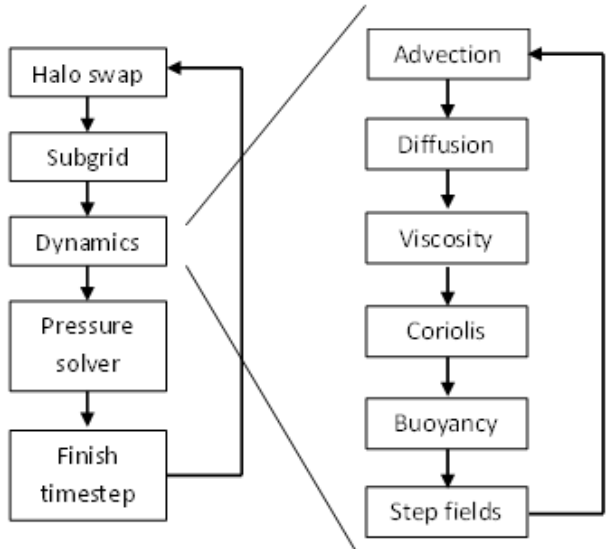
Crossing the chasm: Accelerating HPC codes using FPGAs

Dr Nick Brown, Research Fellow

n.brown@epcc.ed.ac.uk

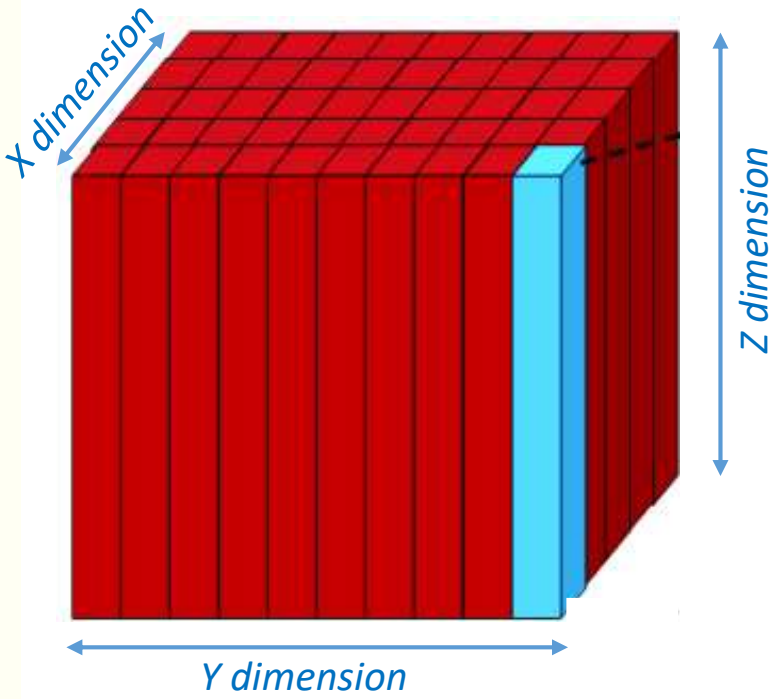
Met Office NERC Cloud (MONC) model

- MONC is a model we developed with the Met Office for simulating clouds and atmospheric flows
  - Advection is the most computationally intensive part of the code at around 40% runtime
  - Stencil based code
  - Porting to the advection to the ADM8K5 board
  - Advecting over three fields (wind in x, y, and z dimensions)



Advection code sketch

```
do i=1, x_size
  do j=1, y_size
    do k=2, z_size
      su(k, j, i) = tcx * (u(k,j,i-1) * (u(k,j,i) +
       u(k,j,i-1)) - u(k,j,i+1) * (u(k,j,i) + u(k,j,i+1)))
      su(k, j, i) = su(k, j, i) + tcy * (u(k,j-1,i) *
              (v(k,j-1,i) + v(k,j-1,i+1)) - u(k,j+1,i) *
              (v(k,j,i) + v(k,j,i+1)))

      if (k .lt. z_size) then
        su(k, j, i) = su(k, j, i) + tzc1(k) * u(k-1,j,i) *
              (w(k-1,j,i) + w(k-1,j,i+1)) - tzc2(k) * u(k+1,j,i) *
              (w(k,j,i) + w(k,j,i+1))
      else
        su(k, j, i) = su(k, j, i) + tzc1(k) * u(k-1,j,i) *
              (w(k-1,j,i) + w(k-1,j,i+1))
      end if
    end do
  end do
end do
```

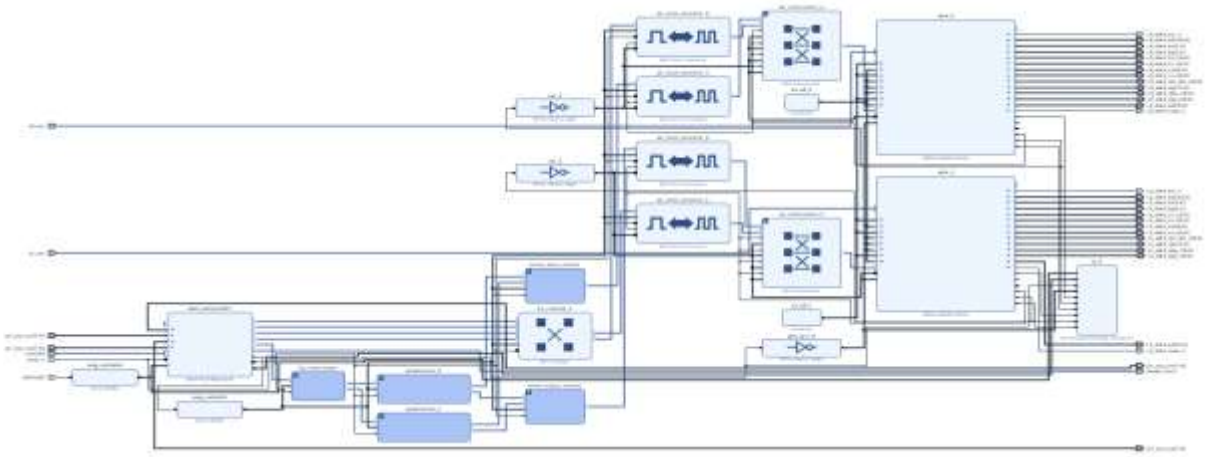


- 53 double precision floating point operations per grid cell for all three fields in this kernel
  - 32 double precision floating point multiplications, 21 floating point additions or subtractions

The journey to HLS kernel performance

Description	Runtime (ms)
Sandybridge CPU core	676.4
Broadwell CPU core	540.2

- Following Xilinx's UltraFast High-Level Productivity Design Methodology
  - Write the kernel(s) using HLS to generate RTL
  - Use the block design in Vivado to hook up



Description	Runtime (ms)
Initial port	51498
Pipeline directive on inner loop	14130
Local BRAM as cache and re-order loops	621.3
Tune double precision cores and clock to 310Mhz	584.6
Concurrent loading and storing via dataflow directive	189.64
X dimension of cube in the dataflow region	163.43
256 bit DRAM connected ports	65.41
Issue 4 doubles per cycle	63.49

These timings are the compute time of a single HLS kernel, ignoring DMA transfer, for problem size of 16.7 million grid cells

Initial High Level Synthesis kernel

```
int pw_advection(double * u, double * su, ..., int size_x, int size_y, ...) {
  #pragma HLS INTERFACE m_axi port=u offset=slave
  #pragma HLS INTERFACE m_axi port=su offset=slave
  #pragma HLS INTERFACE s_axilite port=size_x bundle=CTRL_BUS
  #pragma HLS INTERFACE s_axilite port=size_y bundle=CTRL_BUS
  #pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
  .....
}
```

- Convert into C and apply appropriate directives on interface

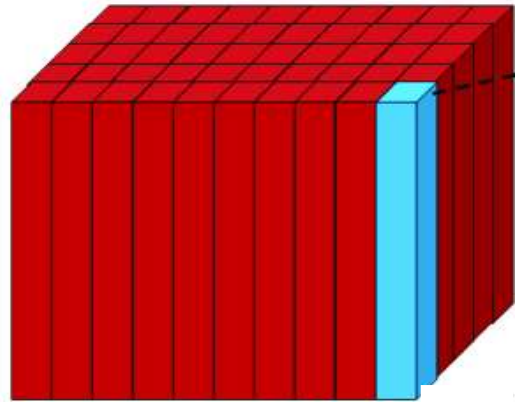
- Pipeline the inner loop with initiation interval of one
  - Decreases runtime from 51 seconds to 14 seconds
  - But data ports are the limit here, maximum two accesses (as dual ported) any one clock cycle and-so HLS identifies possible conflict and limits pipeline accordingly

```
for (unsigned int i=start_x;i<end_x;i++) {
  for (unsigned int j=start_y;j<end_y;j++) {
    for (unsigned int k=1;k<size_z;k++) {
      #pragma HLS PIPELINE II=1
      su(i,j,k)=tcx*(u(i-1,j,k) * (u(i,j,k) + u(i-1,j,k)) - u(i+1,j,k) *
                 (u(i,j,k) + u(i+1,j,k)));
      su(i,j,k)=su(i,j,k) + tcy*(u(i-1,j,k) *(v(i,j-1,k) + v(i+1,j-1,k)) -
                 u(i,j+1,k) * (v(i,j,k) * v(i+1,j,k)));
      ……..
    }
  }
}
```

Use BRAM as a cache

```
double u_vals[MAX_VERTICAL_SIZE], u_xp1_vals[MAX_VERTICAL_SIZE],
u_vals2[MAX_VERTICAL_SIZE], ....;

for (unsigned int i=start_x;i<end_x;i++) {
  for (unsigned int j=start_y;j<end_y;j++) {
    memcpy(u_vals, &u(i,j,0), sizeof(double) * size_z);
    memcpy(u_xp1_vals, &u(i+1,j,0), sizeof(double) * size_z);
    memcpy(u_vals2, &u(i,j,0), sizeof(double) * size_z);
    ....
    for (unsigned int k=1;k<size_z;k++) {
      #pragma HLS PIPELINE II=1
      .....
    }
  }
}
```

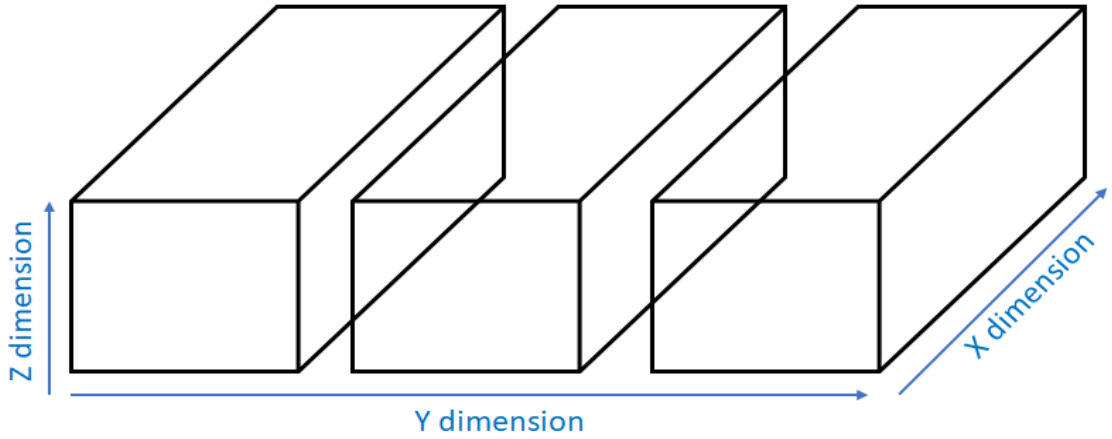
- Use local BRAM to hold data required for working with a single column
  - Copy all data required for a column from the external data ports, then process the column
  - MAX_VERTICAL_SIZE is required as an not dynamically size these in HLS
  - Either single or dual ported, but more than 2 accesses can be needed at any time – and therefore duplicate them



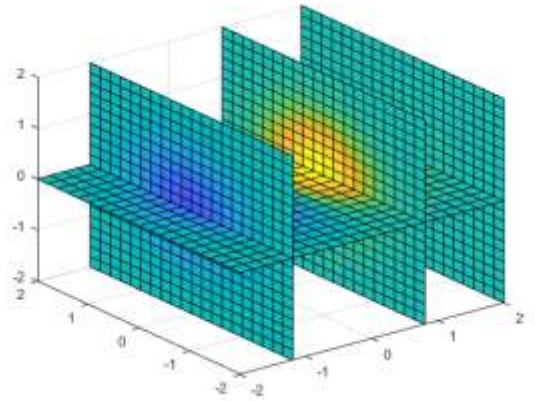
- Runtime is now 3.2 seconds, having sped kernel up by further four times
  - But a major limit is must stop and copy before each column, draining the pipeline
  - 71 cycles deep with II of 2, with column size of 64 elements then each column the pipeline will run for 199 cycles but for only 28% of cycles is the pipeline full utilised ☹

Extending the use of BRAM as a cache

```
for (unsigned int m=start_y;m<end_y;m+=BLOCKSIZE_IN_Y) {
  ...
  for (unsigned int i=start_x;i<end_x;i++) {
    for (unsigned int c=0; c < slice_size; c++) {
      #pragma HLS PIPELINE II=1
      // Move data in slice+1 and slice down by one in X dimension
    }
    for (unsigned int c=0; c < slice_size; c++) {
      #pragma HLS PIPELINE II=1
      // Load data for all fields from DRAM
    }
    for (unsigned int j=0;j<number_in_y;j++) {
      for (unsigned int k=1;k<size_in_z;k++) {
        #pragma HLS PIPELINE II=1
        // Do calculations for U, V, W field grid points
        su_vals[jk_index]=su_x+su_y+su_z;
        sv_vals[jk_index]=sv_x+sv_y+sv_z;
        sw_vals[jk_index]=sw_x+sw_y+sw_z;
      }
    }
    for (unsigned int c=0; c < slice_size; c++) {
      #pragma HLS PIPELINE II=1
      // Write data for all fields to DRAM
    }
  }
}
```



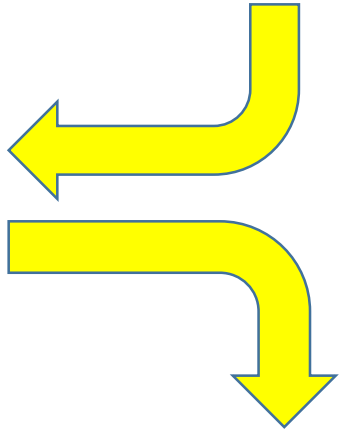
- Reduces runtime down to 0.62 seconds
  - Faster than a core of Sandybridge but slower than a Broadwell core



Tuning double precision FP cores

```
su(k, j, i) = tcx * (u(k,j,i-1) * (u(k,j,i) + u(k,j,i-1)) - u(k,j,i+1) * (u(k,j,i) + u(k,j,i+1)))
```

```
unsigned int jk_index=(size_z * j) + k;
double u_data=u_vals[jk_index];
double um1_data=um1_vals[jk_index];
double up1_data=up1_vals[jk_index];

double t1=u_data+um1_data;
double t2=u_data+up1_data;
double t7=um1_data * t1;
double t8=up1_data * t2;
double su_x=tcx*(t7 - t8);
```

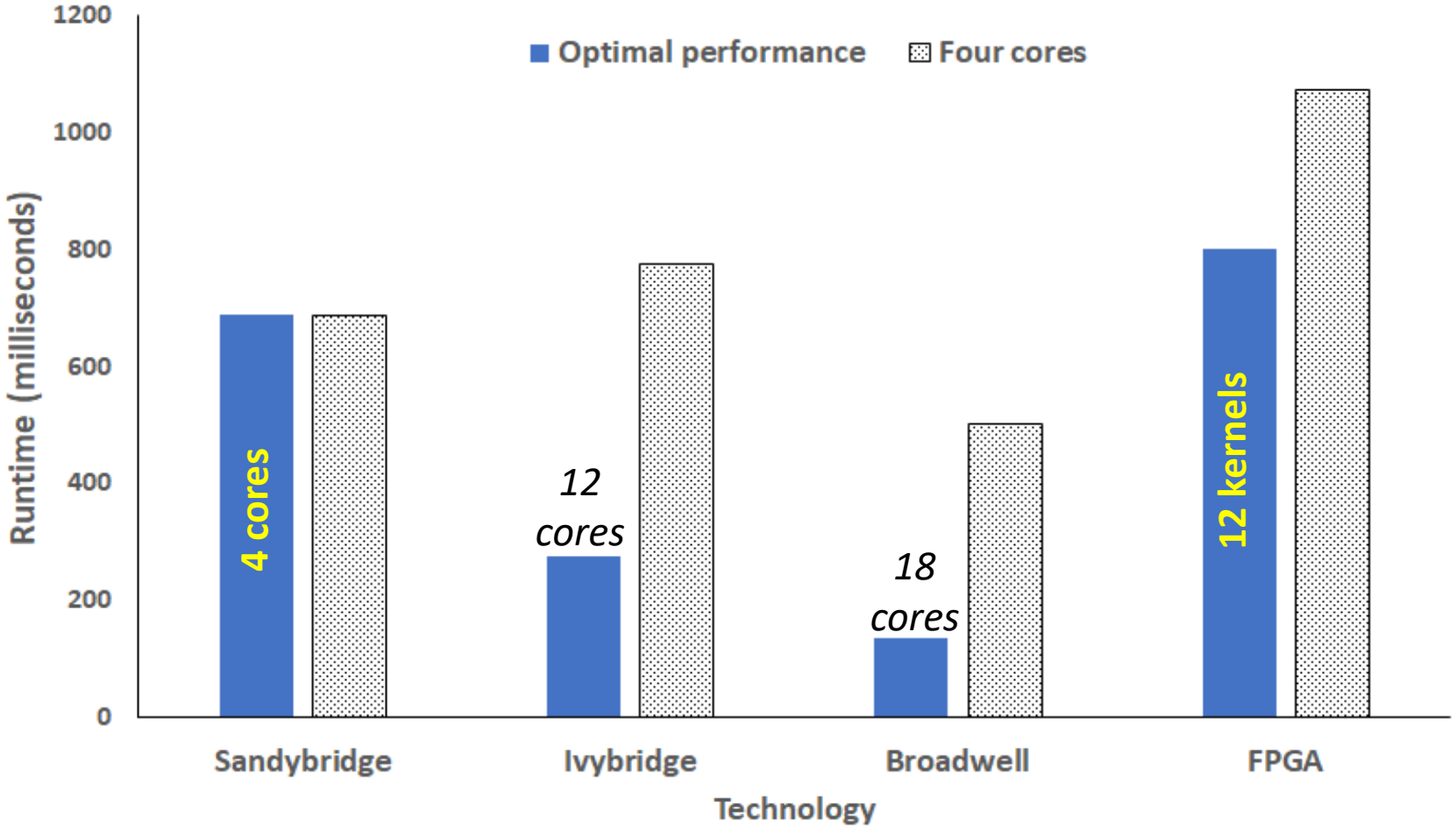


```
#pragma HLS RESOURCE variable=t1 core=DAddSub_fulldsp
#pragma HLS RESOURCE variable=t2 core=DAddSub_fulldsp
#pragma HLS RESOURCE variable=t7 core=DMul_meddsp latency=14
#pragma HLS RESOURCE variable=t8 core=DMul_meddsp latency=14
#pragma HLS RESOURCE variable=su_x core=DMul_meddsp latency=14

double t1=u_data+um1_data;
double t2=u_data+up1_data;
double t7=um1_data * t1;
double t8=up1_data * t2;
double su_x=tcx*(t7 - t8);
```

- Tuned all HLS double precision cores
- The major benefit here was the multiplication
  - Using medium DSP reduced DSP usage by about 1/5th
  - Further pipelined the core to 14 stages, provided period of 2.75 ns meaning we could up the clock frequency to 310MHz
  - Increases pipeline depth from 65 to 72, but latency for a piece of data has decreased from 2.6e-7 seconds to 2.3e-7 seconds.
  - 0.58 seconds runtime

Overall performance at this point

- 67 million grid points with a standard stratus cloud test-case
- Approximately 7 times slower than 18 cores of Broadwell
  - DMA transfer time accounted for over 70% of runtime

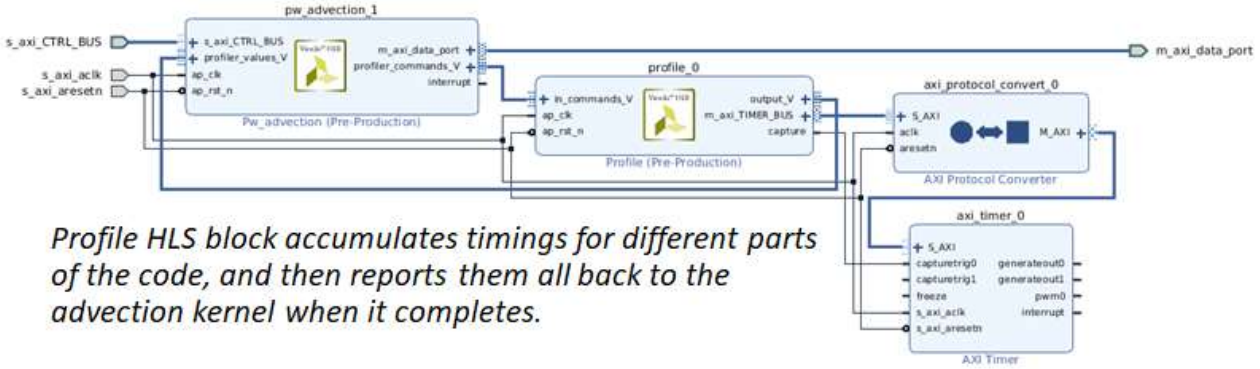


Finding out where the bottlenecks are

Description	Total Runtime (ms)	% in compute	Load data (ms)	Prepare stencil & compute results (ms)	Write data (ms)
Current version	584.65	14%	320.82	80.56	173.22
Run concurrent loading and storing via dataflow directive	189.64	30%	53.43	57.28	75.65
Include X dimension of cube in the dataflow region	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
Issue 4 doubles per cycle	63.49	85%	2.72	53.88	3.60

These timings are the compute time of a single HLS kernel, ignoring DMA transfer, for problem size of 16.7 million grid cells

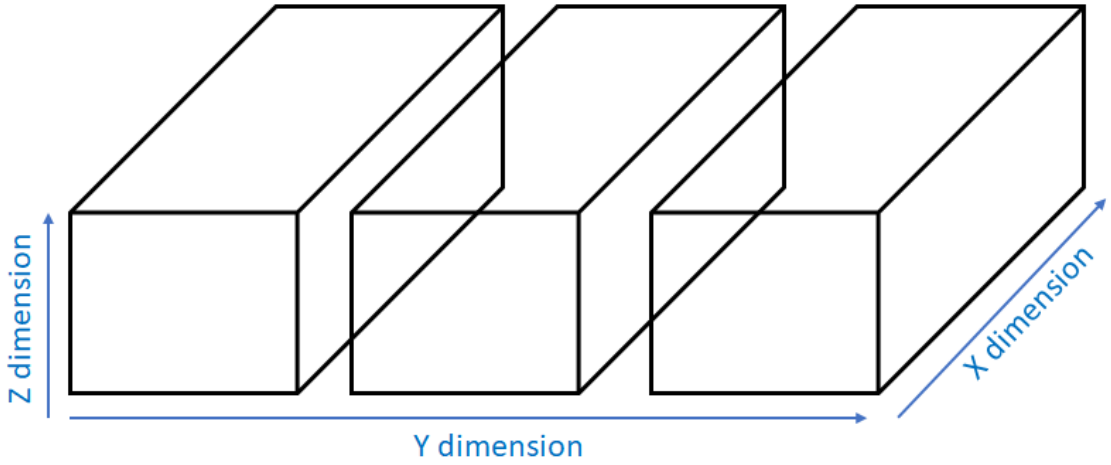
- Found that 14% of runtime was doing compute by the kernel, 86% on memory access!



Profile HLS block accumulates timings for different parts of the code, and then reports them all back to the advection kernel when it completes.

Run concurrent loading and storing via dataflow directive

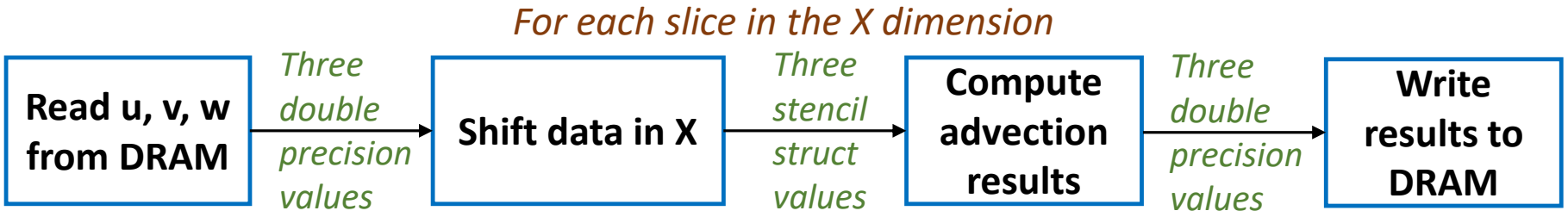
```
for (unsigned int m=start_y;m<end_y;m+=BLOCKSIZE_IN_Y) {
  ...
  for (unsigned int i=start_x;i<end_x;i++) {
    for (unsigned int c=0; c < slice_size; c++) {
      #pragma HLS PIPELINE II=1
      // Move data in slice+1 and slice down by one in X dimension
    }
    for (unsigned int c=0; c < slice_size; c++) {
      #pragma HLS PIPELINE II=1
      // Load data for all fields from DRAM
    }
    for (unsigned int j=0;j<number_in_y;j++) {
      for (unsigned int k=1;k<size_in_z;k++) {
        #pragma HLS PIPELINE II=1
        // Do calculations for U, V, W field grid points
        su_vals[jk_index]=su_x+su_y+su_z;
        sv_vals[jk_index]=sv_x+sv_y+sv_z;
        sw_vals[jk_index]=sw_x+sw_y+sw_z;
      }
    }
    for (unsigned int c=0; c < slice_size; c++) {
      #pragma HLS PIPELINE II=1
      // Write data for all fields to DRAM
    }
  }
}
```



- But each part runs sequentially for each slice:
  1. Move data in slice+1 and slice down in X by 1
  2. Load data for all fields into DRAM
  3. Do calculations for U,V,W field grid points
  4. Write data for fields to DRAM

- Instead, can we run these concurrently for each slice?

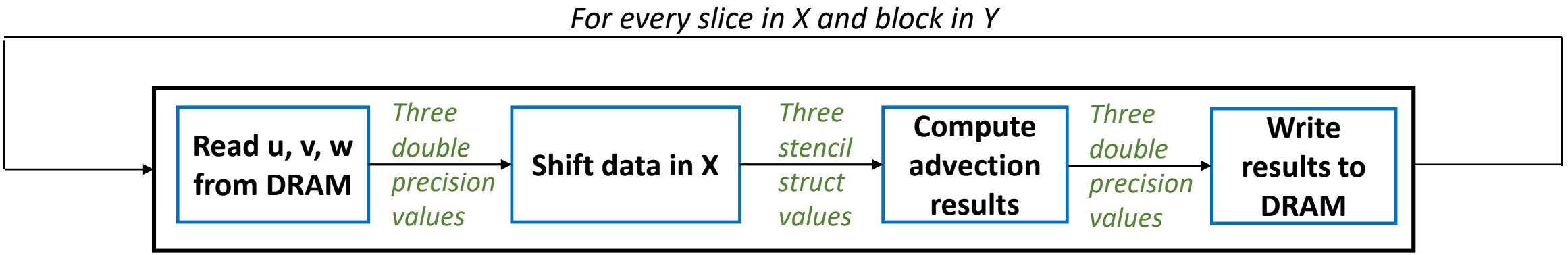
Run concurrent loading and storing via dataflow directive

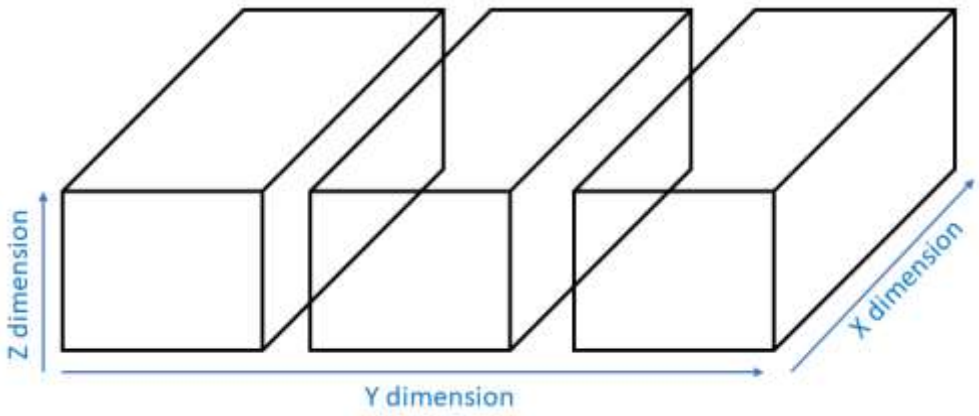


- Using the HLS Dataflow directive create a pipeline of these four activities
  - These stage use HLS streams (FIFO queues) to connect them
- Resulted in 2.60 times runtime reduction
  - Reduced computation runtime by around 25%
  - Reduced data access time by over 3x
  - Time spent in computation now 30%

Description	Total Runtime (ms)	% in compute	Load data (ms)	Prepare stencil & compute results (ms)	Write data (ms)
Current version	584.65	14%	320.82	80.56	173.22
Run concurrent loading and storing via dataflow directive	189.64	30%	53.43	57.28	75.65
Include X dimension of cube in the dataflow region	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
Issue 4 doubles per cycle	63.49	85%	2.72	53.88	3.60

Where we are….



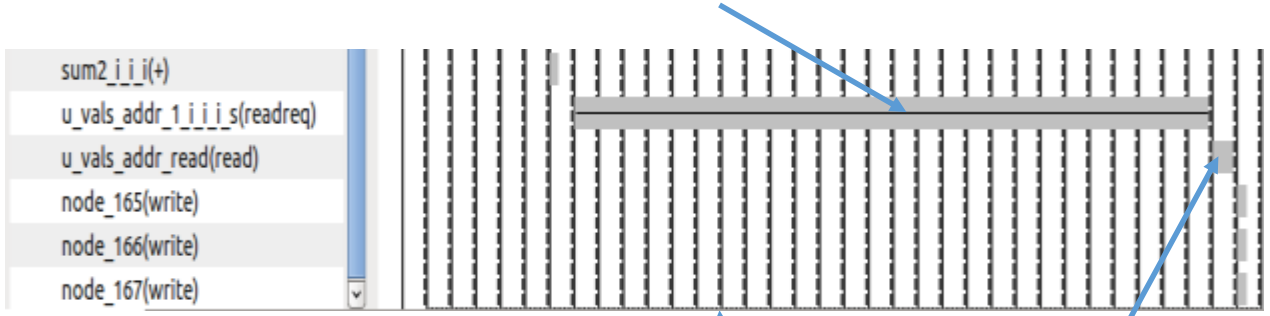


Description	Total Runtime (ms)	% in compute	Load data (ms)	Prepare stencil & compute results (ms)	Write data (ms)
Current version	584.65	14%	320.82	80.56	173.22
Run concurrent loading and storing via dataflow directive	189.64	30%	53.43	57.28	75.65
Include X dimension of cube in the dataflow region	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
Issue 4 doubles per cycle	63.49	85%	2.72	53.88	3.60

Include X dimension of cube in dataflow region

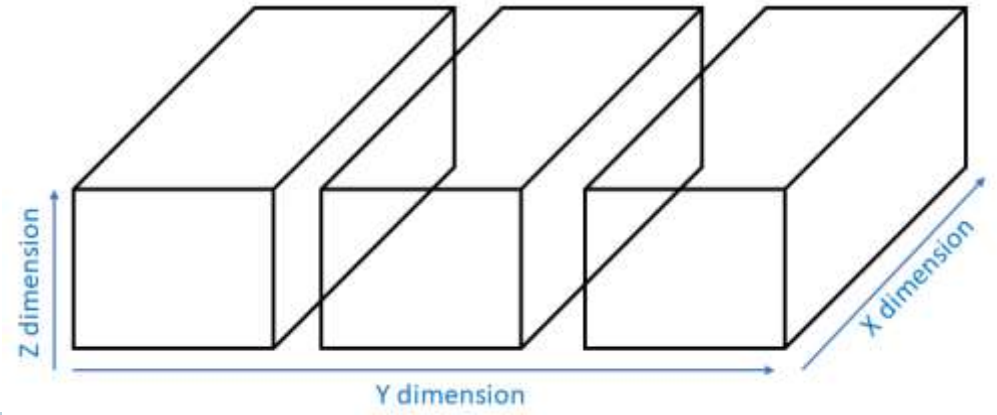
```
void retrieve_input_data(double*u,hls::stream<double>& ids){
   for (unsigned int i=start_x;i<end_x;i++) {
      int start_read_index=……;
      for (unsigned int c=0;c<slice_size;c++) {
#pragma HLS PIPELINE II=1
         int read_index=start_read_index+x;
         ids.write(u[read_index]);
      }
   }
}

void perform_advection(double * u) {
   for (unsigned int m=start_y;m<end_y;m+=BLOCKSIZE_IN_Y) {
      ...
#pragma HLS DATAFLOW
      retrieve_input_data(u, in_data_stream_u, ...);
      ...
   }
}
```

Readreq done for every element 25 cycles



Read 1 cycle

The inner loop is 28 cycles total



Sped up the compute slightly, but data access was 3.6 times slower!

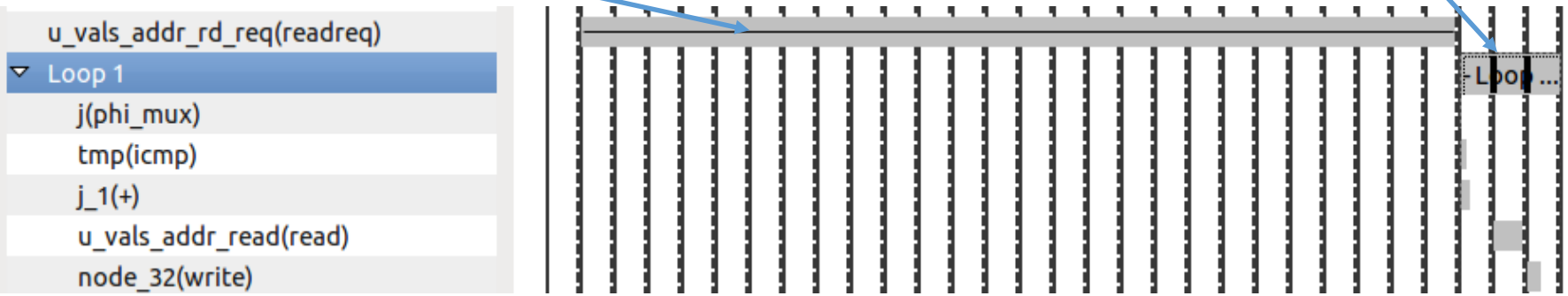
Include X dimension of cube in dataflow region

```
void retrieve_input_data(double*u,hls::stream<double>& ids){
   for (unsigned int i=start_x;i<end_x;i++) {
      int start_read_index=……;
      for (unsigned int c=0;c<slice_size;c++) {
#pragma HLS PIPELINE II=1
         int read_index=start_read_index+x;
         ids.write(u[read_index]);
      }
   }
}
```

```
void retrieve_input_data(double*u,hls::stream<double>& ids){
   for (unsigned int i=start_x;i<end_x;i++) {
      int start_read_index=……;
      do_retrieve(i, u, ids);
   }
}

void do_retrieve(int i, double*u, hls::stream<double>& ids){
   for (unsigned int c=0;c<slice_size;c++) {
#pragma HLS PIPELINE II=1
      int read_index=start_read_index+x;
      ids.write(u[read_index]);
   }
}
```

Readreq moved outside loop and now only done once per slice

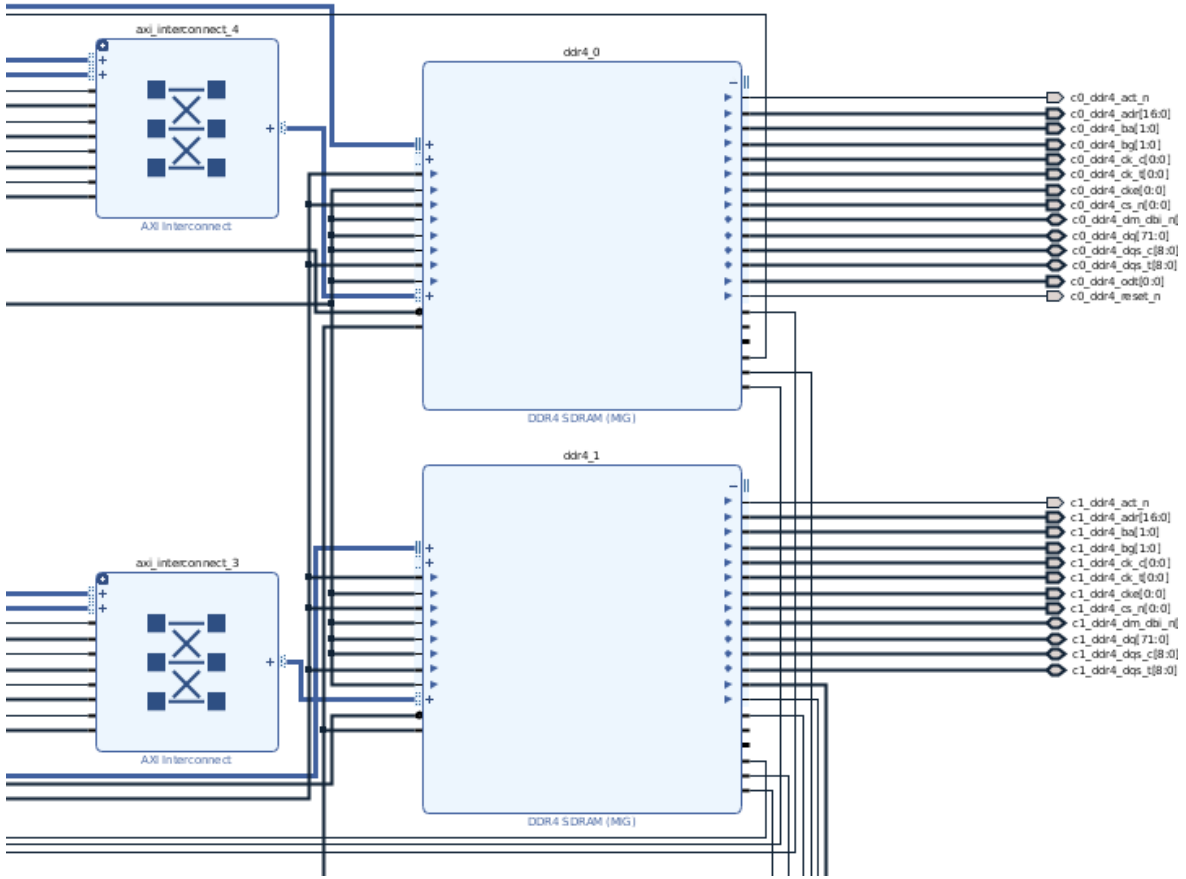
The inner loop is 3 cycles total



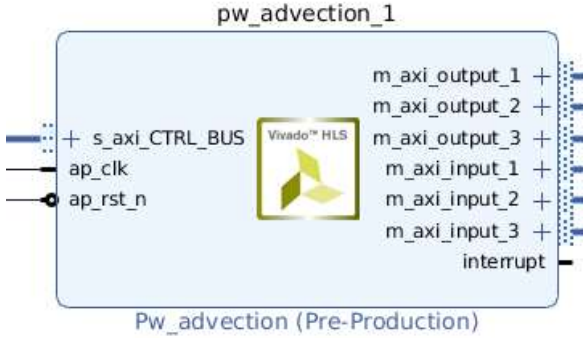
Reduced data access by 4.5 times compared to readreq in every iteration

- Slight improvement overall, compute now 33% of runtime

256 bit DRAM connected ports



- At the block design level, DRAM controllers are at 256 bits width
  - Which Alpha Data tell us is optimal
- But our kernels work with 64 bit values (double precision)



  - Using a data width converter in the AXI interconnects
- Are we throwing away bandwidth and/or creating overhead at the controller block?

256 bit DRAM connected ports

```
struct dram_data {
   double vals[4];
};

void pw_advection(struct dram_data * su, struct dram_data * sv,
struct dram_data * sw, struct dram_data * u, struct dram_data *
v, struct dram_data * w, …) {
#pragma HLS DATA_PACK variable=su
#pragma HLS DATA_PACK variable=sv
#pragma HLS DATA_PACK variable=sw
#pragma HLS DATA_PACK variable=u
#pragma HLS DATA_PACK variable=v
#pragma HLS DATA_PACK variable=w

   ...
}
```

```
void do_retrieve(int i, struct dram_data *u,
                                  hls::stream<double>& ids){
   for (unsigned int c=0;c<y_size;c++) {
      for (unsigned int j=0;j<z_size/4;j++) {
#pragma HLS PIPELINE II=1
         ...
         struct dram_data u_dram_data=u[read_index];
         for (unsigned int m=0;m<4;m++) {
            ids.write(u_dram_data.vals[m]);
         }
      }
   }
}
```

Due to conflict on ids the best II is 4

- Very significantly reduced DMA data access time by 13X
  - Now compute is 82% of the overall runtime

Description	Total Runtime (ms)	% in compute	Load data (ms)	Prepare stencil & compute results (ms)	Write data (ms)
Current version	584.65	14%	320.82	80.56	173.22
Run concurrent loading and storing via dataflow directive	189.64	30%	53.43	57.28	75.65
Include X dimension of cube in the dataflow region	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
Issue 4 doubles per cycle	63.49	85%	2.72	53.88	3.60

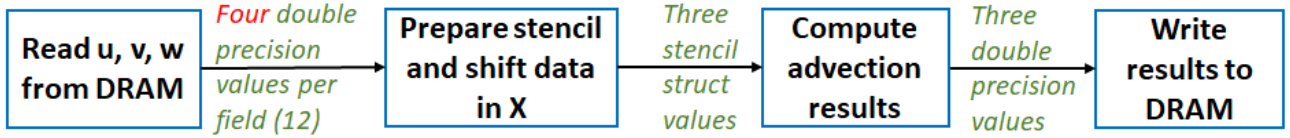
Issue 4 doubles per cycle

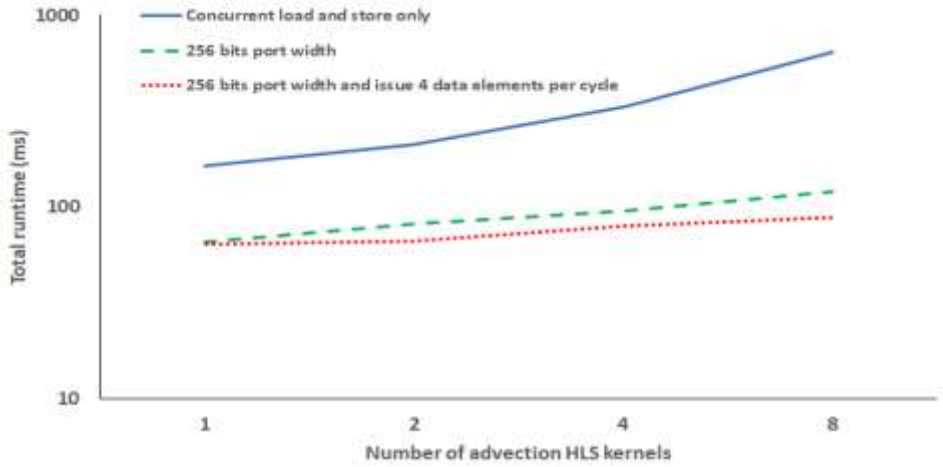
```
void do_retrieve(int i, double*u, hls::stream<double>& ids){
   for (unsigned int c=0;c<y_size;c++) {
      for (unsigned int j=0;j<z_size/4;j++) {
#pragma HLS PIPELINE II=1
         ...
         struct dram_data u_dram_data=u[read_index];
         for (unsigned int m=0;m<4;m++) {
            ids.write(u_dram_data.vals[m]);
         }
      }
   }
}
```

```
void do_retrieve(int i, struct dram_data *u,
                                hls::stream<double> ids[4]){
   for (unsigned int c=0;c<y_size;c++) {
      for (unsigned int j=0;j<z_size/4;j++) {
#pragma HLS PIPELINE II=1
         ...
         struct dram_data u_dram_data=u[read_index];
         for (unsigned int m=0;m<4;m++) {
            ids[m].write(u_dram_data.vals[m]);
         }
      }
   }
}
```

No conflict on ids so the II is now 1

- Effectively, once the pipeline is filled, every cycle we are loading 4 doubles per field into our FIFO queues



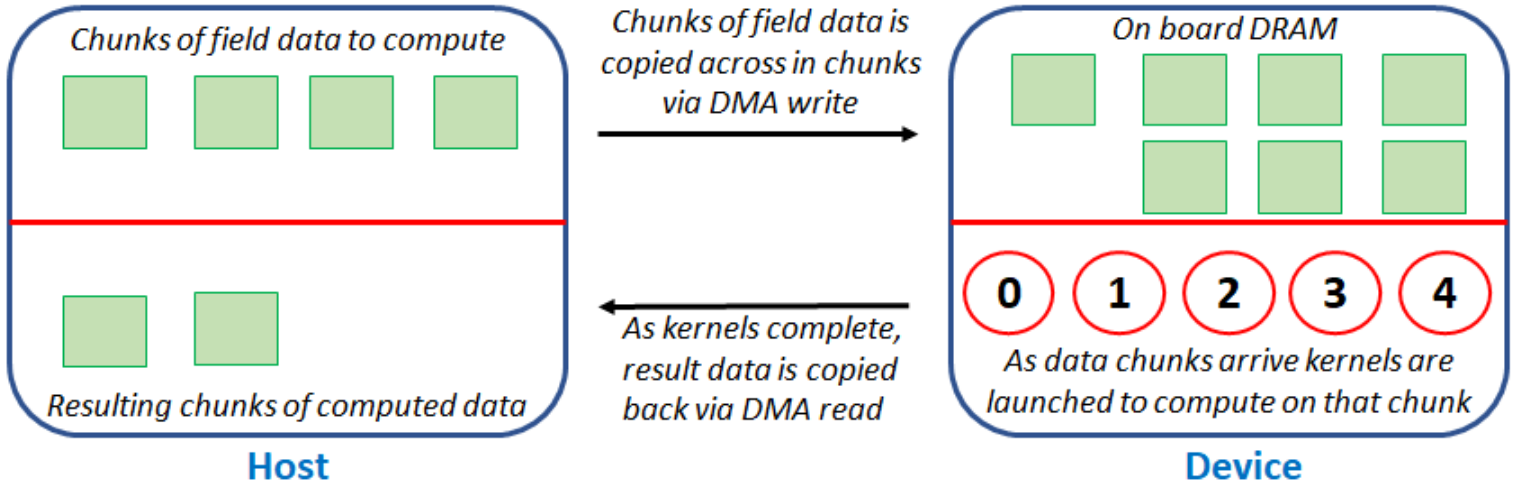


Aggregate HLS kernel only (no DMA transfer) time for problem size of 16.7 million grid points (strong scaling)

Addressing DMA transfer

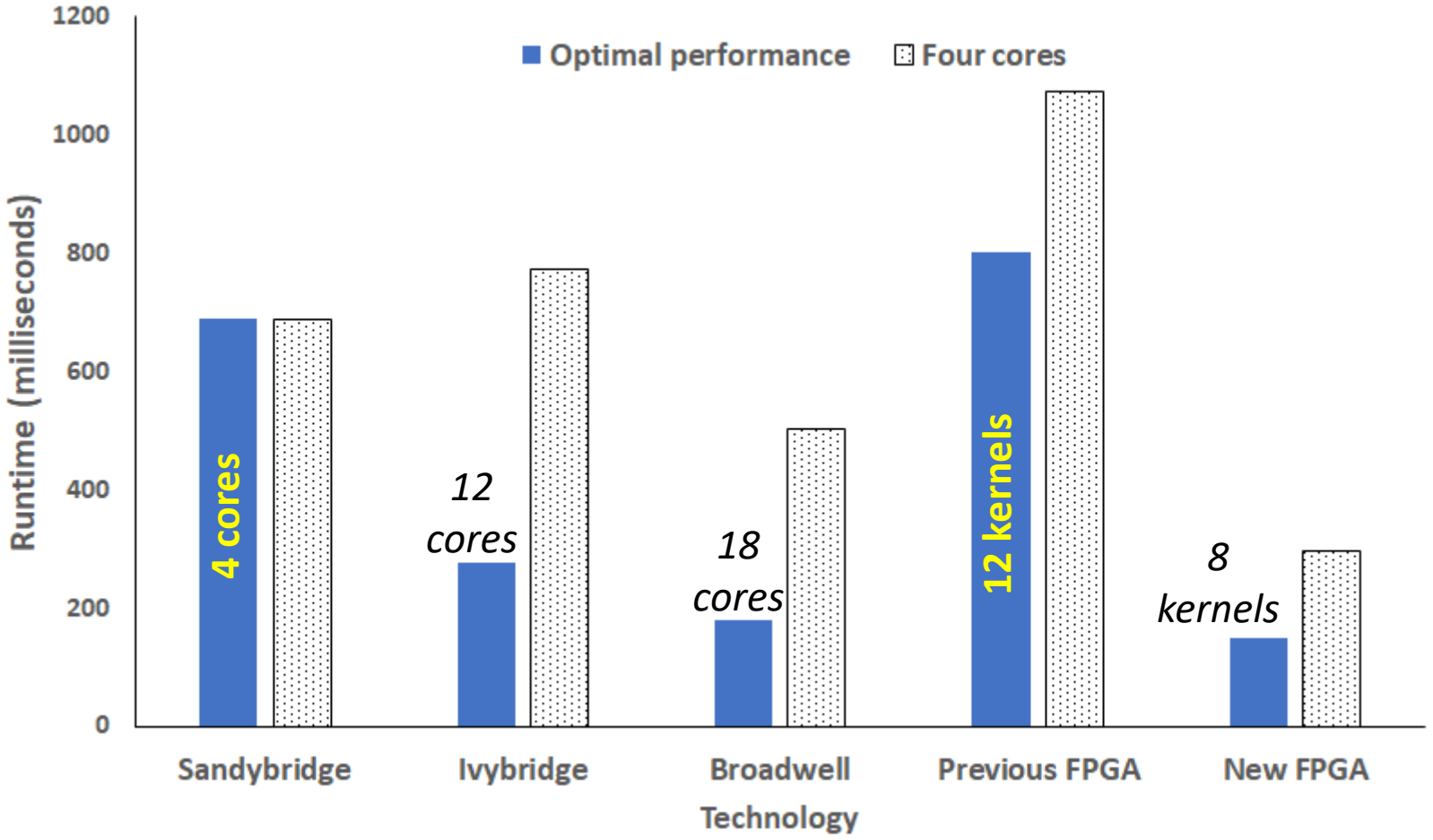
- Previously we waited for all PCIe data transfer to complete, and then kernels were started based on a static decomposition. Only once all computation was completed did results get transferred back
  - DMA was responsible for over 70% of the runtime!

- Modified to be far more dynamic
  - Split data into chunks and when complete start a kernel if one is idle
  - As soon as kernel completes begin results transfer back to the host

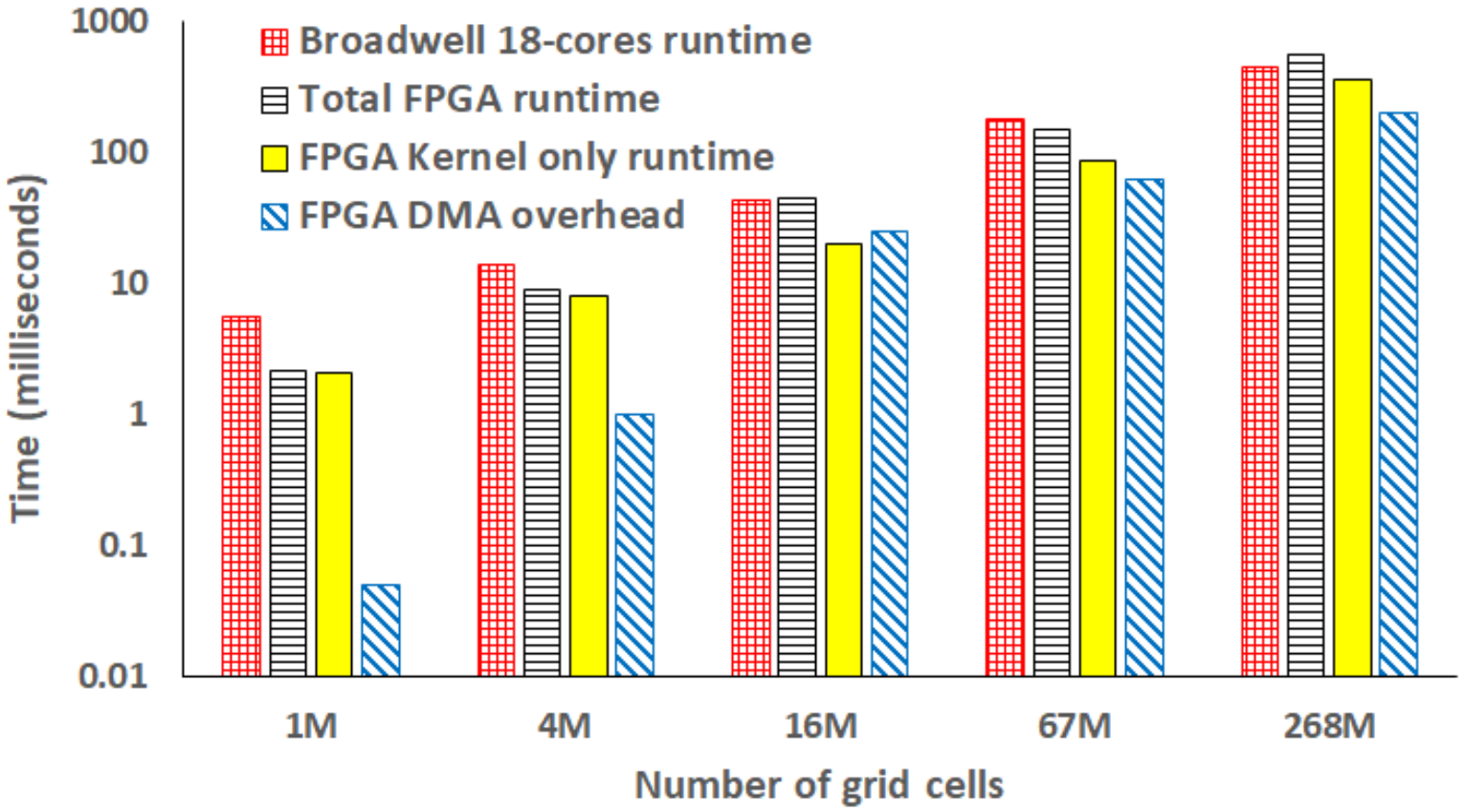


Full performance comparison

- 67 million grid points with a standard stratus cloud test-case
  - Including DMA transfer
- Now only 8 HLS kernels as new version required increased resources
- We outperform 18 cores of Broadwell now
  - 8 HLS kernels: 148ms
  - 18 Broadwell: 180ms



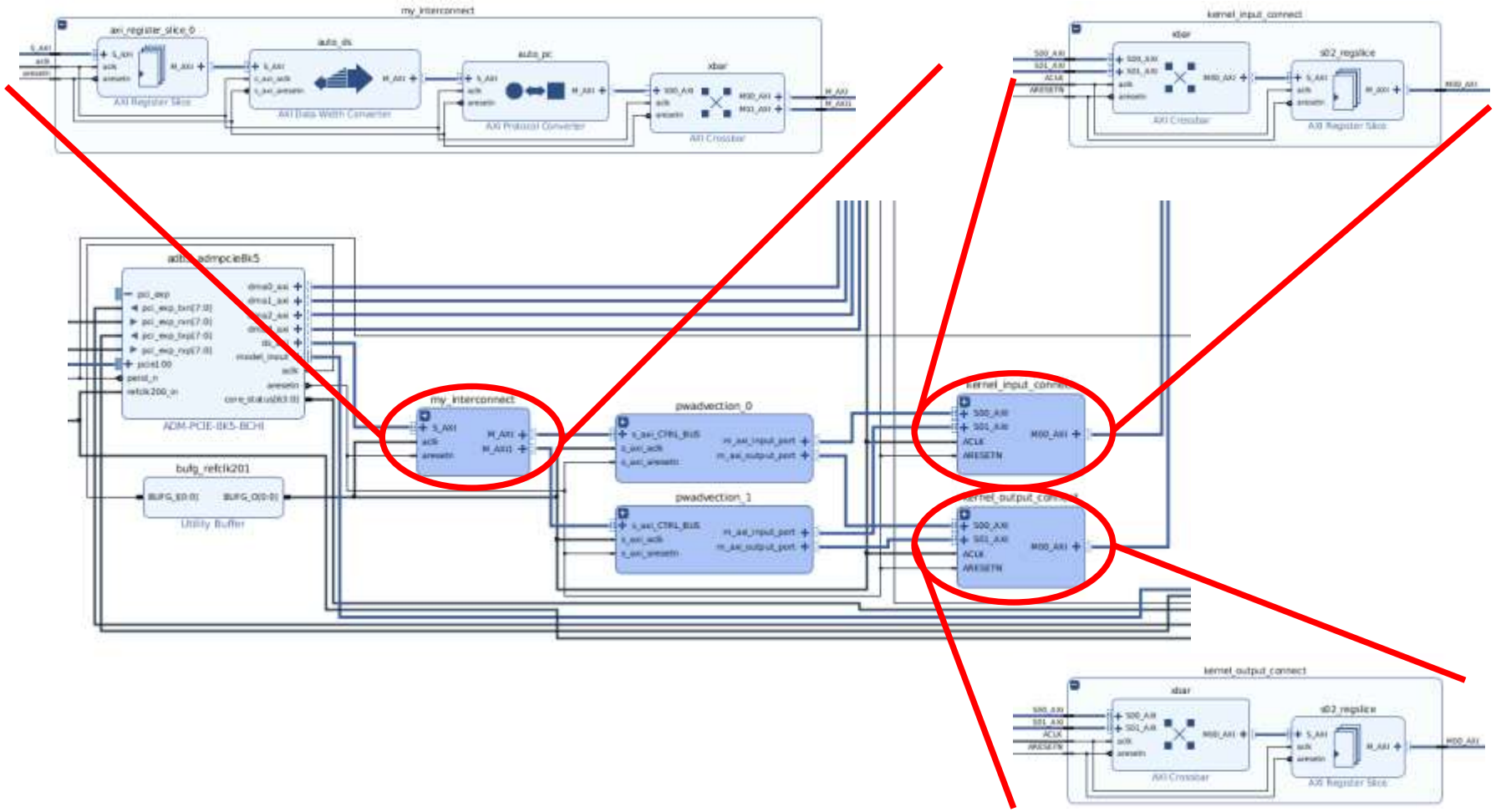
Performance comparison

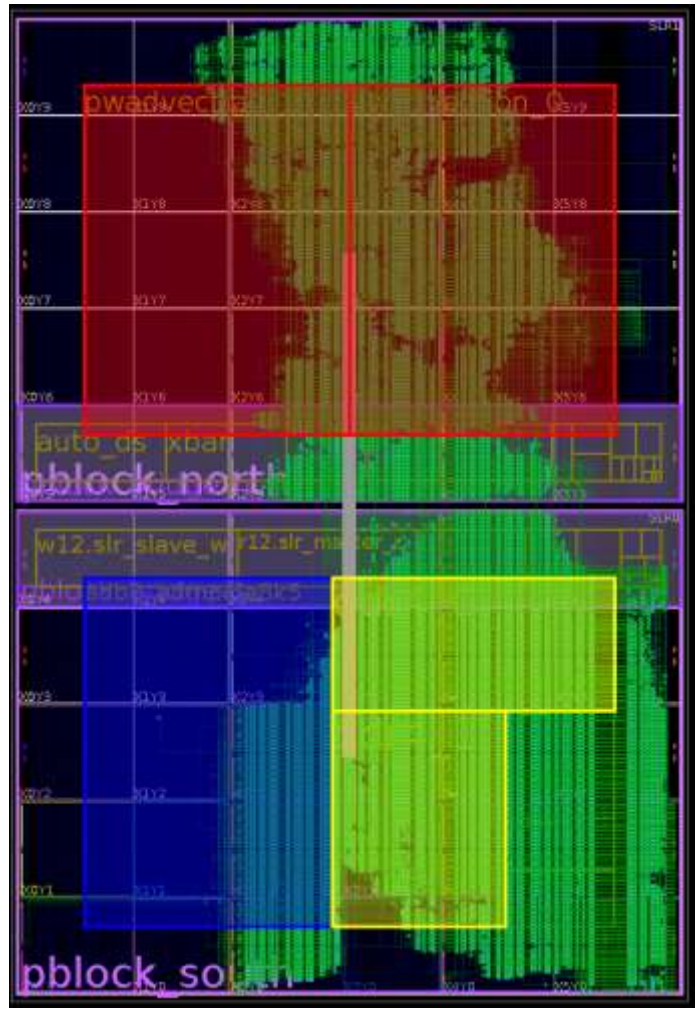


- FPGA draws 28.9 Watts idle and 35.7 Watts under load
  - Vivado estimates power draw to be 23 Watts

- Scaling size of the domain
  - We outperform 18 cores of Broadwell until 268M grid points
  - 1M: FPGA 2.59 times faster
    - DMA accounts for 2% of RT
  - 4M: FPGA 1.52 times faster
  - 16M: Approaches are comparable
  - 67M: FPGA 1.22 times faster
  - 268: Broadwell 1.23 times faster
    - DMA accounts for > 40% of RT
    - Over 12GB of data transferred to or from the PCIe card

Floor planning to meet timing





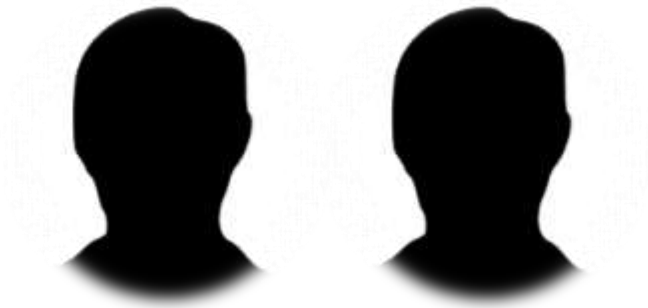
Conclusions and further work

- When accelerating codes on FPGAs *have to think dataflow*
  - But difficult to know if *thinking dataflow* enough!
  - Critically important for us to have a rich profiling environment enabling detailed performance analysis of kernels.
- Going to experiment with HBM to see if we can increase our 85% of time in compute even further
  - Will different code patterns suit HBM?
  - Double the resources of the ADM8k5
- Further developing our DMA streaming approach to be driven more by the FPGA rather than the host explicitly starting kernels
  - Use of OpenCAPI to avoid data copying in the first place
- Detailed power analysis and comparison on the CPU would be interesting

Ongoing FPGA efforts at EPCC

CASTEP on FPGAs via secondment with Alpha Data

MONC, Nek5000, Alya, AVBP on FPGAs via EXCELLERAT EU CoE



Two industrial MSc dissertation projects with Xilinx around financial modelling on FPGAs

Industrial MSc dissertation project with Alpha Data around machine learning on FPGAs