# OpenMP for Computational Scientists

## 5: Programming your GPU with OpenMP

Dr Tom Deakin
University of Bristol

Tuesday 1 December, 2020

# Outline

University of BRISTOL

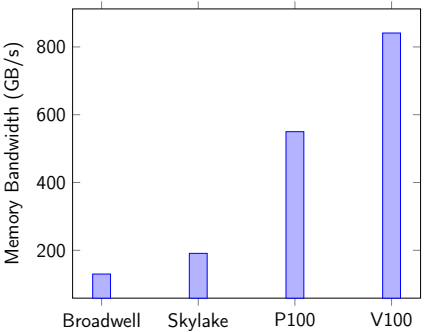
- GPU introduction
- The OpenMP `target` construct
- Memory movement
- Parallelism
- Device data environment

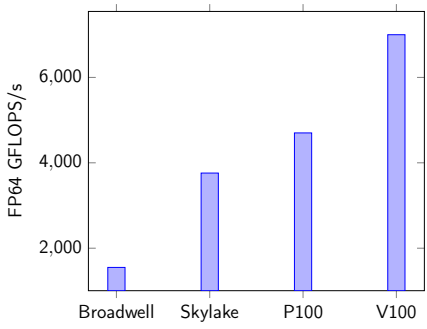
# GPU performance

University of BRISTOL

Why use a GPU?

- Hardware trends developing highly parallel processors.
- Many simple cores vs few complex cores is one approach.
- E.g.: NVIDIA Volta GPUs offer 4.4X memory bandwidth and 1.9X the FLOPS/s of dual-socket Intel Xeon (Skylake).



# Unlocking this potential

University of BRISTOL

- GPUs made of many cores. NVIDIA call them Streaming Multiprocessors (SMs):
  - V100 has 80 SMs.
  - P100 has 56 SMs.
- Each SM consists of 64 FP32 CUDA cores.
- CUDA cores are really organised as 2 vector units 32 wide (called warps).

**Take away**

GPUs are really vector-architectures made up of smaller blocks which execute together.

# GPUs need lots of parallelism

University of BRISTOL

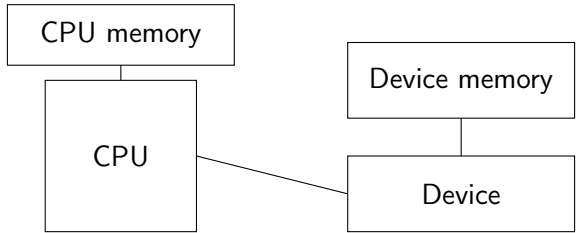
- GPUs are *throughput optimised*, whereas CPUs are *latency optimised*.
- Throughput optimised also called *latency tolerant*.
- GPUs achieve this by running many operations at once, and overlapping these with each other.
- Hence need many (many) operations...
- A V100 has 5,120 processing elements, each needing multiple units of work to overlap.

**Take away**

Massive amounts of parallelism to exploit.

# Device model

OpenMP has a host/device model.



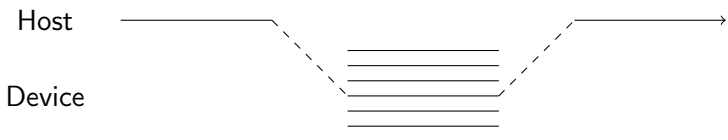

- Can have more than one device.
- Devices are connected to a host CPU via interconnect, such as PCIe or NVLink.
- Devices come with their own memory. On NVIDIA HPC GPUs Pascal/Volta this is HBM.

# Execution model

- Execution begins on the host CPU, with zero or more devices connected to the host.
- Memory spaces *not* shared!
- In OpenMP, some data copied automatically, plus controls for explicit copying.
- Directives are used to transfer execution to the device.

```
!$omp target [clause [clause] ...]
!$omp end target
```

- Host execution idles until target region completes (exact semantics based on tasks: see next session!).



# Practically programming a GPU

University of BRISTOL

- Programming the kernels/loops themselves is often the "easy" bit!
- Most of your programming time will be spent in getting minimal memory movement between host and device.
- Performance of the kernels themselves is often good right away assuming you're working with code that was good on a CPU.
- Optimisations for memory layout for vectorisation often apply to GPUs.

# The target construct

Get code region running on the device.

```
!$omp target [clause [clause]...]
  ...
!$omp end target
```

- Starts executing *in serial* on the target device.
- Need other constructs to expand parallelism.
- `nowait` clause:
  - Allows host thread to continue working. Must synchronise later using tasks.
- Other clauses mainly about memory movement.

# Data movement

University of BRISTOL

- Remember: memory is *not* shared between host and target.
- OpenMP uses a combination of implicit and explicit memory movement.
- This is the most complicated part of the offload specification.

- Memory movement is often a performance killer.
  - A V100 has 900 GB/s peak memory bandwidth.
  - Connected to the host via PCIe with 32 GB/s peak bandwidth.
  - Transfers between host and device are relatively very slow: minimise them.

# Data regions

University of BRISTOL

- Data needs mapping between host and device memory spaces.
- Variable names exist in host and device space: the compiler sorts out which one you mean when you use them in your code.

```
x(i) = 1.0

!$omp target
x(i) = 1.0
!$omp end target
```

- OpenMP runtime and compilers must work out when `x` is in host memory or device memory.
- Like having two arrays: `h_x` and `d_x` on the host and device respectively.

# Data movement

University of BRISTOL

- Mapping/transfers between host and device memory spaces occur when
  - enter/exit a `target` region.
  - `target enter/exit data` constructs.
  - `update` construct.

- Default behaviour:
  - Scalars are mapped `firstprivate`.
  - This means they *do not* get copied back to the host.
  - Actually saves a memory copy as passed like a kernel argument.

  - Stack arrays are mapped `tofrom`.
  - Heap arrays are not mapped by default.

# The map clause

University of BRISTOL

- Specify the transfer of data between host and device on a `target` region.

- Assume we have an array `A(1:N)` and a scalar `x`.
- Sizes of arrays are generally known in Fortran so don't need to specify amount of data to copy.
- Can use array slicing.
- (you shouldn't be using assume sized arrays in your kernels anyway!)

```
!$omp target map(...)
   ...
!$omp end target
```

# The map clause

Direction defined from the *host* perspective.

- `map(to: A, x)`

  On entering the region, copy from host to device.

- `map(from: A, x)`

  On exiting the region, copy from device to host. At start of `target` region, these are uninitialised on the device.

- `map(tofrom: A, x)`

  Same as applying `map(to: ...)` and `map(from: ...)`

- `map(alloc: A)`

  Allocate data on the device without copying from the host. It is uninitialised. Can later be copied back to the host (with `update` etc.) as long as allocated on host too.

# Exercise

University of BRISTOL

- Port your 5-point stencil code to the GPU.
- Use the `target` construct to transfer execution.
- Use the `map` clauses to transfer data.

# The one construct you'll need

University of BRISTOL

In general, you'll run loops on the device using:

```
!$omp target teams distribute parallel do
do i = 1, N
  ... ! Loop body
end do
!$omp end target teams distribute parallel do
```

We'll walk through what the constituent parts mean.

**Warning!**

Not using this combined statement can have severe performance issues.

# Execution model: teams

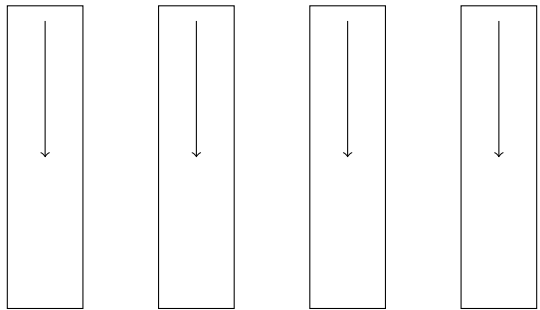
University of BRISTOL

```
!$omp target teams
... ! Code
!$omp end target teams
```

- OpenMP *threads* on a device are grouped into a *team*.
- Can synchronise threads *within* a team.
- *Cannot* synchronise between teams (must exit `target` region for this).
- Groups of teams are called a *league*.
- `target` construct offloads (serial) execution to device.
- `teams` construct creates a league of times.
- Master thread in *each* team (redundantly) executes the code.

# Execution model: teams

- The `target teams` construct creates a number of teams on the GPU, each containing one thread.
- All threads execute code block.



# Execution model: distribute

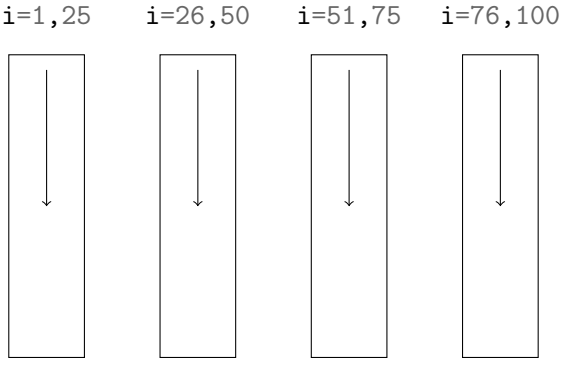
University of BRISTOL

- Distribute iterations of a loop across teams.
- Each team gets part of the iteration space.
- Change default assignment with `dist_schedule(static)` clause. Optionally include chunk size.
- Still only the master thread in the team executes them.

```
!$omp target teams distribute
do i = 1, N
... ! Code
end do
!$omp end target teams distribute
```

# Execution model: distribute

- The `target teams distribute` construct distributes loop iterations to the teams.
- Teams still only contain one thread.
- Each team computes a different iteration range.



# Execution model: parallel do

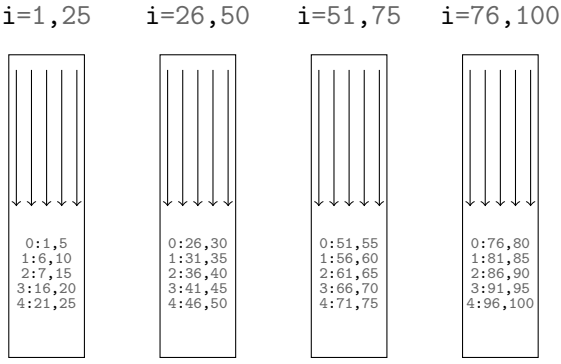
University of BRISTOL

- Same semantics as on the CPU!
- Launches threads within the team and distributes iterations across those threads.
- Note, iterations that were assigned to the team by the `distribute` construct are distributed across threads in the team.
- Can use the `schedule` clause too.

```
!$omp target teams distribute parallel do
do i = 1, N
... ! Code
end do
!$omp end target teams distribute parallel do
```

# Execution model: parallel do

- The `target teams distribute parallel do` construct launches threads in each team.
- Threads in the team share iteration space assigned by `distribute` construct.
- Finally have lots of parallel execution!



# SIMD construct

University of BRISTOL

- The `simd` construct is also valid on the `distribute parallel do` construct.
- OpenMP says this means SIMD instructions are generated.
- Minor implementation details differ between compilers.
- Compilers sometimes ignore different parts of the construct, depending on the situation.
- *`!$omp target teams distribute parallel do`* is a portable solution for practically obtaining the same parallelism across compilers.

# OpenMP: mapping the parallelism

University of BRISTOL

- OpenMP defines three levels of parallelism:
  1. Teams.
  2. Parallel threads.
  3. SIMD.
- But most GPU hardware really has two levels of parallelism:
  1. Compute units.
  2. Processing elements.
- Implementations have flexibility in how they associate OpenMP concepts to the underlying hardware.
- LLVM-based compilers, including Cray CCE >= 9, usually associate:
  - OpenMP teams to compute units.
  - OpenMP threads to processing elements.
  - OpenMP SIMD is ignored.
- Cray classic compiler maps SIMD to processing elements instead.

# How is this parallelism applied?

University of BRISTOL

Consider: `!$omp teams distribute`

- Loop iterations distributed between teams.
- Remember, you can't synchronize between teams.
- So all iterations are independent.
- Implementations can, and will, share the work across the whole GPU:
  - OpenMP teams being mapped to processing elements.
  - Doesn't matter how the work-items are grouped into work-groups (compute units) as no synchronisation.
- Behaves somewhat like SIMD auto-vectorization.

# What parallelism are you getting?

University of BRISTOL

- With more than one possible mapping, sometimes you need to find out what is really happening.
- Compiler documentation:
  - Cray: `man intro_openmp`
- Compiler output:
  - In CCE 10, `-rm` flag.
- Profiling:
  - `$ nvprof --print-gpu-trace`
  - Look for the number of threads per block, and number of blocks.
  - Combine that with knowledge of pragma and number of loop iterations.

## Coffee break!

# Memory movement best practice

University of BRISTOL

```
call initilise(A,B,C)

do t = 1, N
  call update(A)
  call update(B)
  call update(C)
end do
```

- Scientific codes tend to initialise data at the start, then run many kernels.
- Generally don't worry about timing of initialisation.
- Poor practice in MPI would for example gather and scatter data to reinitialise data every iteration.
- Same applies between host and device memory spaces!

# Data transfer revisited

University of BRISTOL

- Data transfers may happen at start and end of every `target` region.
- Often want to perform initial device data environment setup once, run through iterative loop, copying back at end.
- **Do not** want to copy the data every iteration! Very expensive.

# Enter/exit data constructs

University of BRISTOL

- Use `target enter data` and `target exit data` constructs to control device data environment.
- Bulk transfers happen at beginning and end, not for every `target` region in the big loop.
- Target regions inherit the existing data environment.

```
!$omp target enter data map(to: A, B, C)

do t = 1, N
  !$omp target
  ... ! E.g. Read A and B, write C
  !$omp end target
end do

!$omp target exit data map(from: C)
```

# Vector addition with memory movement

University of BRISTOL

```
! Initialise on host
allocate(A(N), B(N), C(N))
A = 1.0
B = 2.0

! Copy A and B to device, and allocate space for C
!$omp target enter data map(to: A, B) map(alloc: C)

! Run vector add on device
!$omp target teams distribute parallel do
do i = 1, N
  c(i) = a(i) + b(i)
end do
!$omp end target teams distribute parallel do

! Copy C back to host
!$omp target exit data map(from: C)
```

# Update construct

University of BRISTOL

- Often need to transfer data between host and device between different `target` regions.
- E.g. the host does something between the two regions.
- Example on next slide...
- Use the `update` construct to move the data explicitly between host and device, in either direction.
- Remember: direction is from the *host's* perspective.

# Update construct

```
1  !$omp target enter data map(to: A, B, C)
2  !$omp target
3  ... ! Use A, B and C on device
4  !$omp end target
5
6  ! Copy A from device to host
7  !$omp target update from(A(1:N))
8
9  ! Change A on the host
10 A = 1.0
11
12 ! Copy A from host to device
13 !$omp target update to(A(1:N))
14
15 !$omp target
16 ... ! Use A, B and C on device
17 !$omp end target
18
19 !$omp target exit data map(from: C)
```

# Halo exchange

Use the `update` clause with a typical halo exchange pattern.

```
1  !$omp target enter data map(...)
2
3  do t = 1, N
4    !$omp target ...
5      ... ! run kernel
6    !$omp end target ...
7
8    ! Copy latest halo data from device to host
9    !$omp target update from(halo)
10
11   ! Exchange with MPI
12   call MPI_Sendrecv(halo, ...)
13
14   ! Copy neighbour rank data to device
15   !$omp target update to(halo)
16 end do
17
18 !$omp target exit data map(...)
```

# Pointer swapping

Mapping between addresses on host and device is done when the target constructs are encountered.

```
1  !$omp target enter data map(to: A, B)
2
3  do t = 1, N
4    !$omp target
5      A(1) = B(1)
6
7    tmp => A
8    A => B
9    B => A
10 end do
```

- On the `target` construct, the pointers are evaluated every time.
- The compiler/runtime makes sure the mappings occur as expected.
- If use `map(from: )` then data transferred back to location specifed on *entry* to region, even if pointers swapped.
- Unlikely to run into confusing states if use `update` or `target exit data` to bring back the data.

# Asynchronous offload

University of BRISTOL

- By default, the host thread will idle and wait for the `target` region to complete.
- The `nowait` clause causes the `target` region to be offloaded as a task.
- The host thread can continue working asynchronously with the device!
- Must synchronise using `taskwait`, or at a `barrier` (explicit or implicit) depending on host threading design.
- Uses the OpenMP tasking semantics.

# Asynchronous offload

University of BRISTOL

```
1  !$omp target nowait
2  !$omp teams distribute parallel do
3  do i = 1, 10000000
4    ... ! Lots of work
5  end do
6  !$omp end teams distribute parallel do
7  !$omp end target
8  ! Host just continues because of nowait
9
10 call expensive_io_routine()
11
12 ! Wait for target task to finish
13 !$omp tastwait
```

# Compiler support

University of BRISTOL

- **Cray** provided first vendor supported implementation targeting NVIDIA GPUs in late 2015. Latest version of CCE now supports all of OpenMP 4.5 and some of 5.0.
- **LLVM/Clang** supports OpenMP 4.5 offload to NVIDIA GPUs. Used as base for many compilers.
- **IBM** XL compiler suite utilises their prior work with Clang to provide OpenMP target support for NVIDIA GPUs.
- **Intel** began support for OpenMP 4.0 targetting Intel Xeon Phi coprocessors in 2013 (version 15.0). Compiler versions 17.0+ support OpenMP 4.5 (targetting Xeon Phi), and oneAPI targetting Intel GPUs.
- **GCC** 6.1 introduced support for OpenMP 4.5 targetting Intel Xeon Phi and HSA-enabled AMD GPUs. GCC 10.0 can target Intel Xeon Phi, AMD GCN GPUs and NVIDIA GPUs.

`www.openmp.org/resources/openmp-compilers-tools/`.

# CUDA Toolkit: NVprof

University of BRISTOL

- The CUDA toolkit works with code written in OpenMP 4.5 without any special configuration.
- Useful to use the profiler `nvprof`.
- Particularly useful to check it ran on a GPU! Can silently fallback to CPU execution.
- Can generate high level profiling information, a timeline, and generate data for NVIDIA's `nvvp` profiler.

# nvprof output

```
nvprof ./stencil_target

==176642== Profiling application: ./stencil_target
==176642== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 86.24%  424.60ms        30  14.153ms  13.776ms  14.637ms  stencil_$ck_L49_1
  9.44%  46.496ms        33  1.4090ms     895ns  24.381ms  [CUDA memcpy HtoD]
  4.31%  21.242ms        32  663.82us  1.0240us  11.176ms  [CUDA memcpy DtoH]

==176642== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 53.72%  424.69ms        31  13.700ms  1.8730us  14.641ms  cuStreamSynchronize
 37.11%  293.35ms         1  293.35ms  293.35ms  293.35ms  cuCtxCreate
  5.96%  47.091ms        33  1.4270ms  6.9970us  24.584ms  cuMemcpyHtoD
  2.76%  21.844ms        32  682.63us  13.226us  11.304ms  cuMemcpyDtoH
  0.18%  1.4557ms         1  1.4557ms  1.4557ms  1.4557ms  cuMemHostAlloc
  0.17%  1.3477ms         5  269.54us  5.2270us  580.42us  cuMemAlloc
  0.04%  320.63us        30  10.687us  8.6930us  43.243us  cuLaunchKernel
  0.04%  317.57us         1  317.57us  317.57us  317.57us  cuModuleLoadData
  0.01%  45.755us         1  45.755us  45.755us  45.755us  cuStreamCreate
  0.00%  26.802us        34     788ns     283ns  4.3010us  cuEventCreate
  0.00%  4.3840us        11     398ns     309ns     585ns  cuDeviceGetAttribute
  0.00%  3.7440us         5     748ns     460ns  1.2540us  cuDeviceGet
  0.00%  3.6880us         3  1.2290us     356ns  2.7150us  cuDeviceGetCount
  0.00%  1.0500us         1  1.0500us  1.0500us  1.0500us  cuCtxSetCurrent
  0.00%  1.0230us         2     511ns     193ns     830ns  cuFuncGetAttribute
  0.00%     976ns         1     976ns     976ns     976ns  cuModuleGetGlobal
  0.00%     957ns         1     957ns     957ns     957ns  cuMemHostGetDevicePointer
  0.00%     806ns         1     806ns     806ns     806ns  cuModuleGetFunction
  0.00%     604ns         1     604ns     604ns     604ns  cuCtxGetCurrent
  0.00%     442ns         1     442ns     442ns     442ns  cuFuncSetCacheConfig
```

# nvprof output

```
nvprof --print-gpu-trace ./stencil_target

==176680== Profiling application: ./stencil_target
==176680== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device
     Context    Stream  Name
429.51ms  17.860ms                    -               -         -         -         -  122.19MB  6.6813GB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy HtoD]
447.94ms  15.150ms                    -               -         -         -         -  122.19MB  7.8763GB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy HtoD]
463.46ms  1.4080us                    -               -         -         -         -        8B  5.4186MB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy HtoD]
463.47ms     992ns                    -               -         -         -         -        4B  3.8455MB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy HtoD]
463.90ms  14.176ms           (128 1 1)       (128 1 1)       103  1.0078KB        0B         -           -  Tesla P100-PCIE
     1        14  stencil_$ck_L49_1 [43]
478.15ms  1.5360us                    -               -         -         -         -        8B  4.9671MB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy DtoH]
478.61ms     992ns                    -               -         -         -         -        8B  7.6909MB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy HtoD]
478.63ms  14.277ms           (128 1 1)       (128 1 1)       103  1.0078KB        0B         -           -  Tesla P100-PCIE
     1        14  stencil_$ck_L49_1 [79]
492.92ms  1.1200us                    -               -         -         -         -        8B  6.8120MB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy DtoH]
492.96ms     992ns                    -               -         -         -         -        8B  7.6909MB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy HtoD]
492.97ms  14.402ms           (128 1 1)       (128 1 1)       103  1.0078KB        0B         -           -  Tesla P100-PCIE
     1        14  stencil_$ck_L49_1 [83]
507.38ms  1.3120us                    -               -         -         -         -        8B  5.8151MB/s  Tesla P100-PCIE
     1         7  [CUDA memcpy DtoH]
```

# Exercise

University of BRISTOL

- Parallelise your 5-point stencil code on the GPU.
- Use the `target teams distribute parallel do` construct for parallelism.
- Use the `target enter/exit data` constructs to transfer data.
- Print out the grid sum for every iteration:
  - Need to use `reduction` clause.
  - Remember to `map` the reduction result!
- Experiment with different variations of the construct.
- Try using the `collapse` clause to increase parallelism.
- Profile the code with `nvprof` to explore the mapping.

# Summary

University of BRISTOL

- Can program a GPU using OpenMP with a single pragma!

```
!$omp target teams distribute parallel do
```

- Host/device execution model.
- Data movement between host and device data regions using:
  - `map` clauses,
  - `target enter/exit data` constructs,
  - `target update` constructs.
- The need to map back result of a reduction in OpenMP 4.5.