International Fortran Conference 2020

Interfacing with OpenCL from Modern Fortran for Highly Parallel Workloads

Laurence Kedward

University of Bristol

Presentation Outline

I. Background
II. Focal – A pure Fortran interface library for OpenCL
III. Results and Conclusions

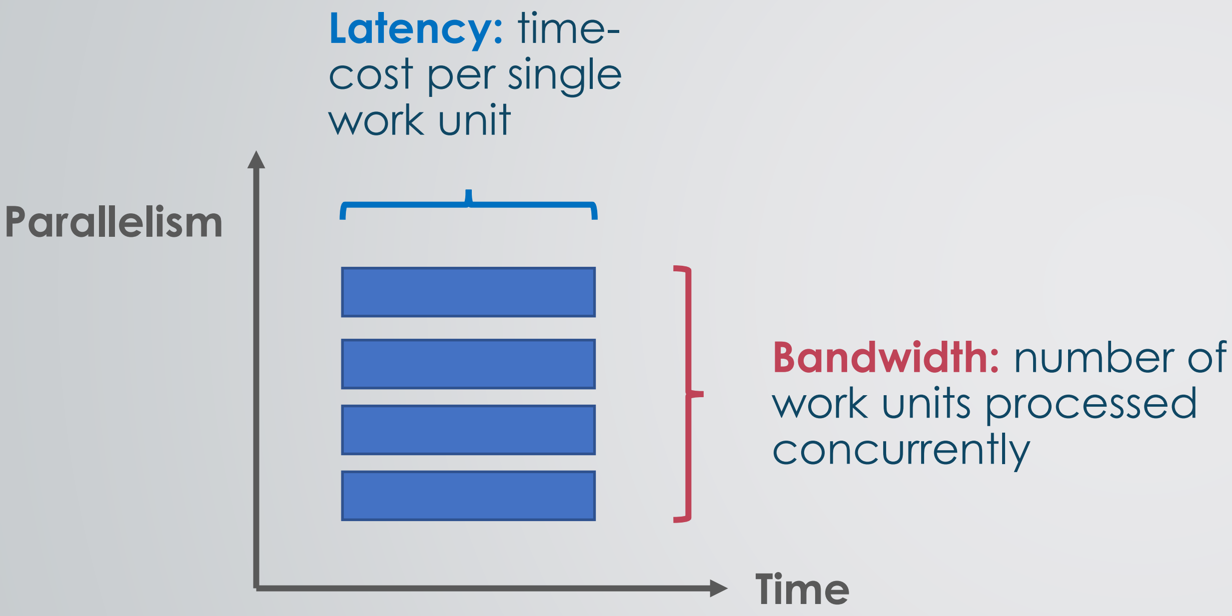
I. Background

Current and future trends in HPC

Programming CPUs vs GPUs

OpenCL overview

Maximising throughput: CPUs vs GPUs



GPUs: maximise bandwidth

- Highly parallel: 1000s of simple cores
- Very high bandwidth memory

$$\textbf{Throughput} = \frac{\textbf{Bandwidth}}{\textbf{Latency}}$$

(amount of work processed per unit time)

CPUs: minimise latency

- Large cache hierarchies (memory latency)
- Instruction pipelining / branch prediction
- Hyper-threading (hide latency)

Programming GPU architectures

Problem type: SIMD

- Running the same instructions on many thousands of different data
- Little or no global synchronisation between threads

Workload: arithmetic intensity

- Maximise the amount of computation and minimise the amount of memory accessed

Memory access: coalesced, structured, direct

- Consecutive threads should access contiguous memory (no striding)
- Hierarchical memory structure

Physically distinct memory space: minimise host-device data transfer

- PCIe bandwidth between GPU and CPU is very slow

Programming models

Languages
- CUDA (NVIDIA)
- HIP (AMD)
- OpenCL (Khronos)
- SYCL (Khronos)
- OneAPI (Intel)

Libraries
- cuBLAS, cuFFT, etc.
- clSPARSE
- Kokkos
- ArrayFire
- RAJA
- Loopy

Domain-specific languages
- PyFR
- OP2
- OPS

Directives
- OpenMP 4.0
- OpenACC

Programming models

Languages

- CUDA (NVIDIA)
- **HIP** (AMD)
- OpenCL (Khronos)
- **SYCL** (Khronos)
- **OneAPI** (Intel)

Libraries

- cuBLAS, cuFFT, etc.
- clSPARSE
- **Kokkos**
- **ArrayFire**
- **RAJA**
- Loopy

Domain-specific languages

- PyFR
- OP2
- OPS

Directives

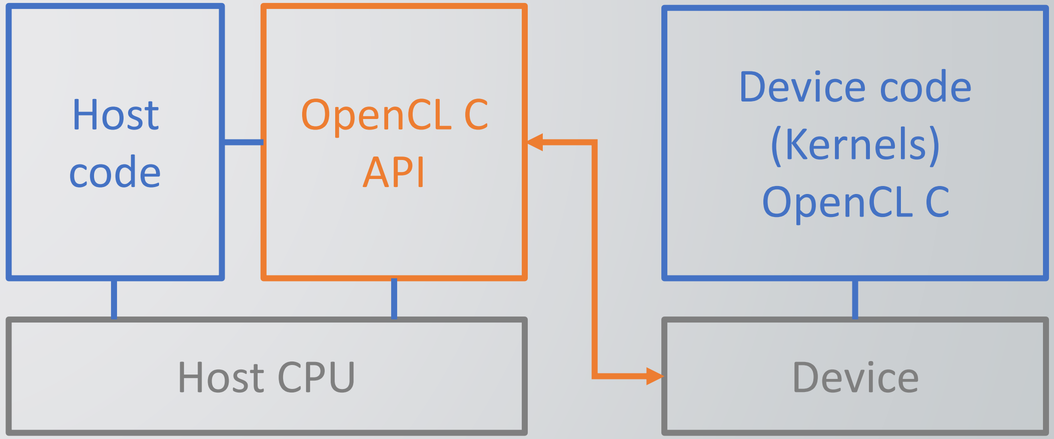
- OpenMP 4.0
- OpenACC

Is C++ becoming the *de facto* language for massive parallelism?

OpenCL

An open cross-platform standard for programming a diverse range of architectures for parallel workloads.

- **Portable:** Intel, AMD, NVIDIA, ARM, Qualcomm
- **Mature:** initial release in 2009
- **Explicit control**
  - Memory management
  - Dependencies
  - Execution & synchronisation



OpenCL & Fortran: Motivation

- OpenCL is an extensive and powerful framework for programming highly parallel workloads
- Low-level C API - bad for domain scientists and engineers:
  - Very verbose: distracts from research problem
  - Pointers aplenty: unsafe, prone to user-error
- Low-level C API – good for developing libraries and abstractions
  - Modern Fortran allows simplification and abstraction of C interfaces

II. The Focal Library

A modern Fortran abstraction library for OpenCL

Focal: An overview

➢ Focal is a modern Fortran abstraction of the OpenCL API

A Pure Fortran Library

- Not a language extension
- Tested with **`gfortran`** and **`ifort`**

Requirements

- Compiler support for **Fortran 2008**
- An OpenCL SDK (*e.g. NVIDIA CUDA SDK*)

Features

- Simple but **explicit syntax** to wrap API calls
- Remove use of C pointers
- Adds level of **type-safety**
- Built-in **error handling** of OpenCL API calls
- Built-in **'debug' mode** for checking correctness
- Built-in routines for collecting and presented **profiling information**

Focal: Using the library

Building

- Requires: F2008 compiler

```
$ make -j
```

```fortran
program test
    use Focal
    implicit none

end program test
```

Using

```
$ $FC -c test.f90 -J/path/to/focal/mod
```

Linking

```
$ $FC *.o -L/path/to/focal/lib -lFocal -lOpenCL
```

By example: initialising OpenCL

```
cl_uint errcode;
cl_platform_id* platform_ids;
cl_device_id* device_ids;
cl_uint num_devices;
cl_uint num_platforms;

// Get number of platforms
errcode = clGetPlatformIDs(0, NULL, &num_platforms);

// Allocate space for platforms
platformIDs = (cl_platform_id *) malloc(sizeof(cl_platform_id)*num_platforms);

// Get platforms
cl_int errcode = clGetPlatformIDs(num_platforms, platformIDs, num_platforms);

// INSERT logic to choose a platform

// Get number of devices
errcode = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 0,
        NULL, &num_devices);

// Allocate space for devices
device_ids = (cl_device_id *) malloc(sizeof(cl_device_id)*num_platforms);

// Get devices
errcode = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, num_devices,
        device_ids, num_devices);

// Create an OpenCL context
cl_context context = clCreateContext( NULL, num_devices, device_ids, NULL, NULL, &errcode);
```

Very verbose

- Pointers
- Low-level memory allocation
- Size in bytes
- API error handling (not shown)

By example: initialising OpenCL **with Focal**

Simplified and self-explanatory interface

Pointers replaced with derived types

```
type(fclDevice) :: device
...
device = fclInit(vendor='nvidia,amd',type='gpu',sortBy='cores')

call fclSetDefaultCommandQueue( fclCreateCommandQ(device) )
```

Additional commands allow multi-device and multi-platform initialisation

Initialise device memory

Derived types for device buffers:

```fortran
integer, parameter :: N = 1000000
type(fclDeviceInt32) :: device_int
type(fclDeviceFloat) :: device_float
type(fclDeviceDouble) :: device_dbl
...
call fclInitBuffer(cmdq,device_int,N)  ! Use specific cmdq
call fclInitBuffer(device_float,N)     ! Use default command queue
call fclInitBuffer(device_dbl,N)
```

Derived types in Focal bring a level of type-safety to the OpenCL API

Host-device memory transfers

```fortran
use iso_c_binding, only: c_int32_t, c_float, c_double
integer, parameter :: N = 1000000
type(fclDeviceInt32) :: device_int
type(fclDeviceFloat) :: device_float
type(fclDeviceDouble) :: device_dbl1, device_dbl2

integer(c_int32_t) :: host_int(N)
real(c_float) :: host_float(N)
real(c_double) :: host_dbl(N)
...
device_int = host_int             ! Host to device
host_float = device_float         ! Device to host
device_dbl2 = device_dbl1         ! Host to host
```

- ***Overloaded assignment** for buffer types gives simple syntax for buffer transfer operations.*
- *Only matching types are overloaded: adds **type safety***
- *Supports both blocking and non-blocking transfers*

Device kernels

Host code:

Compile program to obtain a kernel object:

```
type(fclProgram) :: prog
type(fclKernel) :: sumKernel
character(:), allocatable :: kernelSrc
...
call fclSourceFromFile('kernels/sum.cl',kernelSrc)
prog = fclCompileProgram(kernelSrc)
sumKernel = fclGetProgramKernel(prog,'sum')
```

Device code:

A simple OpenCL kernel to add two vectors:

```
kernel void sum(const int N,
                const global float * v1,
                global float * v2){
  int i = get_global_id(0);
  if(i < N) v2[i] += v1[i];
}
```

Host code:

Launch kernel:

```
type(fclDeviceFloat) :: array1, array2
...
sumKernel%global_work_size = [Nelem, 1, 1]
call sumKernel%launch(Nelem,array1,array2)
```

Synchronisation & Dependencies

Wait on host:

Synchronise host and device execution

```
! Wait on default command queue
call fclWait()

! Wait on a specific command queue
call fclWait(cmdq)

! Wait on last kernel launch
call fclWait(fclLastKernelEvent)

! Wait on last transfer on cmdq
call fclWait(cmdq%lastCopyEvent)
```

Set event dependencies:

Event objects allow easy dependencies

```
type(fclCommandQ) :: cmdq
type(fclKernel) :: myKernel
type(fclDeviceFloat) :: deviceArray
type(fclEvent) :: e
...
myKernel%launch(cmdq,deviceArray) ! Launch kernel
e = cmdq%lastKernelEvent

call fclSetDependency(e) ! Data transfer dependent
hostData1 = deviceData1  !  on kernel completion
```

Profiling device operations

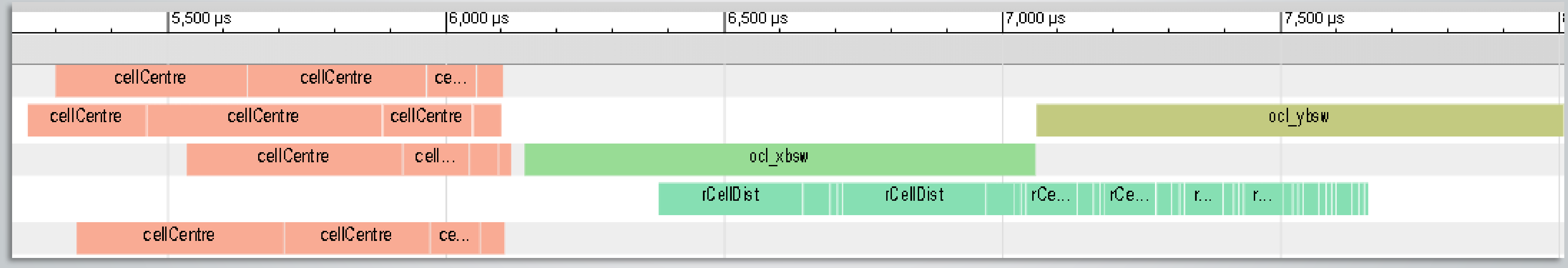
OpenCL has built-in support for **event profiling**

```fortran
type(fclProfiler) :: profiler
...
call fclDumpProfileData(profiler,[unit])

call fclDumpTracingData(profiler, &
                        filename='trace.json')
```

Profile name (Kernel)	No. of events	T_avg (ns)	T_max (ns)	T_min (ns)	Local Mem.	Privat Mem.	PWGS
initialise	1	308750	308750	308750	0	0	32
collide	5000	846595	1527166	735833	0	0	32
boundaryConditions	5000	154161	234083	131333	0	0	32
stream	5000	1067590	1419500	921000	0	0	32
macroVars	50	897174	1104250	802833	0	0	32

ns: nanoseconds, PWGS: Preferred work group size, Mem: Memory in bytes.



Extra details

Fortran submodules

- *Absolute separation of interface & implementation*
- *Fast compilation*
    - *Parallel*
    - *Incremental*
- *Interface parent module can be included as 'header' file*

Documentation

- *Website and user-guide (mkdocs): lkedward.github.io/focal-docs*
- *API reference (FORD): lkedward.github.io/focal*

github.com/LKedward/focal

License MIT

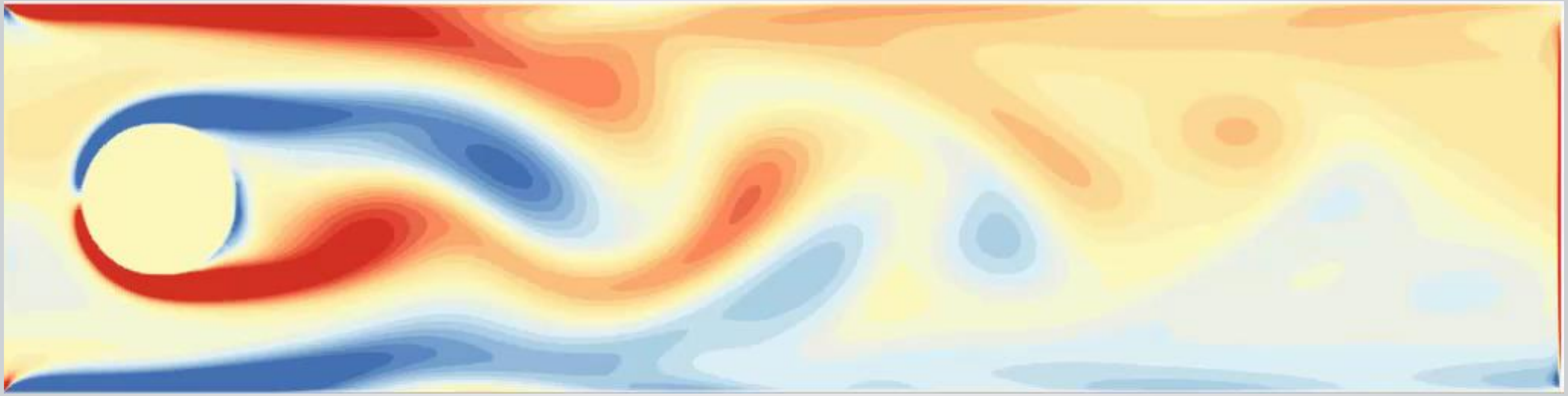
Development

- *Automated tests (Travis): using Intel OpenCL on x86*
- *Code coverage: codecov.io/gh/LKedward/focal*

III. Results & Conclusions

A modern Fortran abstraction library for OpenCL

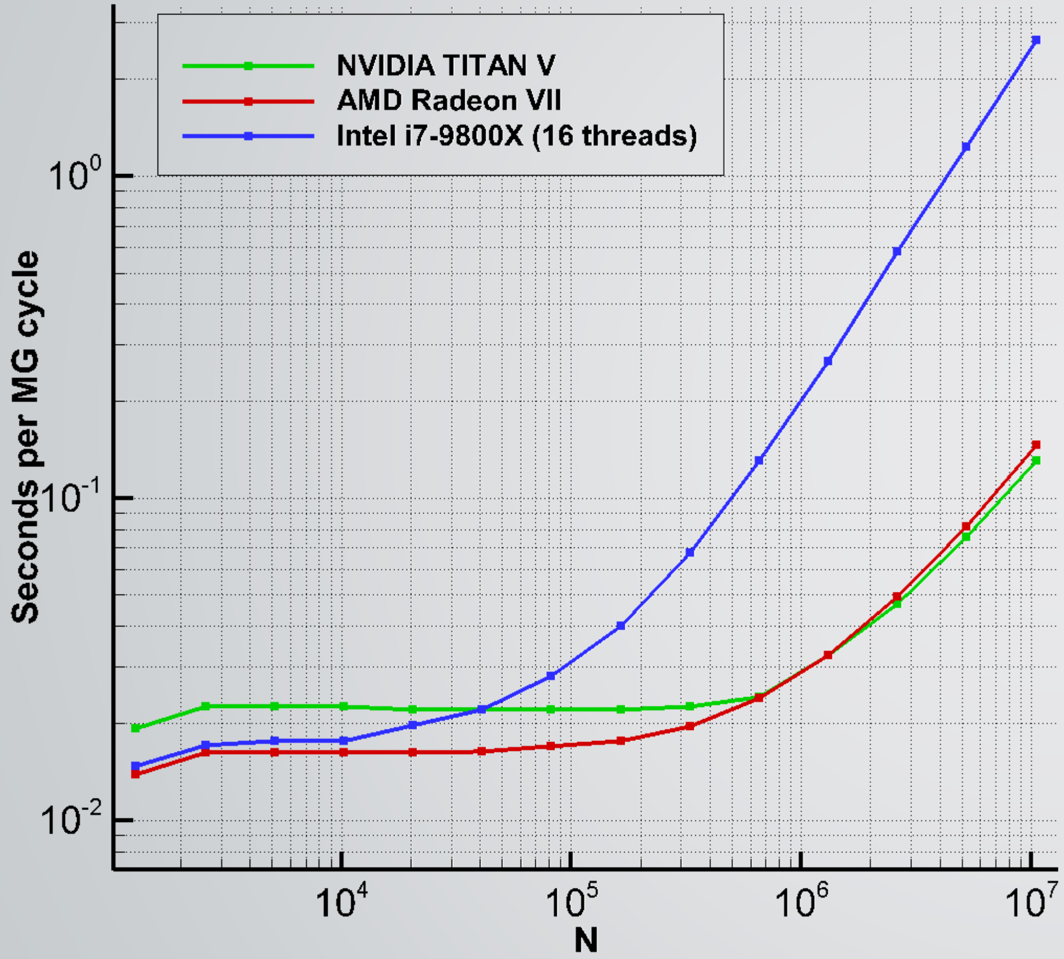
Demonstration: Lattice Boltzmann



github.com/LKedward/lbm2d_ocl

- **128 lines of Fortran code**
- **142 lines of OpenCL kernel code**
- **12x speedup** on Tesla P100 GPU **versus 28 Xeon cores**

Demonstration: Finite Volume Euler



Speedup over Intel CPU (16 threads):

AMD Radeon VII: 18x

NVIDIA Titan V: 20x

Very memory-bound application:

- Various optimisations for maximising arithmetic intensity
- 20% improvement from exploiting work-group shared memory
- Demonstrate mixed-precision solver

Kedward, L. J. and Allen, C. B., "Implementation of a Highly-Parallel Finite Volume Test Bench Code in OpenCL" in AIAA Aviation Forum, June 2020, doi.org/10.2514/6.2020-2923

Summary

- **Portable**
  - Pure standard-conforming Fortran library
  - Only dependency is OpenCL SDK from hardware vendor
  - No vendor/compiler lock-in
  - OpenCL is a mature and portable standard
- **Easy to use**
  - Easy to build and use
  - Simple and explicit syntax
- **Powerful**
  - Supports most features of OpenCL 1.2
  - Allows fine-grain control and optimisation
- **Dual source**
  - Still need to write kernel code in OpenCL C dialect

Ideas for the Future

Fortran already contains the abstractions required to simplify accelerator programming

```fortran
kernel subroutine fluxIntegral(grid,primitives,resid,...)
  type(TGRID), intent(in) :: grid
  type(TPRIM), intent(in) :: primitives
  real(c_double), intent(out) :: resid(:,:)

  real(c_double), address(shared) :: wavespeeds(size(resid,1))
  integer :: i = get_kernel_index(0)

  ...

end subroutine fluxIntegral
```

- Assumed-shape arguments
- Native multi-dimensional arrays
- Custom array bounds
- Derived types/structs in kernels
- Optional kernel arguments
- Work-group shared buffers
- Operator overloading

Acknowledgements

This work was funded by the MENtOR project, a UKVLN project supported by the Engineering and Physical Sciences Research Council (EPSRC) of the UK. Grant reference number EP/S010378/1

EPSRC
Engineering and Physical Sciences Research Council

MENtOR
A UKVLN Project
Methods and Experiments for NOvel Rotorcraft

UK VERTICAL LIFT NETWORK

International Fortran Conference 2020

Interfacing with OpenCL from Modern Fortran for Highly Parallel Workloads

Thank you for listening!

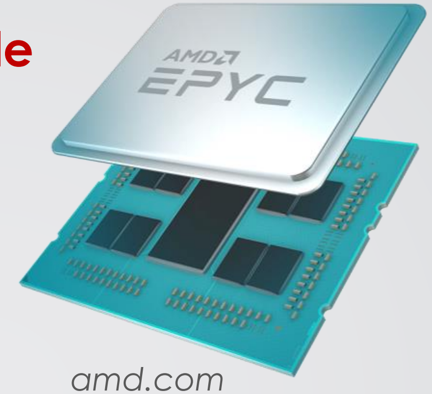
University of BRISTOL

Current and future trends in HPC

Intel.com

Intel 2nd gen. (2019) Xeon scalable

- 56 cores & 112 threads
- Up to 3.80 GHz
- 77MB cache



amd.com

AMD 2nd gen. (2018) EPYC

- 64 cores & 128 threads
- Up to 3.4GHz clock
- 256MB cache
- 30 bn transistors

Fujitsu ARM A64FX many-core processor

- 2.7 TFLOPs
- 8bn transistors
- 48 cores
- 32GB on-chip memory
- 1024GB/s on-chip bandwidth

insidehpc.com

NVIDIA A100 'Ampere' (late 2020)

- 19.5 TFLOPs
- 54bn transistors
- 8192 compute cores
- 1866 GB/s memory bandwidth