EXCELLERAT

Exploring the acceleration of Nekbone on reconfigurable architectures

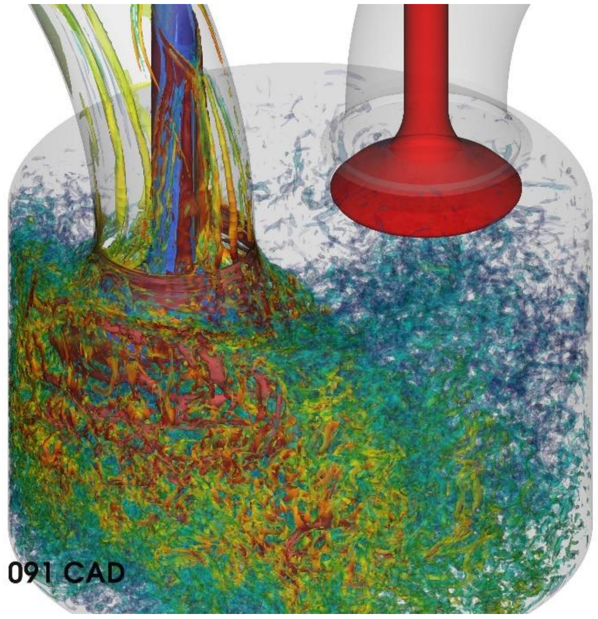
Nick Brown,

EPCC at the University of Edinburgh

Background

- We are interested in the role of FPGAs in future exa-scale machines to provide high performance and power efficiency
  - In the EXCELLERAT CoE this is mainly focussed on engineering codes

- Nekbone is a mini-app that captures the basic structure of Nek5000
  - Solves a standard Poisson equation using a Conjugate Gradient (CG) iterative method with a simple preconditioner
  - A useful tool for exploring the algorithmic elements that are pertinent to Nek5000, and many other HPC codes
  - Has been used extensively on CPUs and GPUs, so can FPGAs can provide any performance/power efficiency benefits?



Where our focus is: The AX kernel

EXCELLERAT

```
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
```

Iterate over elements

```
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
```

Multiply and add values calculated in local_grad3

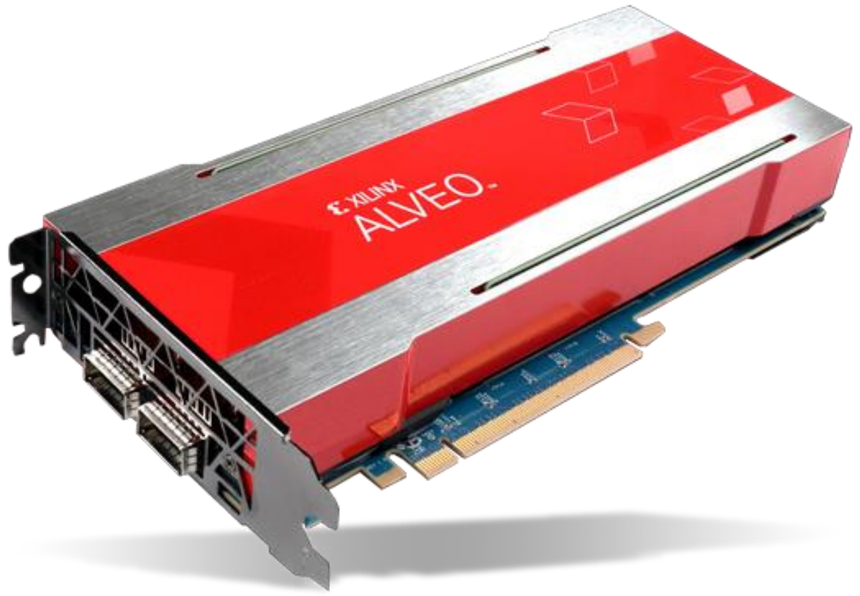
```
subroutine local_grad3(ur, us, ut, u, n, dxm1, dxm2)
  integer, intent(in) :: n
  real(n,n,n), intent(in) :: u, dxm1, dxm2
  real(n,n,n), intent(out) ::ur, us, ut

  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

- This AX kernel of the CG solver accounts for around 75% of the overall runtime of Nekbone
- Our experiments utilise 800 elements, and N=16 which means 4096 grid points per element
  - There are 831488 double precision floating point operations per element
- Some challenges on the CPU
  - 35% of L1, and 10% of L2, cache reads missed for this kernel
  - Runs out of memory BW as we scale the CPU cores

Key question: If we port this to FPGAs and move to a dataflow algorithm relying on streaming data, can we ameliorate such memory overhead?

Experimental set-up



- All FPGA runs done on a Xilinx Alveo U280
  - 1.08 million LUTs, 4.5MB of on-chip BRAM, 30MB of on-chip URAM, 9024 DSP slices, 8GB HBM2
- We use Xilinx's Vitis 2020.1 throughout, writing our code in C++
  - From the view point of HPC software developers exploring the role of FPGAs to accelerate their codes
- All Nekbone runs use 800 elements, and polynomial order (N) of 16

- For comparison, CPU runs performed on a 24 core Intel Xeon Platinum Cascade Lake (8260M), and unless otherwise stated all cores were used.
- GPU runs (a little later in the paper) were done on a NVIDIA V100 GPU using CUDA

Overview of single kernel performance

Description	Performance GFLOPs	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Initial FPGA port	0.020	0.03%	0.29%
Optimised for dataflow	0.28	0.43%	4.06%
Optimised memory access	0.42	0.63%	6.09%
Optimise matrix multiplications	12.72	19.35%	20.85%
Ping-pong buffering	27.78	42.26%	45.54%
Remove pipeline stalls	59.14	89.96%	96.95%
Increase clock frequency to 400 Mhz	77.73	118%	95.73%

Von-Neumann based algorithm

Approx. 4000 times difference in performance

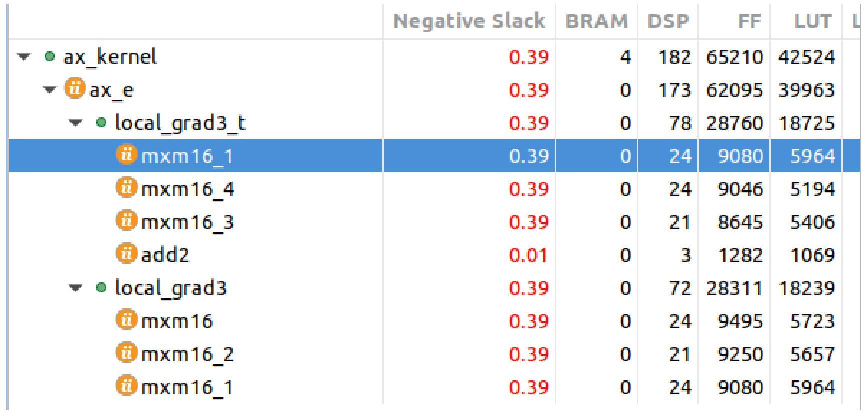
Optimised dataflow based algorithm

The first step….

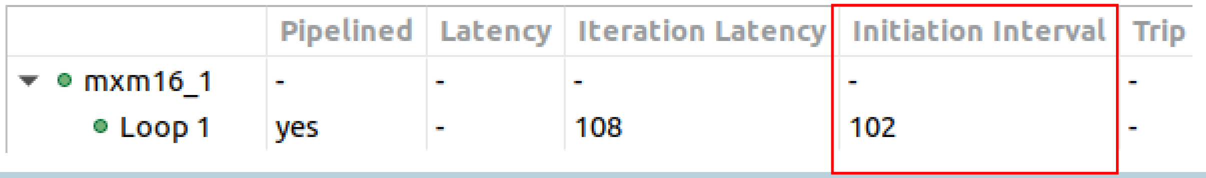
```
void ax_kernel(double * w, double * u, double * gxyz, double * dxm1, double * dxtm1, double * ur, double * us,
    double * ut, double * wk, int nx1, int ny1, int nz1, int nelt, int ldim) {
  #pragma HLS INTERFACE m_axi port=w offset=slave
  #pragma HLS INTERFACE m_axi port=u offset=slave
  #pragma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

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

- The initial version simply used pragmas to decorate arguments as ports
- On host side hooked it up via OpenCL



	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

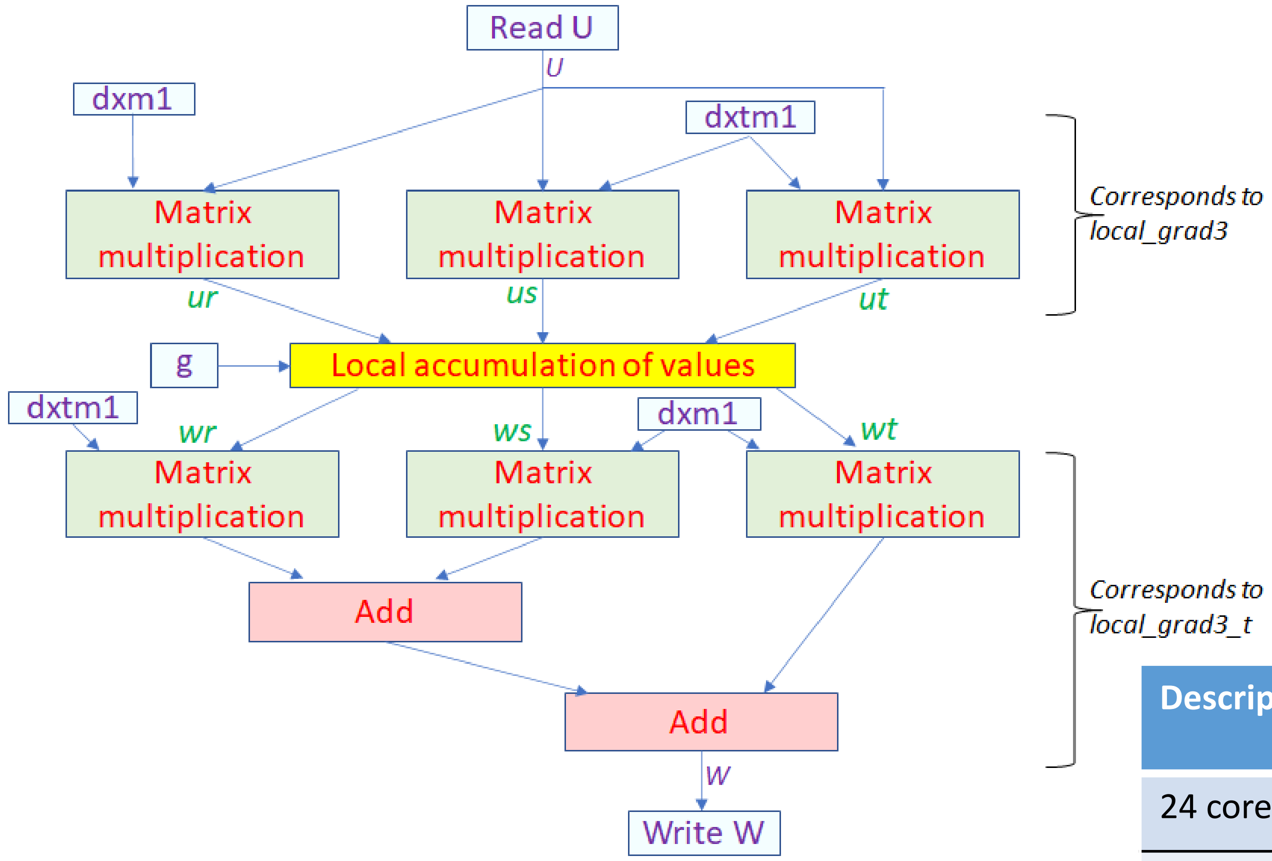


	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip
mxm16_1	-	-	-	-	-
Loop 1	yes	-	108	102	-

Description	Performance GFLOPs	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Initial version	0.020	0.03%	0.29%

Initial version around 3287 times slower than the CPU – Thing can only get better!

Redesigning the algorithm for dataflow



For each element e in nelt, execute this dataflow, with grid points of U, D and Dt as input, generating result grid points of W. All stages connected via HLS streams and (ideally) running concurrently.

```
double a_temp[NX], c_temp[NX*NX][NX];
#pragma HLS array_partition variable=a_temp complete

for (int k = 0; k < NX; k++) {
  for (int j=0;j<NX*NX;j++) {
    double b_val=b.read();
    for (int i=0;i<NX;i++) {
      if (k==0) c_temp[j][i]=0.0;
      if (j==0) a_temp[i]=a.read();
      c_temp[j][i]+=a_temp[i] * b_val;
      if (k==NX-1) c.write(c_temp[j][i]);
    }
  }
}
```

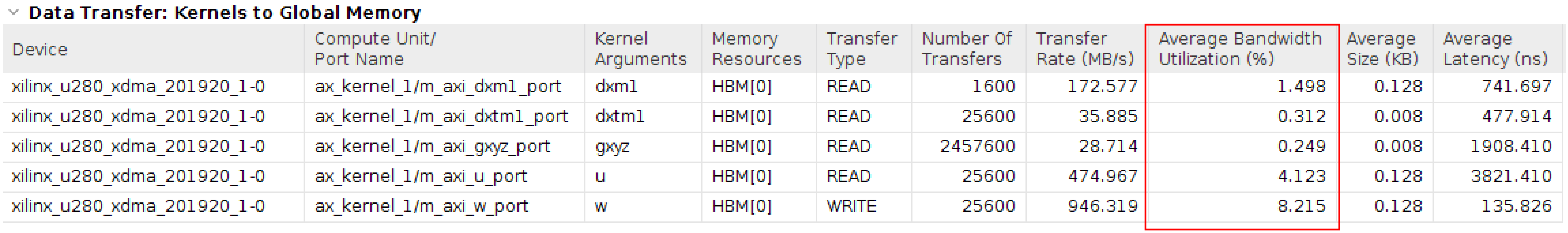
The MM algorithm from Vitis open source BLAS library

Description	Performance GFLOPs	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Optimised for dataflow	0.28	0.43%	4.06%

Over ten times faster than our initial version, but performance still sucks!

Getting smart on data transfer

- Profiled via Vitis analyser to understand where the bottlenecks might be

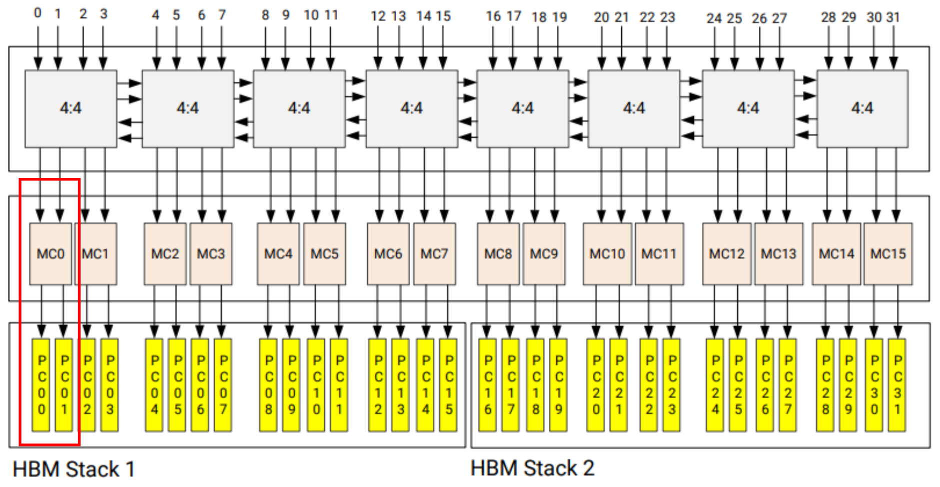


Data Transfer: Kernels to Global Memory

Device	Compute Unit/ Port Name	Kernel Arguments	Memory Resources	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Latency (ns)
xilinx_u280_xdma_201920_1-0	ax_kernel_1/m_axi_dxm1_port	dxm1	HBM[0]	READ	1600	172.577	1.498	0.128	741.697
xilinx_u280_xdma_201920_1-0	ax_kernel_1/m_axi_dxtm1_port	dxtm1	HBM[0]	READ	25600	35.885	0.312	0.008	477.914
xilinx_u280_xdma_201920_1-0	ax_kernel_1/m_axi_gxyz_port	gxyz	HBM[0]	READ	2457600	28.714	0.249	0.008	1908.410
xilinx_u280_xdma_201920_1-0	ax_kernel_1/m_axi_u_port	u	HBM[0]	READ	25600	474.967	4.123	0.128	3821.410
xilinx_u280_xdma_201920_1-0	ax_kernel_1/m_axi_w_port	w	HBM[0]	WRITE	25600	946.319	8.215	0.128	135.826

- Data transfer between the on-device HBM2 and kernel is terrible!
  - Aggregate BW of 952 MB/s, whereas the HW specification says we could expect a maximum of 460 GB/s
  - Lots of individual small transfers too

Getting smart on data transfer



```
struct packaged_double {
  double data[8];
};

void ax_kernel(struct packaged_double * w, ......) {
  #pragma HLS INTERFACE m_axi port=w offset=slave bundle=w_port
  #pragma HLS data_pack variable=w
  ......
}
```

- 8GB of HBM is split up into 32 banks of 256MB
  - 16 memory controllers, each with a channel connecting two banks.
  - By default, all memory in bank 0
- We made each argument an explicit, separate, AXI4 port and then configured Vitis to place each input or output argument in different HBM banks (ideally with different memory controllers too!)
- HBM memory controllers optimised for 256- or 512-bit accesses
  - As we are double precision, all our accesses were 64 bits so combined these into 512-bit width structures

Description	Performance GFLOPs	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Optimised memory access	0.42	0.63%	6.09%

Doubled our performance. Memory B/W now on average 95% for accesses, so worth doing but not a silver bullet!

Improving the MM algorithm

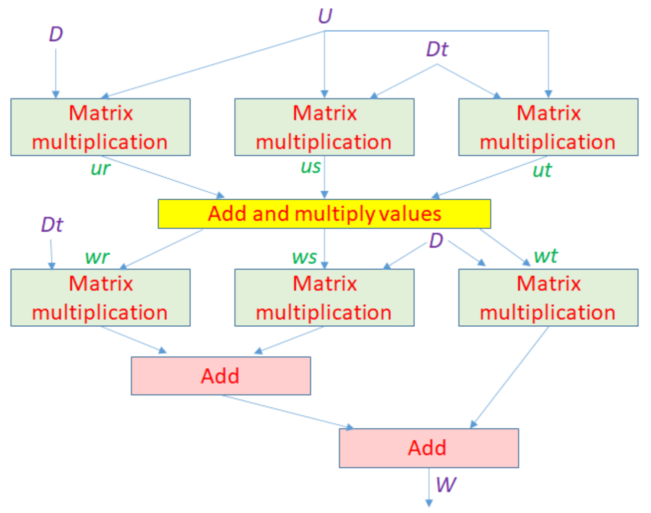
```
double a_temp[NX], c_temp[NX*NX][NX];
#pragma HLS array_partition variable=a_temp complete

for (int k = 0; k < NX; k++) {
  for (int j=0;j<NX*NX;j++) {
    double b_val=b.read();
    for (int i=0;i<NX;i++) {
      if (k==0) c_temp[j][i]=0.0;
      if (j==0) a_temp[i]=a.read();
      c_temp[j][i]+=a_temp[i] * b_val;
      if (k==NX-1) c.write(c_temp[j][i]);
    }
  }
}
```

Only generates results on the last iteration of k

```
double a_temp[NX][NX], b_temp[NX];
#pragma HLS array_partition variable=a_temp dim=1 complete

for (int j=0;j<NX*NX;j++) {
  // Load b values that are needed by the inner loop
  struct packaged_double in_data=b.read();
  for (int q=0;q<8;q++) b_temp[q]=in_data.data[q];
  in_data=b.read();
  for (int q=0;q<8;q++) b_temp[q+8]=in_data.data[q];

  for (int i=0;i<NX;i++) {
    if (j==0) a_temp[0][i]=a[0].read();
    double temp_0=a_temp[0][i] * b_temp[0];

    if (j==0) a_temp[1][i]=a[1].read();
    double temp_1=a_temp[1][i] * b_temp[1];

    ....
    c.write(C_temp_0 + C_temp_1);
  }
}
```

Generates immediately (or as soon as pipeline is filled anyway)

- Only generated a result on the last iteration of k
  - Subsequent pipeline stages stalling on this.
  - Algorithmic issues limiting what parts can run concurrently

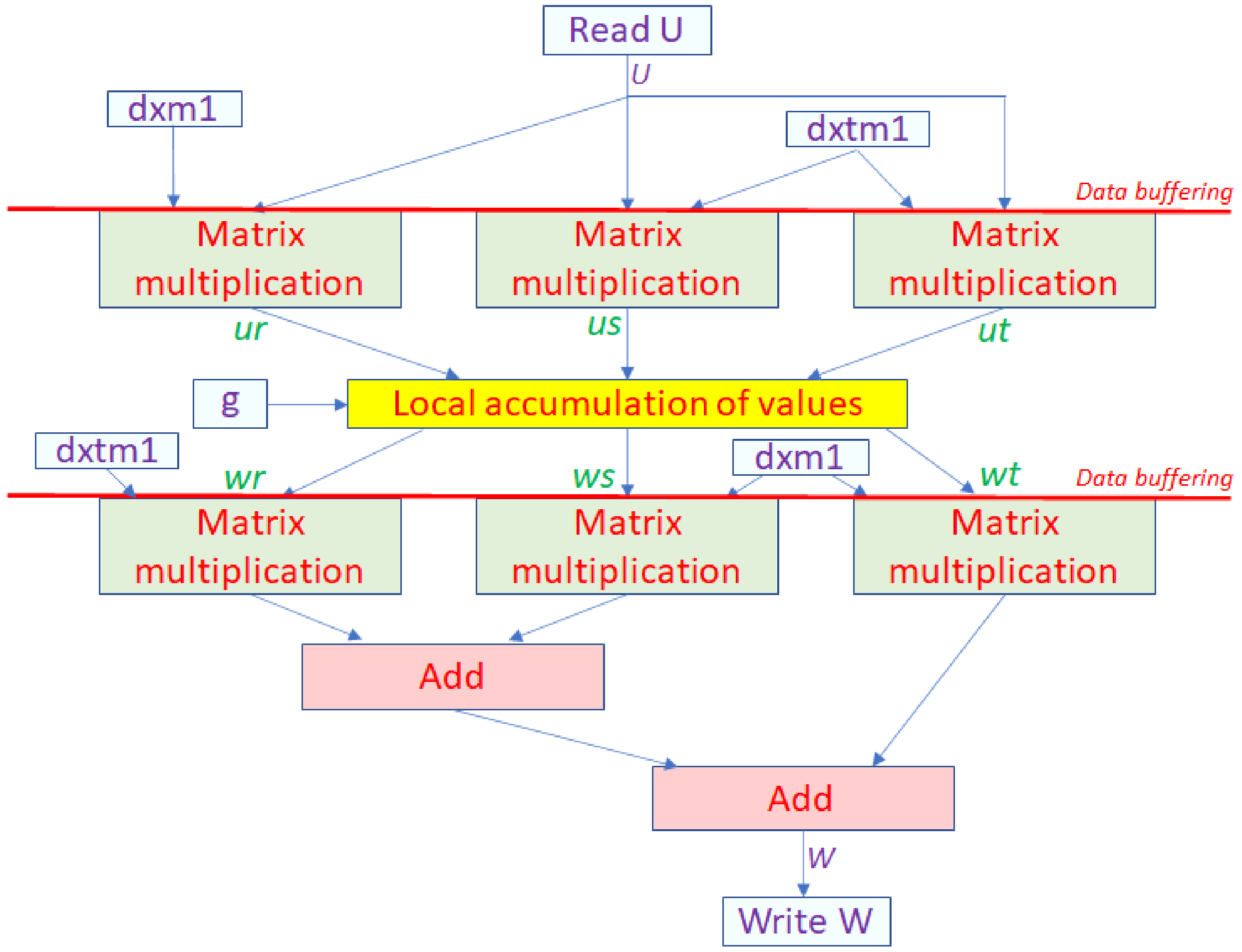


- By refactoring reduced this delay to 45 cycles (the depth of the pipeline) & significantly more DP ops running concurrently

Description	Performance GFLOPS	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Optimise matrix multiplications	12.72	19.35%	20.85%

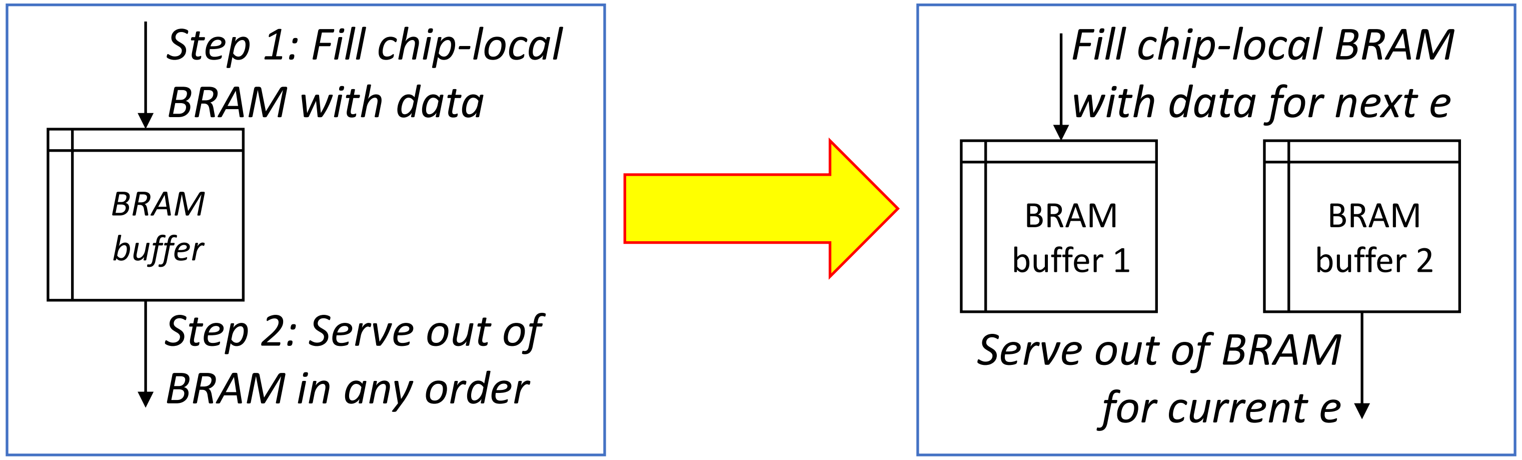
Increases performance of previous version by around 30 times. Theoretical performance increased from 6.9 to 61 GFLOPS

Ping pong buffering data between stages

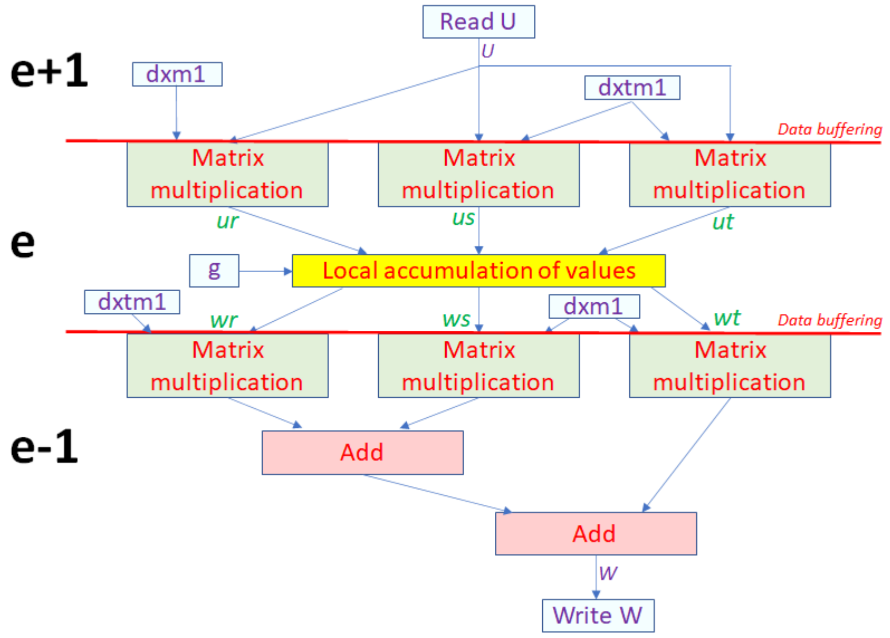


- Our current design is limited
  - Each MM requires U in a different order
    - This is also the case for D and Dt too
    - Also data for wr, ws, wt needs to be reordered
  - Each MM is associated with a buffer of grid points for that element.
    - Once full, data is then served from the buffers into their respective MM in the specific order required.
    - Causes three implicit phases of operation, with only one active at any one time

Ping pong data between stages


Step 1: Fill chip-local BRAM with data
BRAM buffer
Step 2: Serve out of BRAM in any order
Fill chip-local BRAM with data for next e
BRAM buffer 1
BRAM buffer 2
Serve out of BRAM for current e

- Initially did this explicitly in the code with buffers
  - But this resulted in high resource usage so moved to HLS's ping pong buffers (PIPO) with an inner dataflow region

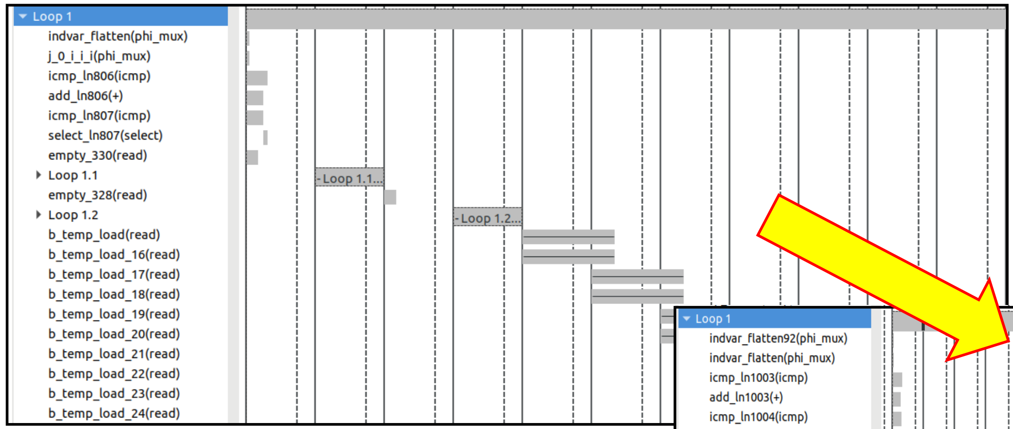


Description	Performance GFLOPS	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Ping pong buffering	27.78	42.26%	45.54%

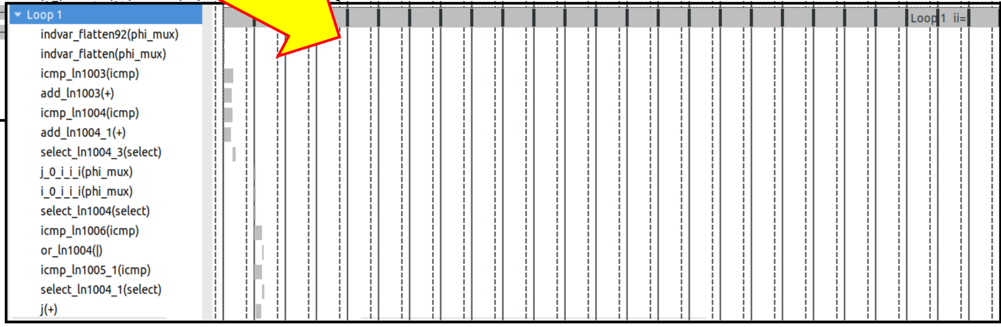
Increased the performance of our kernel by over two times, but still less than half the performance of either the CPU or our theoretical performance

Removing pipeline stalls

```
double a_temp[NX][NX], b_temp[NX];
#pragma HLS array_partition variable=a_temp dim=1 complete

for (int j=0;j<NX*NX;j++) {
  // Load b values that are needed by the inner loop
  struct packaged_double in_data=b.read();
  for (int q=0;q<8;q++) b_temp[q]=in_data.data[q];
  in_data=b.read();
  for (int q=0;q<8;q++) b_temp[q+8]=in_data.data[q];

  for (int i=0;i<NX;i++) {
    if (j==0) a_temp[0][i]=a[0].read();
    double temp_0=a_temp[0][i] * b_temp[0];

    if (j==0) a_temp[1][i]=a[1].read();
    double temp_1=a_temp[1][i] * b_temp[1];

    ....
    c.write(C_temp_0 + C_temp_1);
  }
}
```



- Brought the reading of the b stream into the inner loop
- For our problem size meant going from 204800 batches of 16 cycles (having to drain between each batch), to 1 batch of 3276800 cycles



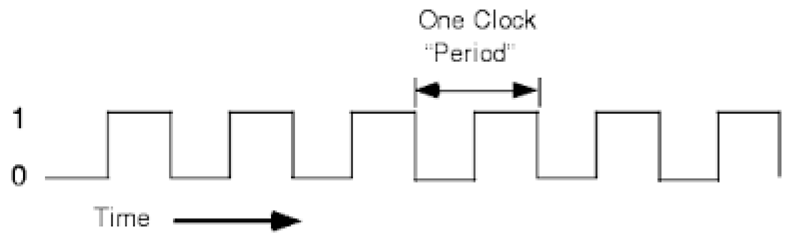
- Dependency between loading *b_temp* and reading it
  - Our inner loop was being pipelined nicely, but was filling and draining for every inner iteration (n1) rather than nelt*n3*n1
  - With a pipeline depth of 45 cycles, this was expensive

Description	Performance GFLOPS	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Remove pipeline stalls	59.14	89.96%	96.95%

Achieving around 90% the performance of the 24 core Xeon CPU. The theoretical performance of our HLS kernel was 61 GFLOPS, of which we were achieving almost 97%.

Upping the clock frequency

- The theoretical performance of our kernel is 61 GFLOPS and the 24 core CPU is achieving around 66 GFLOPS
  - So focussed on the kernel itself, in order to increase performance and potentially beat the CPU we need to increase the theoretical performance



- The default clock on the Alveo U280 is 300Mhz
  - This can be increased via a simple configuration change
  - But increasing the clock frequency impacts the overall complexity of the kernel, for instance by increasing to 400Mhz the depth of our matrix multiplication pipeline increased to 61 cycles.
- We found empirically that 400Mhz was the optimal clock frequency
  - Beyond this the complexity of the matrix multiplications increased very significantly, with the pipeline II increased to two.
  - It was possible to reduce this back down to one by using the *bind_op* Vitis HLS pragma to increase the latency of the double precision floating point cores, but the performance we obtained by doing so never matched that of 400Mhz.

Description	Performance GFLOPS	% CPU performance	% theoretical performance
24 cores of Xeon (Cascade Lake) CPU	65.74	-	-
Increase clock frequency to 400 Mhz	77.73	118%	95.73%

For the first time, with a single kernel beating the 24-core Xeon Platinum CPU

Scaling to multiple kernels

- Now we had a good performing FPGA kernel, let’s see what we can get when we scale it up!
  - Also comparing power efficiency, not only against CPU but also a V100 GPU

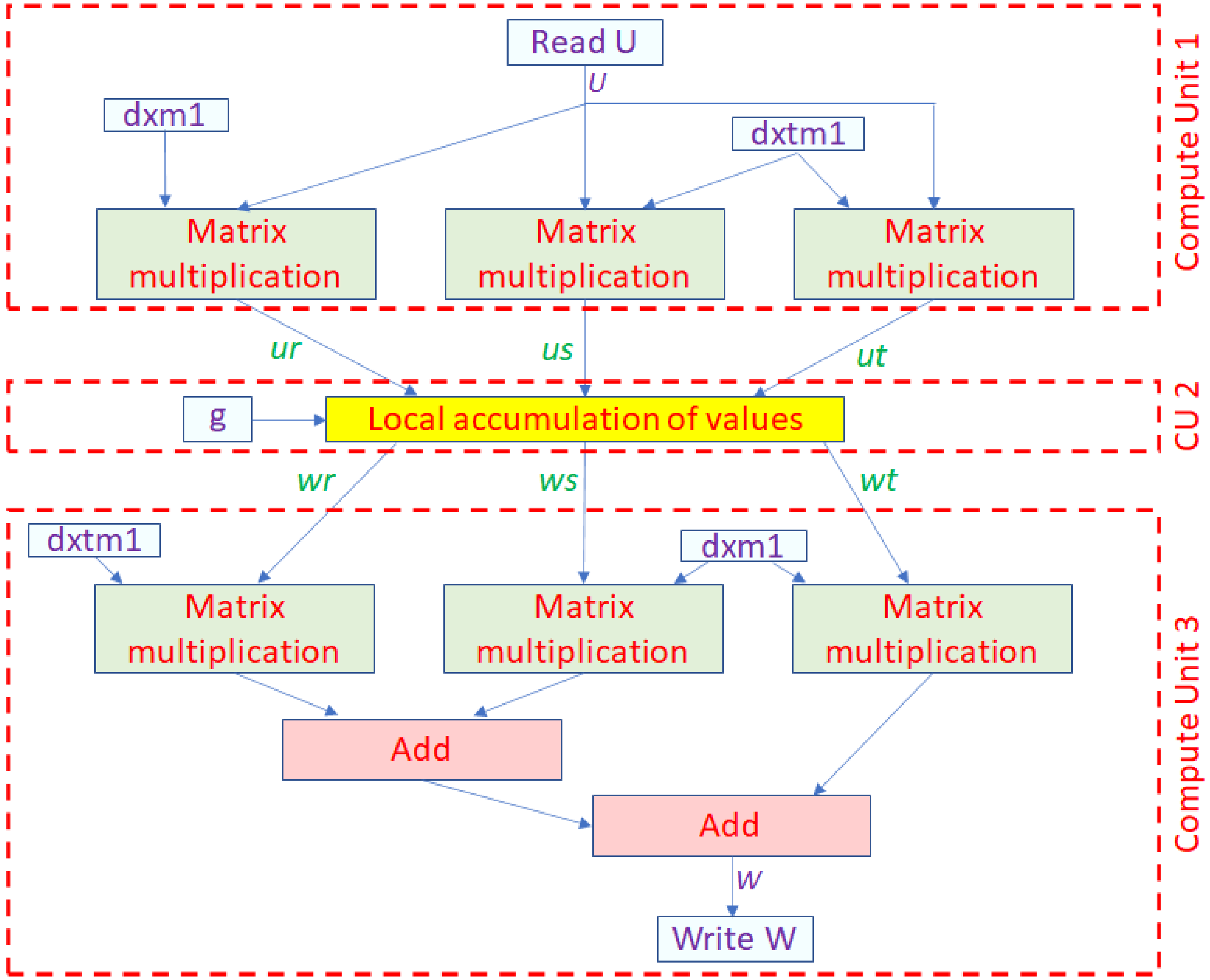
Chip Component	SLR usage default	SLR usage with explicit memory placement
DSP slices	34%	34%
LUTs	28%	36%
Flip Flops	24%	29%
BRAM	115%	32%
URAM	0%	30%

The Alveo U280 has three Super Logic Regions (SLRs)

- Used Vitis HLS’s *bind_storage* pragma to direct explicit memory placement
  - FIFO queues associated with the HLS streams, and arrays associated with the data-reordering ping pong buffers into LUTRAM
  - Data storage associated with each matrix multiplication dataflow region into the on-chip Ultra-RAM (URAM)
  - Was rather time consuming to figure out the best placement of different data regions and their associated resource usage.

Splitting kernels up into Compute Units

- We initially found that, as we scaled kernels up the performance was surprisingly poor
  - Vitis/Vivado was dynamically down clocking our kernels to meet timing
  - Fixed it by splitting a single kernel up into three CUs connected via AXI streams
- Initially this resulted in routing errors due to congestion in the matrix multiplication of the first CU.
  - Using the *Congestion_SpreadLogic_high* implementation strategy fixed the issue but resulted in poor performance.
  - Found was due to naming conflicts between the first and third CUs. Specifically, the names of the MM functions were the same in each CU, and place and route was attempting to perform some optimisation by consolidating these together
    - Fixed by giving functions unique names between the CUs



Performance and power comparison

Description	Performance (GFLOPS)	Power usage (Watts)	Power efficiency (GFLOPS/Watt)
1 CPU core	5.38	65.16	0.08
24 CPU cores	65.74	176.65	0.37
V100 GPU	407.62	173.63	2.34
1 FPGA kernel	74.29	45.61	1.63
2 FPGA kernels	146.94	52.47	2.80
4 FPGA kernels	289.02	71.98	4.02

- Four kernels achieve over four times the performance of the CPU, and 71% the performance of the V100 GPU
- On average, adding an extra FPGA kernel requires approximately an additional 7 Watts, with a performance increase close to 74 GFLOPS per kernel
- 4 kernels on FPGA is almost twice as power efficient as the GPU

- We found it important to connect different FPGA kernels to different HBM memory controllers and keep them separate in this manner
    - Not doing so meant that we were prone to hold conflicts during building
    - This is potentially one of the reasons why our kernels scale well, as there is no contention on memory access between them

Conclusions and further work

- In summary, I think our results on the Alveo U280 are positive for FPGAs:
  - Significantly out-performs the CPU at two and a half times less power consumption
  - Achieves 71% the performance of the V100 but at 2.4 times less power draw and almost twice the power efficiency
  - We had a few headaches scaling up to four kernels, but doable with some trial and error
- Lots of steps required to optimise the kernel for dataflow and the performance difference by doing so is approximately 4000 times from the Von-Neumann to optimised dataflow version
  - We found the theoretical performance a very helpful measure to calculate and compare against
  - Found that it's still critical to use the Vitis-HLS IDE for analysis of code to understand what potential issues there might be

---

- In the future could potentially increase the number of kernels by:
  - Exploring reduced precision and fixed point, along with the accuracy impacts it makes within Nekbone
  - Experiments with other polynomial orders as N=16 is rather high, and reducing this will reduce our resource requirements
- Exploring next generation FPGAs such as Versal, although to be fair would need to compare against the A100 GPU which is also likely to provide improved performance.
- Based our work on the original Fortran Nekbone version, updating this to the newer C++ version would be useful and enable more convenient use of our dataflow code by the community.