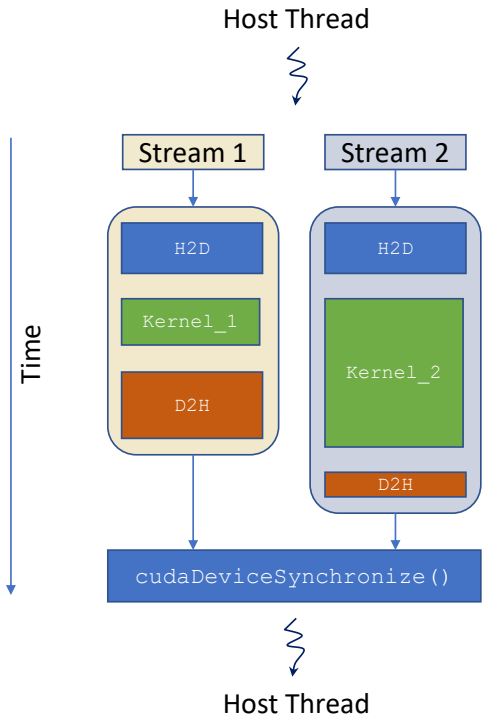
CUDA Streams

Quoting from section 6.2.8.5 in the CUDA Programming Guide:

Applications manage concurrency through streams.

A stream is a sequence of commands (*possibly issued by different host threads*) that execute in order.

Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently.



http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

Multiple CUDA Streams

When using streams in CUDA, you must supply a "stream" variable as an argument to:

- kernel launch
- `cudaMemcpyAsync()`

Which is created using `cudaStreamCreate();`

As shown over the last couple of slides:

- Operations within the same stream are ordered - (i.e. FIFO – first in, first out) – they cant overlap.
- Operations in different streams are unordered wrt each other and can overlap.

Use multiple streams to increase performance by overlapping memory communication with compute.

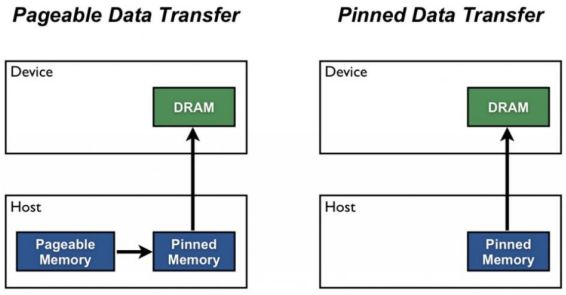
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
my_kernel_one<<<blocks,threads,0,stream1>>>(…);
cudaStreamDestroy(stream1);
```

An example of launching a kernel in a stream that isn't the "default stream".

Page-locked / Pinned memory

Section 6.2.6 of the cuda programming guide:

- Host memory is usually paged, so run-time system keeps track of where each page is located.
- For higher performance, pages can be fixed (fixed address space, always in RAM), 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
- You must use page-locked memory with `cudaMemcpyAsync();`
- Page-locked memory is allocated using `cudaHostAlloc()`, or registered by `cudaHostRegister();`



Pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory.

https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

The default stream

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

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

This forces old (bad) behaviour in which a cudaMemcpy or kernel launch on the default stream blocks/synchronizes with other streams.

Or `--default-stream per-thread`

This forces new (good) behaviour in which the default stream doesn't affect the others.

For more info see the nvcc documentation:

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-cuda-compilation

Practical 11

An example is given in practical 11 for those interested, try with the two different flags:

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

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

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

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

The default stream

The second (most useful?) 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-interacting) default streams for each thread.

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

However “proper” streams within cuda are very versatile and fully featured, so might be worth the time and complexity investment.

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

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

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

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

cudaDeviceSynchronize();
```


Stream commands

As previously shown, each stream executes a sequence of cuda calls. However to get the most out of your heterogeneous computer you might also want to do something on the host.

There are at least two ways of coordinating this:

Use a separate thread for each stream

- It can wait for the completion of all pending tasks, then do what’s needed on the host.

Use just one thread for everything

- For each stream, add a callback function to be executed (by a new thread) when the pending tasks are completed.
- It can do what’s needed on the host, and then launch new kernels (with a possible new callback) if wanted.

Stream commands

Some useful stream commands are:

`cudaStreamCreate(&stream)`
Creates a stream and returns an opaque "handle" – the "stream variable".

`cudaStreamSynchronize(stream)`
Waits until all preceding commands have completed.

`cudaStreamQuery(stream)`
Checks whether all preceding commands have completed.

`cudaStreamAddCallback()`
Adds a callback function to be executed on the host once all preceding commands have completed.

http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

Stream commands

Functions 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 (doesn't block the host).

`cudaEventQuery(event)`
Check whether event has occurred.

`cudaEventElapsedTime(time,event1,event2)`
Times between event1 and event2.

Multi-GPU computing

Multiple devices

What happens if there are multiple GPUs?

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

- By default a CUDA application uses the lowest number device which is “visible” and available (this might not be what you want).
- Visibility controlled by environment variable `CUDA_VISIBLE_DEVICES`.
- The current device can be chosen/set by using `cudaSetDevice()`
- `cudaGetDeviceProperties()` does what it says, and is very useful.
- Each stream is associated with a particular device, which is the “current” device for a kernel launch or a memory copy.
- see `simpleMultiGPU` example in SDK or section 6.2.9 for more information.

https://www.flickr.com/photos/ebastian/8804000077

Multiple devices

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

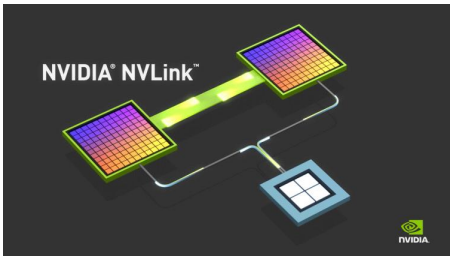
This is the premise of the NVlink interconnect, which is much faster than PCIe (900GB/s P2P on Hopper).

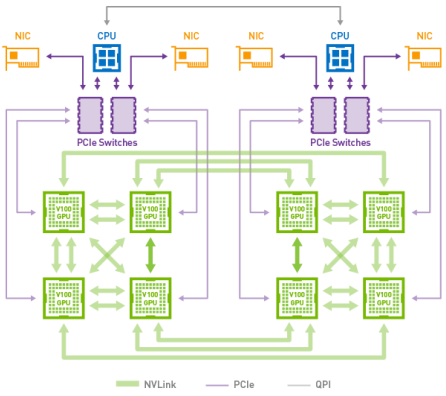
`cudaMemcpy()` can do direct copy from one GPU's memory to another.

A kernel on one GPU can also read directly from an array in another GPU's memory, or write to it.
This even includes the ability to do atomic operations with remote GPU memory.

For more information see Section 6.13, "Peer Device Memory Access" in CUDA Runtime API documentation:
https://docs.nvidia.com/cuda/cuda-runtime-api/

https://fuse.wikichip.org/news/1224/a-look-at-nvidias-nvlink-interconnect-and-the-nvswitch/





Multi-GPU computing

Multi-GPU computing exists at all scales, from cheaper workstations using PCIe, to more expensive Quadro / Titan products using fewer NVLink, to high-end NVIDIA DGX servers.

Single workstation / server:

- a big enclosure for good cooling!
- up to 4 high-end cards in 16x PCIe v4 slots – up to 16GB/s interconnect.
- 2x high-end CPUs.
- 2-3kW power consumption – not one for the office!

NVIDIA DGX H100 Deep Learning server:

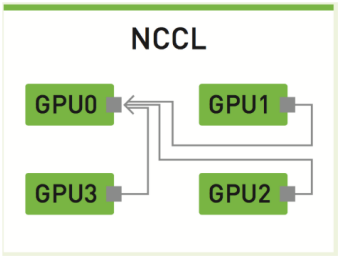
- 8 NVIDIA GH100 GPUs, each with 80GB HBM2.
- 2× 56-core Intel Xeons (Platinum 8480C 2.0 GHz).
- 2 TB RAM memory, 8x 3.84TB NVMe.
- 900GB/s NVlink interconnect between the GPUs.
- ~£379,000

Multi-GPU Computing

How do you use these machines?

This depends on hardware choice:

- For single machines, use shared-memory multithreaded host application.
- For DGX products you must use the NVIDIA Collective Communications Library (NCCL).
- For clusters / supercomputers, use distributed-memory MPI message-passing.



https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/

MPI approach

In the MPI approach:

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

Multi-user support

What if different processes try to use the same device?

The behaviour of the device depends on the system compute mode setting (section 3.4):

In “default” mode, each process uses the fastest device:
- This is good when one very fast card, and one very slow card.
- But not very useful when you have two identical fast GPUs (one sits idle).

In “exclusive” mode, each process is assigned to first unused device;
However code will return an error if none are available.

`cudaGetDeviceProperties()` reports the mode setting

The mode can be changed by a user account with sys-admin privileges using the `nvidia-smi` command line utility.

Some tips and tricks

Loose ends – Loop unrolling

Section 10.37 (of the programming guide):
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];
```

This is a standard compiler trick to avoid the cost of incrementing and looping.

The pragma

```
#pragma unroll 5
```

will also force unrolling for loops that do not have explicit limits.

Loose ends – const __restrict__

Section 10.2.6 (of the programming guide):

`__restrict__` keyword

The qualifier asserts that there is no overlap (in memory space) between `a,b,c`, for example we do not have:

```
a[i]=q[i]
b[i]=q[i+1]
```

(you have no pointer aliasing) so the compiler can perform more optimisations.

The following blog post demonstrates how this can achieve a good speed increase:

https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/#disqus_thread

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
               float* __restrict__ c) {

        for (i=1; i<N; i++) {
                a[i] = b[i] + c[i];
        }
        ...
}
```

Loose ends - volatile

Section 17.5.3.3 (of the programming guide):
`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 because the compiler can optimize locations in shared memory by locating them in registers (but register scope is specific to a single thread), for any thread.

Loose ends - Compilation

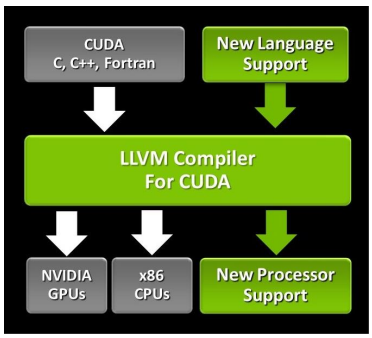
Compiling:

The `Makefile` for first few practicals uses `nvcc` to compile both the host and the device code.

Internally `nvcc` uses `gcc` for the host code (at least by default). The device code compiler is based on the open source LLVM compiler.

It often makes sense to use different compilers, for example `icc` which is for host code which does not have kernel launches.

To do this you must use the `-fPIC` flag to produce position-independent-code (this just generates machine code that will execute properly, independent of where it's held in memory).



https://developer.nvidia.com/cuda-llvm-compiler

Loose ends - Compilation

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)
```


Loose ends - Compilation

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)
```

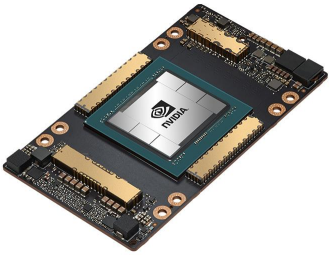
Loose ends - Compilation

Other useful compiler options:

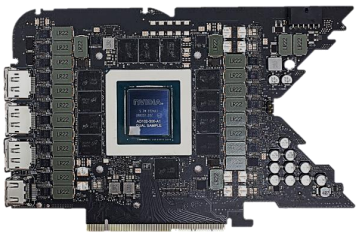
`-arch=sm_80`
This specifies GPU architecture (in this case sm_80 is for Ampere A100).

`-maxrregcount=n`
This asks the compiler to **generate code using at most n registers**; the compiler may ignore this if it's not possible, but it may also increase register usage up to this limit.

This is less important now since threads can have up to 255 registers, but can be useful in some instances to reduce register pressure and enable more thread blocks to run.



or



Loose ends – Compilation

Launch bounds (10.36):

`-maxrregcount` is given as an argument to the compiler (`nvcc`) and modifies the default for all kernels.

A per kernel approach can be taken by using the `__launch_bounds__` qualifier:

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,minBlocksPerMultiprocessor)
MyKernel(...) {
…
}
```

Summary

This lecture has discussed a number of more advanced topics. As a beginner, you can ignore almost all of them As you get more experienced, you will probably want to start using some of them to get the very best performance.

Lecture 7:
tackling a new application

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Initial planning

1) Has it been done before?

- check with Google
- ask a local expert
- check CUDA sample codes
- sign up to the CUDA Developer Program (free) and check out relevant Video-on-Demand talks from the last GTC (GPU Technology Conference)
- check out the NVIDIA Developer blogs: https://developer.nvidia.com/blog (very good for info on new hardware architectures as well as new software features)

Initial planning

2) Where is the parallelism?

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

Initial planning

Sometimes you need to think about "the bigger picture"

Already considered 3D finite difference example:

- lots of grid nodes so lots of inherent parallelism
- even for ADI method, a grid of 256^3 has 256^2 tri-diagonal solutions to be performed in parallel so OK to assign each one to a single warp

  (optional lecture 8 on how best to solve tri-diagonal equations on GPUs – involves doing more computation to reduce the amount of communication)
- but what if we have a 2D or even 1D problem to solve?

Initial planning

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

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

- different initial data
- different model constants

This adds to the available parallelism

Initial planning

2D:

- 128KB of shared memory on Ampere == 32K `float`
  so grid of 64^2 could be held within shared memory
  - one kernel for entire calculation
  - each block handles a separate 2D problem; possibly two block per SM
- for bigger 2D problems, might need to split each one across more than one block
  - separate kernel for each timestep / iteration

Initial planning

1D:

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

Initial planning

Parallel Cyclic Reduction (PCR): starting from

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

with $a_0 = c_{N-1} = 0$, subtract a_n times row $n-1$, and c_n times row $n+1$ and re-normalise to get

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

with $a^*_m{=}0$ for $m{<}2$ and $c^*_m{=}0$ for $m{\geq}N{-}2$.

Repeating this $\log_2 N$ times gives the value for x_n (since the values of the final a's and c's will be zero) and each step can be done in parallel.

(Practical 7 uses shared memory, but if $N \leq 32$ it fits in a single warp and can be implemented using shuffles.)

Initial planning

3) Break the algorithm down into its constituent pieces

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

Initial planning

4) Is there a problem with warp divergence?

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

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

- . . . or again seek expert help

Initial planning

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

- usually best to move whole application onto GPU, so not limited by PCIe v4 bandwidth (32GB/s)
- occasionally, OK to keep main application on the host and just off-load compute-intensive bits
- dense linear algebra is a good off-load example; data is $O(N^2)$ but compute is $O(N^3)$ so fine if N is large enough

Heart modelling

Heart modelling is another interesting example:

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

Initial planning

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

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

Initial planning

If compute-intensive:

- don't worry (too much) about cache efficiency
- minimise integer index operations
- if using double precision, think whether it's needed

If data-intensive:

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

Initial planning

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

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

Initial planning

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

- look for NVIDIA sample codes demonstrating use of the feature
- write some trivial little test problems of your own
- check you really understand how they work

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

Initial planning

Read NVIDIA documentation on performance optimisation:

- Section 5 of CUDA C++ Programming Guide
- CUDA C++ Best Practices Guide
- Volta Tuning Guide
- Ampere Tuning Guide
- Hopper Tuning Guide

Programming and debugging

Many of my comments here apply to all scientific computing

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

debugging can be hard!

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

Programming and debugging

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

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

Programming and debugging

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

Very important!

Forgetting a

```
__syncthreads();
```

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

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

Reminder: `compute-sanitizer --tool racecheck` to check for race condition

Programming and debugging

In developing `laplace3d`, my approach was to

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

Performance improvement

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

- often hard to predict optimal size *a priori*
- optimal size can also vary on different hardware
- with early GPUs, could gain $2\times$ improvement by re-optimising the block sizes
- probably not as much change these days between successive generations

  (not so much change in SMs, more a change in the number of SMs, the size of L2 cache, and new features like Tensor Cores)

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 a good idea for GPU programming, though I have not seen it used by others:

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

Performance improvement

Use profiling to understand the application performance:

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

The Nsight Compute profiler is powerful

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

The Nsight Systems profiler gives a top-level view and is relatively easy to use.

Going further

In some cases, a single GPU is not sufficient

Shared-memory option:

- single system with up to 16 GPUs
- GPUs linked by either PCIe (direct or via CPU) or NVlink (much faster)
- single process with a separate host thread for each GPU, or use just one thread and switch between GPUs
- can transfer data directly between GPUs – NVIDIA software will use the fastest route, avoiding the CPU if possible

Going further

Distributed-memory option:

- a cluster, with each node having 1 or 2 GPUs
- nodes connected by high-speed Ethernet/Infiniband networking with PCIe network cards
- simplest approach is MPI message-passing, with separate process for each GPU
- modern MPI software has full support for CUDA, with direct data transfers (no intermediate copies in CPU) where possible

`https://developer.nvidia.com/mpi-solutions-gpus`
`https://developer.nvidia.com/gpudirect`

Final words

- it continues to be an exciting time for HPC
- coding to get a good fraction of peak performance remains challenging – computer science objective should be to simplify this for developers through
  - libraries
  - domain-specific high-level languages
  - code transformation
- confident prediction: GPUs and other accelerators such as vector units will remain dominant in HPC for next 10 years, so it's worth your effort to re-design and re-implement your algorithms

OP2 – an open-source library for unstructured grid applications

(2023 comment: slides originally from 2013)

Mike Giles, Gihan Mudalige, Istvan Reguly

mike.giles@maths.ox.ac.uk

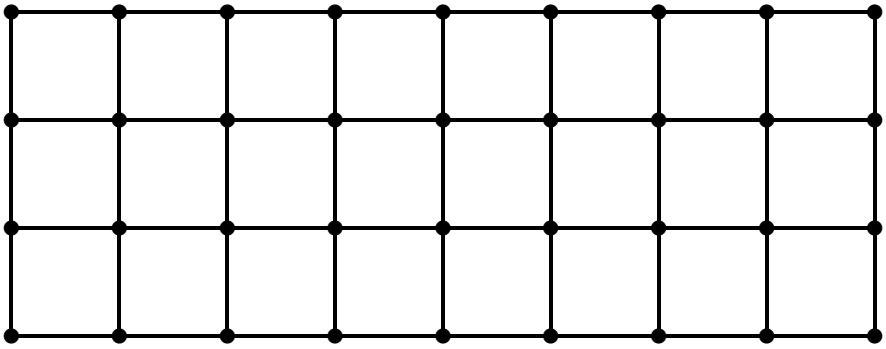
Oxford University Mathematical Institute

Oxford e-Research Centre

Outline

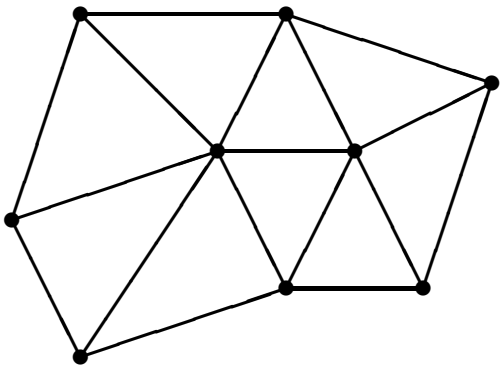
- structured and unstructured grids
- software challenge
- user perspective (i.e. application developer)
  - API
  - build process
- implementation issues
  - hierarchical parallelism on GPUs
  - data dependency
  - code generation
  - auto-tuning

Structured grids



- logical (i, j) indexing in 2d; (i, j, k) in 3D
- implicit connectivity – neighbours of node (i, j, k) are $(i \pm 1, j \pm 1, k \pm 1)$
- fairly easy to parallelise – see `laplace3d` example

Unstructured grids



- a collection of nodes, edges, faces, cells, etc., each addressed by a 1D index
- explicit connectivity – mapping tables define connections from edges to nodes, or faces to cells, etc.
- much harder to parallelise (not in concept so much as in practice) but a lot of existing literature on the subject
- used a lot because of geometric flexibility

Software Challenge

- Application developers want the benefits of the latest hardware but are very worried about the software development effort, and the expertise required
- Status quo is not really an option – running lots of single-thread MPI processes on multiple CPUs won't give great performance
- Want to exploit GPUs using CUDA, and CPUs using OpenMP/AVX
- However, hardware is likely to change rapidly in next few years, and developers can not afford to keep changing their software implementation

Software Abstraction

To address this challenge, need to move to a suitable level of abstraction:

- separate the user's specification of the application from the details of the parallel implementation
- aim to achieve application level longevity with the user specification not changing for perhaps 10 years
- aim to achieve near-optimal performance through re-targetting the back-end implementation to different hardware and low-level software platforms

History

OPlus (Oxford Parallel Library for Unstructured Solvers)

- developed for Rolls-Royce 10 years ago
- MPI-based library for HYDRA CFD code on clusters with up to 200 nodes

OP2:

- open source project
- keeps OPlus abstraction, but slightly modifies API
- an “active library” approach with code transformation to generate CUDA for GPUs and OpenMP/AVX for CPUs

OP2 Abstraction

- sets (e.g. nodes, edges, faces)
- datasets (e.g. flow variables)
- mappings (e.g. from edges to nodes)
- parallel loops
  - operate over all members of one set
  - datasets have at most one level of indirection
  - user specifies how data is used (e.g. read-only, write-only, increment)

OP2 Restrictions

- set elements can be processed in any order, doesn't affect result to machine precision
  - explicit time-marching, or multigrid with an explicit smoother is OK
  - Gauss-Seidel or ILU preconditioning is not
- static sets and mappings (no dynamic grid adaptation)

OP2 API

```
void op_init(int argc, char **argv)

op_set op_decl_set(int size, char *name)

op_map op_decl_map(op_set from, op_set to,
                   int dim, int *imap, char *name)

op_dat op_decl_dat(op_set set, int dim,
                   char *type, T *dat, char *name)

void op_decl_const(int dim, char *type,
                   T *dat)

void op_exit()
```

OP2 API

Example of parallel loop syntax for a sparse matrix-vector product:

```
op_par_loop(res,"res", edges,
  op_arg_dat(A,-1,OP_ID, 1,"float",OP_READ),
  op_arg_dat(u, 1,pedge,1,"float",OP_READ),
  op_arg_dat(du,0,pedge,1,"float",OP_INC));
```

This is equivalent to the C code:

```
for (e=0; e<nedges; e++)
   du[pedge[2*e]] += A[e] * u[pedge[1+2*e]];
```

where each “edge” corresponds to a non-zero element in the matrix `A`, and `pedge` gives the corresponding row and column indices.

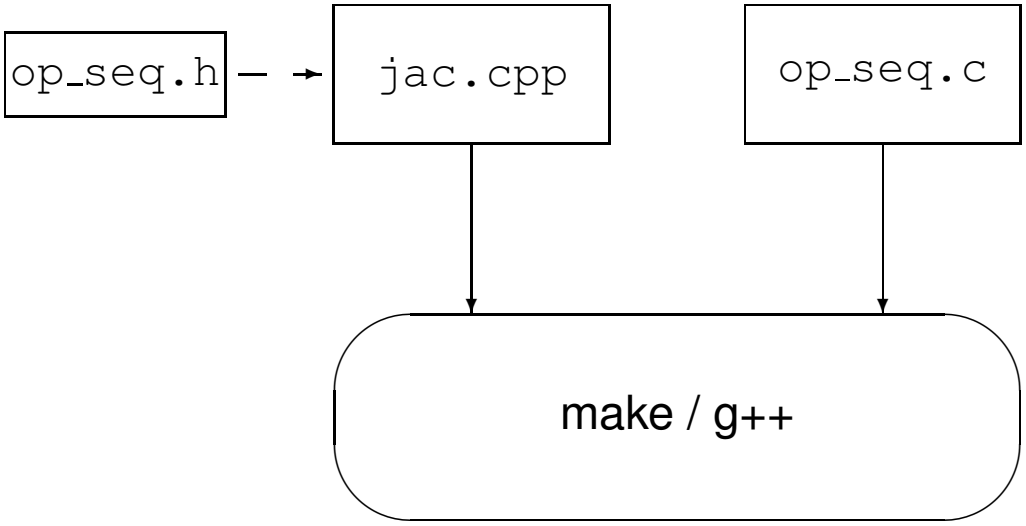
User build processes

Using the same source code, the user can build different executables for different target platforms:

- sequential single-thread CPU execution
  - purely for program development and debugging
  - very poor performance
- CUDA for single GPU
- OpenMP/AVX for multicore CPU systems
- MPI plus any of the above for clusters

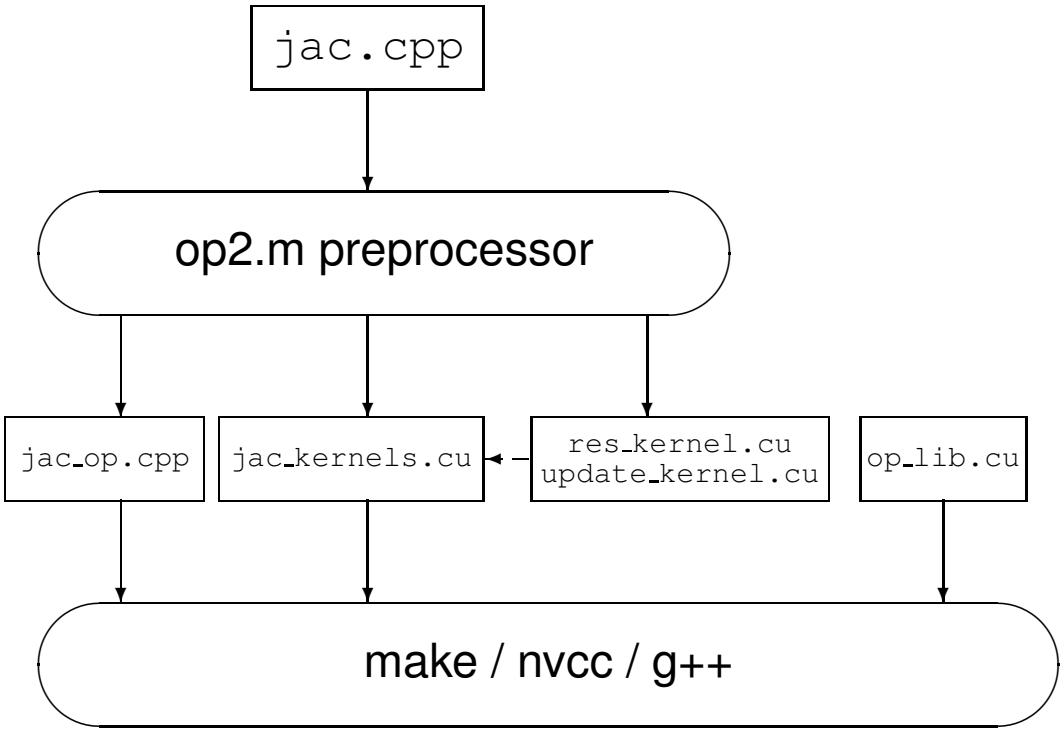
Sequential build process

Traditional build process, linking to a conventional library in which many of the routines do little but error-checking:



CUDA build process

Preprocessor parses user code and generates new code:



Implementation Approach

The question now is how to deliver good performance on multiple GPUs

Initial assessment:

- lots of natural parallelism on grids with up to 10^9 nodes/edges
- not a huge amount of compute per node/edge so important to
  - avoid PCIe transfers as much as possible
  - achieve good data reuse to minimise GPU / global memory transfers
- have to be careful with data dependencies

GPU Parallelisation

Could have up to 10^6 threads in 3 levels of parallelism:

- MPI distributed-memory parallelism (1-100)
  - one MPI process for each GPU
  - all sets partitioned across MPI processes, so each MPI process only holds its data (and halo)
  - each partition sized to fit within global memory of GPU (up to 6GB)
  - only halos need to be transferred from one GPU to another, via the CPUs
  - hopefully, this will give a balanced implementation – slight possibility that MPI networking will end up being the primary bottleneck, so will work hard to overlap computation and MPI communication

GPU Parallelisation

- block parallelism (50-1000)
  - on each GPU, data is broken into mini-partitions, worked on separately and in parallel by different SMs within the GPU
  - each mini-partition is sized so that all of the indirect data can be held in shared memory and re-used as needed
  - implementation requires re-numbering from global indices to local indices – tedious but not difficult
  - can use different mini-partitions for different parallel loops – "execution plan" generated during startup
- thread parallelism (32-128)
  - each mini-partition is worked on by a block of threads in parallel

Shared memory or L1 cache?

Caches:

- easy to use, but hard to predict/understand performance
- good performance for structured grids where often all of the cache line is used
- not so good for unstructured grids with indirect addressing

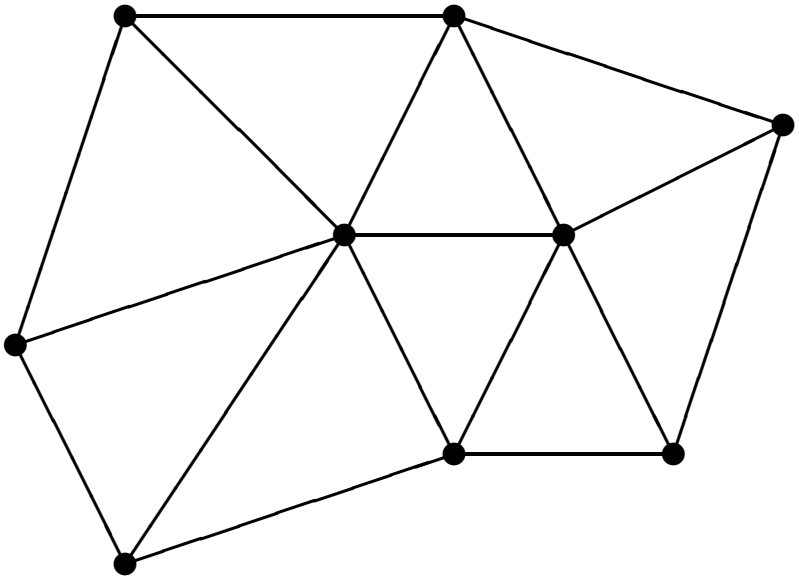
Shared memory:

- full control means you understand performance
- only store the data which is actually needed
- tedious to implement, but that's the point of a library, to do the tedious things so users don't have to

Data dependencies

Key technical issue is data dependency when incrementing indirectly-referenced arrays.

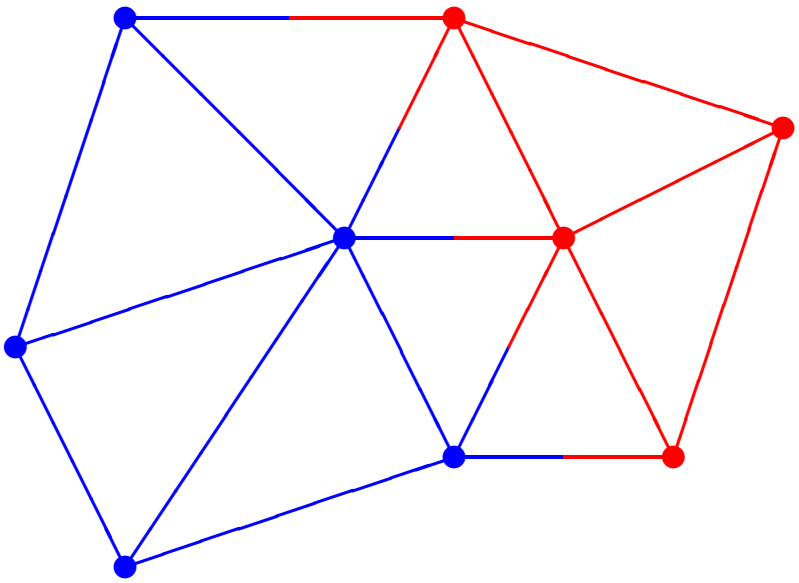
e.g. potential problem when two edges update same node



Data dependencies

Method 1: “owner” of nodal data does edge computation

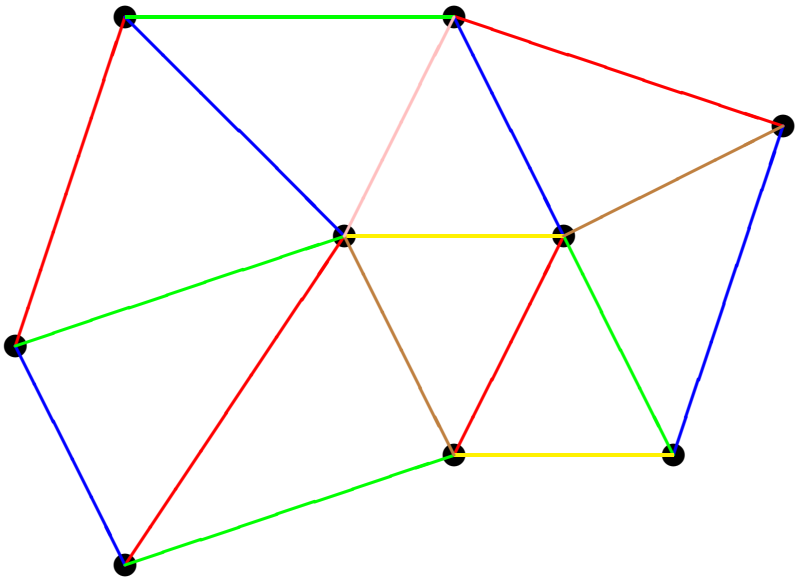
- drawback is redundant computation when the two nodes have different “owners”



Data dependencies

Method 2: “color” edges so no two edges of the same color update the same node

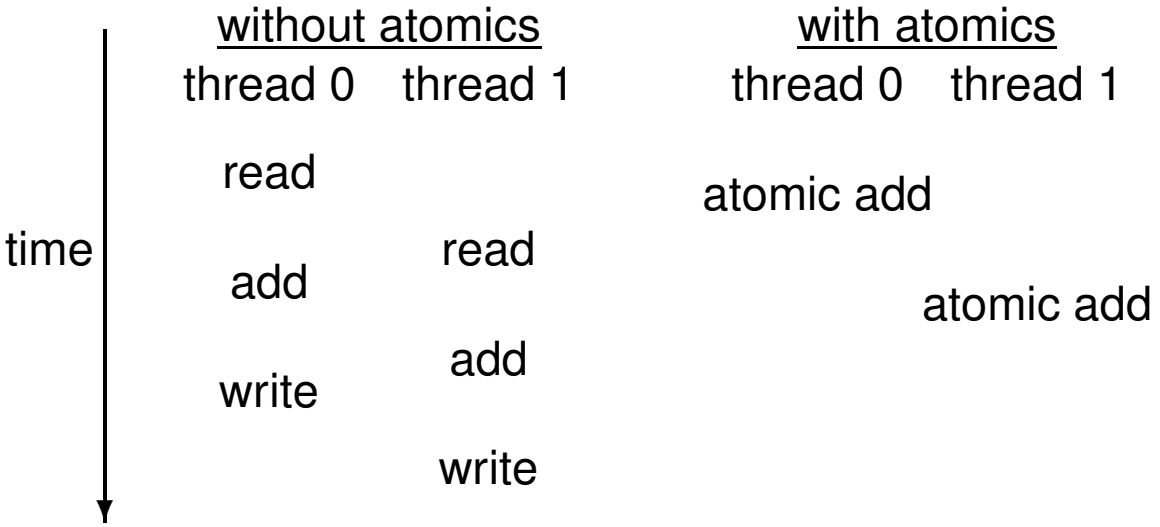
- parallel execution for each color, then synchronize
- possible loss of data reuse and some parallelism



Data dependencies

Method 3: use “atomic” add which combines read/add/write into a single operation

- avoids the problem but needs hardware support
- drawback is slow hardware implementation



Data dependencies

Which is best for each level?

- MPI level: method 1
  - each MPI process does calculation needed to update its data
  - partitions are large, so relatively little redundant computation
- GPU level: method 2
  - plenty of blocks of each color so still good parallelism
  - data reuse within each block, not between blocks
- block level: method 2 (2023 update: method 3)
  - indirect data in local shared memory, so get reuse
  - individual threads are colored to avoid conflict when incrementing shared memory

Code Generation

Initial prototype, with code parser/generator written in MATLAB (2023 update: later switched to python), can generate:

- CUDA code for a single GPU
- OpenMP code for multiple CPUs

The parallel loop API requires redundant information:

- simplifies MATLAB program generation – just need to parse loop arguments, not entire code
- numeric values for dataset dimensions enable compiler optimisation of CUDA code
- "programming is easy; it's debugging which is difficult" – not time-consuming to specify redundant information provided consistency is checked automatically

Auto-tuning

In the CUDA implementation there are various parameters and settings which apply to the whole code:

- compiler flags, such as whether to use L1 caching
- (whether to use AoS or SoA storage for each dataset)

and others which can be different for each CUDA kernel:

- number of threads in a thread block
- size of each mini-partition
- (whether to use a 16/48 or 48/16 split for the L1 cache / shared memory)

Auto-tuning

In each case, the optimum choice / value is not obvious, but it is possible to

- give a small set of possible values for each (usually two or three)
- state which can be optimised independently (e.g. the parameters for one kernel don't affect the execution of another kernel)

What is then needed is a flexible auto-tuning system to select the optimum combination by exhaustive "brute force" search.

The parameter independence is essential to making this viable.

Auto-tuning

A flexible auto-tuning package has been developed:

- written in Python
- input specification includes
  - parameters and possible values
  - a mechanism to compile the code, perhaps using some of the parameter values
  - a mechanism to run the code, again perhaps using some of the parameter values
  - by default, the run-time is used as the "figure-of-merit" to be optimised
- at present only brute-force optimisation is supported, but in the future other strategies may be included

Auto-tuning

Example configuration file:

```
#
# parameters and values
#
PARAMS = { flag, {block0, part0}, {block1, part1} }

flag   = {"-Xptxas -dlcm=ca", "-Xptxas -dlcm=cg" }  # compiler flag
block0 = {64, 96, 128}   # thread block size for loop 0
part0  = {128, 192, 256} # partition size for loop 0
block1 = {64, 96, 128}   # thread block size for loop 1
part1  = {128, 192, 256} # partition size for loop 1

#
# compilation and evaluation mechanisms
#
COMPILER = make -B flag=%flag% block0=%block0% part0=%part0%
                               block1=%block1% part1=%part1%
EVALUATION = ./executable
```

Airfoil test code

- 2D Euler equations, cell-centred finite volume method with scalar dissipation (miminal compute per memory reference – should consider switching to more compute-intensive “characteristic” smoothing more representative of real applications)
- roughly 1.5M edges, 0.75M cells
- 5 parallel loops:
  - `save_soln` (direct over cells)
  - `adt_calc` (indirect over cells)
  - `res_calc` (indirect over edges)
  - `bres_calc` (indirect over boundary edges)
  - `update` (direct over cells with RMS reduction)

Airfoil test code

Library is instrumented to give lots of diagnostic info:

```
new execution plan #1 for kernel res_calc
number of blocks       = 11240
number of block colors = 4
maximum block size     = 128
average thread colors  = 4.00
shared memory required = 3.72 KB
average data reuse     = 3.20
data transfer (used)   = 87.13 MB
data transfer (total)  = 143.06 MB
```

- factor 2-4 data reuse in indirect access, but up to 40% of cache lines not used on average

Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using initial parameter values:

- mini-partition size (PS): 256 elements
- blocksize (BS): 256 threads

count	time	GB/s	GB/s	kernel name
1000	0.23	107.8		save_soln
2000	1.26	61.0	63.1	adt_calc
2000	5.10	32.5	53.4	res_calc
2000	0.11	4.8	18.4	bres_calc
2000	1.07	110.6		update
TOTAL	7.78			

Second B/W column includes whole cache line

Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

count	time	GB/s	GB/s	kernel name	PS	BS
1000	0.22	101.8		save_soln		512
2000	1.09	74.1	75.4	adt_calc	256	128
2000	4.95	36.9	60.6	res_calc	128	128
2000	0.10	5.3	20.0	bres_calc	64	128
2000	1.03	94.7		update		64
TOTAL	7.40					

This is a 5 % improvement relative to baseline calculation. Switching from AoS to SoA storage would increase `res_calc` data transfer by approximately 120%.

Airfoil test code

Double precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

count	time	GB/s	GB/s	kernel name	PS	BS
1000	0.44	104.9		save_soln		512
2000	2.62	52.9	53.8	adt_calc	256	128
2000	10.35	30.5	50.8	res_calc	128	128
2000	0.08	11.2	27.9	bres_calc	64	128
2000	1.87	104.5		update		64
TOTAL	15.36					

This is a 7.5 % improvement relative to baseline calculation. Switching from AoS to SoA storage would again increase `res_calc` data transfer by approximately 120%.

Airfoil test code

Single precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 16

count	time	GB/s	GB/s	kernel name	PS
1000	1.68	13.7		save_soln	
2000	11.15	7.3	7.5	adt_calc	128
2000	16.57	10.3	11.2	res_calc	1024
2000	0.16	3.2	11.9	bres_calc	64
2000	4.67	20.9		update	
TOTAL	34.25				

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.

Airfoil test code

Double precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 12

count	time	GB/s	GB/s	kernel name	PS
1000	2.51	18.3		save_soln	
2000	11.68	11.8	11.9	adt_calc	1024
2000	20.99	12.8	13.5	res_calc	1024
2000	0.17	5.0	12.4	bres_calc	512
2000	9.29	21.1		update	
TOTAL	44.64				

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.

Conclusions

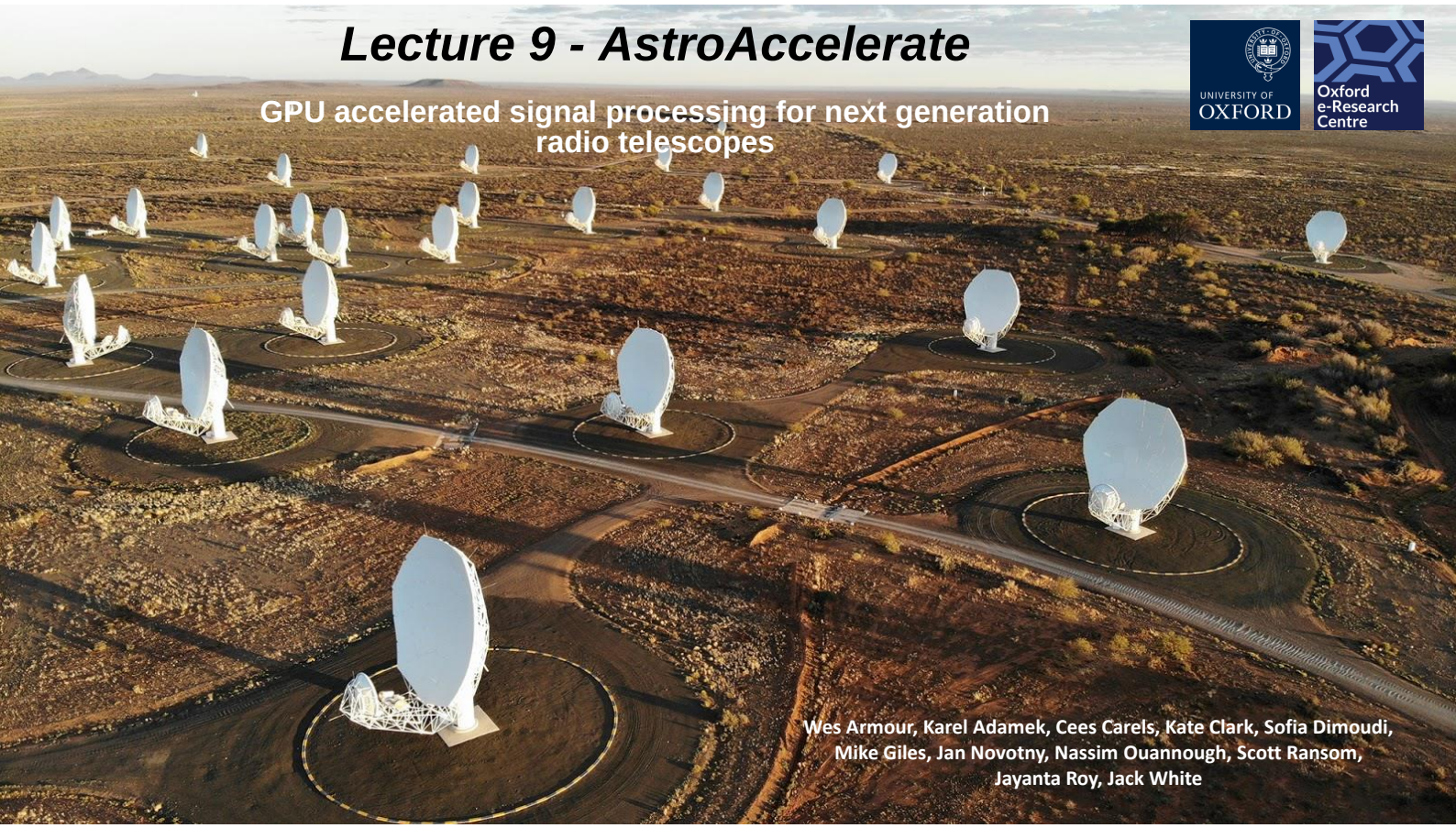
- have created a high-level framework for parallel execution of unstructured grid algorithms on GPUs and other many-core architectures
- looks encouraging for providing ease-of-use, high performance and longevity through new back-ends
- auto-tuning is useful for code optimisation, and a new flexible auto-tuning system has been developed
- C2070 GPU speedup versus two 6-core Westmere CPUs is roughly $5\times$ in single precision, $3\times$ in double precision
- currently working on MPI layer in OP2 for computing on GPU clusters
- key challenge then is to build user community

Acknowledgements

- Carlo Bertolli, David Ham, Paul Kelly, Graham Markall and others (Imperial College)
- Nick Hills (Surrey) and Paul Crumpton (original OPlus development)
- Yoon Ho, Leigh Lapworth, David Radford (Rolls-Royce) Jamil Appa, Pierre Moinier (BAE Systems)
- Tom Bradley, Jon Cohen and others (NVIDIA)
- EPSRC, TSB, NVIDIA and Rolls-Royce for financial support
- Oxford Supercomputing Centre

2023 update

- OP2 continues, funded by Rolls-Royce and EPSRC
- subsequent OPS development supported calculations on unstructured collections of structured blocks
- project is now being led by Gihan Mudalige (Warwick) and Istvan Reguly (PPCU in Budapest)
- primary codes using OP2 are
  - HYDRA – Rolls-Royce's primary CFD code
  - VOLNA – tsunami simulation code
- a huge HYDRA calculation and visualisation won an award at Supercomputing 2022
- main project webpage is `https://op-dsl.github.io/`



Lecture 9 - AstroAccelerate

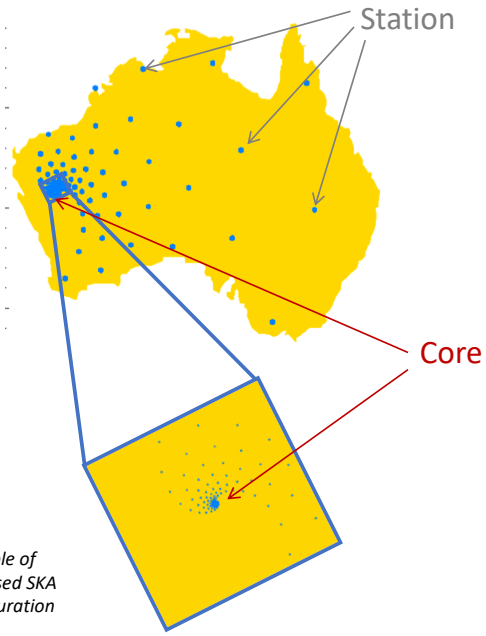
GPU accelerated signal processing for next generation radio telescopes

Wes Armour, Karel Adamek, Cees Carels, Kate Clark, Sofia Dimoudi, Mike Giles, Jan Novotny, Nassim Ouannough, Scott Ransom, Jayanta Roy, Jack White

Part One

A brief introduction to

What is SKA?



Example of proposed SKA configuration

What is SKA?

SKA is a ground based radio telescope that will span continents.

What does SKA stand for?

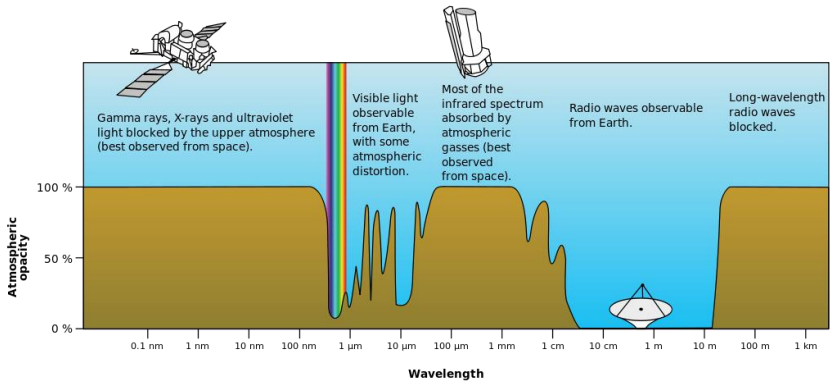
Square Kilometre Array, so called because it will have an effective collecting area of a square kilometre.

Where will SKA be located?

SKA will be built in South Africa and Australia.

Graphic courtesy of Anne Trefethen

What is SKA?



SKA is a ground based telescope. This means that it is most sensitive to the radio range of frequencies. The radio range of frequencies that can be observed from here on Earth is very wide, specifically SKA will be sensitive to frequencies in the range of 50MHz to 20GHz (wavelengths 15 mm to 6 m). This makes SKA ideal for studying lots of different science cases.

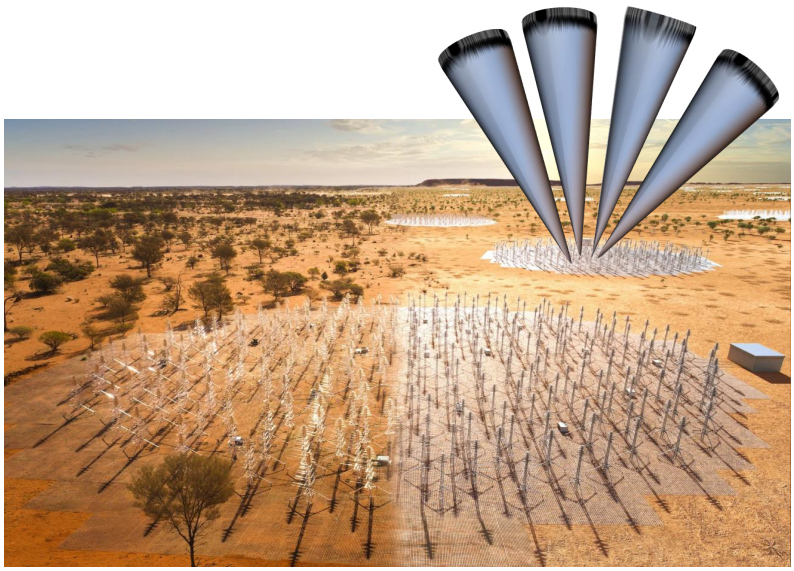
Image source Wikipedia. Authors: NASA (original); SVG by Mysid

What is SKA?

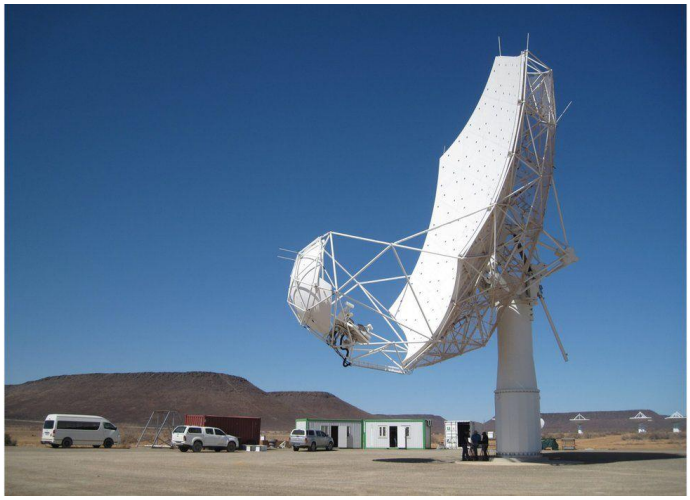
SKA will have the ability to use all of its antennas to produce images of the radio sky in unprecedented accuracy and detail.

It will also be able to use combinations of antennas to perform multiple observations of different regions of the sky at the same time.

In this scenario data from each beam can be computed in parallel.



SKA science



SKA will study a wide range of science cases and aims to answer some of the fundamental questions mankind has about the universe we live in.

- How do galaxies evolve
  – What is dark energy?
- Tests of General Relativity
  – Was Einstein correct?
- Probing the cosmic dawn
  – How did stars form?
- The cradle of life
  – Are we alone in the Universe?

SKA time domain - signal processing

Time Domain Team

The time domain team is an international team led by Oxford and Manchester.

It aims to deliver an end-to-end signal processing pipeline for time domain science performed by SKA (see right).

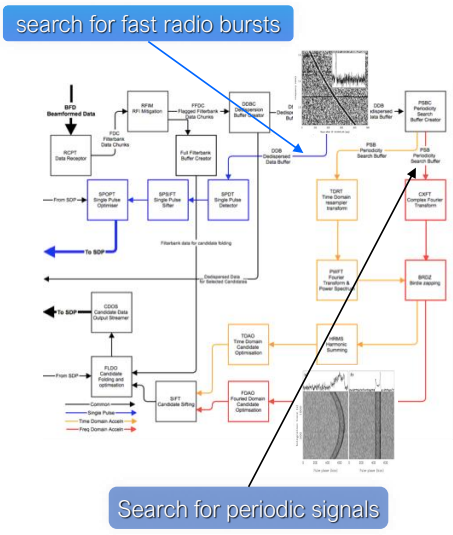


Image courtesy of Aris Karastergiou

SKA time domain - signal processing

Our work focussed on vertical prototyping activities.

We delivered accelerated algorithms for many-core technologies, such as GPUs to perform the processing steps within the signal processing pipeline with the aim of achieving real-time processing for the SKA.

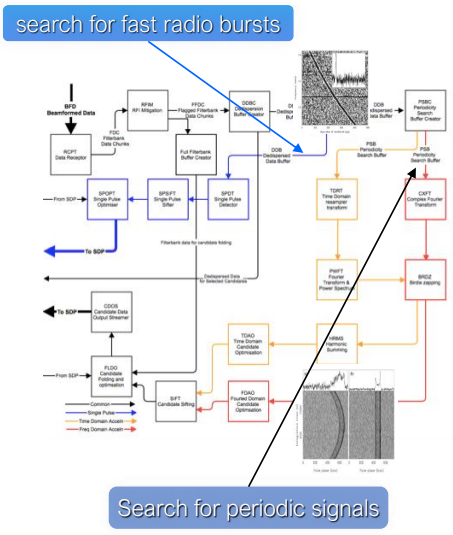


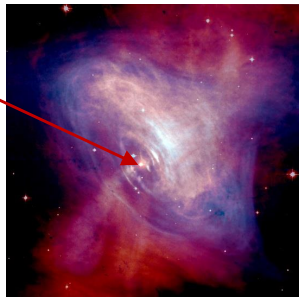
Image courtesy of Aris Karastergiou

SKA time domain science - Pulsars

Pulsars are magnetized, rotating neutron stars. They emit synchrotron radiation from the poles, e.g. Crab Nebula.

Their magnetic field is offset from the axis of rotation as such (as observed from here on Earth, they act as cosmic lighthouses.

They are extremely periodic and so make excellent clocks!



Hester et al.

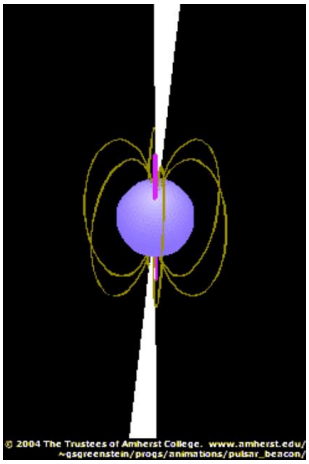
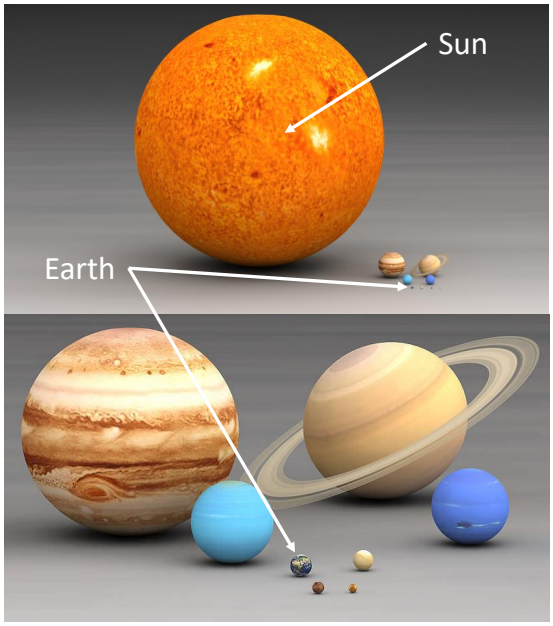


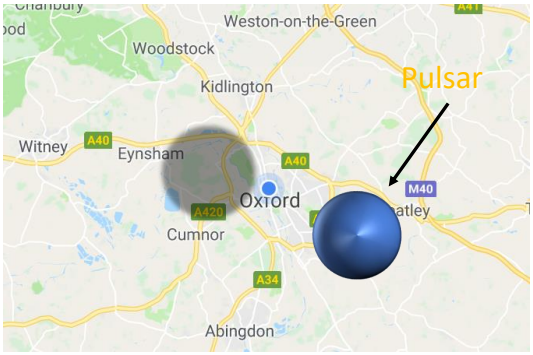
Image: Amherst College

Pulsars – size and scale



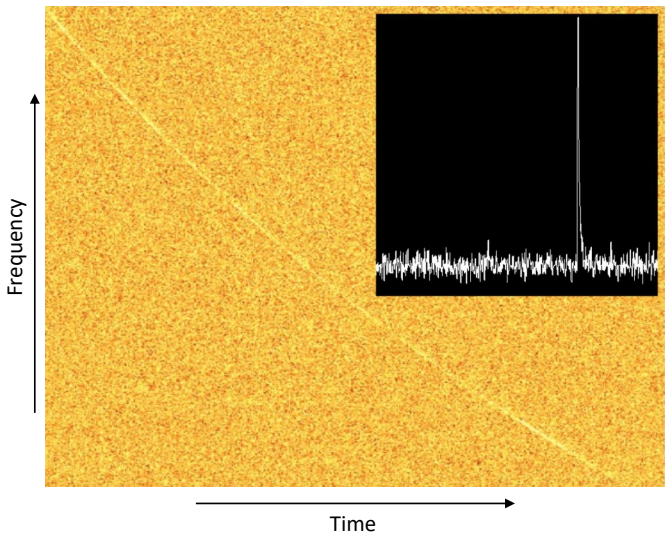
Pulsars are typically 1-3 Solar masses in size, they have a diameter of 10-20 Kilometres and a pulse period ranging from milliseconds to seconds.

Meaning that they are very small, very dense and rotate extremely quickly.



https://commons.wikimedia.org/wiki/File:Planets_and_sun_size_comparison.jpg (Author: Lsmpascal)

SKA time domain science - FRBs



Fast Radio Bursts (FRBs), were first discovered in 2005 by Lorimer et al.

They are observed as extremely bright single pulses that are extremely dispersed (meaning that they are likely to be far away, maybe extra galactic).

Now hundreds of FRBs have been observed or found in survey data. They are of unknown origin, but likely to represent some of the most extreme physics in our Universe.

Hence they are extremely interesting objects to study.

Credit: FRB110220 Dan Thornton (Manchester)

SKA time domain - data rates

The SKA will produce vast amounts of data. In the case of time-domain science we expect the telescope to be able to place ~2000 observing beams on the sky at any one time (there are trivially parallel to compute).

The telescope will take 20,000 samples per second for each of those beams and then it will measure power in 4096 frequency channels for each time sample. Each of those individual samples will comprise of 4x8 bits, although we are only really interested in one of the 8 bits of information.

Doing the math tells us that we will need to process 160GB/s of relevant data. This is approximately equal to analysing 50 hours of HD television data per second.

The most costly computational operations in data processing pipeline are

DDTR ~ $O(n_{dms} * n_{beams} * n_{samps} * n_{chans})$

FDAS ~ $O(n_{dms} * n_{beams} * n_{samps} * n_{acc} * \log(n_{samps}) * 1/t_{obs})$

Requiring ~2 PetaFLOP of Compute!

SKA time domain – data challenges

Because we would like to monitor interesting and exotic events as they occur we need to process data in real-time (or as near to as possible).

So storing the data and processing later isn't feasible. The data rates mean transporting data offsite would be challenging and costly.

So processing must happen close to the telescope. But how do we put a computer capable of processing big-data streams in real-time in the middle of a desert?

Connectivity, power, operation all pose significant problems.

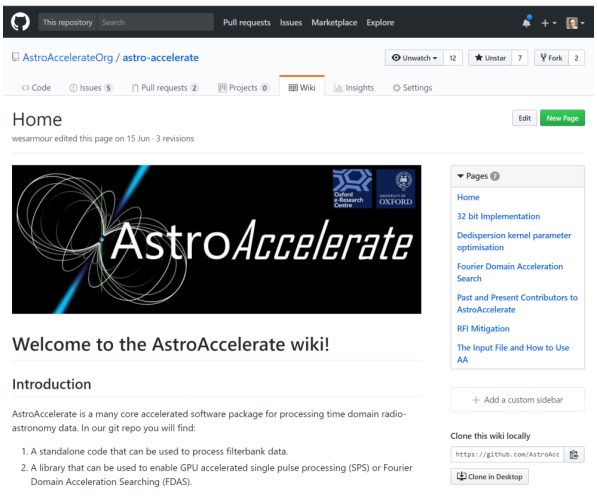
Part Two

Astro*Accelerate* – A case study

AstroAccelerate

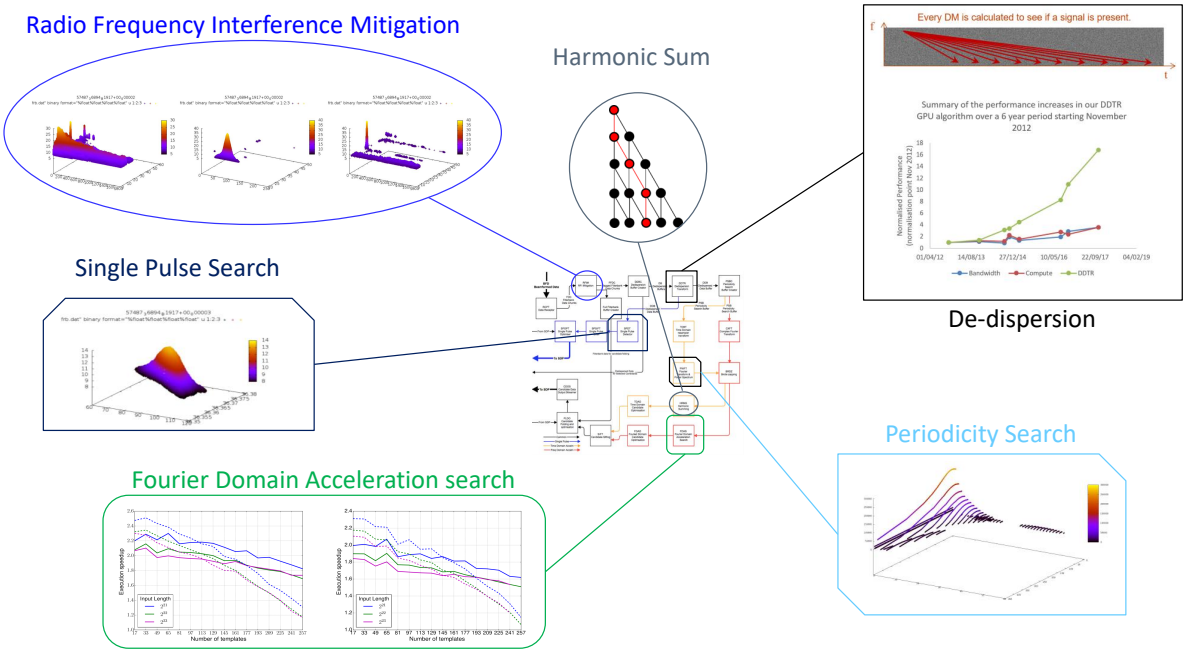
AstroAccelerate is a GPU enabled software package that focuses on achieving real-time processing of time-domain radio-astronomy data. It uses the CUDA programming language for NVIDIA GPUs.

The massive computational power of modern day GPUs allows the code to perform algorithms such as de-dispersion, single pulse searching and Fourier Domain Acceleration Searching in real-time on very large data-sets which are comparable to those which will be produced by next generation radio-telescopes such as the SKA.

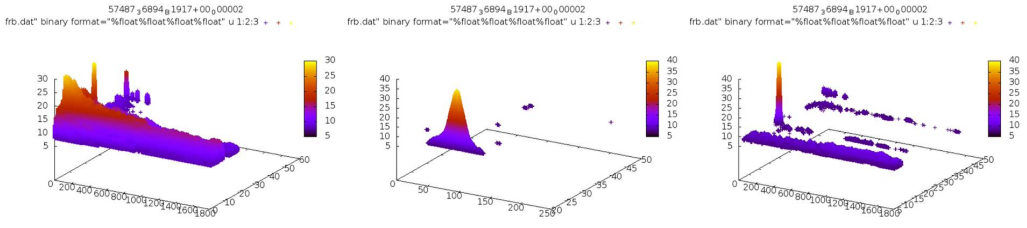


https://github.com/AstroAccelerateOrg/astro-accelerate

AstroAccelerate - Signal Processing

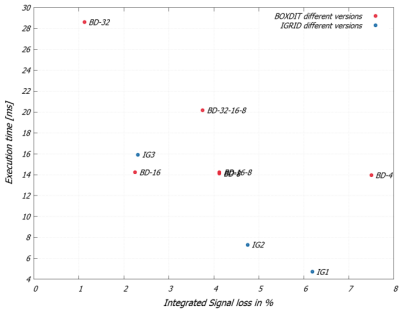


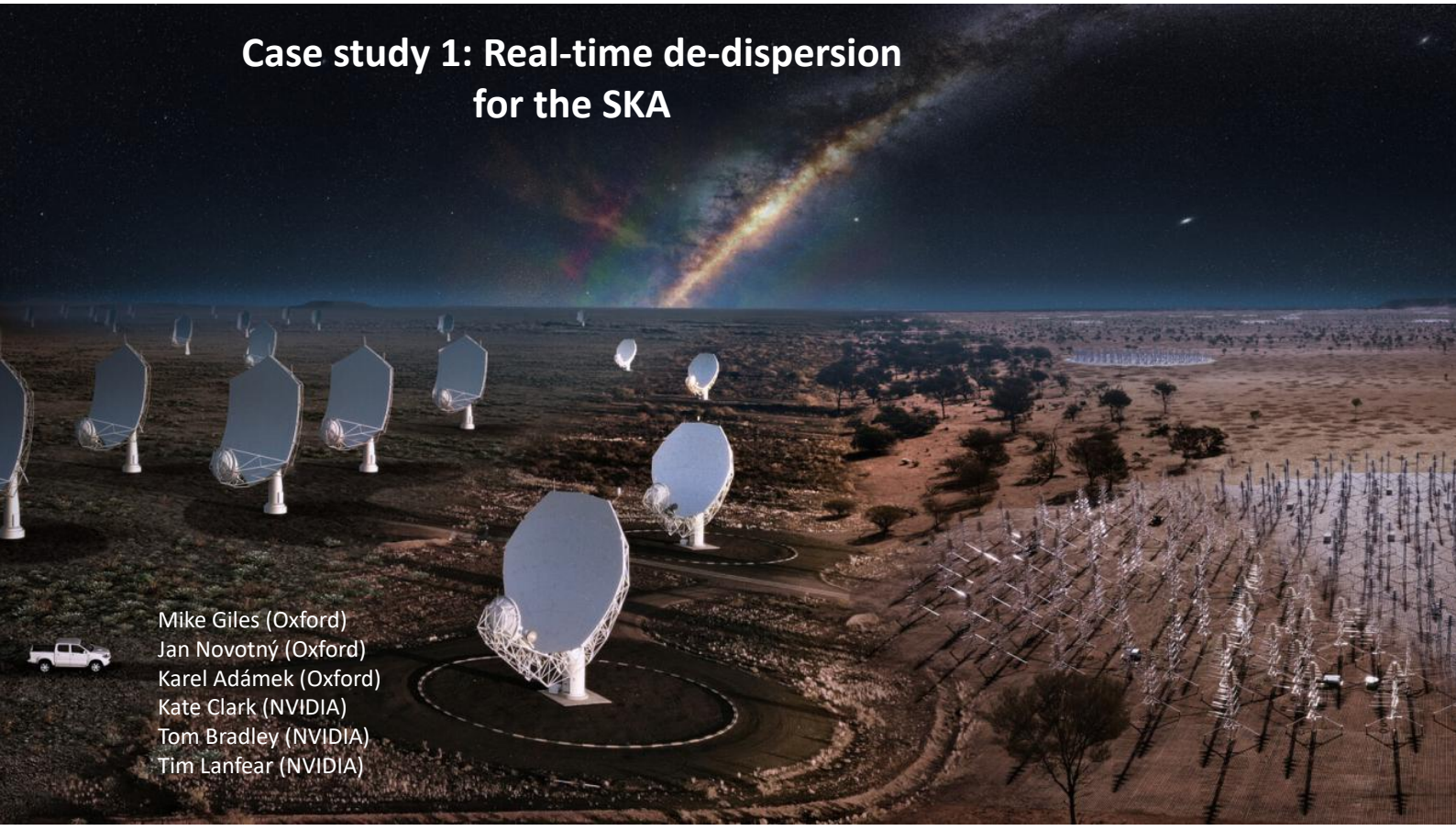
AstroAccelerate - Features



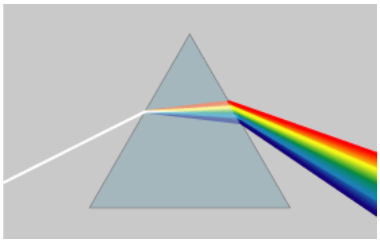
AstroAccelerate has the following features…

- Zero DM and basic RFI Mitigation
- DDTR
- Single Pulse Search
- Fourier Domain Acceleration Search
- Periodicity search with harmonic sum



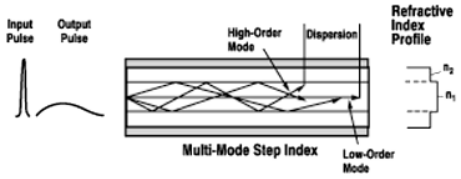


What is dispersion?



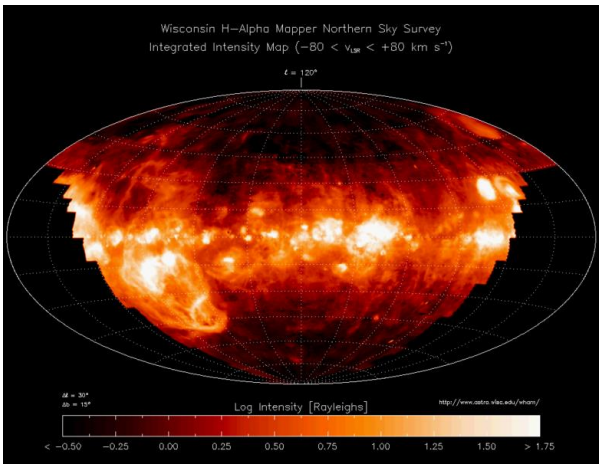
Chromatic dispersion is something we are all familiar with. A good example of this is when white light passes through a prism.

Group velocity dispersion occurs when pulse of light is spread in time due to its different frequency components travelling at different velocities. An example of this is when a pulse of light travels along an optical fibre.



Dispersion by the ISM

The interstellar medium (ISM) is the matter that exists between stars in a galaxy.



Haffner et al. 2003

In warm regions of the ISM (~8000K) electrons are free and so can interact with and affect radio waves that pass through it.

The dispersion measure - DM

The time delay, $\Delta\tau$, between the detection of frequency f_{high} and f_{low} is given by:

$$\Delta\tau = C_{DM} \times \mathrm{DM} \times \left(\frac{1}{f_{low}^2} - \frac{1}{f_{high}^2}\right)$$

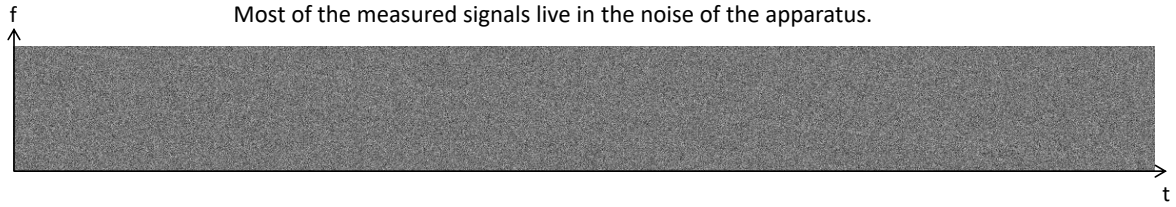
Where C_{DM} is the dispersion constant. DM is the dispersion measure:

$$DM = \int_0^d n_e dl$$

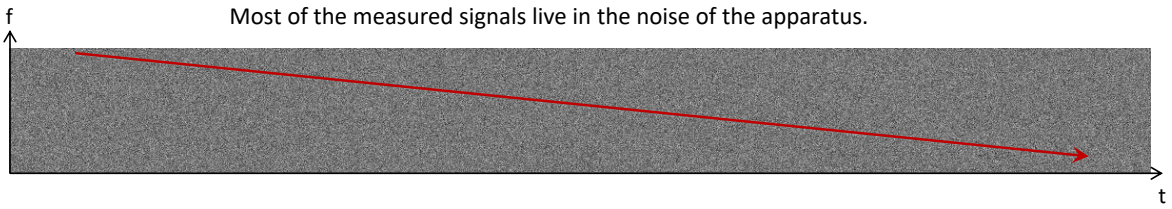
This is the free electron column density between the radio source and observer.

We can measure $\Delta\tau$ and f and so can study DM

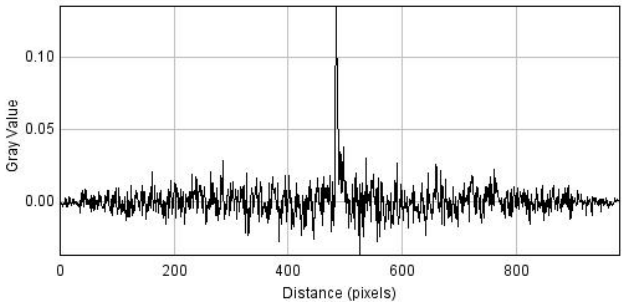
Experimental Data



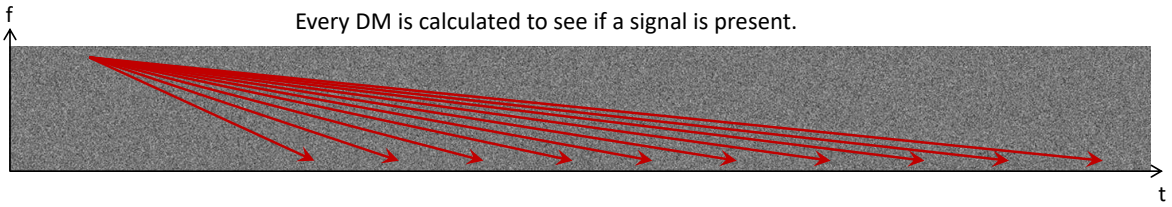
Experimental Data



Hence frequency channels have to be “folded”



De-dispersion



In a blind search for a signal many different dispersion measures are calculated.

This results in many data points in the (f,t) domain being used multiple times for different dispersion searches.

This allows for data reuse in a GPU algorithm.

All of this must happen in real-time i.e. the time taken to process all of our data must not exceed the time taken to collect it

De-dispersion Transform

Our DDTR is an implementation of incoherent brute force de-dispersion.

1. We brute force optimise the tuneable parameters of the code, such as the thread block size and number of registers used.

2. It utilises GPU shared memory and typically achieves 60-80% of peak throughput.

3. It uses SIMD in work to process multiple time samples per machine word for data less than or equal to 16 bits.

```
for (i = 0; i < SNUMREG; i++)
{
        local = 0;
        unroll = ( i * 2 * SDIVINT );
        for (j = 0; j < UNROLLS; j++)
        {
                stage = *(int*) &f_line[j][( shift[j] + unroll )];
                local += stage;
        }
        local_kernel_one[i] += (local & 0x0000FFFF);
        local_kernel_two[i] += (local & 0xFFFF0000) >> 16;
}
```

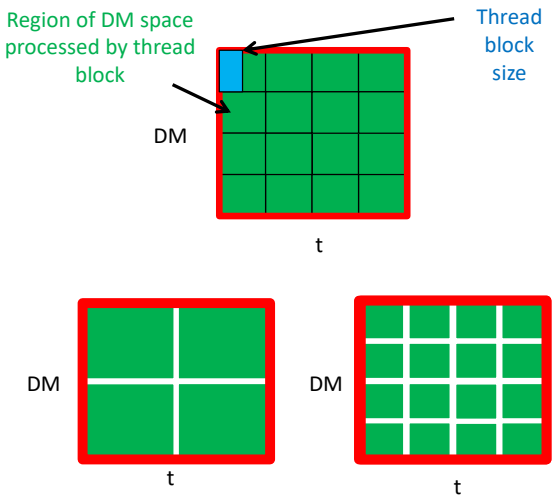
https://github.com/AstroAccelerateOrg/astro-accelerate/blob/master/lib/device_dedispersion_kernel.cu

De-dispersion Transform – 1. tuning

Each thread processes a tunable number of time samples, each de-dispersion trial associated with one time sample is stored in a GPU register.

Along with this the number of time samples per thread block and the number of de-dispersion trials (which is where data reuse comes from) are tuned.

Finally the code performs a tunable number of SIMD in word operations which are periodically unloaded to a floating point accumulator.



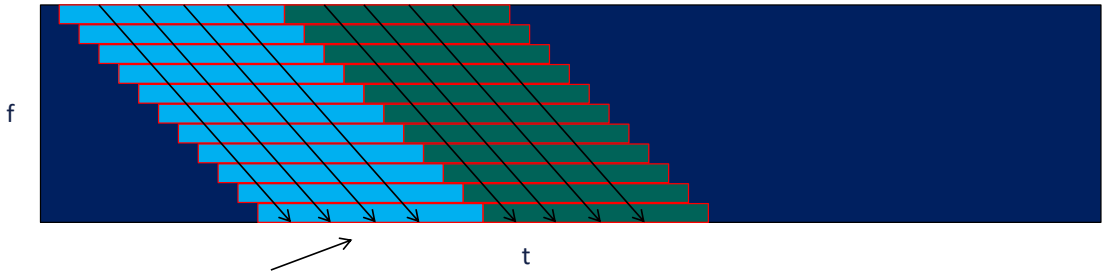
Optimising the parameterisation

https://github.com/AstroAccelerateOrg/astro-accelerate/blob/master/lib/device_dedispersion_kernel.cu

De-dispersion Transform – 2. shared memory

Exploiting registers and fast shared memory…

Each dispersion measure for a given frequency channel needs a shifted time value.



Constant DM's with varying time.

In practice a thread will process multiple time samples and a thread block will also process neighboring DM trials to increase data reuse.

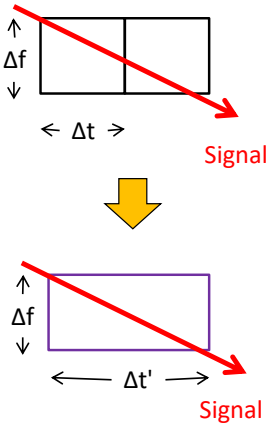
Incrementing all of the registers at every frequency step ensures a high data reuse of the stored frequency time data in the L1 cache or shared memory.

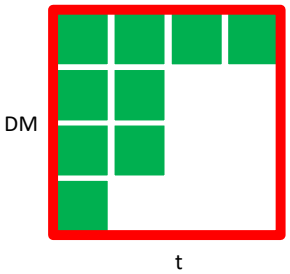
De-dispersion Transform – 2. time binning

One issue with using a shared memory based algorithm is that for high DM trials (those that represent distant objects, forming long broad curves in our input frequency-time data) we need to store increasing lengths of constant frequency varying time data in shared memory.

This ultimately limits the highest DM trial that can be searched at full time resolution.

To overcome this we've added a time binning (scrunching) kernel that decimates data in time. This has the effect of decreasing time resolution and allows us to search to arbitrary high DM trials.





Has the added advantage of reducing the amount threads that are needed to process a region of (DM,t) space, speeding up the code.

De-dispersion Transform – 3. SIMD in word

We exploit the fact that one frequency-time sample of SKA data will be 8 bits.

We pack the data in such a way so that we can perform two de-dispersion trials per integer operation.

We convert the unsigned char to an unsigned short and pack as ushort2, we mask this as an int and add ints.

Once a single trial nears the maximum allowable value for a ushort we store the value in a floating point accumulator. This has the effect of increasing the speed of the code and also it's precision.

Recorded telescope data (t_n = 8 bits) is stored in global as a uchar array

char[] = [t_0,t_1,t_2,t_3,t_4,t_5,t_6 …]

This is converted to ushort when loaded though the texture pipe (doubling the size of the array stored because it is now interleaved with 8 bits of zeros

ushort[] = [0 t_0, 0 t_1, 0 t_2, 0 t_3, 0 t_4, 0 t_5, 0 t_6, …]

Masking this with an int allows us to add two samples per one instruction issued.

De-dispersion Transform – 3. SIMD in word

In reality we have to odd/even interleave the data to ensure correct byte alignment within shared memory banks (4 bytes wide).

ushort2[] = [0 t_0,0 t_1][0 t_1,0 t_2][0 t_2,0 t_3][0 t_3,0 t_4]…

For thread with an even shift (lets say 2)…

(t_0,t_1) (t_1,t_2) (t_2,t_3) (t_3,t_4) (t_4,t_5) (t_5,t_6)

$t_i = t_2$

$t_{i+1} = t_3$

For thread with an odd shift (lets say 3)…

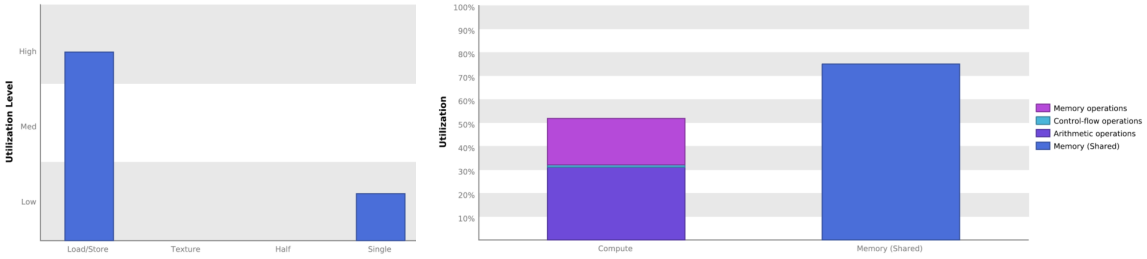
(t_0,t_1) (t_1,t_2) (t_2,t_3) (t_3,t_4) (t_4,t_5) (t_5,t_6)

$t_i = t_3$

$t_{i+1} = t_4$

Now each thread computes the correct two time values and at double data rate

De-dispersion Transform - results



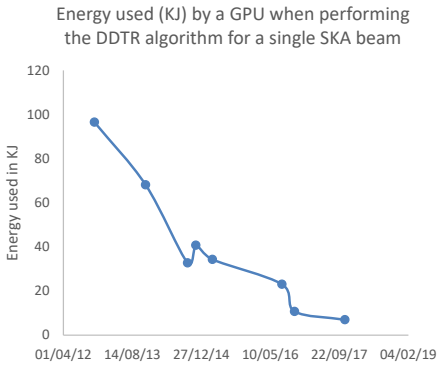
Shared Memory

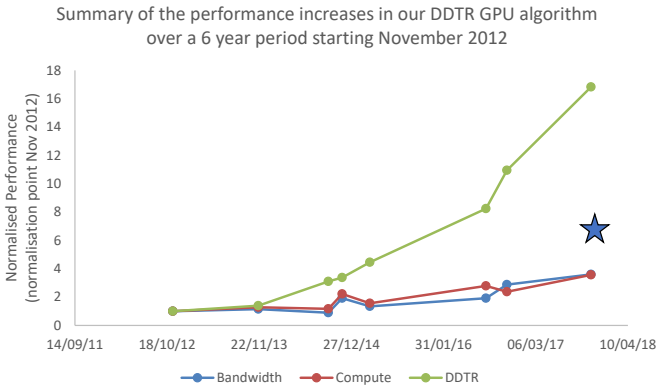
Shared Loads	43769001212	7,976.627 GB/s	
Shared Stores	5655113738	1,030.609 GB/s	
Shared Total	49424114950	9,007.236 GB/s	Idle Low Medium High Max

Results showing the shared memory utilisation, which is this codes limiting factor. We achieve 75% of peak throughput, limited by load/store.

The total shared memory bandwidth throughput achieved on a TITAN V is 9 TB/s.

De-dispersion Transform - results





These two plots demonstrate how we have reduced power consumption and increased performance for the DDTR algorithm over a six year period.

The bule star indicates the performance of our initial (optimised) code running on current hardware. Demonstrating how invested effort algorithm optimisation over a long period can deliver significant gains.

De-dispersion Transform – cost / benefit analysis

But is it worth the effort?

Estimated runtime for DDTR in the PSS pipeline (conservative 25%)
Estimate of speed increase compared to initial code ~17x

➡ Total PSS pipeline acceleration ~ 4x

So to deliver the science in the same wall clock time you'd need 4x the GPU capacity.

Even if you're prepared to wait 4x longer… Energy efficiency has increased by 14x
Very rough estimate of PSS OpEx saving ~ £1M
Estimate of total effort ~ 1.0FTE for four years ~ £250K (FEC)

Hence a £750K saving in OpEx costs alone (this is a conservative estimate).

(You can't just go out and buy this at a later date. Domain expertise in both radio astronomy data processing and many-core acceleration are needed)

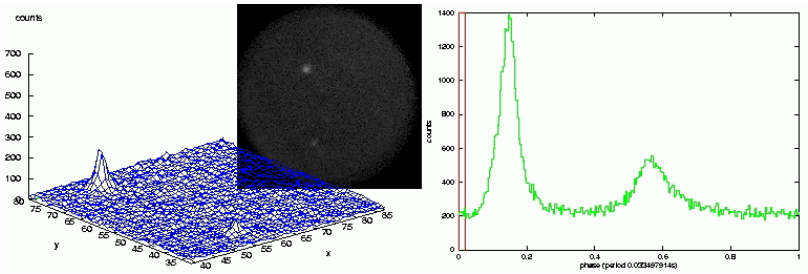
Conclusions – Comparisons of GPUs

Technology	Kepler (K40)	Kepler (K80)	Kepler (780Ti)	Maxwell (980)	Maxwell (Titan X)	Pascal (Titan XP)	Pascal P100	Volta V100	Volta Titan V
Fraction of real-time	1.035	2.5	2.88	2.3	3.3	6.1	8.1	12.5	10.9
Watts per beam (Average)	127W	76 W	~70W	~61W	~64W	~43W	~24W	13W	10W
Cost per beam (capital, accelerator only)	£3K?	£4K?	£250	£200	£240	~£200	~£420	~£530	~£270
Cost per beam (2 year survey, GPU only, based on 1KWh costing £0.2)	~£430	~£265	~£245	~£213	~£224	~£151	~£84	~£45	~£35

Improvement between generations comes from a combination of advances in both the hardware and algorithm

Case study 2: Fourier Domain Acceleration Searching for the SKA

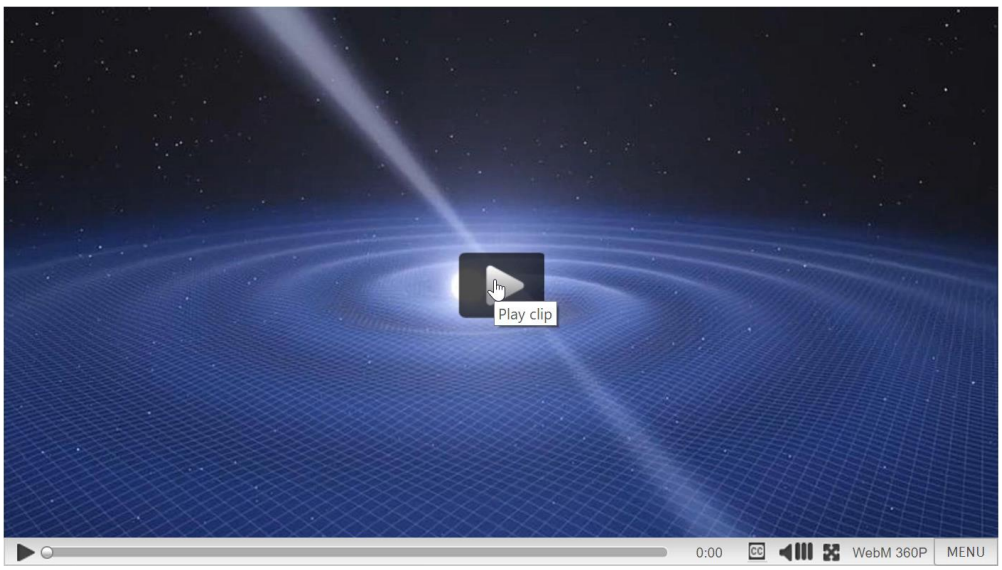
Sofia Dimoudi, Karel Adámek, Jack White, Mike Giles
(Oxford)



Light curve and slow motion picture of the **solitary** pulsar located in the centre of the Crab Nebula.

Image taken with a photon counting camera on the 80cm telescope of the Wendelstein Observatory, Dr. F. Fleischmann, 1998

Binary pulsars and gravitational waves



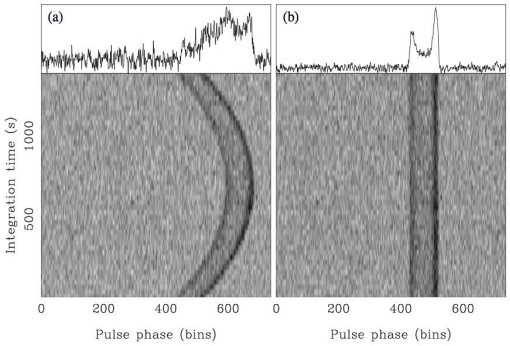
http://www.eso.org/public/videos/eso1319a/ Author: ESO/L. Calçada

Fourier Domain Acceleration Search - FDAS

Signals from binary systems can undergo a Doppler shift due to accelerated motion experienced over the orbital period.

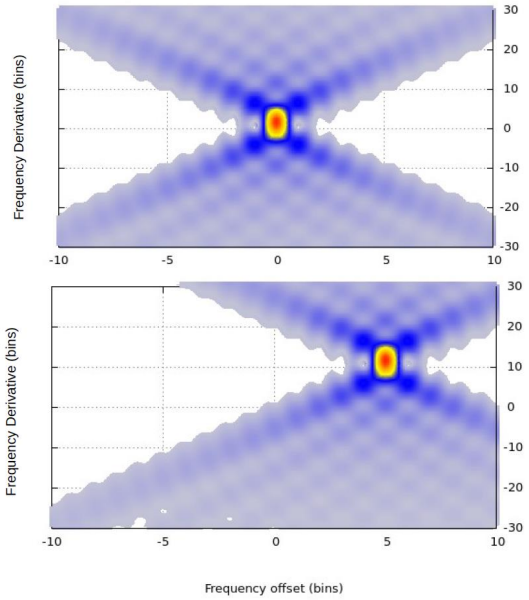
Much like the sound of a siren approaching you and then speeding away.

This can be corrected by using a matched filter approach.



Ransom, Eikenberry, Middleditch: AJ, Vol 24, Issue 3, pp. 1788-1809

FDAS Example



The two plots illustrate the effect of orbital acceleration.

The first plot shows a signal without acceleration, the signal is centred on its frequency and lies on the f-dot template corresponding to zero acceleration.

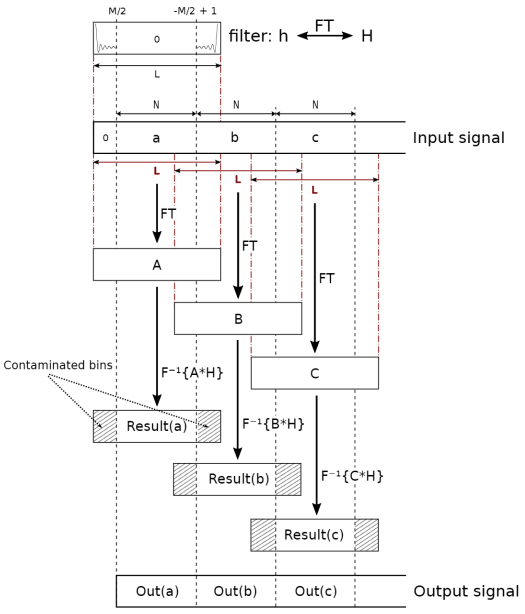
The second plot shows a signal with a frequency derivative, and has drifted from the original frequency by a number of bins.

Fourier Domain Acceleration Search

Use overlap-save algorithm to compute cyclic N-point convolution of template with signal segment.

Avoids the need for synchronisation because contaminated ends of convolved data are discarded (as opposed to overlap-add).

Code calculates the convolution, powers and extracts peaks.

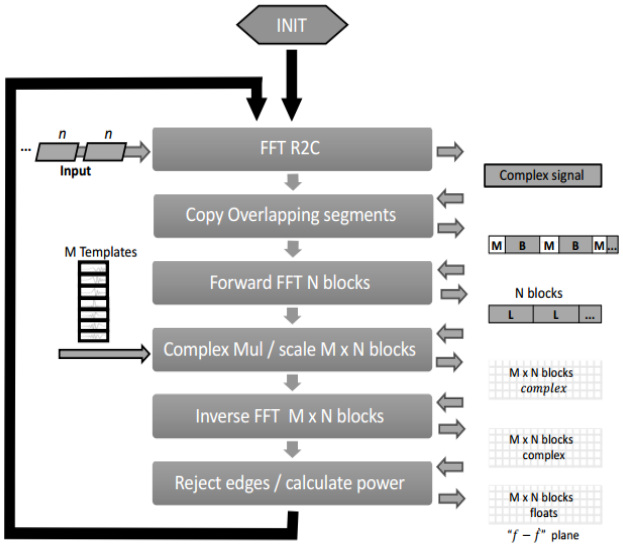


Fourier Domain Acceleration Search

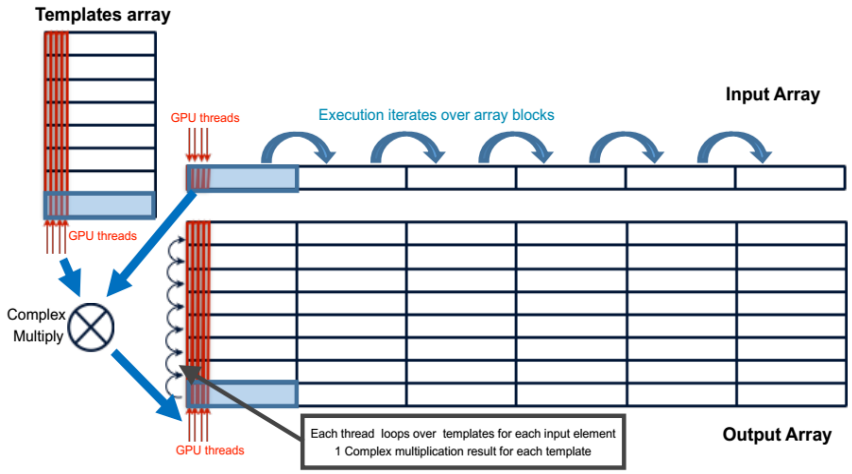
Using cuFFT means many transactions to device memory on the GPU (represented by grey arrows on the right of the diagram).

This causes the computation to be limited by global memory bandwidth (the lowest common denominator on a GPU).

This means that a cuFFT based implementation is very slow.



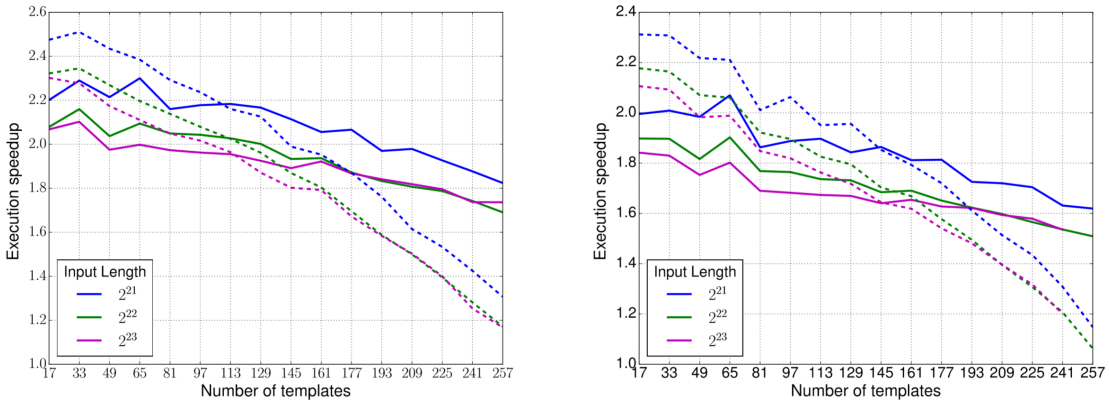
Fourier Domain Acceleration Search



By writing our own custom I/FFT codes to work on shared memory we can perform the FFT, pointwise multiply and scale, IFFT and edge rejection all in one kernel.

Karel Adámek, Sofia Dimoudi, Mike Giles, and Wesley Armour. 2020. GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory. *ACM Trans. Archit. Code Optim.* 17, 3, Article 18 (August 2020). **https://doi.org/10.1145/3394116**

Fourier Domain Acceleration Search



Initial results from our tests on a Tesla P100. In the SKA region of interest – signal length 2^{23}, template size of 512 (solid line) and no interbinning (left graph)

Futher optimisation achieved approximately a 3.5x speed increase on a V100

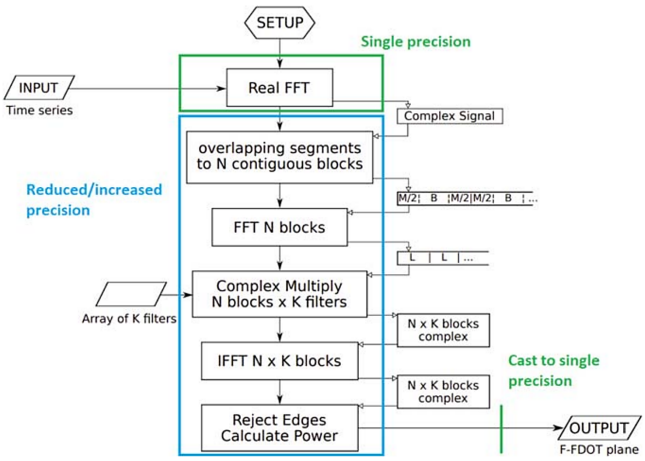
Sofia Dimoudi *et al* 2018 *ApJS* **239** 28 https://iopscience.iop.org/article/10.3847/1538-4365/aabe88

Karel Adámek, Sofia Dimoudi, Mike Giles, and Wesley Armour. 2020. GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory. *ACM Trans. Archit. Code Optim.* 17, 3, Article 18 (August 2020). **https://doi.org/10.1145/3394116**

FDAS further optimisation

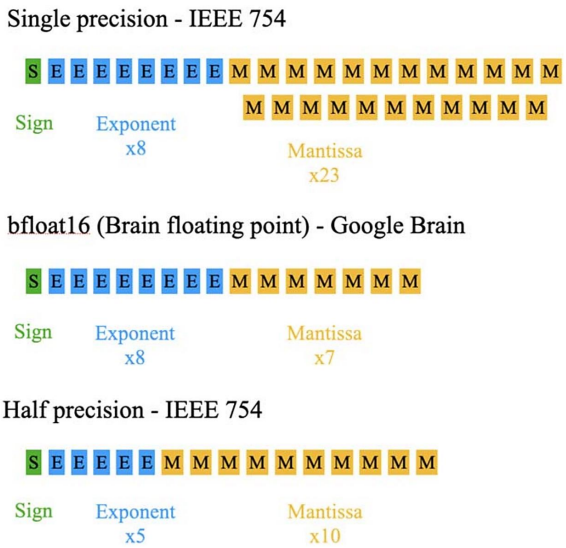
Jack's work has focused on implementing mixed precision for FDAS.

By reducing the precision in the convolution par tof the algorithm (where FFTs can be shorter, so we have far smaller accumulated errors we are able to double the bandwidth throughput for the convolution parts of the code.



FDAS further optimisation

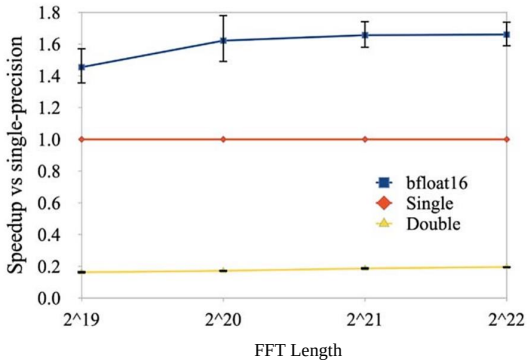
Jack used bfloat16 (16 bits) for the convolution part of FDAS. This is because it has the same range as float (fp32) so there is no need to scale numbers, and due to the fact that the exponent is the same as fp32 we are easily able to type convert between the two (on GPUs the SFU performs type conversion and it is a limited resource).



FDAS further optimisation

The work provides roughly a 1.7x speed increase compared to a single precision implementation.

So do consider whether you really need to compute in double or float or…..



Jack White *et al* 2023 *ApJS* **265** 13 https://iopscience.iop.org/article/10.3847/1538-4365/acb351/meta

FDAS Energy optimisation

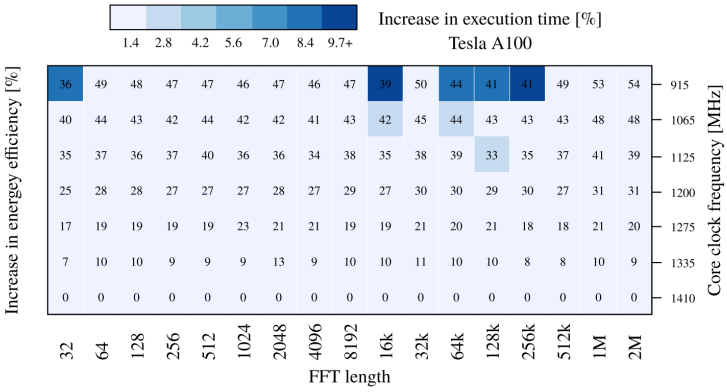
For a bandwidth bound algorithm it makes no sense to have the GPU cores running as quickly as they can.

The reason is that your memory cannot deliver data to them quickly enough to utilise all of the computation that they can perform.

FDAS Energy optimisation

Because the FFT is bandwidth bound we are able to reduce the GPU core clock frequency without having significant impact on execution time.

The figure (right) shows the energy saving for a given core clock frequency. The shading indicates the increase in execution time.



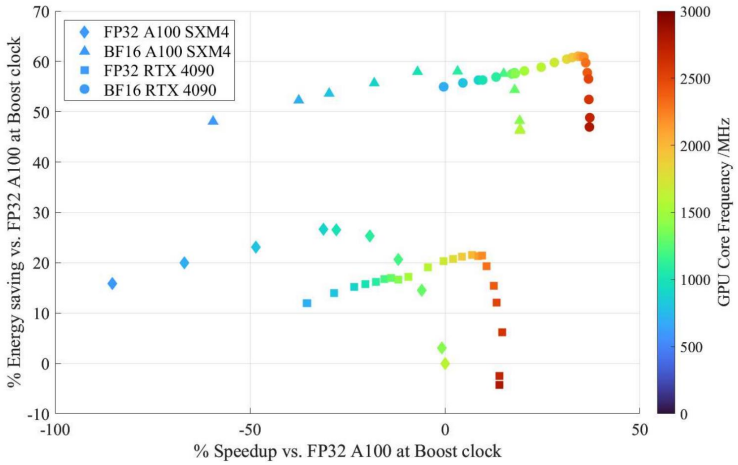
K. Adámek, et al in *IEEE Access* https://ieeexplore.ieee.org/abstract/document/9330509

FDAS Energy optimisation

We can use NVML to change the clock speed of the processing cores to slow them down.

This has the advantage of reducing the energy that the GPU uses.

By combining mixed precision with reduction in core clock frequency we are able to achieve speed increase and energy savings.



Jack White *et al* Astronomical Data Analysis Software and Systems, ADASS XXXII

Acknowledgments and Collaborators

University of Oxford

Mike Giles (Maths)

Aris Karastergiou (Physics)

Chris Williams (Physics)

Steve Roberts (Engineering)

Sofia Dimoudi (OeRC)

Nassim Ouannoughi (OeRC)

Karel Adamek (OeRC)

Jayanth Chennamangalam (Physics)

Griffin Foster (Physics)

University of Manchester

Ben Stappers

Mike Keith

Prabu Thiagaraj

Jayanta Roy

Mitch Mickaliger

NRAO/UVA

Scott Ransom

University of Bristol

Dan Curran (Electrical Engineering)

Simon McIntosh Smith (Electrical Engineering)

ASTRON

Cees Bassa

Jason Hessels

University of Opava

Jan Novotny

NVIDIA

Kate Clark

Tim Lanfear

Tom Bradley

Max Plank

Ewan Barr

AstroAccelerate

http://www.oerc.ox.ac.uk/projects/astroaccelerate

Lecture 10
Future Directions

Prof Wes Armour
wes.armour@eng.ox.ac.uk

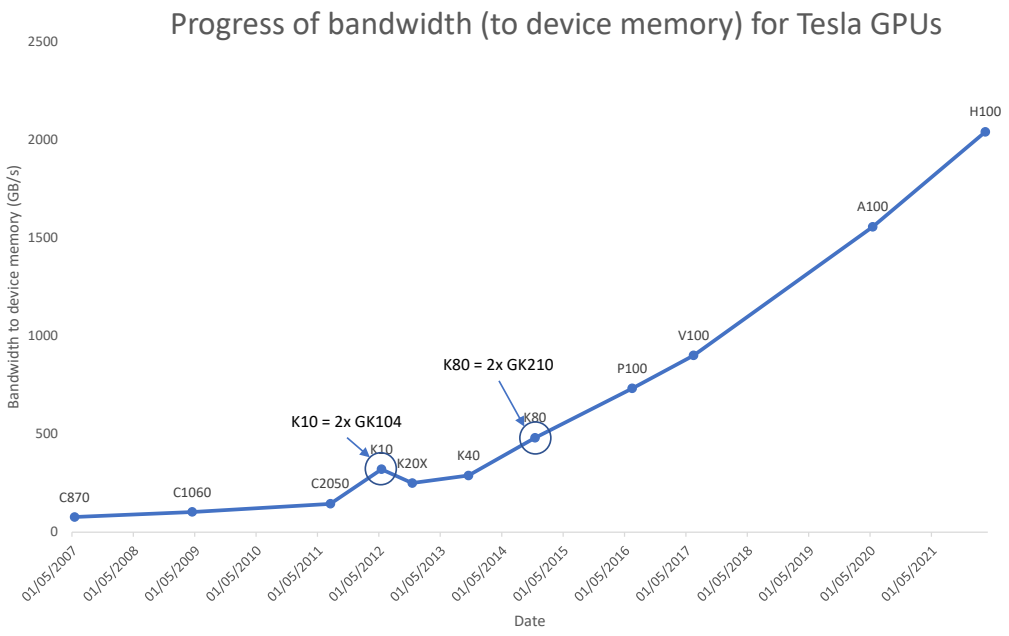
Oxford e-Research Centre
Department of Engineering Science

Learning outcomes

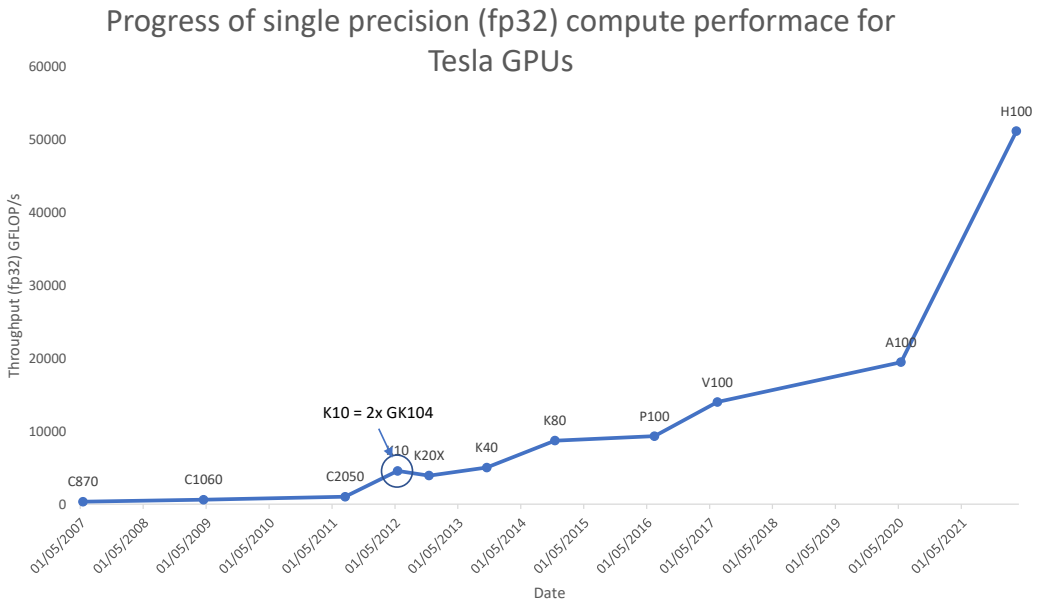
In this final lecture we will look at the current landscape of accelerated computing.

We will look at hardware and software trends and potential future directions for accelerated computing.

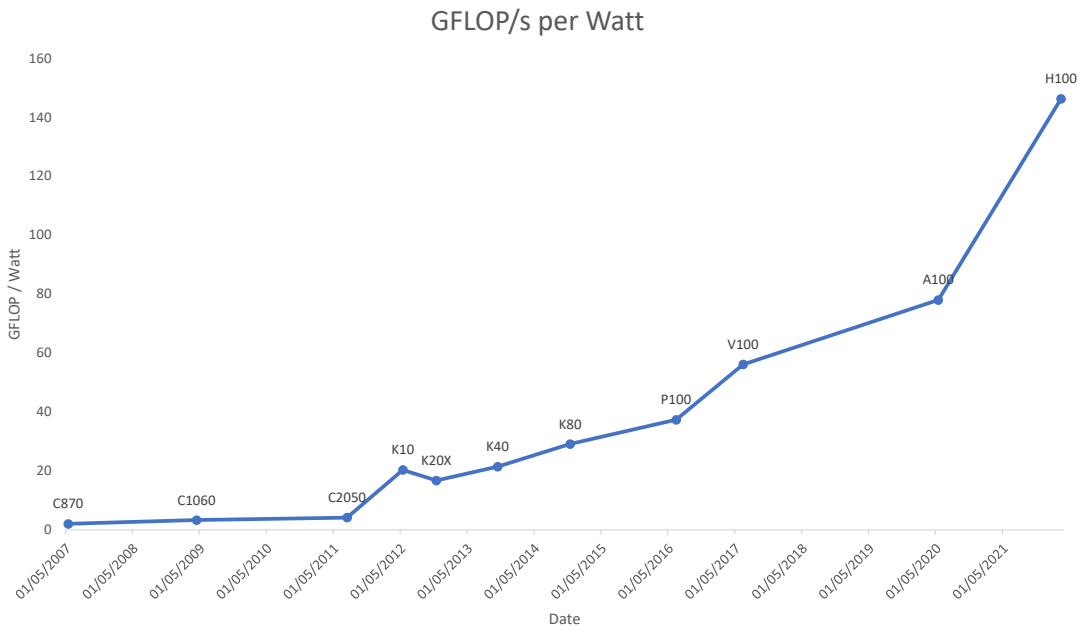
Bandwidth



Compute



Efficiency



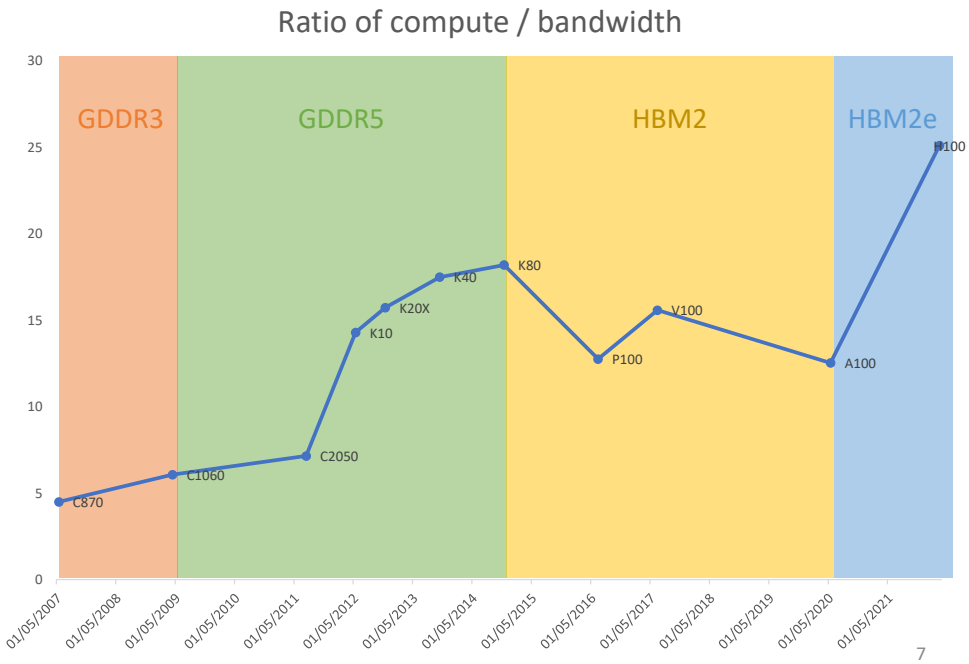
Ratio of compute / bandwidth

The ratio of compute / bandwidth often called **arithmetic intensity** or **operational intensity** (I) tells us how many floating point operations we can perform in the time it takes to move each byte of data from device memory.

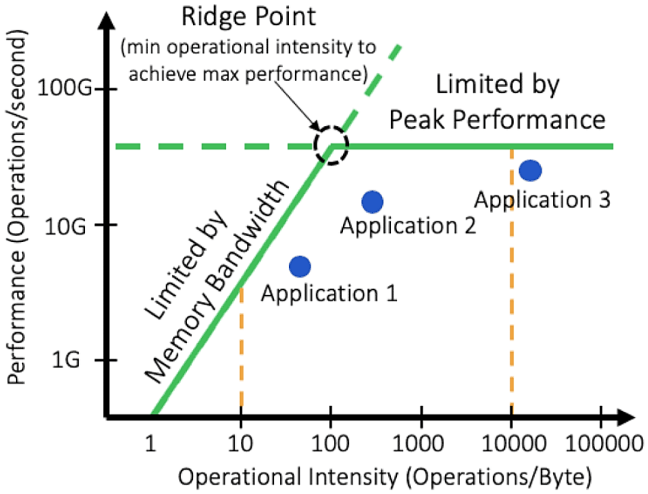
$$I = \frac{W}{Q} = \frac{Work}{Memory\ traffic} = \frac{FLOPs}{Byte}$$

Ratio of (peak) compute / bandwidth

We can see from the plot on the right that, although the introduction of new memory technologies reduces operational intensity for short periods, **the overall trend is for it to increase**.



Roofline model



The roofline model tells us, for given hardware, whether our application will be bandwidth bound or compute bound.

Reminder - recompute not transfer

Given the fact that we can now perform so many FLOPs per byte that we move from device memory (e.g. ~100 for a single float on H100), it is worth considering whether it is more efficient to recompute values rather than transferring them.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The growing cost of owning NVIDIA

Due to market dominance, commercial interest and the boom in AI, the cost of NVIDIA GPUs has increased significantly.

The plot on the right comes from nextplatform and shows the successive increase in launch price for different generations of GPUs.

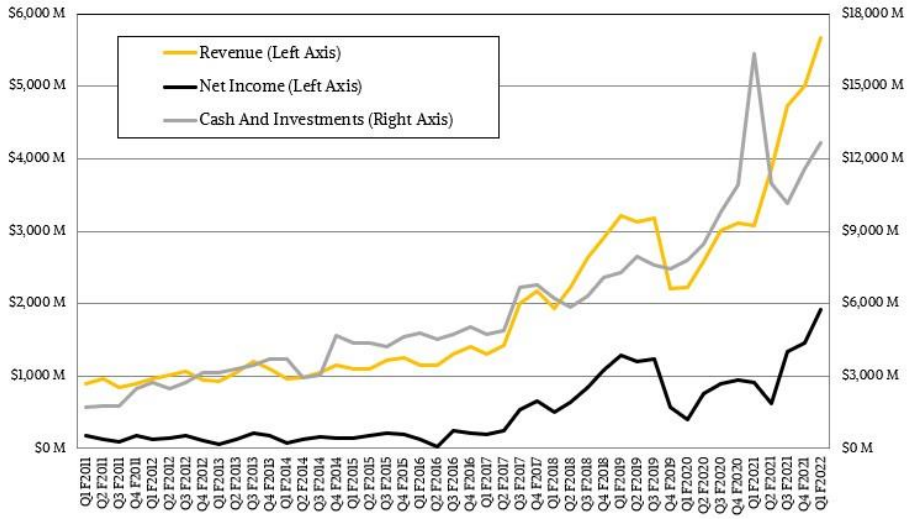
https://www.nextplatform.com/2022/05/09/how-much-of-a-premium-will-nvidia-charge-for-hopper-gpus/

Timothy Prickett Morgan

The growing cost of owning NVIDIA

Here we see the growth in NVIDIA's revenue over the last decade, again we see in the last few years near exponential growth.

So what does this mean in terms of GPU availability and total cost of ownership.



https://www.nextplatform.com/2021/05/26/nvidias-next-major-wave-of-ai-revenues/

Multi-GPU computing

Multi-GPU computing exists at all scales, from cheaper workstations using PCIe, to more expensive Quadro / Titan products using fewer NVLink, to high-end NVIDIA DGX servers.

Single workstation / server:

- a big enclosure for good cooling!
- up to 4 high-end cards in 16x PCIe v4 slots – up to 16GB/s interconnect.
- 2x high-end CPUs.
- 2-3kW power consumption – not one for the office!
- £12K-£18K

NVIDIA DGX H100 Deep Learning server:

- 8 NVIDIA GH100 GPUs, each with 80GB HBM2.
- 2× 56-core Intel Xeons (Platinum 8480C 2.0 GHz).
- 2 TB RAM memory, 8x 3.84TB NVMe.
- 900GB/s NVlink interconnect between the GPUs.
- £???,??? (DGX A100 currently costs ~£350K, launch price was £200K)

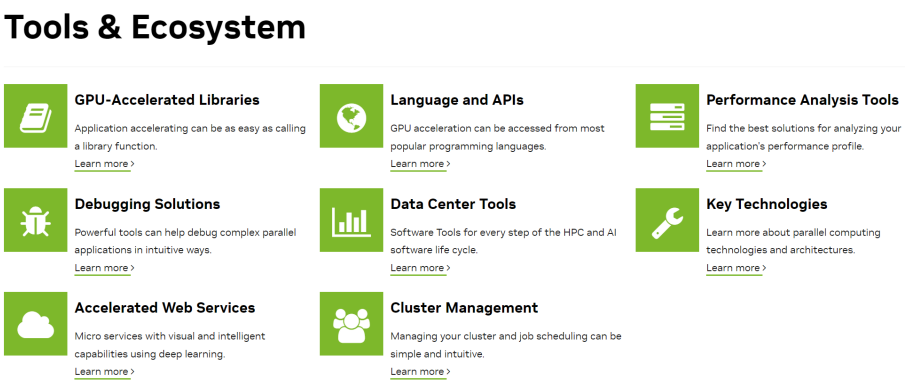
Ease of use

Even though we see the **cost of hardware (the CapEx) increasing significantly** (at the moment), **total cost of ownership should also consider the operating costs** (OpEx) and any upfront costs in adopting GPU technologies.

Hopefully during **this week you will have developed a feel for how easy it will be for you to gain GPU acceleration in your projects** / codes.

NVIDIAs rich software ecosystem makes it *relatively* easy to adopt GPU technology into your codes.

This helps to minimise development time needed to port an existing project to use GPUs.



So we should buy NVIDIA right?

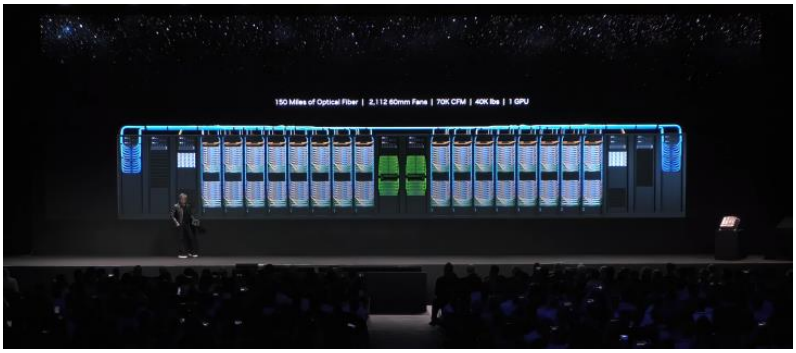
If you want to buy **A100 GPUs** in quantity you will be faced with an **8 month lead time**. For **H100 it is worse, currently about 12 months**.

Why? Because there is so much interest in training LLMs / foundational models using NVIDIA GPUs it has generated a supply and demand issue.

AI start-ups have access to significant funds, allowing them to buy lots of GPUs at a premium price…

"Inflection AI, a new startup found by the former head of deep mind and backed by Microsoft and Nvidia, last week raised $1.3 billion…"

https://www.tomshardware.com/news/startup-builds-supercomputer-with-22000-nvidias-h100-compute-gpus



22,000 H100s…

https://wccftech.com/inflection-ai-develops-supercomputer-equipped-with-22000-nvidia-h100-ai-gpus/

Look at the worlds largest machines, past and future trends - June 2021

The top500 lists the worlds fastest computers. A new list is produced in June and November each year.

Looking back, just 2 years ago, three out of the five fastest computers in the world were powered by NVIDIA GPUS.

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371
5	**Perlmutter** - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	706,304	64.59	89.79	2,528

Look at the worlds largest machines, past and future trends - June 2023

Two years later…

New machines, taking the top places in the top500, including the worlds first exaflop machine, are based on AMD, not NVIDIA.

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,824,768	238.70	304.47	7,404
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Frontier – The worlds first Exaflop machine

Hosted at the Oak Ridge Leadership Computing Facility (OLCF) Tennessee, **Frontier is the worlds only ExaFLOP supercomputer.**

It was delivered in partnership with HPE (Cray) and was also the worlds "greenest" supercomputer when it became operational in May 2022.
https://www.top500.org/lists/green500/2022/06/

Great presentation by Bronson Messer (Director of Science):

http://www.phys.utk.edu/archives/colloquium/2022/10-03-messer.pdf

By OLCF at ORNL - https://www.flickr.com/photos/olcf/52117623843/, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=119231238

Frontier – Compute configuration

The HPE Cray EX rack is a **liquid cooled and blade-based system**. This allows for very high density in a small footprint.

The EX4000 cabinet is a sealed unit that **uses closed-loop cooling to ensure minimal heat is exhausted into the data centre**.

Both Atos and Lenovo have similar technology.

All solutions use direct attached liquid cooled cold plates to remove heat from compute components.

This allows densities of up to 250KW per rack.

https://www.hpe.com/psnow/doc/a00094635enw

Frontier – Specs

- 9472 AMD Epyc "Trento" 64 core 2 GHz CPUs.
- 37888 Radeon Instinct MI250X GPUs.
- HPE Slingshot interconnect.
- Frontier is liquid-cooled, allowing 5x the density of an air-cooled architecture.
- Each rack holds 64 blades, each blade has two nodes.
- A node consists of one CPU, 4x GPUs (each having 128GB memory), 512 GB RAM and 4TB of flash memory.
- 21 Megawatts

https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

By OLCF at ORNL - https://www.flickr.com/photos/olcf/52117623843/, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=119231238

AMD as a solution? Hardware

We see from the change in the top500, **AMD GPUs are now gaining traction in HPC and scientific computing**.

This is because when the **total cost of ownership was considered for both Frontier and LUMI**, it was decided that **AMD GPUs would be more cost effective.**

DoE spent approximately 1/3 of their budget on hardware, the other 2/3 was on software porting and running costs.

A bit more on the MI250X that Frontier uses: 2x 64GB of HBM2e, 3.2TB/s bandwidth, 48TFLOP/s (fp32 and fp64) and 500 Watts TDP.

The forthcoming MI300 will be used in the 2 Eflop El Capitan machine

https://www.amd.com/en/products/specifications/professional-graphics/4476,19496

AMD as a solution? Cost vs performance

Currently, for a reasonable server expect to pay:

2x A100 server £25K
2x H100 server £37K
2x MI250X server £22K

AMD claims the MI250X is between 1.5x and 2.5x faster than A100 for a range of representative scientific codes (it should be though, it's 18 months newer).

YMMV…

LAMMPS Molecular Dynamics

Classical molecular dynamics package.

Application	Metric	Test Modules	Bigger is Better	4xMI250	4xA100 (SXM)	MI250/A100
LAMMPS	ATOM-Time Steps/s	Reaxff	Yes	19,482,180.48	8,850,000	Up to 2.2x[2]

LSMS Physics

Locally Self-Consistent Multiple Scattering (LSMS) is an order-N approach to the calculation of the electronic structure of large systems.

Application	Metric	Test Modules	Bigger is Better	1xMI250	1xA100 (SXM)	MI250/A100
LSMS	ATOM-Interactions/s	FePt54	Yes	3.95E_09	2.44E+09	Up to 1.6x[3]

MILC Physics

Lattice Quantum Chromodynamics (LQCD) codes simulate how elemental particles are formed and bound by the "strong force" to create larger particles like protons and neutrons.

Application	Metric	Test Modules	Bigger is Better	1xMI250	1xA100(SXM)	MI250/A100
MILC	Total Time (Sec)	Apex Medium	No	1,604.6	2,262	Up to 1.4x[1]

OpenMM Molecular Dynamics

High performance, customizable molecular simulation. Times in seconds to complete 8 identical simulations on one GPU.

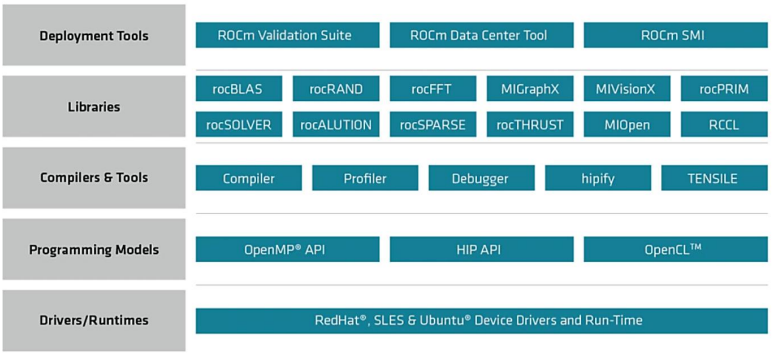
Application	Metric	Test Modules	Bigger is Better	1xMI250	1xA100 (SXM)	MI250/A100
OpenMM	Total Time (Sec) / 10,000 steps	*amoebakg*	No	387	921	Up to 2.4x[4]

Software - Radeon Open Compute Platform (ROCm)

AMD ROCm

One of the reasons **NVIDIA has been so dominant in the HPC space is its software ecosystem** and its ability to run on basic gaming cards (GeForce), to prosumer (Titan) to high end data centre cards (Tesla).

AMD now has a similar, growing (and in some parts rather familiar) software ecosystem called ROCm.



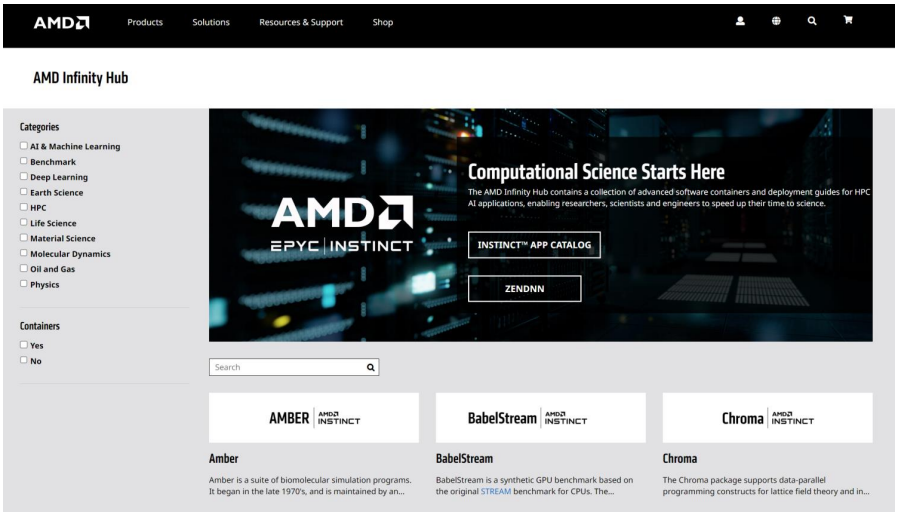
https://www.amd.com/en/graphics/servers-solutions-rocm

Software - Infinity hub

Have a growing number of leading packages optimised of Instinct. For example:

- Amber
- Gromacs
- Chroma
- QUDA
- CP2K
- PyTorch

Largely driven by DoE contracts.



https://www.amd.com/en/technologies/infinity-hub

https://www.amd.com/system/files/documents/gpu-accelerated-applications-catalog.pdf

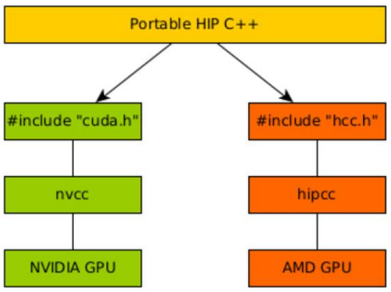
Heterogeneous-Compute Interface for Portability (HIP)

HIP is AMDs "version" of CUDA, it's a Kernel Language that looks, in many parts, similar to CUDA.

It **aims** to allow you **to create applications that are portable**, so when you write in HIP, your code will be able to run not only AMD GPUs, but NVIDIA also (at least that's the aim, just like OpenCL…).

AMD Claim:

- HIP has little (or no) performance impact compared to coding directly in CUDA.
- HIP allows coding in a single-source C/C++ programming language.
- The HIPIFY tools automatically convert most source from CUDA to HIP.
- Developers can specialize for the platform (CUDA or AMD) to tune for performance or handle tricky cases.



https://github.com/ROCm-Developer-Tools/HIP

https://www.youtube.com/watch?v=hSwgh-BXx3E

https://www.lumi-supercomputer.eu/preparing-codes-for-lumi-converting-cuda-applications-to-hip/

Heterogeneous-Compute Interface for Portability (HIP)

Let's look at some HIP (the main() code)…

```
…
char* inputBuffer;
char* outputBuffer;

hipMalloc((void**)&inputBuffer, (strlength + 1) * sizeof(char));
hipMalloc((void**)&outputBuffer, (strlength + 1) * sizeof(char));

hipMemcpy(inputBuffer, input, (strlength + 1) * sizeof(char), hipMemcpyHostToDevice);

hipLaunchKernelGGL(helloworld,
          dim3(1),
          dim3(strlength),
          0, 0,
          inputBuffer ,outputBuffer );

hipMemcpy(output, outputBuffer,(strlength + 1) * sizeof(char), hipMemcpyDeviceToHost);

hipFree(inputBuffer);
hipFree(outputBuffer);
…
```

https://github.com/ROCm-Developer-Tools/HIP-Examples/blob/master/HIP-Examples-Applications/HelloWorld/HelloWorld.cpp

Heterogeneous-Compute Interface for Portability (HIP)

Let's look at some HIP (the kernel code)…

```
__global__ void helloworld(char* in, char* out)
{
          int num = hipThreadIdx_x + hipBlockDim_x * hipBlockIdx_x;
          out[num] = in[num] + 1;
}
```

It all looks rather familiar, almost like someone has done a global "find `cuda` replace with `hip`"…

https://github.com/ROCm-Developer-Tools/HIP-Examples/blob/master/HIP-Examples-Applications/HelloWorld/HelloWorld.cpp

HIPIFY

HIPIFY is a set of scripts that will (try) to translate your CUDA source code into HIP auto~~matically~~magically for you.

The scripts are based on perl and clang.

Jack tried to take our AstroAccelerate code base (admittedly it is large and in parts quite complicated) and use HIPIFY to generate an AMD executable code.

He wasn't able (through no fault of his own!!).

When Jack emailed support he was pointed to the git repo and asked to raise an issue.

So some work to do before this is truly automagical.

Supported CUDA APIs

- Runtime API
- Driver API
- cuComplex API
- Device API
- RTC API
- cuBLAS
- cuRAND
- cuDNN
- cuFFT
- cuSPARSE
- CUB

https://github.com/ROCm-Developer-Tools/HIPIFY

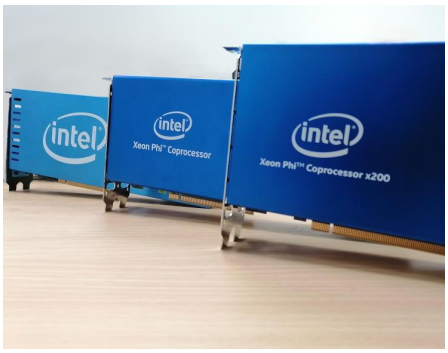
What about Intel?

Whilst Intel didn't invent the idea of a coprocessor, they did popularise it with the x87, dedicated to accelerating and adding functionality for floating point computations.

Since then Intel have had several failed attempts at entering the accelerator computing market.

- i860
- Larabee
- Xeon Phi (MIC)

In 2018 Intel revived the idea of a GPGPU accelerator and this has now become the Intel Xe (eXascaler for everyone).



By D2theW - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=114913985

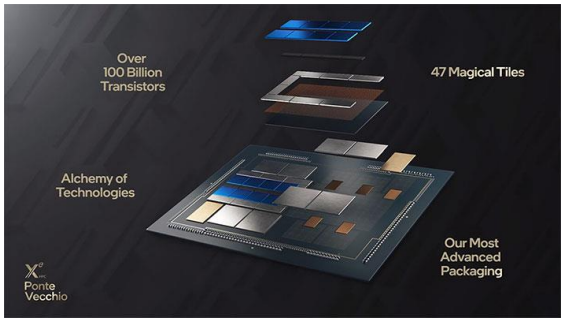
Intel Xe-HPC - Ponte Vecchio GPU

The new **Ponte Vecchio GPU is only just reaching the market.**

Intel specs are: 45 TFLOP/s, 5 TB/s bandwidth and 2 TB/s connectivity (I think this is Xe Link).

Tests on development silicon (very tentatively) indicate that **for some applications this reaches about 80% of the performance of an A100**.

We should keep in mind though, that the A100 is two years older than this, NVIDIAs current flagship GPU is the H100.



OneAPI

OneAPI is Intel's answer to HIP. It is an open standard and aims to deliver a single unified API that **can be used across all of its products from FPGAs to GPUs to CPUs.**

It aims to go further that just Intel products. **OneAPI has some functionality for both NVIDIA and AMD GPUs (via Codeplay plugins).**

This work is part of Intel's plan to make oneAPI the preferred alternative for heterogeneous, parallel programming.

One ring to rule them all…

oneAPI™

https://www.intel.com/content/www/us/en/developer/tools/oneapi/code-samples.html#gs.2twqvx

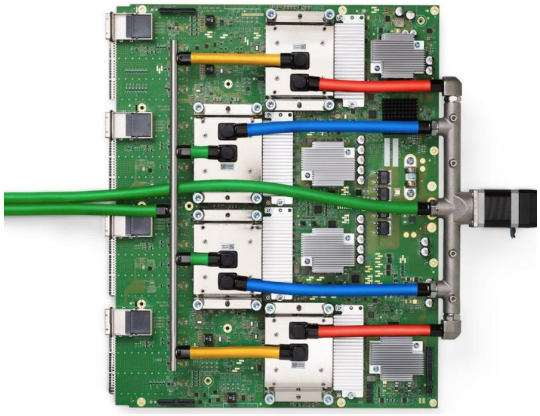
https://www.eejournal.com/article/intels-latest-version-of-oneapi-takes-advantage-of-new-intel-xeon-improvements-supports-amd-and-nvidia/?cid=org&source=linkedin_organic_cmd&campid=ww_23_oneapi&content=art-idz_&linkId=100000207031089

Google – Tensor Processing Unit

Google offer the TPU. This only suited to AI/ML training, but again, it's not general purpose in the way a CPU or GPU is.

TPUs can be accessed through google cloud. **To use them you write your code in TensorFlow / Torch or JAX and it's compiled to use TPU acceleration**.

TPUs are application specific integrated circuits (ASICs) that focus on the acceleration of matrix operations (performing similar operations to NVIDIAs tensor cores).



https://arxiv.org/ftp/arxiv/papers/2304/2304.01433.pdf

https://cloud.google.com/tpu

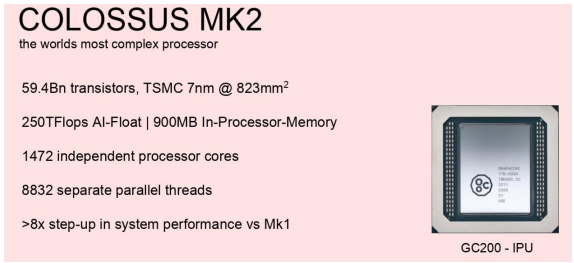
https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

Graphcore

Graphcore produce the Colossus Intelligent Processing Unit.

The Mark 2 IPU was released in 2020. The system design is aimed at sparse problems and has a memory system that is ideal for large AI models.

For **certain application spaces** graphcore products are more than competitive with NVIDIA GPUs.



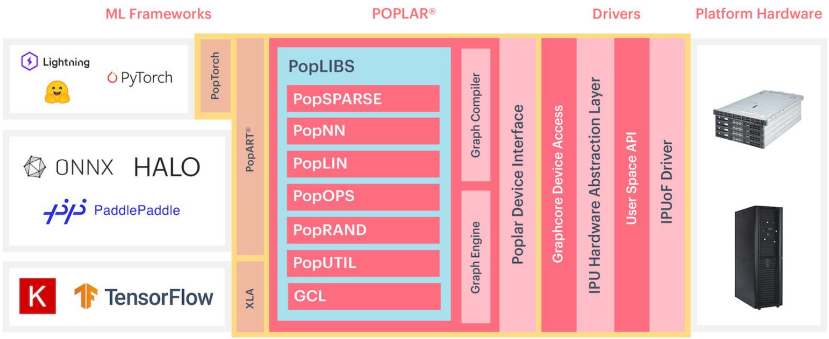
	NVIDIA DGX-A100 (8x A100)	GRAPHCORE 8x M2000	
FP32 compute	156TFLOP	2PFLOP	>12x
AI compute	2.5PFLOP[1]	8PFLOP[2]	>3x
AI Memory	320GB[3]	3.6TB[4]	>10x
System Price	$199,000MSRP	$259,600MSRP	

Graphcore - software

Graphcore have a **software stack called Poplar.**

This will take **code written using TensorFlow, PyTorch and Keras and generate code to run on the IPU.**

But be aware – the IPUs cannot do anything else. They are designed specifically for AI/ML training and work really well in areas such as NLP where models need large memory capacity close to the compute.



https://docs.graphcore.ai/projects/ipu-overview/en/latest/programming_tools.html

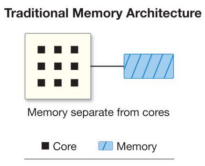
Cerebras

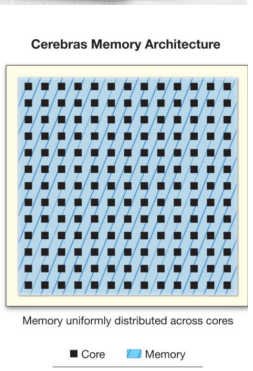
Cerebras produce **wafer level processors**. - Quite amazing.

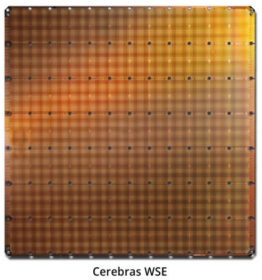
In terms of **software**, Cerebras has a similar approach to Graphcore. It has the **Csoft environment**. It too integrates Torch and TensorFlow to produce code that runs on the CS-2 platform.

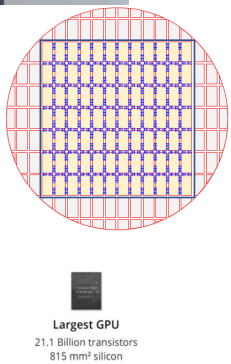
It also has a SDK to allow developers to write custom kernels.

I haven't seen a good comparison to other technology as yet.







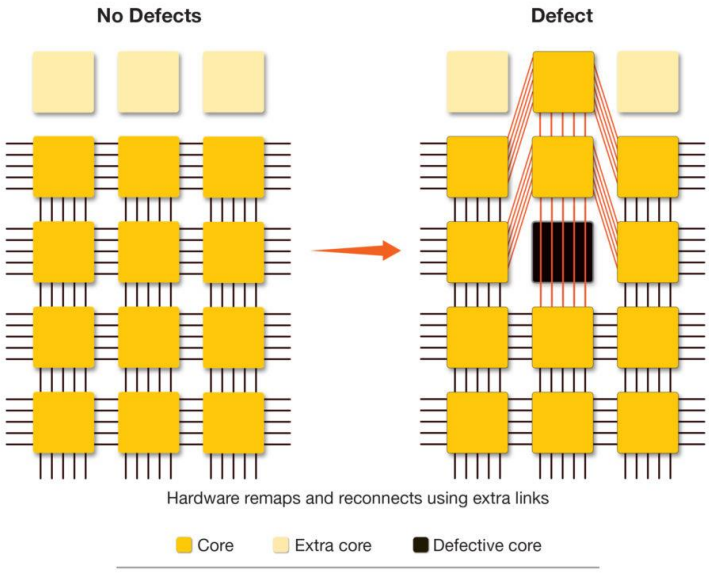


Cerebras

From a 12 inch wafer Cerebras produce a single processor (NVIDIA would get about 60 H100).

For those interested –
TMSC can produce ~ 8K wafers per month*

To ensure high yield, defective cores are identified the time of manufacturing and then the interconnect between cores is configured to avoid defective cores. Then added for that chip.



*CoWoS, TSMC has capacity to produce ~15M wafers per year.

https://www.cerebras.net/blog/wafer-scale-processors-the-time-has-come/#:~:text=A%20wafer%20is%20a%20circle,called%20scribe%20lines%20between%20them.

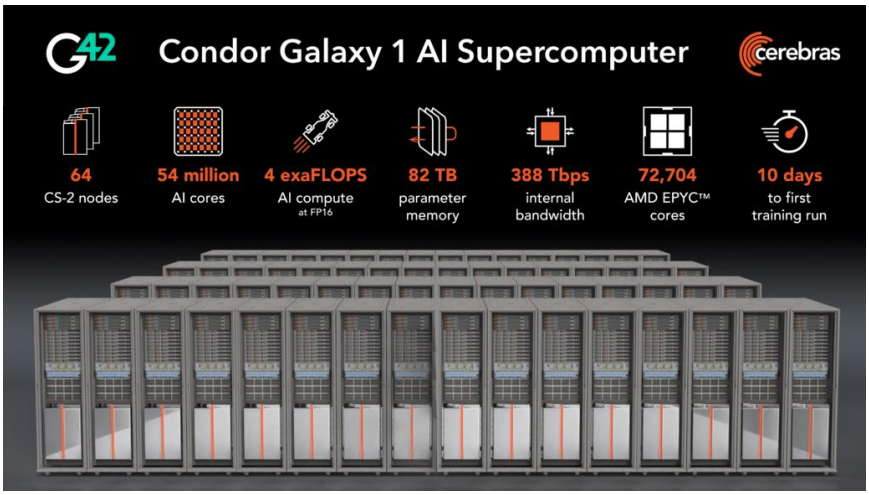
Cerebras and G42

Cerebras have very recently announced that they will supply G42 (UAE AI company) 3x Condor Galaxy systems as part of $100M deal.

A single CG system is capable of 4 EFLOP (fp16), has 54M Cerebras cores and 82TB of memory.

More on CG here:
https://www.condorgalaxy.ai/

Access to CG-1 here:
https://www.cerebras.net/product-cloud/

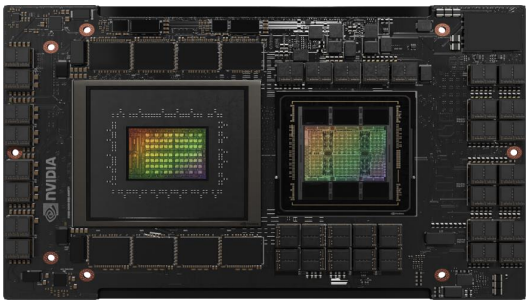


NVIDIA – Grace-Hopper

Grace-Hopper is NVIDIAs answer to the likes of Cerebras and Graphcore. The "Superchip" combines a Grace CPU and a Hopper GPU using NVLink C2C to deliver a CPU+GPU coherent memory model. The fruition of project Denver begun by NVIDIA in (Circa) 2014.

This kind of design will be crucial in progressing exascale computing in the years to come.

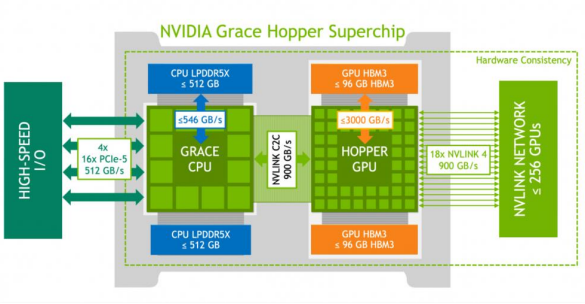
Whitepaper:
https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper



NVIDIA – Grace-Hopper

NVIDIA Grace + Hopper:

- 72x Arm Neoverse V2 cores (4×128-bit SIMD units per core).
- Up to 117 MB of L3 Cache.
- Up to 512 GB of LPDDR5X memory (546 GB/s of memory bandwidth).
- Up to 64x PCIe Gen5 lanes.
- NVIDIA Scalable Coherency Fabric (SCF) mesh and distributed cache with up to 3.2 TB/s memory bandwidth.
- NVIDIA Hopper GPU.
- NVIDIA NVLink-C2C - Up to 900 GB/s total bandwidth.
- Unified address space - each Hopper GPU can address up to 608 GB of memory within a superchip.
- NVIDIA NVLink Switch System connects up to 256x NVIDIA Grace Hopper Superchips using NVLink 4.
- Each NVLink-connected Hopper GPU can address all HBM3 and LPDDR5X memory of all superchips in the network, **for up to 150 TB of GPU addressable memory**.



DGX GH200 AI Supercomputer

The DGX GH200 was announced at Computex May 2023. It's NVIDIAs answer to the likes of Cerebras.

It connects 256 Grace-Hopper "superchips" via NVLink.

- **Single 144 terabytes GPU memory space**.
- **900 GB/s GPU-to-GPU bandwidth**.
- **1 exaFLOPS of FP8 AI performance**.

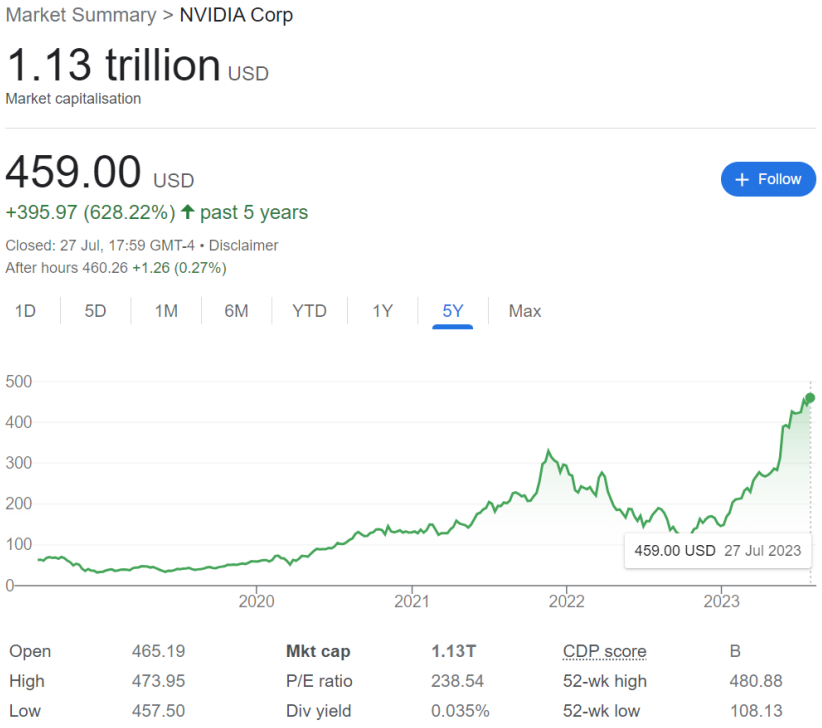
Whilst aimed at AI, this is a general purpose machine and so could be used for other areas of scientific computing.

https://nvidianews.nvidia.com/news/nvidia-announces-dgx-gh200-ai-supercomputer

https://resources.nvidia.com/en-us-dgx-gh200/nvidia-dgx-gh200-datasheet-web-us

The future?

NVIDIAs value continues to grow



The future?

AMDs, even with the success of Frontier is some way behind.

Market Summary > Advanced Micro Devices, Inc.

178.91 billion USD

Market capitalisation

111.10 USD

+92.61 (500.87%) ↑ past 5 years

Closed: 27 Jul, 17:57 GMT-4 • Disclaimer

After hours 111.50 +0.40 (0.36%)

+ Follow

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max

Open	111.79	**Mkt cap**	**178.91B**	CDP score	B
High	115.08	P/E ratio	460.59	52-wk high	132.83
Low	110.51	Div yield	-	52-wk low	54.57

The future?

Intel, failed to deliver Aurora, originally contracted to be completed by 2018, Intel are hoping to deliver later this year. Should be 2 ExaFLOP machine, each node will have 2x Intel Sapphire Rapids (CPU) and 6x Ponte Vecchio GPUs.

Out of the three Intel is worth the least!

Market Summary > Intel Corporation

144.11 billion USD

Market capitalisation

34.55 USD

-15.08 (-30.38%) ↓ past 5 years

Closed: 27 Jul, 18:02 GMT-4 • Disclaimer

After hours 37.30 +2.75 (7.96%)

+ Follow

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max

Open	34.82	**Mkt cap**	**144.11B**	CDP score	B
High	35.03	P/E ratio	-	52-wk high	40.42
Low	34.11	Div yield	1.45%	52-wk low	24.59

The future?

It's likely due to the cost of NVIDIA and shortage of supply that AMD will get a growing fraction of the accelerator market, especially given that they seem to be following (very closely!) NVIDIAs strategy – a great software ecosystem.

The HPE El Capitan supercomputer, due to be commissioned in Q4 2023 is an upcoming exascale supercomputer, hosted at the Lawrence Livermore, will be a 2+ ExaFLOP supercomputer and will displace Frontier as the world's fastest supercomputer.

- It's based on… AMD.

Summary

This lecture has looked at some present alternatives to NVIDIA and CUDA. We've also taken a look at some up-coming technologies, both software and hardware that might we worth watching out for over the coming years.

Lots of what you have learnt this week is transferable!

Also keep an eye on Mikes computing webpage here:

https://people.maths.ox.ac.uk/gilesm/computing.html