Welcome to CIUK 2021

Science and Technology Facilities Council

Scientific Computing

CIUK

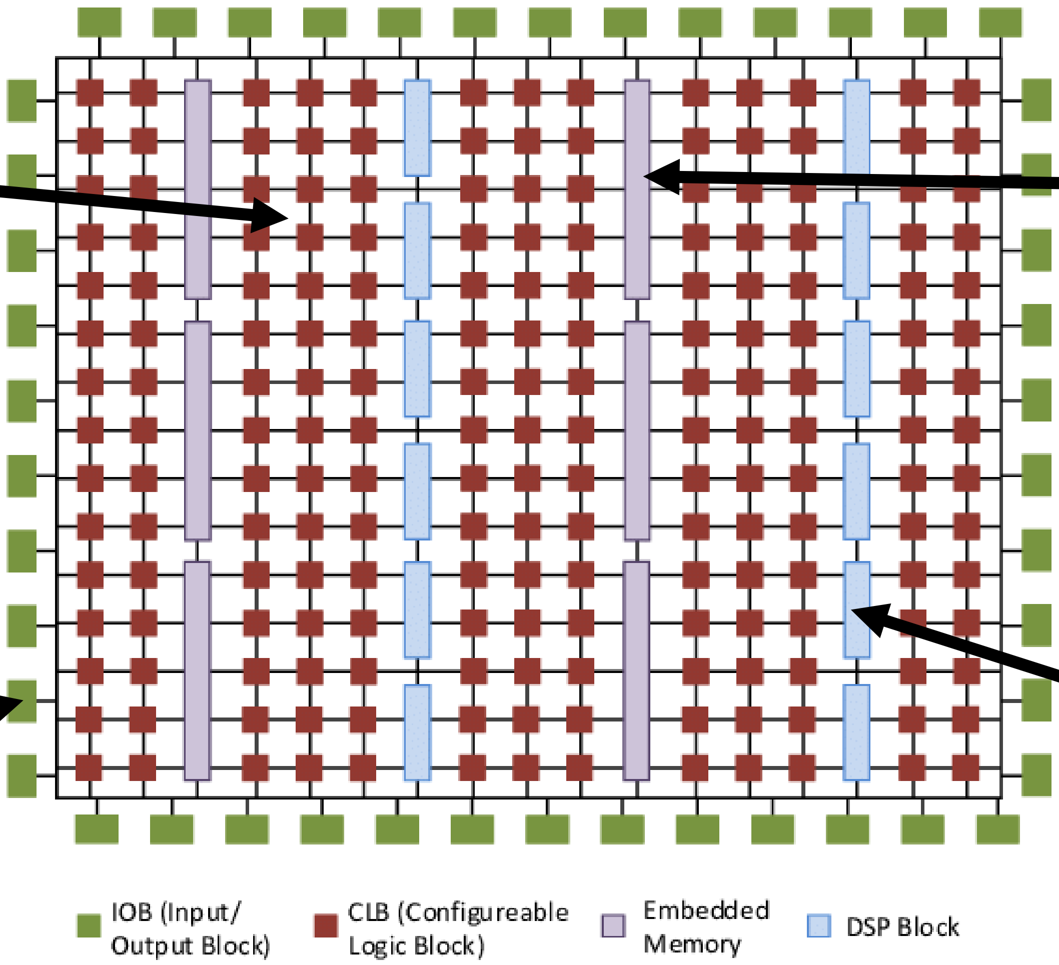
CIUK 2021

FPGAs for scientific workloads: The why and the how

Dr Nick Brown, EPCC
n.brown@epcc.ed.ac.uk

What are Field Programmable Gate Arrays (FPGAs)?

Configurable Logic Block (CLB) contain look up tables which are configured with the application logic. These are sitting within a sea of configurable interconnect



🟩 IOB (Input/ Output Block)
🟥 CLB (Configureable Logic Block)
▨ Embedded Memory
▨ DSP Block

Very fast on-chip memory, known as Block RAM (BRAM) and approx. 40 TB/s, similar to L1 cache and accessible in approx. 1 cycle. Typically a few MB on top end FPGAs

Lots of I/O connections to the outside world, such as PCIe, HBM2, DDR-DRAM, QSFP28 networking

ASIC style components to perform arithmetic, used as the building blocks for floating point arithmetic as this saves a large amount of configurable logic

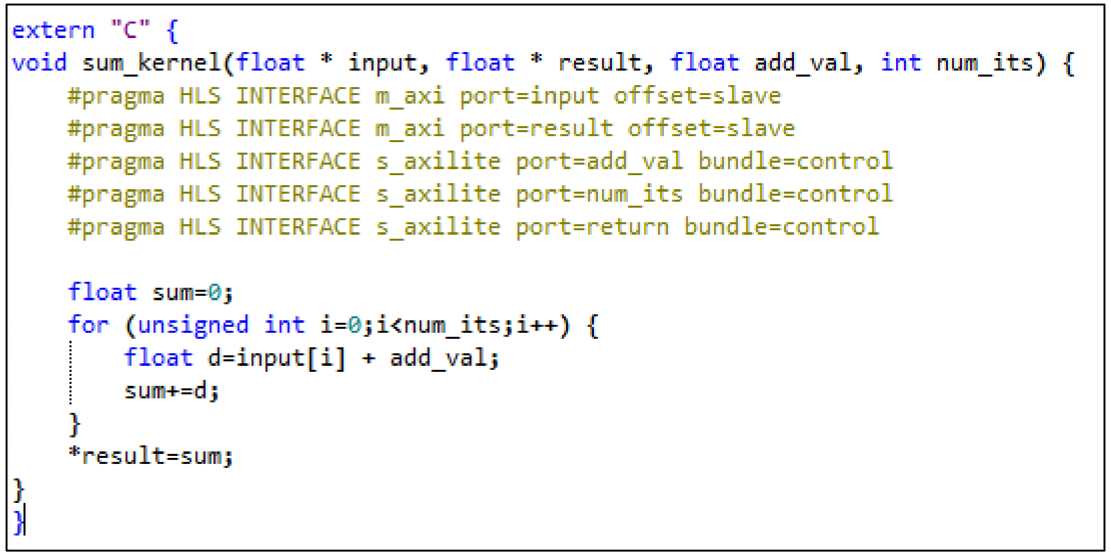
Worth a fresh look for HPC

- Over 10 years ago we had an FPGA cluster in EPCC
  - But immaturity of the hardware (struggling to match CPU performance) and software ecosystem (difficult to program and lack of tooling) ultimately meant that this was not continued into large scale HPC adoption

- But a decade is a long time, and times change!
  - Much more capable hardware and exciting new technologies on the horizon
  - (Very) significantly enhanced software ecosystem allowing the programming of these via C or C++

But isn’t programming these things still hard?

- High Level Synthesis (HLS) allows us to write code in C or C++ for FPGAs – no need to write code in VHDL or Verilog anymore!
    - Xilinx HLS and OpenCL for Xilinx FPGAs, OpenCL for Intel FPGAs
    - OpenCL on the host to drive kernels and transfer data
    - Recent developments make this more a question of software development rather than hardware design, but there are still some challenges!



```
extern "C" {
void sum_kernel(float * input, float * result, float add_val, int num_its) {
    #pragma HLS INTERFACE m_axi port=input offset=slave
    #pragma HLS INTERFACE m_axi port=result offset=slave
    #pragma HLS INTERFACE s_axilite port=add_val bundle=control
    #pragma HLS INTERFACE s_axilite port=num_its bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control

    float sum=0;
    for (unsigned int i=0;i<num_its;i++) {
        float d=input[i] + add_val;
        sum+=d;
    }
    *result=sum;
}
}
```

```
> v++ -t hw --config design.cfg -O3 -c -k sum_kernel -o'sum.hw.xo' device.cpp
```

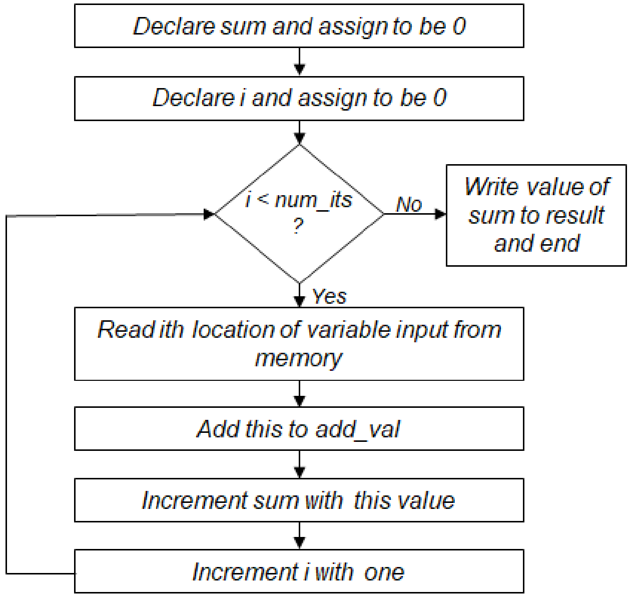
- Some profiling and debugging tools also provided (e.g. Intel integrated with Vtune)
- But building takes a long time (many hours!)
    - Therefore these toolchains provide emulation capabilities where one can build their code in minutes and emulate on the CPU how it would run

What's reconfigurable/spatial/dataflow computing?

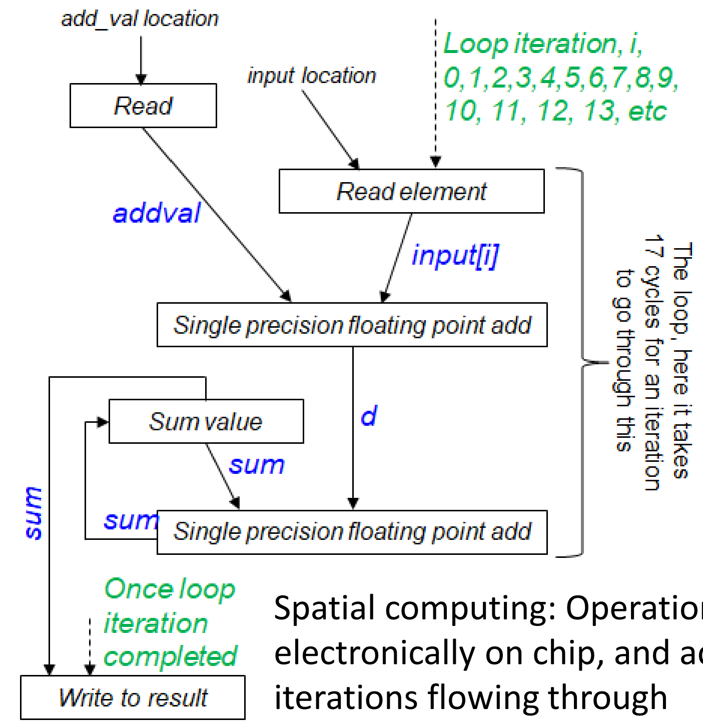
Temporal computing (CPU or GPU)

```
float sum=0;
for (unsigned int i=0;i<num_its;i++) {
  float d=input[i] + add_val;
  sum+=d;
}
*result=sum;
```

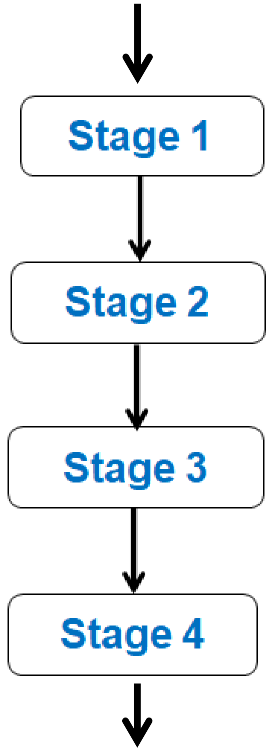
Reconfigurable architecture (dataflow)



Declare sum and assign to be 0
Declare i and assign to be 0
i < num_its ?
No
Write value of sum to result and end
Yes
Read ith location of variable input from memory
Add this to add_val
Increment sum with this value
Increment i with one



add_val location
Read
input location
Loop iteration, i, 0,1,2,3,4,5,6,7,8,9, 10, 11, 12, 13, etc
Read element
addval
input[i]
Single precision floating point add
The loop, here it takes 17 cycles for an iteration to go through this
Sum value
d
sum
sum
sum
Single precision floating point add
Once loop iteration completed
Write to result

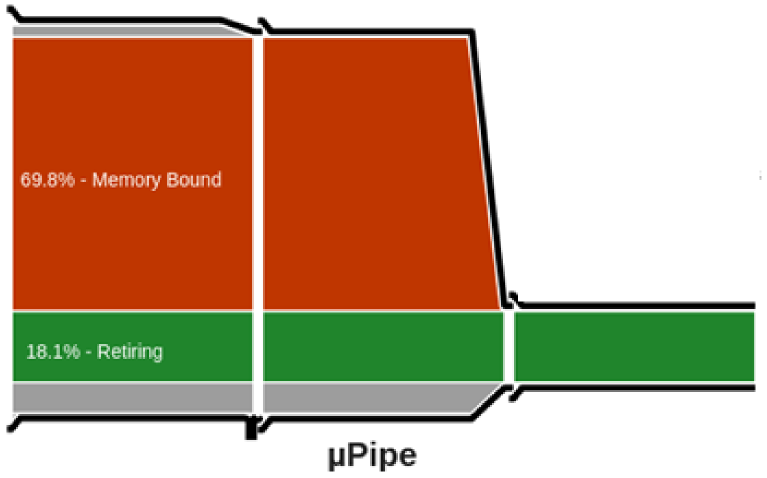


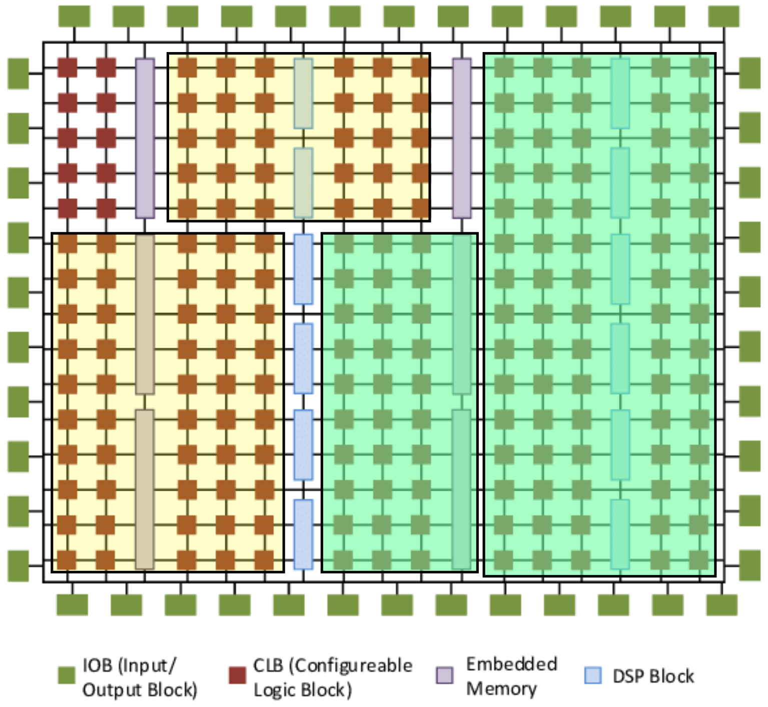
Temporal computing: Can be thought of like a flowchart, with the PE (e.g. CPU or GPU) executing one stage after another

Spatial computing: Operations implemented electronically on chip, and acts as a pipeline, loop iterations flowing through

But what HPC workloads are best suited?

- It's important to pick your battles, and with CPUs and GPUs we have a good idea what performance properties are best suited to the technologies

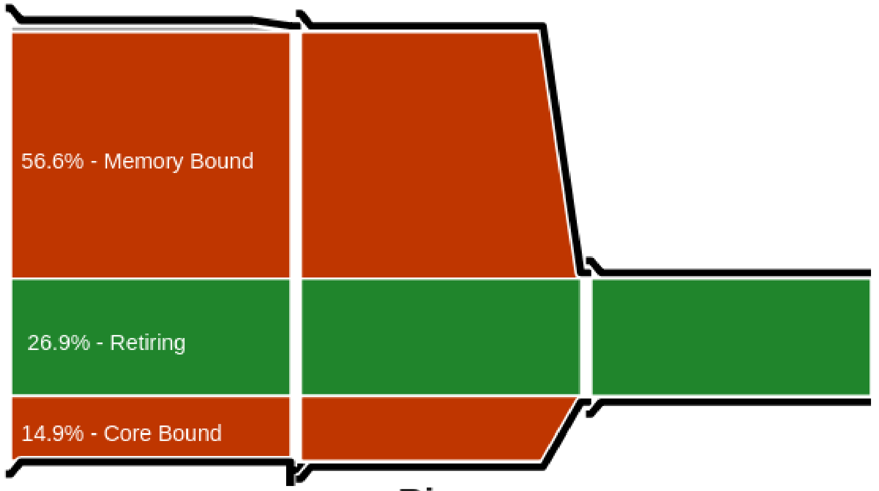




- For FPGAs, if your workload is compute-bound then a GPU is probably a better option!
- However if your code is memory bound or bound by other core issues, this is when you could get benefits
  - Tailoring how you use that fast L1-style BRAM memory
  - Exploiting high-bandwidth off-chip connections and HBM2

Example: AX kernel of Nekbone proxy-app

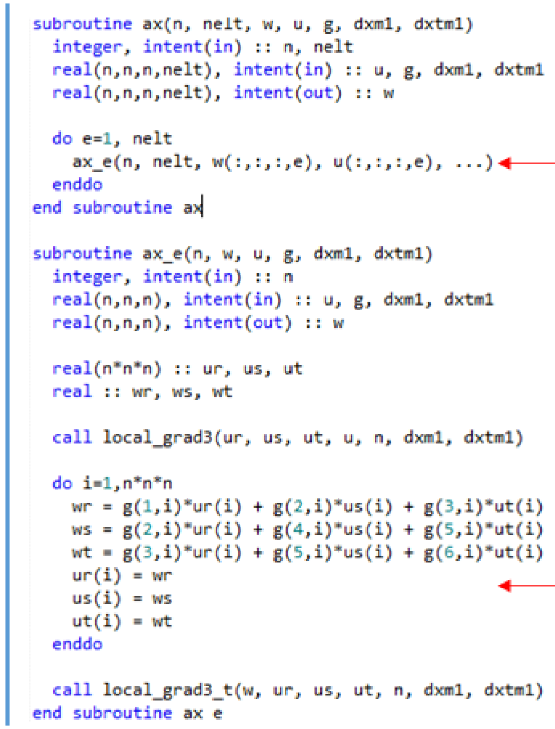
- Nek5000 is used for high fidelity simulation of rotating parts.
  - Nekbone is a proxy-app that captures the basic structure of Nek5000



µPipe

On 24 CPU cores: 65.74 GFLOPs
- Only 11.7 times faster than one CPU core

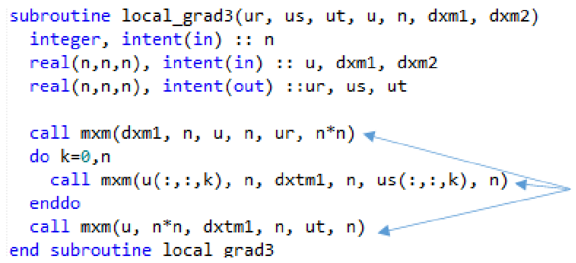
On a Xeon Platinum Cascade Lake with N=16, 800 elements



```
subroutine ax(n, nelt, w, u, g, dxm1, dxtm1)
  integer, intent(in) :: n, nelt
  real(n,n,n,nelt), intent(in) :: u, g, dxm1, dxtm1
  real(n,n,n,nelt), intent(out) :: w

  do e=1, nelt
    ax_e(n, nelt, w(:,:,:,e), u(:,:,:,e), ...)
  enddo
end subroutine ax

subroutine ax_e(n, w, u, g, dxm1, dxtm1)
  integer, intent(in) :: n
  real(n,n,n), intent(in) :: u, g, dxm1, dxtm1
  real(n,n,n), intent(out) :: w

  real(n*n*n) :: ur, us, ut
  real :: wr, ws, wt

  call local_grad3(ur, us, ut, u, n, dxm1, dxtm1)

  do i=1,n*n*n
    wr = g(1,i)*ur(i) + g(2,i)*us(i) + g(3,i)*ut(i)
    ws = g(2,i)*ur(i) + g(4,i)*us(i) + g(5,i)*ut(i)
    wt = g(3,i)*ur(i) + g(5,i)*us(i) + g(6,i)*ut(i)
    ur(i) = wr
    us(i) = ws
    ut(i) = wt
  enddo

  call local_grad3_t(w, ur, us, ut, n, dxm1, dxtm1)
end subroutine ax_e
```

Iterate over elements

Multiply and add values calculated in local_grad3



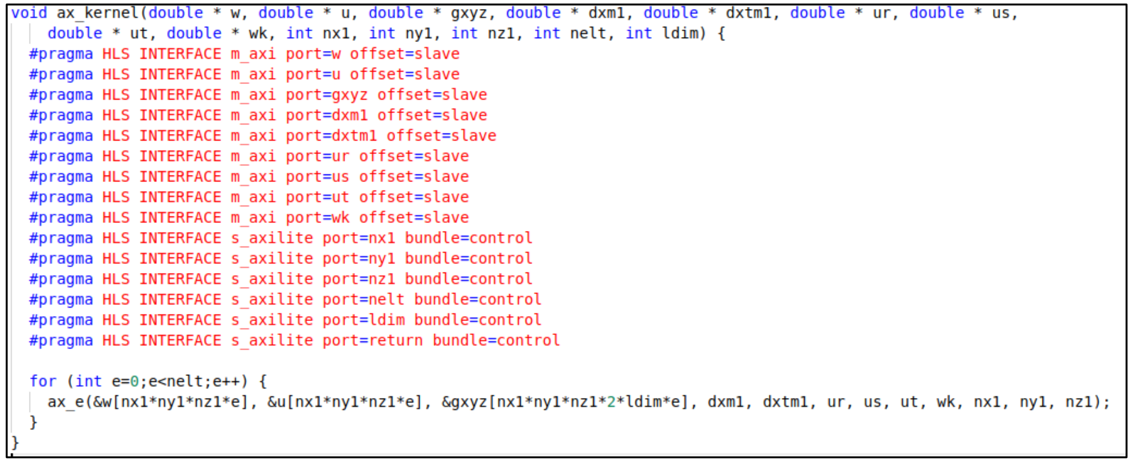
```
subroutine local_grad3(ur, us, ut, u, n, dxm1, dxm2)
  integer, intent(in) :: n
  real(n,n,n), intent(in) :: u, dxm1, dxm2
  real(n,n,n), intent(out) ::ur, us, ut

  call mxm(dxm1, n, u, n, ur, n*n)
  do k=0,n
    call mxm(u(:,:,k), n, dxtm1, n, us(:,:,k), n)
  enddo
  call mxm(u, n*n, dxtm1, n, ut, n)
end subroutine local grad3
```

Matrix multiplications

- All double precision floating point
- AX kernel applies the Poisson operator of the CG solver accounts for approx. 75% of overall runtime of Nekbone
- 800 elements, and a size of N, with the number of grid points equal to N^3
  - For instance with N=16 then there are 831488 double precision floating point operations per element

Use FPGA to ameliorate overhead of memory access and keep compute fed with data

Overview of single kernel performance



```
void ax_kernel(double * w, double * u, double * gxyz, double * dxm1, double * dxtm1, double * ur, double * us,
    double * ut, double * wk, int nx1, int ny1, int nz1, int nelt, int ldim) {
  #pragma HLS INTERFACE m_axi port=w offset=slave
  #pragma HLS INTERFACE m_axi port=u offset=slave
  #pragma HLS INTERFACE m_axi port=gxyz offset=slave
  #pragma HLS INTERFACE m_axi port=dxm1 offset=slave
  #pragma HLS INTERFACE m_axi port=dxtm1 offset=slave
  #pragma HLS INTERFACE m_axi port=ur offset=slave
  #pragma HLS INTERFACE m_axi port=us offset=slave
  #pragma HLS INTERFACE m_axi port=ut offset=slave
  #pragma HLS INTERFACE m_axi port=wk offset=slave
  #pragma HLS INTERFACE s_axilite port=nx1 bundle=control
  #pragma HLS INTERFACE s_axilite port=ny1 bundle=control
  #pragma HLS INTERFACE s_axilite port=nz1 bundle=control
  #pragma HLS INTERFACE s_axilite port=nelt bundle=control
  #pragma HLS INTERFACE s_axilite port=ldim bundle=control
  #pragma HLS INTERFACE s_axilite port=return bundle=control

  for (int e=0;e<nelt;e++) {
    ax_e(&w[nx1*ny1*nz1*e], &u[nx1*ny1*nz1*e], &gxyz[nx1*ny1*nz1*2*ldim*e], dxm1, dxtm1, ur, us, ut, wk, nx1, ny1, nz1);
  }
}
```

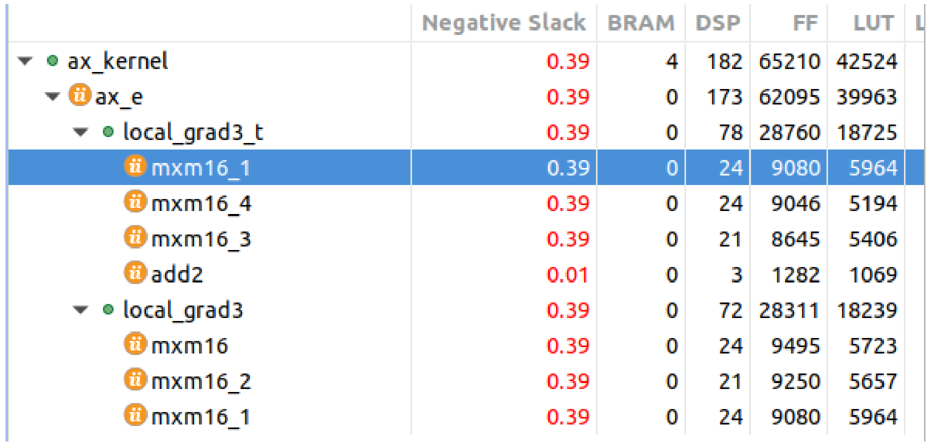
Description	Performance GFLOPs	% CPU performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-
Initial FPGA port	0.020	0.03%
Optimised top down for dataflow	0.28	0.43%
Optimise bottom up	27.78	42.26%
Ping-pong buffering	59.14	89.96%
Increase clock frequency to 400 Mhz	77.73	118%

Von-Neumann

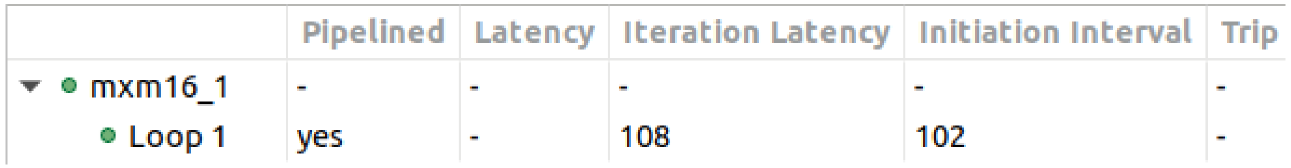
Approx. 4000 times difference in performance

Dataflow

For N=16, Runs performed on a Xilinx Alveo U280



	Negative Slack	BRAM	DSP	FF	LUT
ax_kernel	0.39	4	182	65210	42524
ax_e	0.39	0	173	62095	39963
local_grad3_t	0.39	0	78	28760	18725
mxm16_1	0.39	0	24	9080	5964
mxm16_4	0.39	0	24	9046	5194
mxm16_3	0.39	0	21	8645	5406
add2	0.01	0	3	1282	1069
local_grad3	0.39	0	72	28311	18239
mxm16	0.39	0	24	9495	5723
mxm16_2	0.39	0	21	9250	5657
mxm16_1	0.39	0	24	9080	5964



	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip
mxm16_1	-	-	-	-	-
Loop 1	yes	-	108	102	-

Detailed information available at https://arxiv.org/pdf/2011.04981.pdf

Bottom up optimisations

```
for (int i=0; i<N; i++) {
    double d=x*y;
    double j=d*z;
    double p=d*j
    result=p;
}
```

```
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
    double d=x*y;
    double j=d*z;
    double p=d*j
    result=p;
}
```

Loop pipelining

```
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
    double d=x*y;
    double j=d*z;
    double p=d*j
    result=p;
}
```

```
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
#pragma UNROLL FACTOR=4
    double d=x*y;
    double j=d*z;
    double p=d*j
    result=p;
}
```

Loop unrolling

```
double val=0;
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
    val=val+external[i];
}
```

Spatial dependency

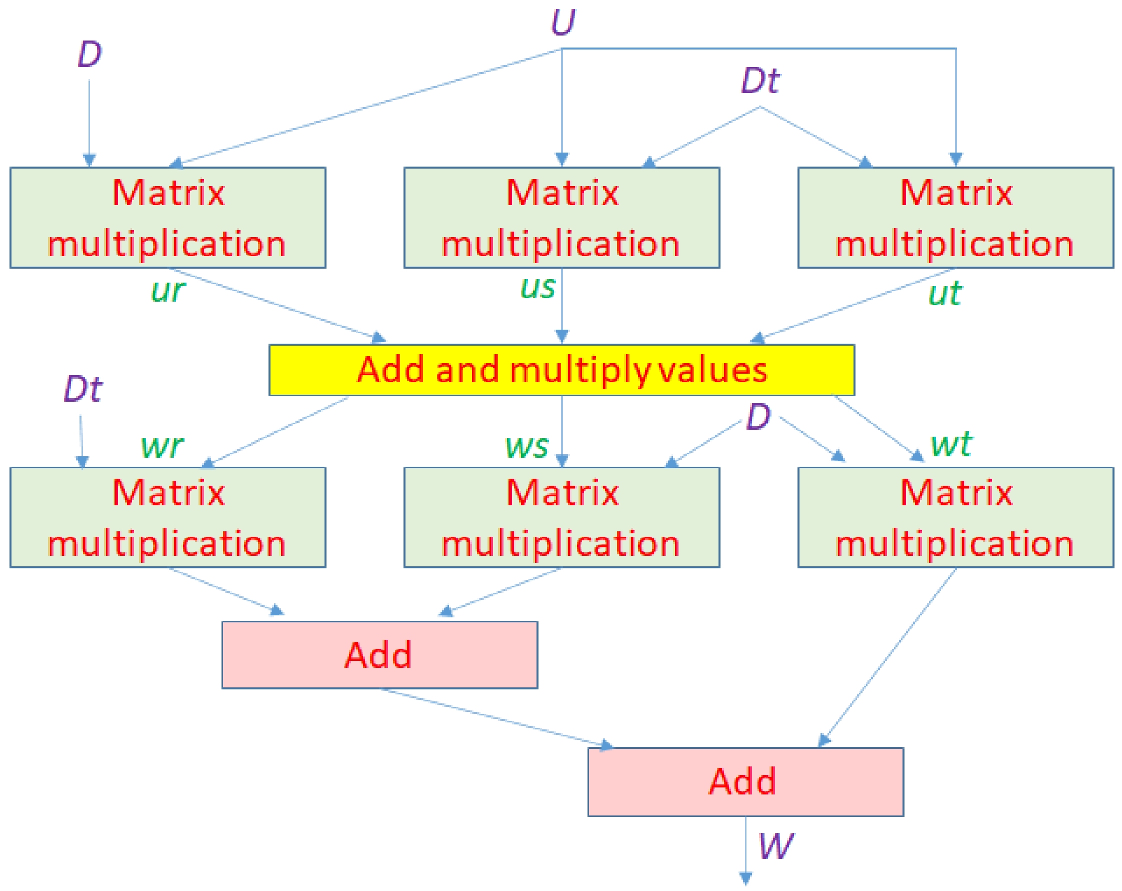
```
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
    double d=external_data[i];
    double j=external_data[i+1];
    ...
}
```

Conflict on external port to HBM2/DDR memory

```
double local_data[M];
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
    local_data[i-2]=a;
    local_data[i-1]=b;
    double v=local_data[i];
}
```

Conflict on (dual-ported) on-chip BRAM memory

Working top down: Adopting a dataflow design



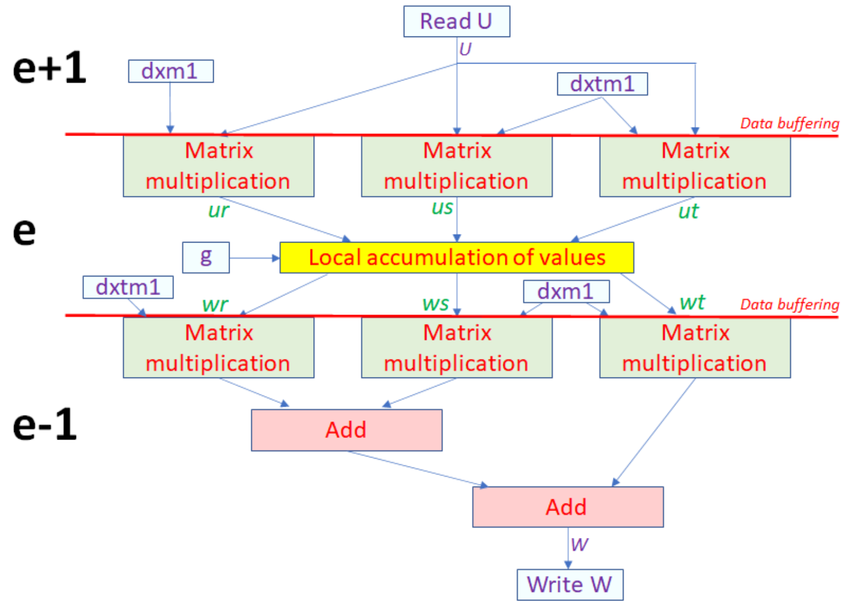
- Each stage is an independent function running concurrently and connected via streams
  - Idea is to keep each part continually fed with data and processing
- Golden rule: Keep the data flowing, each cycle generating a result

Description	Performance GFLOPs	% CPU performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-
Initial FPGA port	0.020	0.03%
Optimised top down for dataflow	0.28	0.43%
Optimise bottom up	27.78	42.26%
Ping-pong buffering	59.14	89.96%
Increase clock frequency to 400 Mhz	77.73	118%

Buffering data between stages

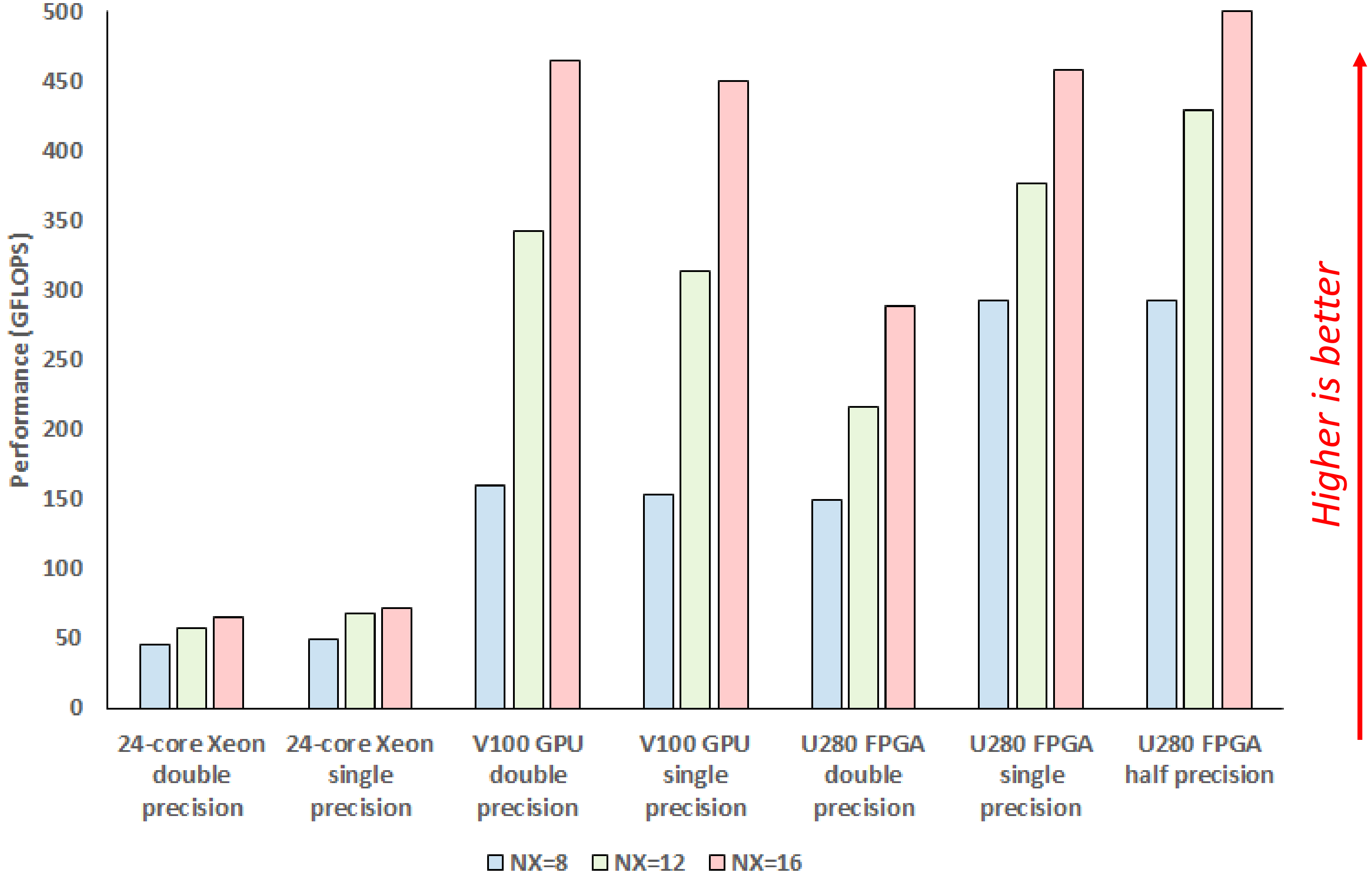
- Adopt ping-pong (double) buffering
  - Works in three phases, loading data for the next element, processing the first three MM for the current element, and the last three MM for the previous element
  - Keeps all parts running concurrently

- Also possible to accurately predict the realistic theoretical best performance our algorithm can deliver
  - Each MM is 31 FLOP/cycle and accumulation is 17/cycle, equals 203 FLOP/cycle. Multiply this by clock frequency for theoretical FLOPS
  - If not achieving this then not keeping data flowing!



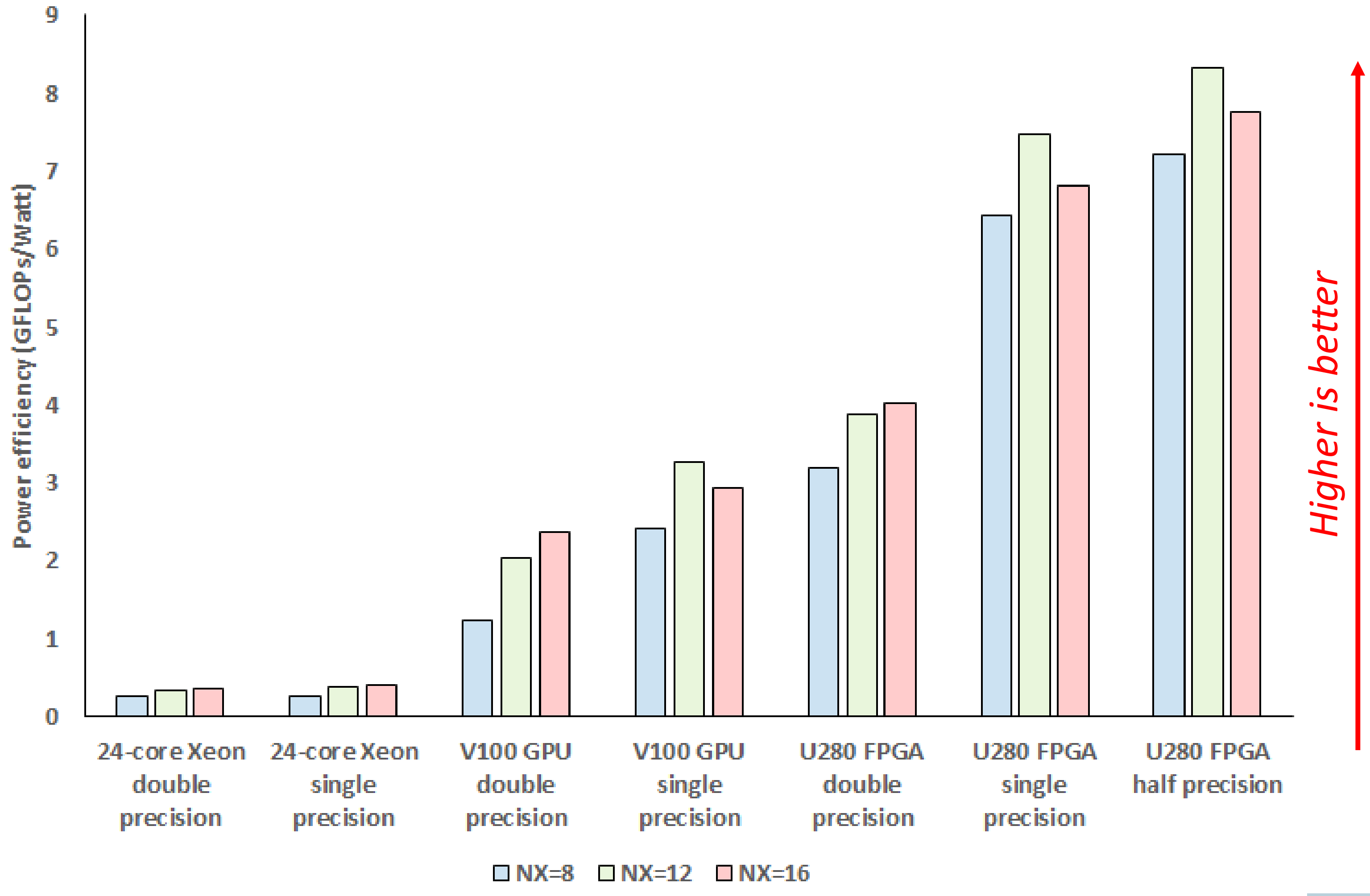
Description	Performance GFLOPs	% CPU performance	Theoretical performance	% Theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-	-
Initial FPGA port	0.020	0.03%	6.9 GFLOPs	0.29%
Optimised top down for dataflow	0.28	0.43%	6.9 GFLOPs	4.06%
Optimise bottom up	27.78	42.26%	61 GFLOPs	45.54%
Ping-pong buffering	59.14	89.96%	61 GFLOPs	96.95%
Increase clock frequency to 400 Mhz	77.73	118%	81.2 GFLOPs	95.73%

Performance against CPU and GPU



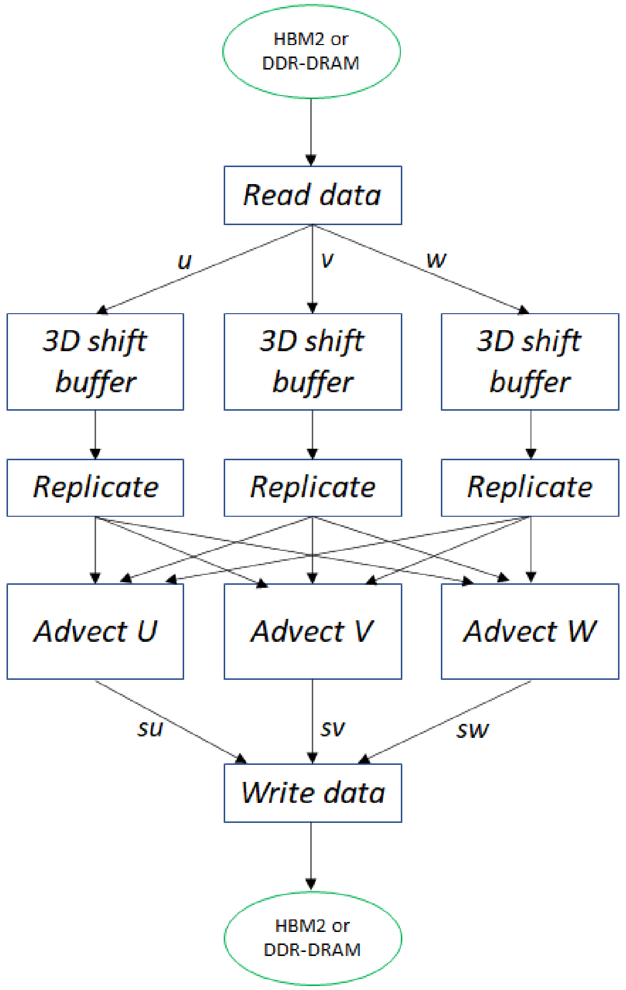
- FPGA runs on an Alveo U280
- CPU on 24-core Cascade Lake Xeon Platinum
- GPU on NVIDIA V100
- 4 FPGA kernels for double precision
  - Depends on exact problem size (NX)
- 7 FPGA kernels for single & half precision
  - Depends on exact problem size (NX)

Power efficiency against CPU and GPU

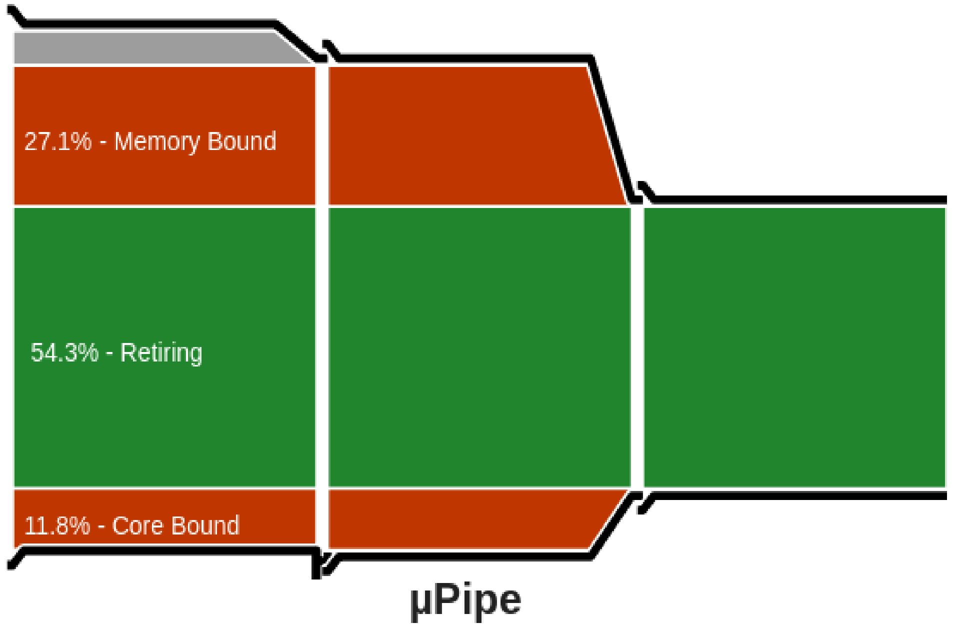


- FPGA runs on an Alveo U280
- CPU on 24-core Cascade Lake Xeon Platinum
- GPU on NVIDIA V100
- 4 FPGA kernels for double precision
  - Depends on exact problem size (NX)
- 7 FPGA kernels for single & half precision
  - Depends on exact problem size (NX)

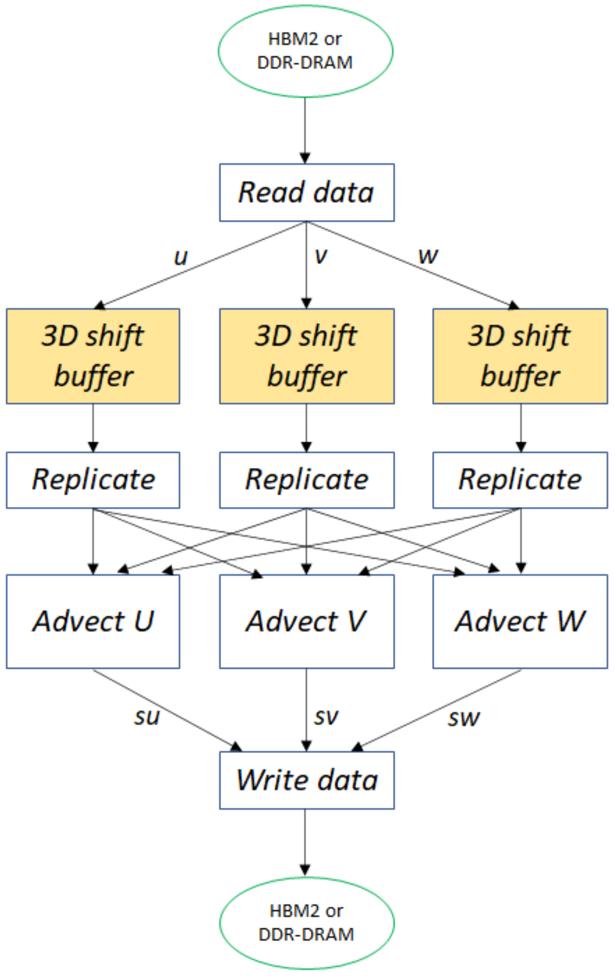
Another example: Atmospheric advection



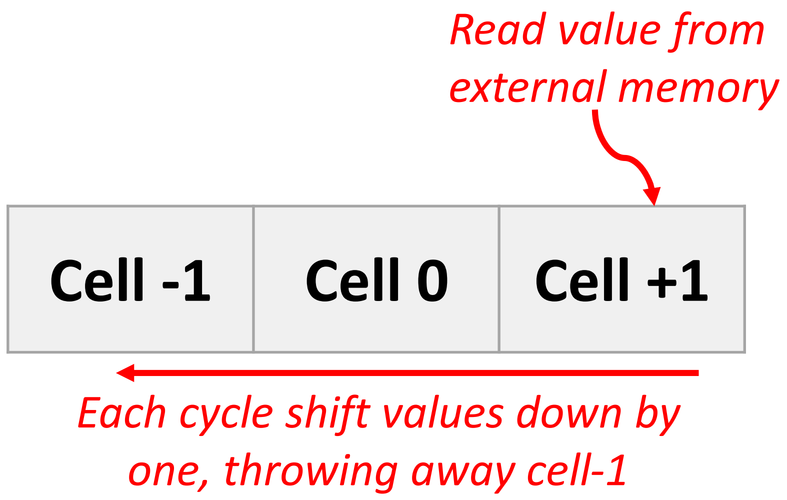
- Part of Met Office NERC Cloud (MONC) model
  - Accounts for around 40% of the model runtime
  - Stencil code working on three fields (U, V, W which is wind in x, y, z dimensions)
- Same ideas as previously
  - Kernel is fairly memory bound
  - Focussed on bottom up and top down dataflow algorithmic techniques
  - Running on both Xilinx Alveo and Intel Stratix-10



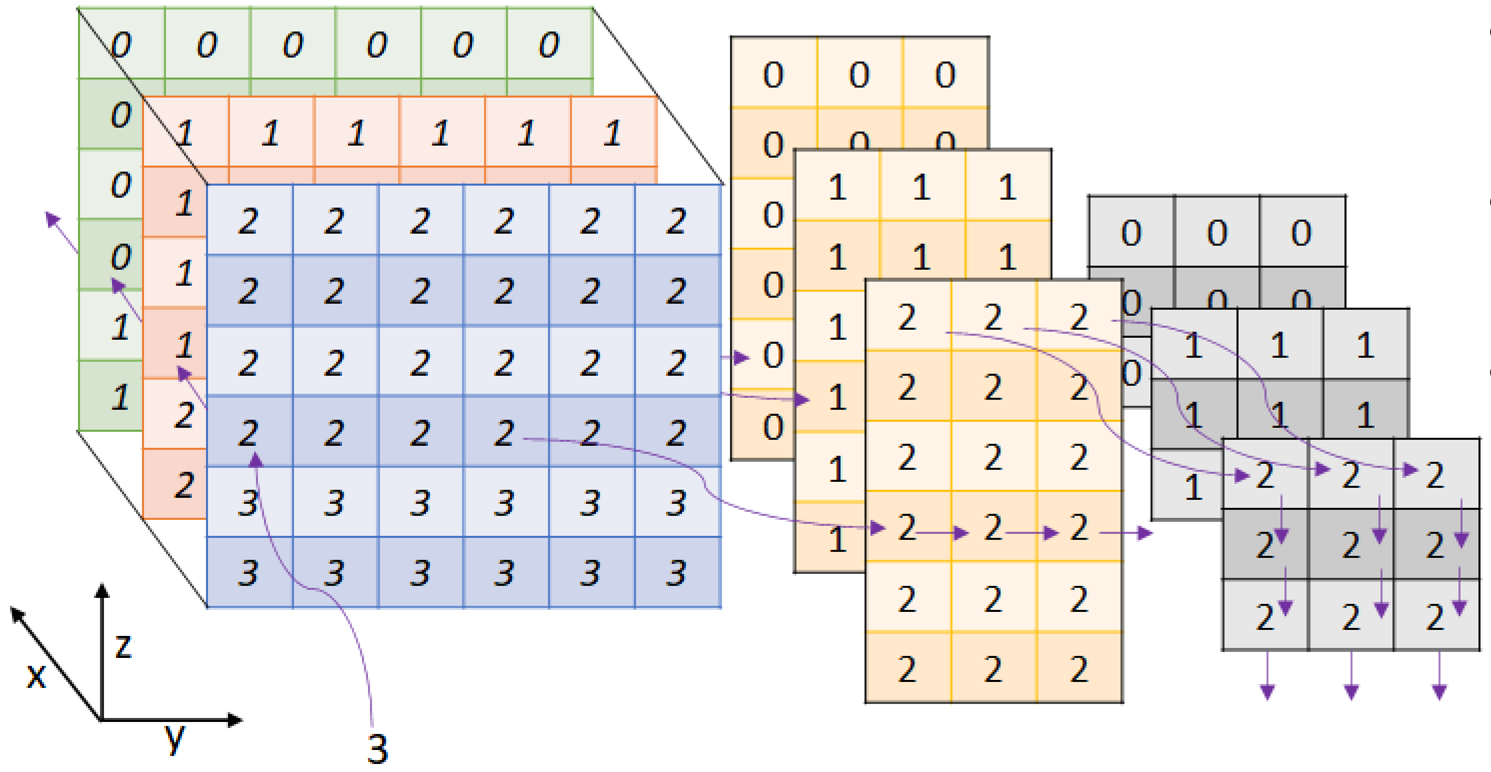
Tailoring caching of data via shift buffer



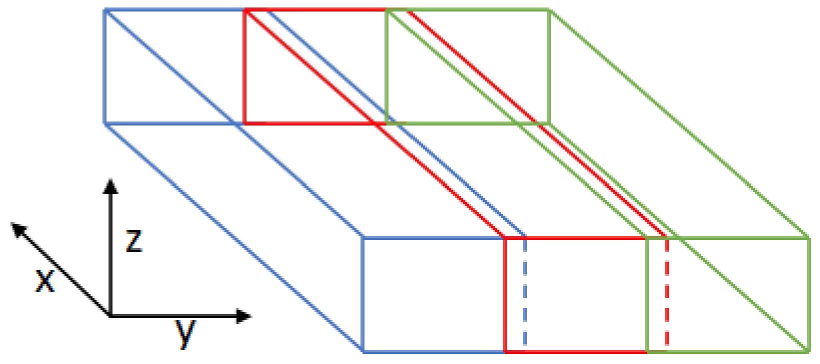
- 3D domain, where stencil computations require up to 27-points to calculate value for each grid cell
- Want to read only one new value from external memory for each field per cycle as otherwise get conflicts on the memory port
  - But need to provide 27 values per cycle to the advect routine in order to achieve a result for each field for each clock cycle
- Use a shift buffer - 1D example


Read value from external memory
Cell -1
Cell 0
Cell +1
Each cycle shift values down by one, throwing away cell-1

Tailoring caching of data via shift buffer



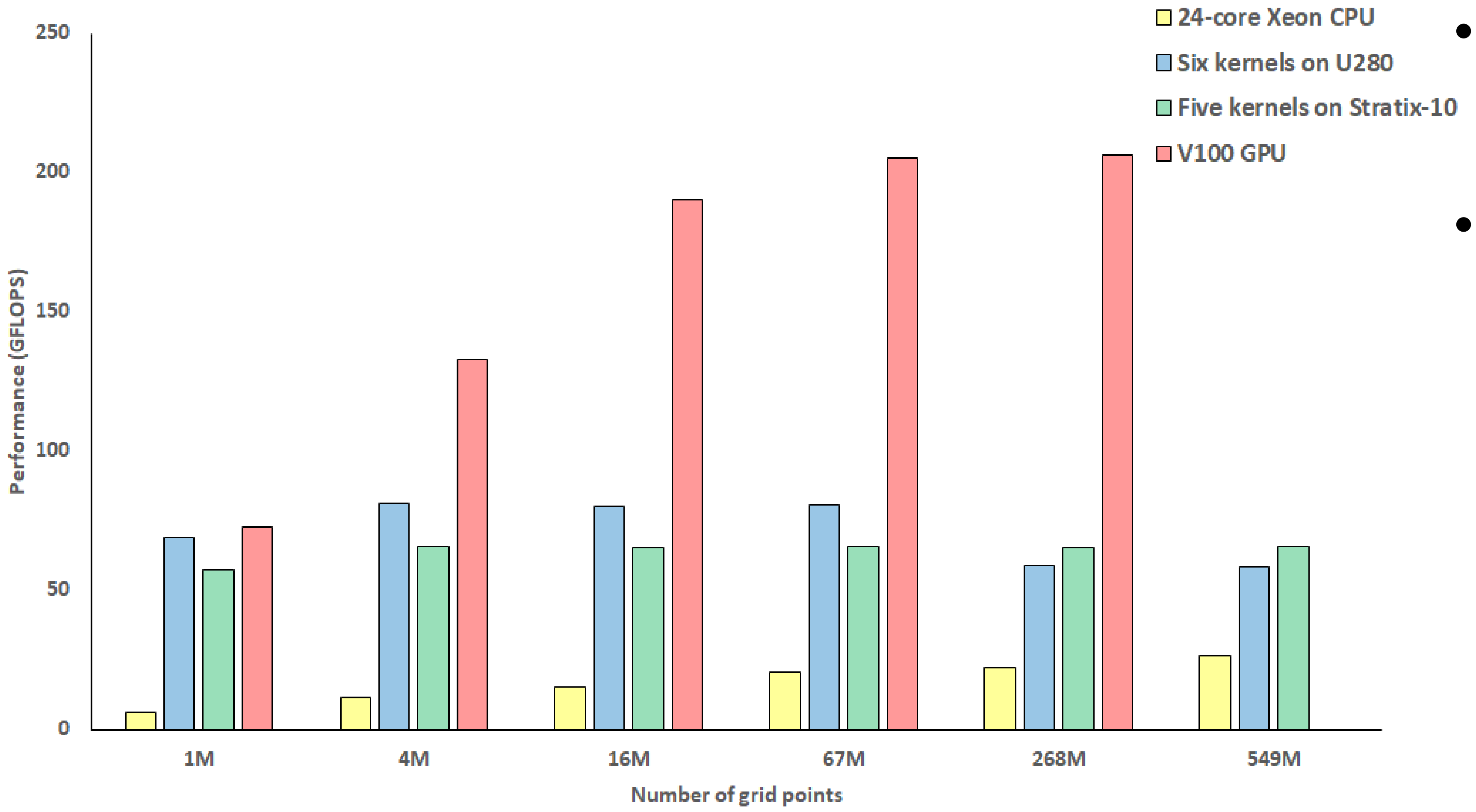
- Have these windows running across the 3D domain
- Generates 27-point stencil each cycle
- Memory on FPGA limits size in Y dimension so work in chunks



- The golden rule of keeping the data flowing and generating a result per cycle necessitated this shift buffer, which then impacted how the code is running and having to chunk in Y

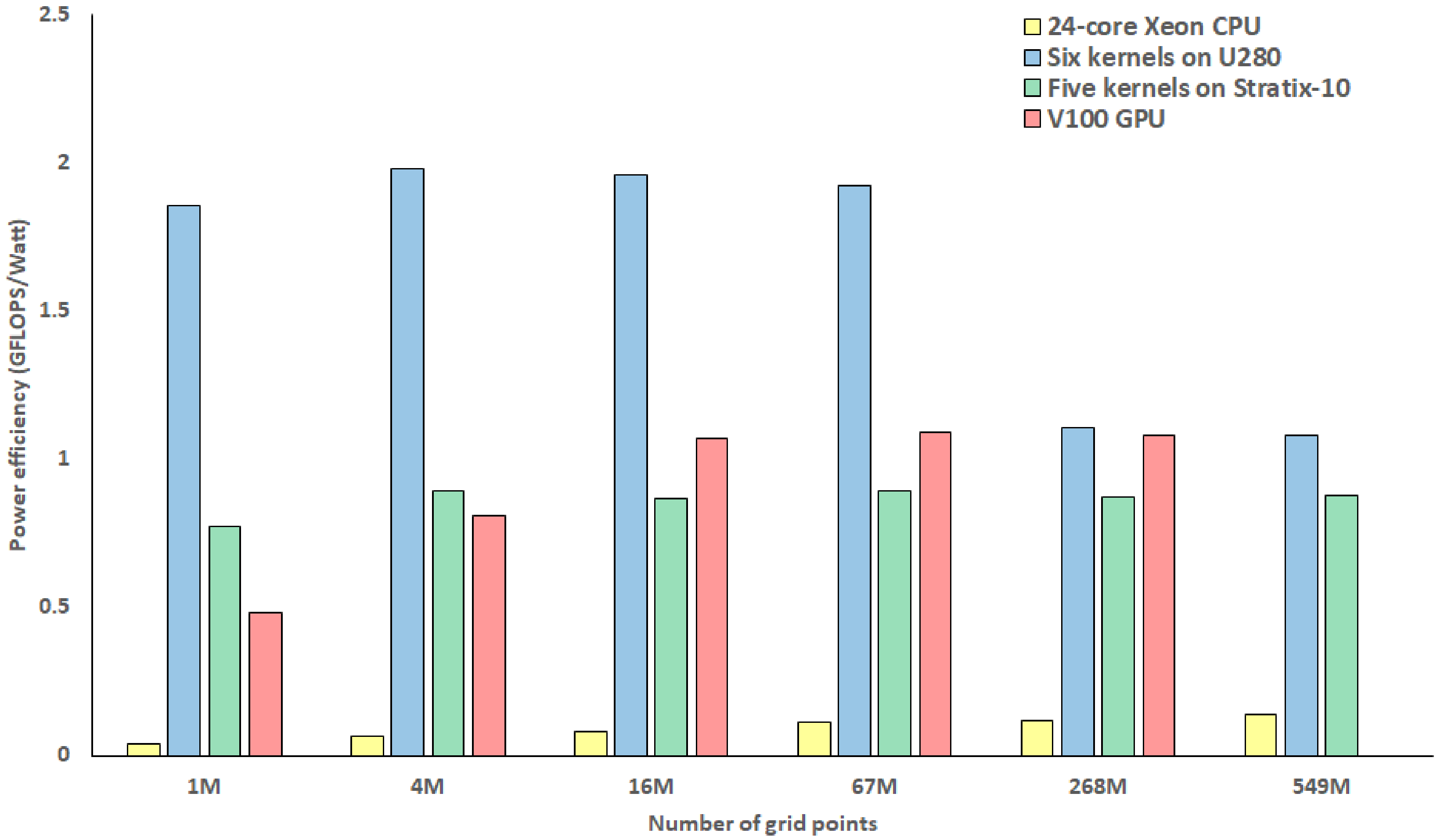
Detailed information available at https://arxiv.org/pdf/2107.13500.pdf

Performance comparison



- FPGAs outperform the CPU by a long way, but GPU is a tough test!
- The Xilinx Alveo U280 tends to outperform the Intel Stratix 10
  - 6 kernels on Alveo vs 5 on the Stratix 10
  - 5 kernels on Stratix 10 are running at 250 MHz, whereas Alveo at 300MHz
  - At largest problem sizes Alveo must use DDR-DRAM rather than 8GB HBM2

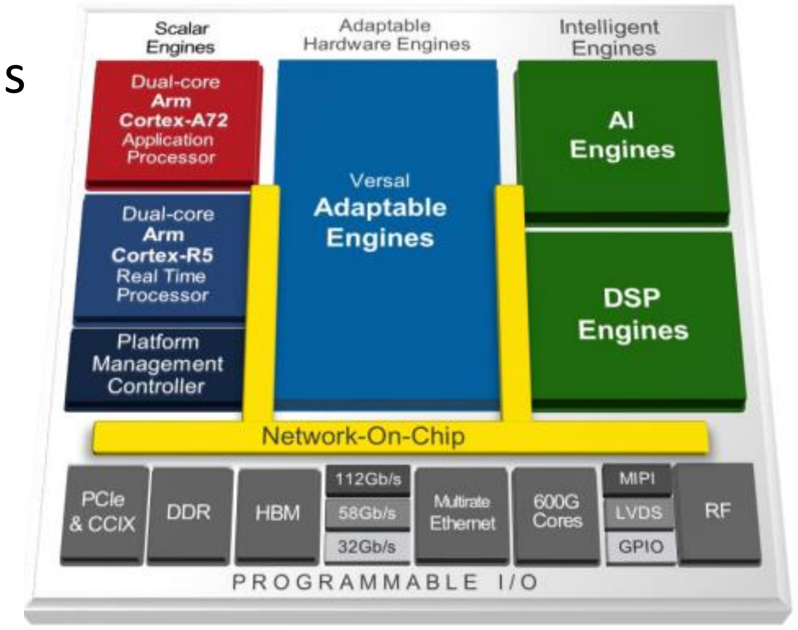
Power efficiency

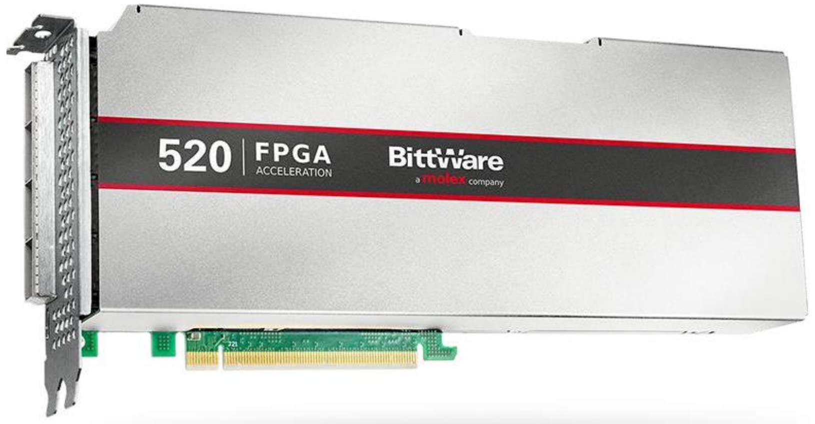


- Alveo U280 has excellent power efficiency until it has to switch from HBM2 to DDR-DRAM
  - Combination of reduced performance and (slightly) increased power draw
- Higher power draw of Stratix 10 means it is competitive against the GPU, especially for smaller problem sizes

Have a play with FPGAs for your code!

- In EPCC we host the ExCALIBUR H&ES FPGA testbed
  - Give HPC code developers access to FPGAs for their workloads
  - Provides a range of FPGAs to see what works best
  - All tooling preinstalled and provide resource for code development (e.g. building and emulation)
  - In collaboration with UCL and Warwick who are developing enabling software

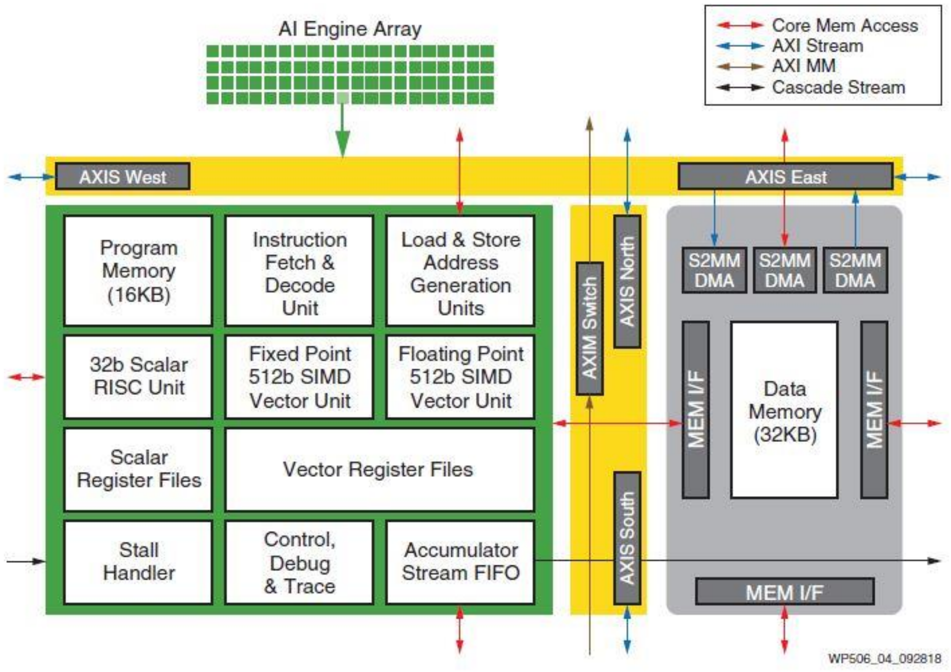




Access is free, visit https://fpga.epcc.ed.ac.uk for more details

Summary

- It's an exciting time in HPC, and FPGAs have a potential role to play
  - But important to pick your battles and focus on solving code level challenges suited to FPGAs, they certainly won't replace GPUs or CPUs!
- FPGAs have become far more capable and next-generation technologies are very exciting
  - Programming FPGAs has become significantly more productive, but need to rethink our algorithms from the perspective of dataflow to achieve good performance
    - Technologies like SYCL on the horizon which look to be very interesting
  - Software experts have a significant contribution to make here around algorithmic techniques and successes stories at the application level.


AI Engine Array
Core Mem Access
AXI Stream
AXI MM
Cascade Stream
AXIS West
AXIS East
Program Memory (16KB)
Instruction Fetch & Decode Unit
Load & Store Address Generation Units
32b Scalar RISC Unit
Fixed Point 512b SIMD Vector Unit
Floating Point 512b SIMD Vector Unit
Scalar Register Files
Vector Register Files
Stall Handler
Control, Debug & Trace
Accumulator Stream FIFO
AXIM Switch
AXIS North
AXIS South
S2MM DMA
S2MM DMA
S2MM DMA
MEM I/F
Data Memory (32KB)
MEM I/F
MEM I/F
WP506_04_092818