It's all about data movement: Optimising FPGA data access to boost performance

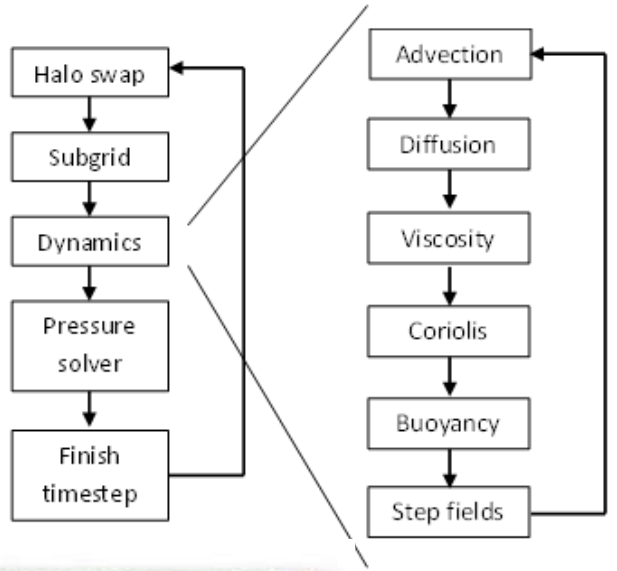
Nick Brown, EPCC at the University of Edinburgh

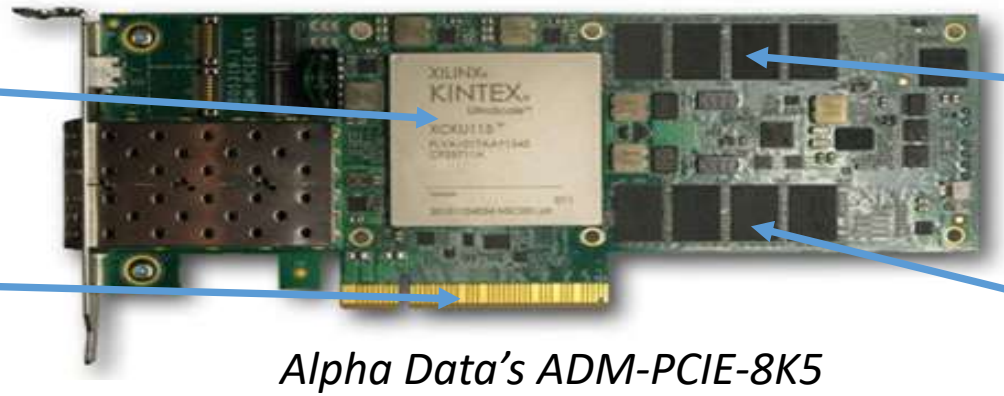
n.brown@epcc.ed.ac.uk

Co-author: David Dolman, Alpha Data

Met Office NERC Cloud (MONC) model

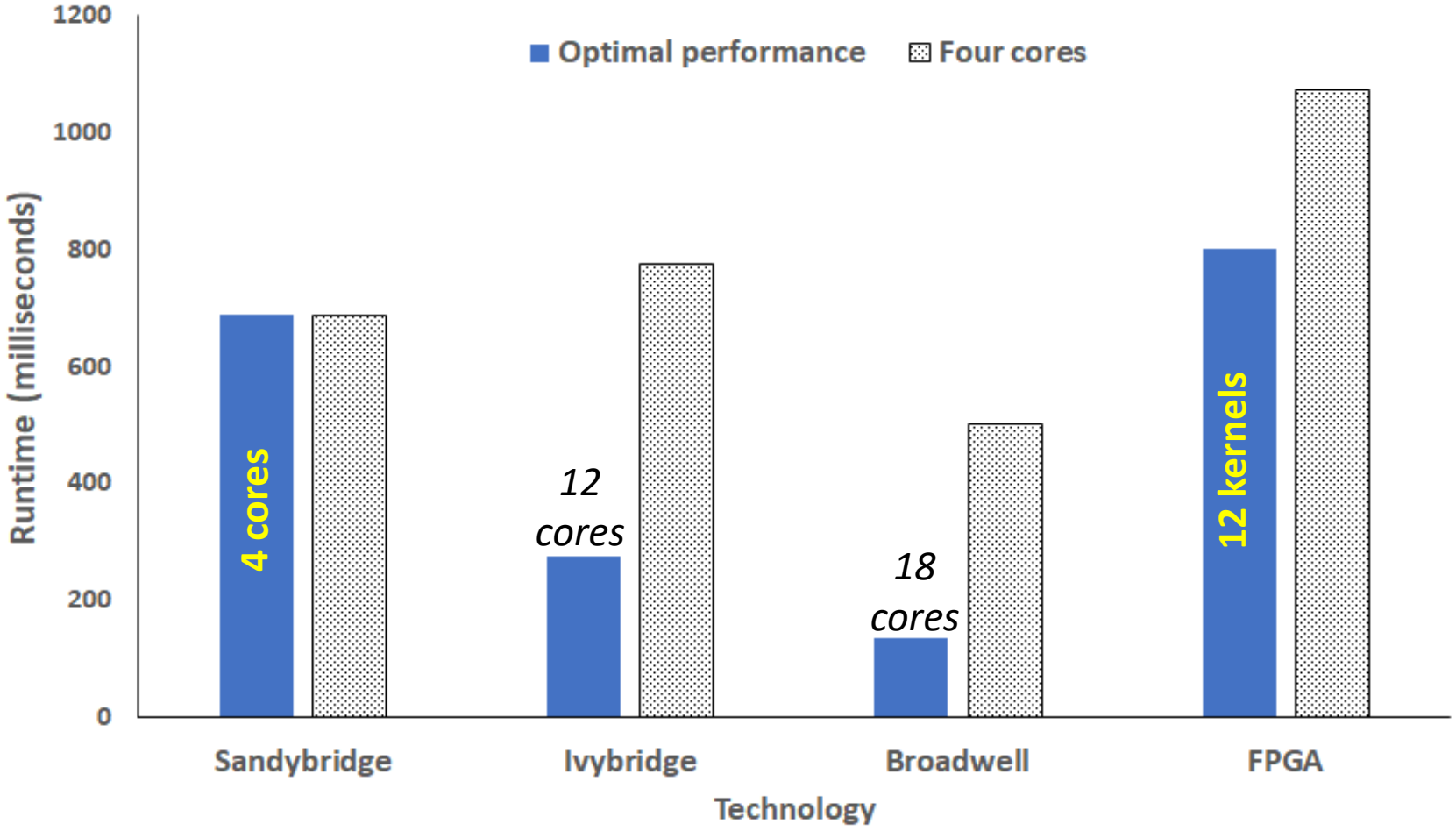
- MONC is a model we developed with the Met Office for simulating clouds and atmospheric flows
  - Advection is the most computationally intensive part of the code at around 40% runtime
  - Stencil based code
  - Previously ported the advection to the ADM8K5 board





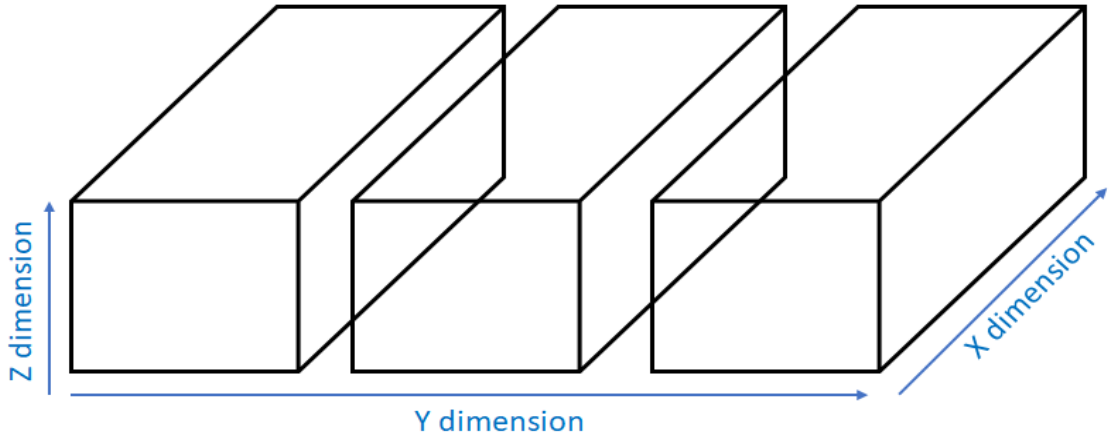
Previous code performance

- 67 million grid points with a standard stratus cloud test-case
- Approximately 7 times slower than 18 core Broadwell
  - DMA transfer time accounted for over 70% of runtime
- Using HLS and Vivado block design
  - Running at 310Mhz

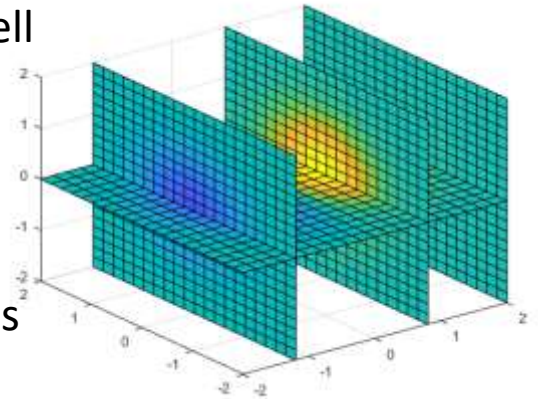


Previous code port

```
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
    }
  }
}
```

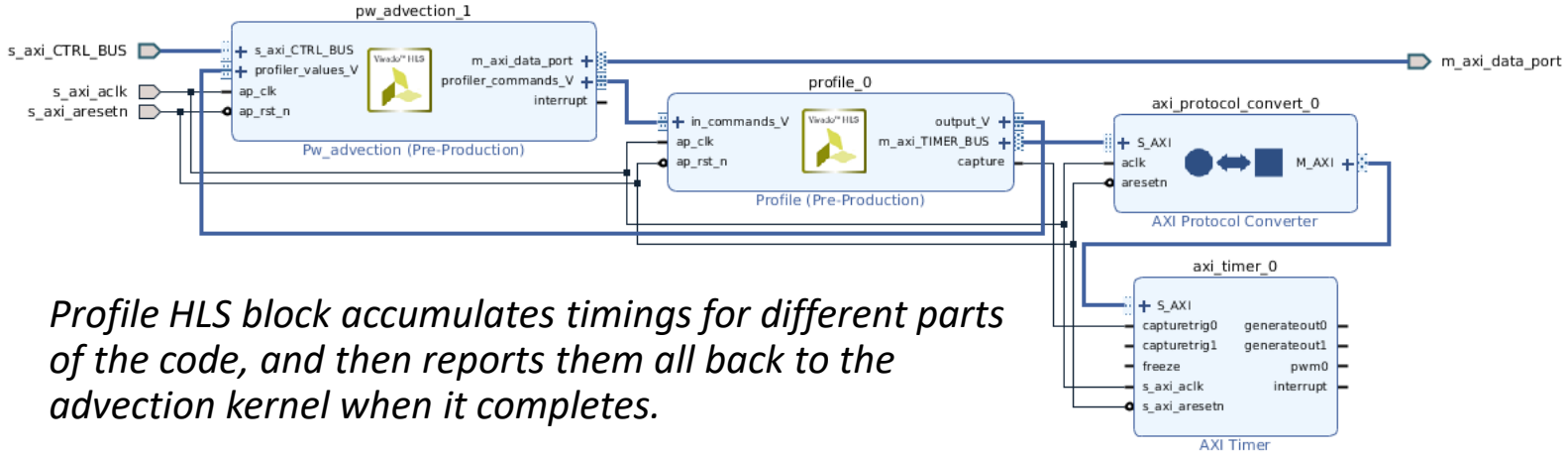


- Operates on 3 fields
- 53 double precision floating point operations per grid cell for all three fields
  - 32 double precision floating point multiplications, 21 floating point additions or subtractions



Finding out where the bottlenecks were

```
profiler_commands->write(BLOCK_1_START);
ap_wait();

function_to_execute(.....);

ap_wait();
{
  #pragma HLS protocol fixed
  profiler_commands->write(BLOCK_1_END);
  ap_wait();
}
```



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

- Wanted to understand the overhead in different parts of the code due to memory access bottlenecks
  - Found that 14% of runtime was doing compute by the kernel, 86% on memory access!
  - But whereabouts in the code should we target?
    - The reading and writing of each slice of data was by far the highest overhead

Acting on the profiling data!

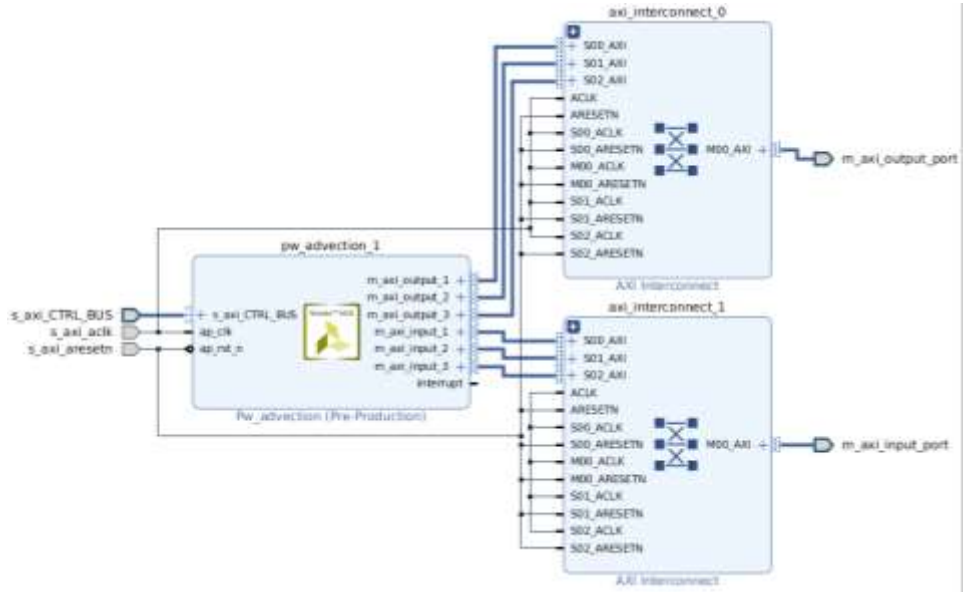
Description	Total Runtime (ms)	% in compute	Load data (ms)	Prepare stencil & compute results (ms)	Write data (ms)
Initial version	584.65	14%	320.82	80.56	173.22
Split out DRAM connected ports	490.98	17%	256.76	80.56	140.65
Run concurrent loading and storing via dataflow directive	189.64	30%	53.43	57.28	75.65
Include X dimension of cube in the dataflow region	522.34	10%	198.53	53.88	265.43
Include X dimension of cube in the dataflow region (optimised)	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
256 bit DRAM connected ports 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

Split out DRAM connected ports

```
for (unsigned int c=0; c < slice_size; c++) {
#pragma HLS PIPELINE II=1
      // Load data for U field from DRAM
      u_vals[c]=u[start_read_index+c];
}
for (unsigned int c=0; c < slice_size; c++) {
#pragma HLS PIPELINE II=1
      // Load data for V field from DRAM
      v_vals[c]=v[start_read_index+c];
}
for (unsigned int c=0; c < slice_size; c++) {
#pragma HLS PIPELINE II=1
      // Load data for W field from DRAM
      w_vals[c]=w[start_read_index+c];
}
```

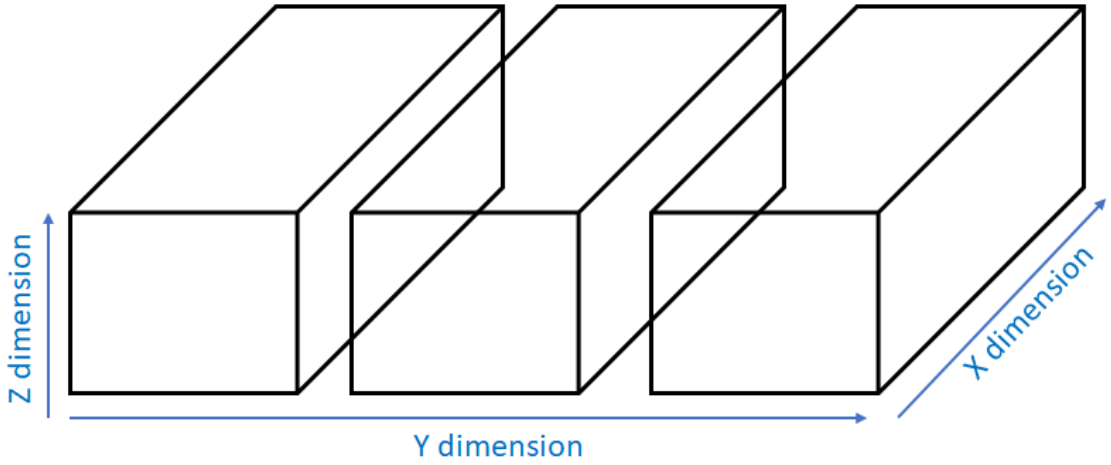
```
for (unsigned int c=0; c < slice_size; c++) {
#pragma HLS PIPELINE II=1
      // Load data for all fields from DRAM
      int read_index=start_read_index+c;
      u_vals[c]=u[read_index];
      v_vals[c]=v[read_index];
      w_vals[c]=w[read_index];
}
```



- By splitting into different ports meant that we can perform external data access concurrently
  - From 14% to 17% - reduced data access overhead from 86% to 82%
  - A slight improvement but clearly a rethink was required!

Run concurrent loading and storing via dataflow directive

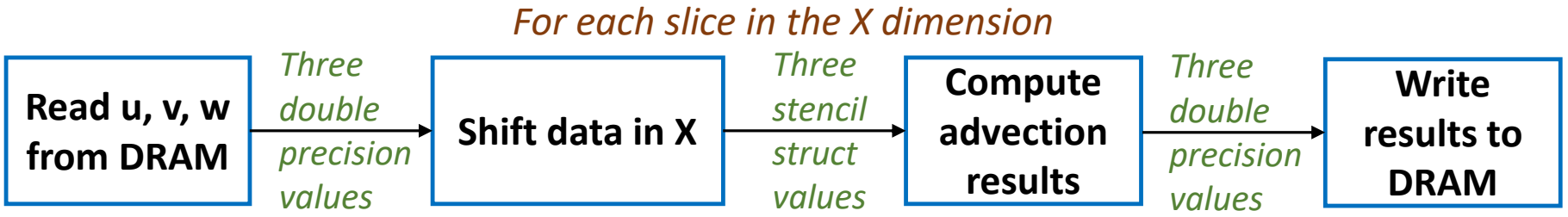
```
for (unsigned int m=start_y;m<end_y;m+=BLOCKSIZE_IN_Y) {
  ...
  for (unsigned int i=start_x;i<end_x;i++) {
    for (unsigned int c=0; c < slice_size; c++) {
      #pragma HLS PIPELINE II=1
      // Move data in slice+1 and slice down by one in X dimension
    }
    for (unsigned int c=0; c < slice_size; 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 FIFO queues to connect them
- Resulted in 2.60 times runtime reduction
  - Reduced computation runtime by around 25%
  - Over three times reduction in data access time
  - Time spent in computation now 30%

Description	Total Runtime (ms)	% in compute	Load data (ms)	Prepare stencil & compute results (ms)	Write data (ms)
Initial version	584.65	14%	320.82	80.56	173.22
Split out DRAM connected ports	490.98	17%	256.76	80.56	140.65
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	522.34	10%	198.53	53.88	265.43
Include X dimension of cube in the dataflow region (optimised)	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
256 bit DRAM connected ports issue 4 doubles per cycle	63.49	85%	2.72	53.88	3.60

Run concurrent loading and storing via dataflow directive


```
struct u_stencil {
   double z, z_m1, z_p1, y_p1, x_p1, x_m1, x_m1_z_p1;
};
```

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

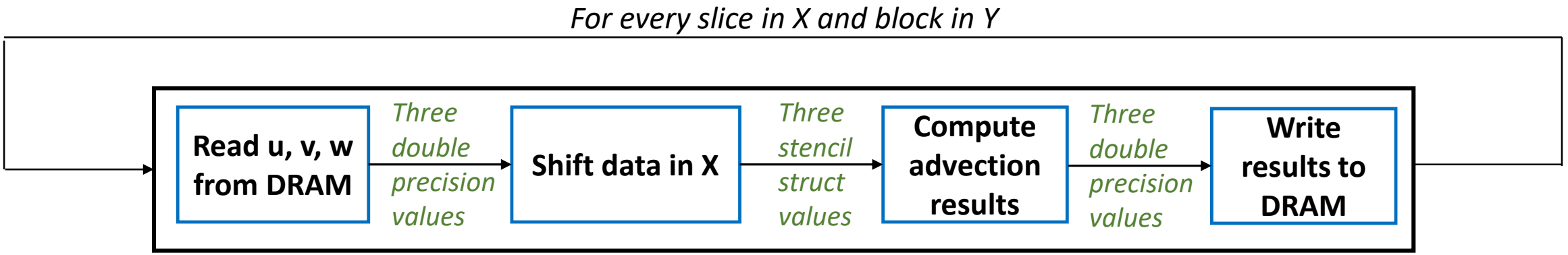
```
void shift_data_in_x(hls::stream<double> & in_data_stream_u,
hls::stream<struct u_stencil> & u_data) {
   for (unsigned int c=0;c<slice_size;c++) {
#pragma HLS PIPELINE II=1
      double x_p1_data_u=in_data_stream_u.read();
      static struct u_stencil u_stencil_data;
      // Pack u_stencil_data and shift in X
      u_data.write(u_stencil_data);
   }
}
```

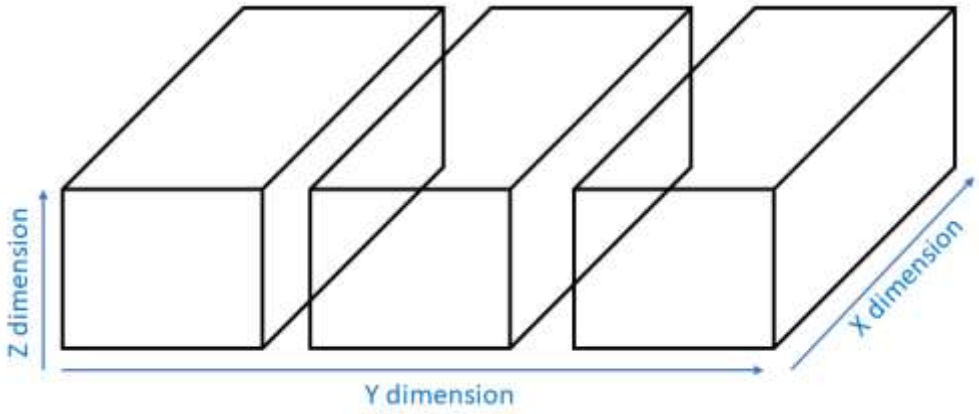
```
void write_input_data(double * u, hls::stream<double>& ids){
   for (unsigned int c=0;c<slice_size;c++) {
#pragma HLS PIPELINE II=1
      u[write_index]=ids.read();
   }
}
```

```
void advect_slice(hls::stream<struct u_stencil> &
u_stencil_stream, hls::stream<double> & data_stream_u) {
   for (unsigned int c=0;c<slice_size;c++) {
#pragma HLS PIPELINE II=1
      double su_x, su_y, su_z;
      struct u_stencil u_stencil_data = u_stencil_stream.read();
      // Perform advection computation kernel
      data_stream_u.write(su_x+su_y+su_z);
   }
}
```

```
void perform_advection(double * u) {
 for (unsigned int m=start_y;m<end_y;m+=BLOCKSIZE_IN_Y) {
    for (unsigned int i=start_x;i<end_x;i++) {
      static hls::stream<double> data_stream_u;
#pragma HLS STREAM variable=data_stream_u depth=16
      static hls::stream<double> in_data_stream_u;
#pragma HLS STREAM variable=in_data_stream_u depth=16
      static hls::stream<struct u_stencil> u_stencil_stream;
#pragma HLS STREAM variable=u_stencil_stream depth=16

#pragma HLS DATAFLOW
      retrieve_input_data(u, in_data_stream_u, ...);
      shift_data_in_x(in_data_stream_u, u_stencil_stream, ...);
      advect_slice(u_stencil_stream, data_stream_u, ...);
      write_slice_data(su, data_stream_u, ...);
} }
```

Where we are….



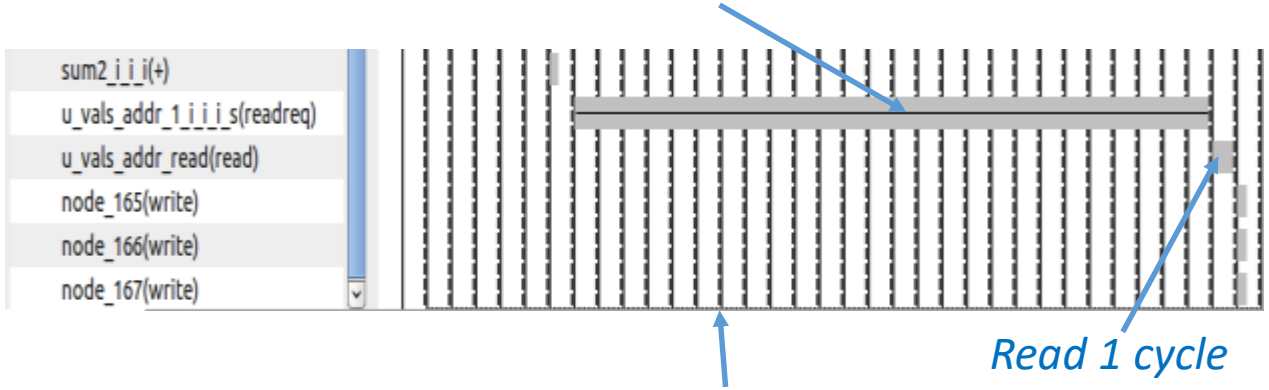


Description	Total Runtime (ms)	% in compute	Load data (ms)	Prepare stencil & compute results (ms)	Write data (ms)
Initial version	584.65	14%	320.82	80.56	173.22
Split out DRAM connected ports	490.98	17%	256.76	80.56	140.65
Run concurrent loading and storing via dataflow directive	189.64	30%	53.43	57.28	75.65
Include X dimension of cube in the dataflow region	522.34	10%	198.53	53.88	265.43
Include X dimension of cube in the dataflow region (optimised)	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
256 bit DRAM connected ports issue 4 doubles per cycle	63.49	85%	2.72	53.88	3.60

Include X dimension of cube in the dataflow region

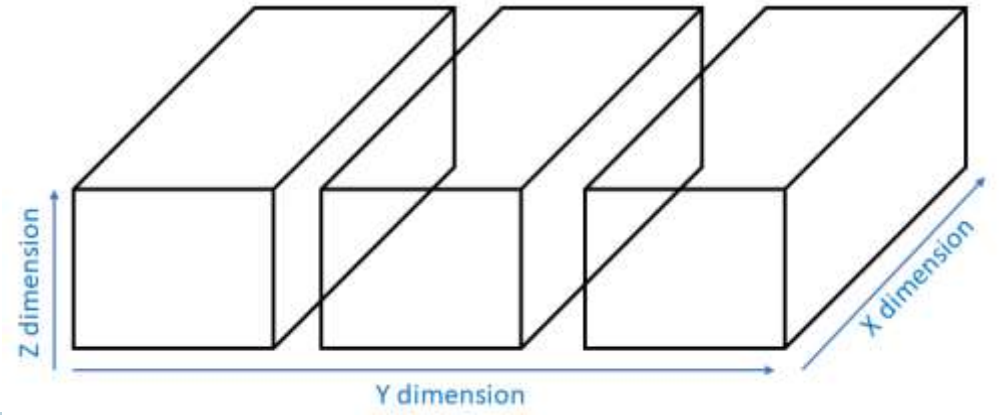
```
void retrieve_input_data(double*u,hls::stream<double>& ids){
   for (unsigned int i=start_x;i<end_x;i++) {
      int start_read_index=……;
      for (unsigned int c=0;c<slice_size;c++) {
#pragma HLS PIPELINE II=1
         int read_index=start_read_index+x;
         ids.write(u[read_index]);
     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

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!

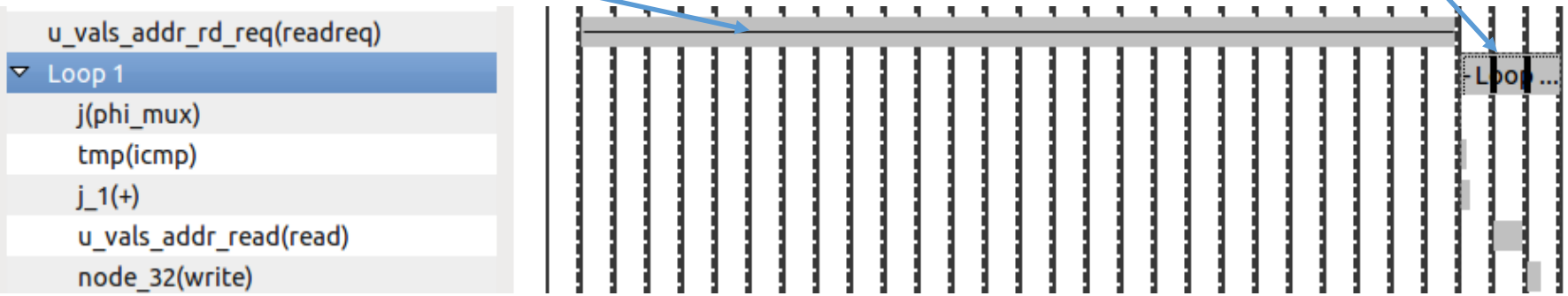
Include X dimension of cube in the dataflow region

```
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

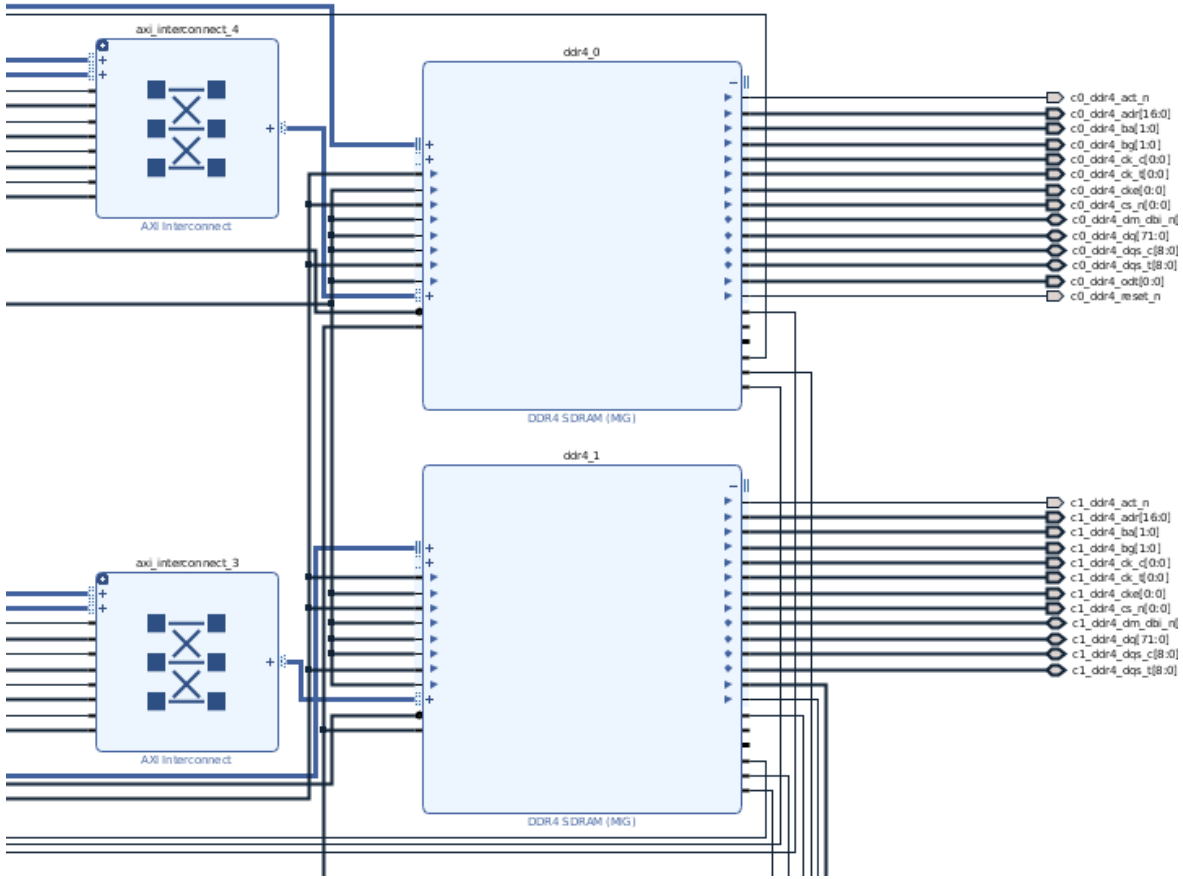
The inner loop is 3 cycles total



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, the DRAM controllers are working at data width of 256 bits
  - Which Alpha Data tell us is optimal for this board
- But our kernels are working 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, 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]);
         }
      }
   }
}
```

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)
Initial version	584.65	14%	320.82	80.56	173.22
Split out DRAM connected ports	490.98	17%	256.76	80.56	140.65
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	522.34	10%	198.53	53.88	265.43
Include X dimension of cube in the dataflow region (optimised)	163.43	33%	45.65	53.88	59.86
256 bit DRAM connected ports	65.41	82%	3.44	53.88	4.48
256 bit DRAM connected ports issue 4 doubles per cycle	63.49	85%	2.72	53.88	3.60

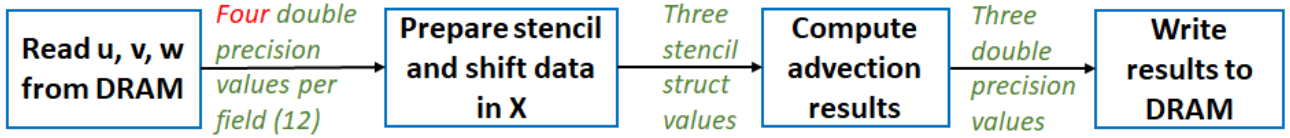
256 bit DRAM connected ports issue 4 doubles per cycle

```
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, double*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

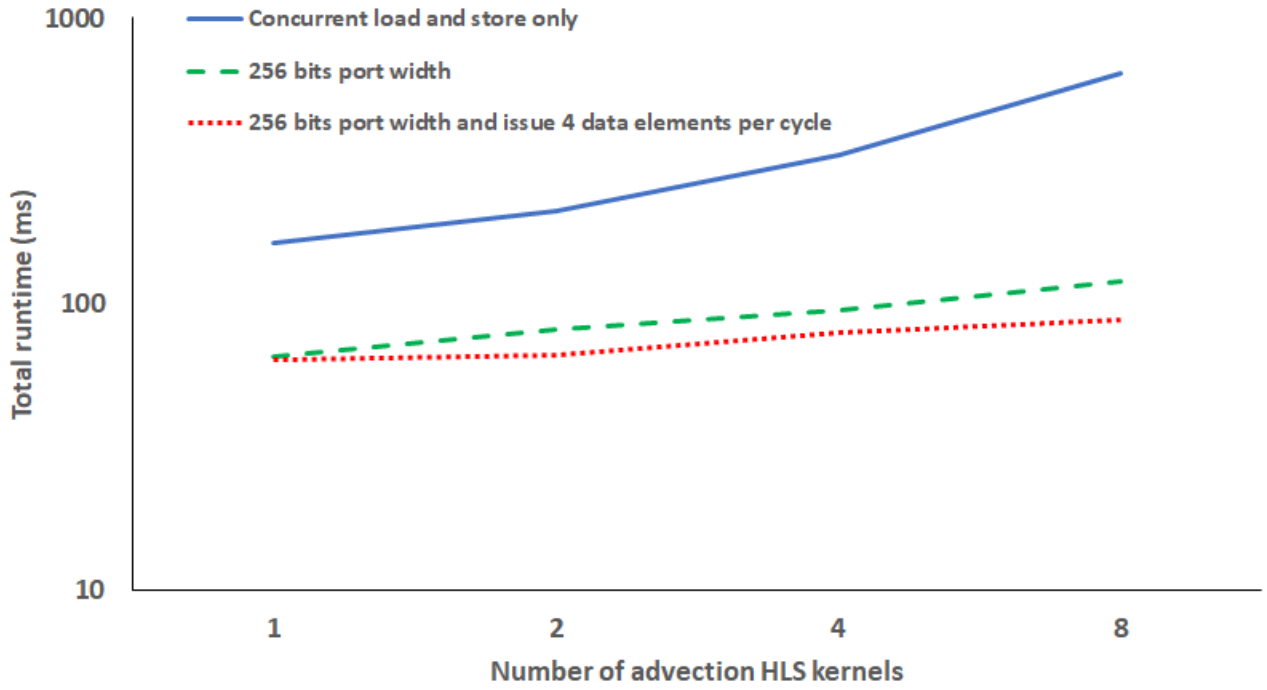
- Effectively, once the pipeline is filled, every cycle we are loading 4 doubles per field into our FIFO queues




```
void perform_advection(double * u) {
 for (unsigned int m=start_y;m<end_y;m+=BLOCKSIZE_IN_Y) {
    for (unsigned int i=start_x;i<end_x;i++) {
      static hls::stream<double> data_stream_u[4];
#pragma HLS STREAM variable=data_stream_u depth=16
      static hls::stream<double> in_data_stream_u[4];
#pragma HLS STREAM variable=in_data_stream_u depth=16
      static hls::stream<struct u_stencil> u_stencil_stream;
#pragma HLS STREAM variable=u_stencil_stream depth=16

#pragma HLS DATAFLOW
        ...
} } }
```

4 double precision values per cycle?

- This means there are FIFO queues of width 64 double values (4 by 16 depth) between the first and second pipeline stages, and the third and fourth.
  - The second stage is still only consuming at a rate of one value per cycle
  - As such, does this provide a buffer against contention on the DRAM, as if loading stalls then there will be plenty of values in the FIFO queues
  - And when access resumes, then the queues will quickly refill based on 4 values per cycle being loaded

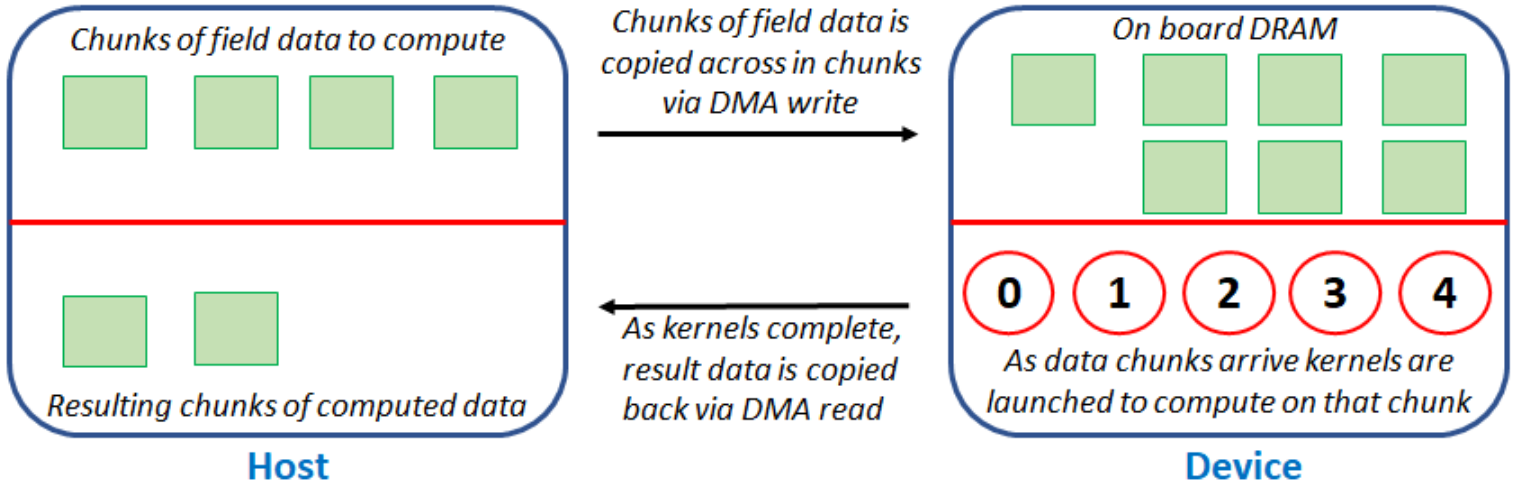


Aggregate HLS kernel only (no DMA transfer) time for problem size of 16.7 million grid points (strong scaling)

Addressing DMA transfer

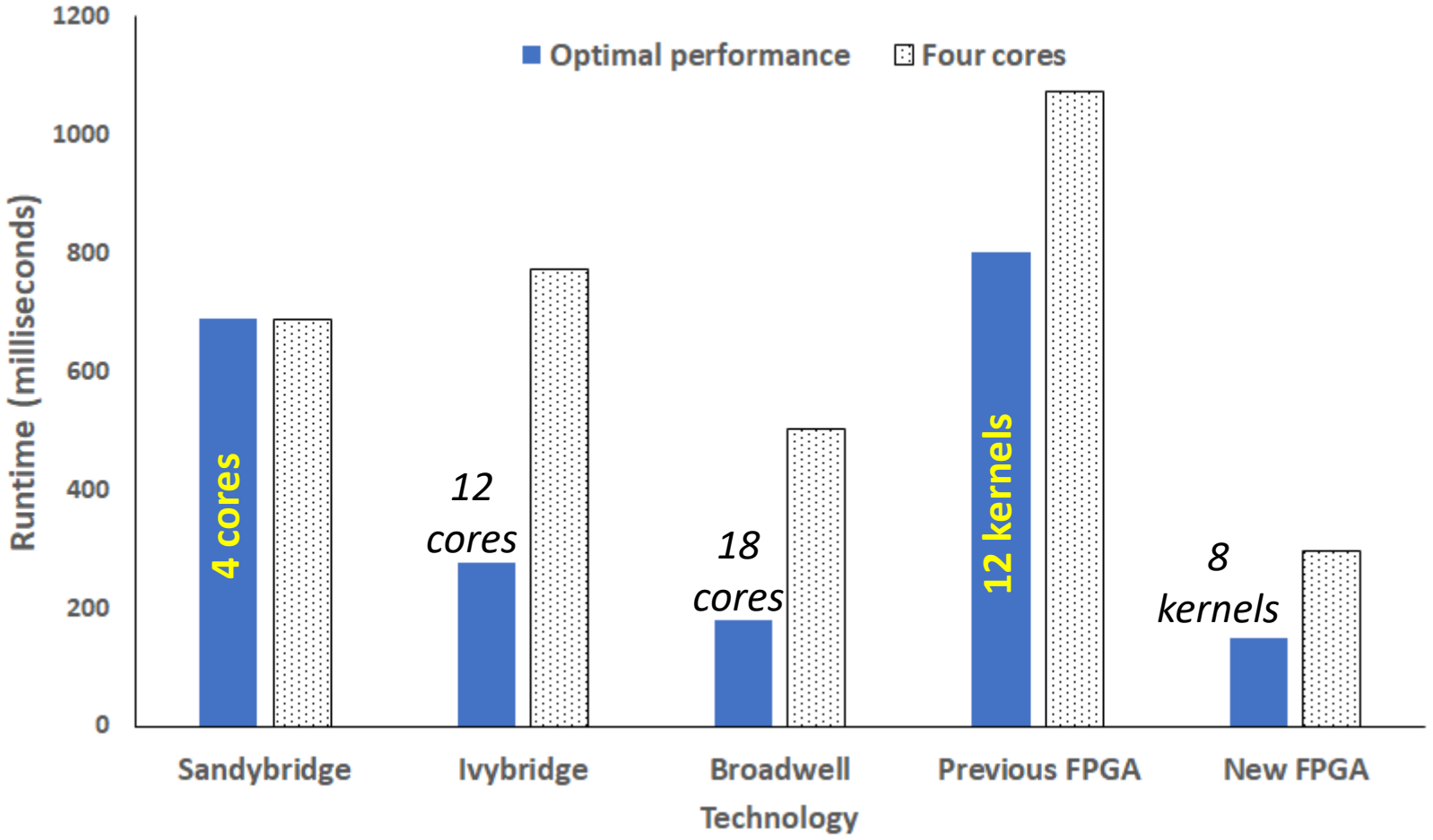
- Previously we waited for all PCIe data transfer to complete, and then kernels were started based on a static decomposition. Only once all computation was completed did results get transferred back
  - DMA was responsible for over 70% of the runtime!

- 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

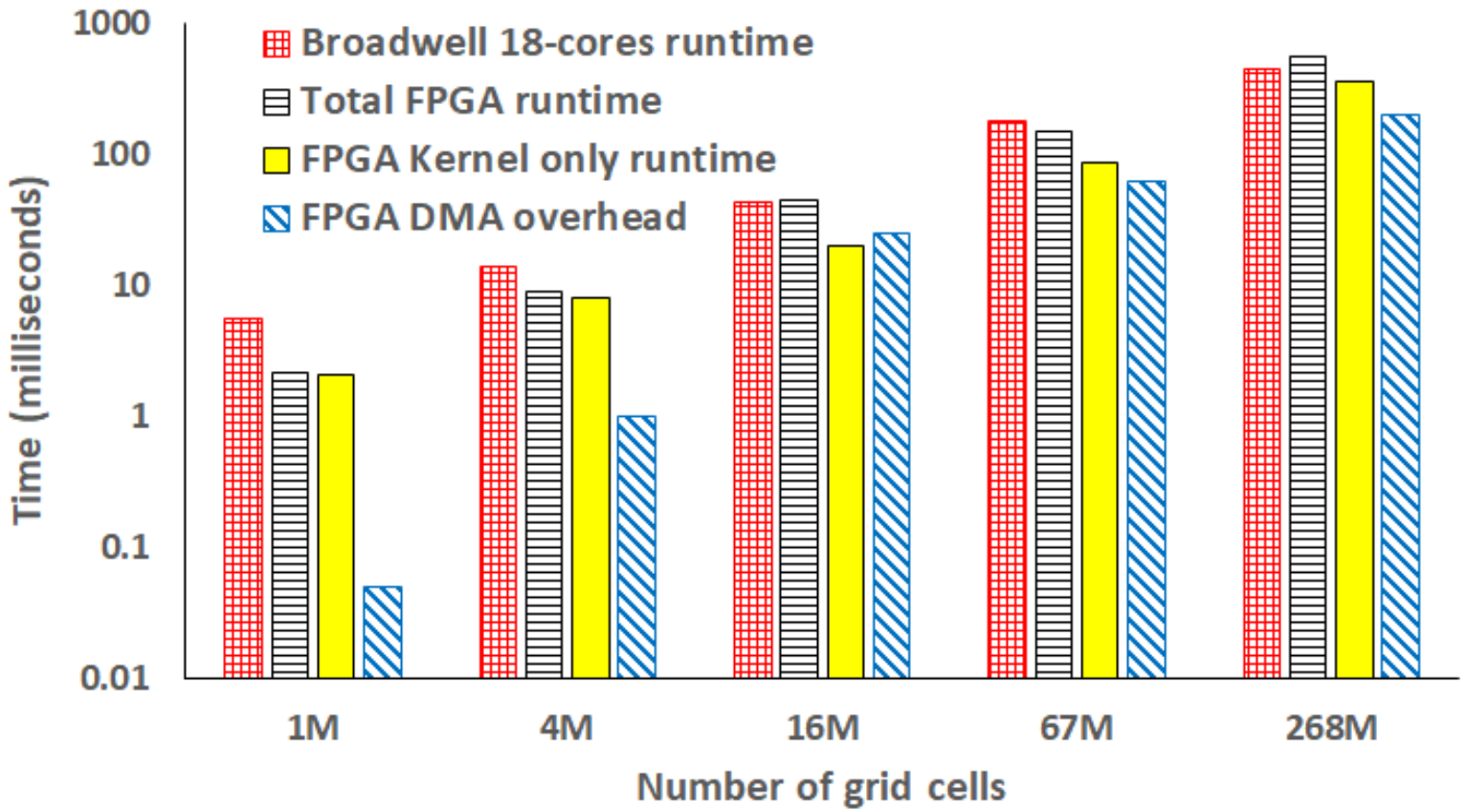


Full performance comparison

- 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



- 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

GFLOP/s

Grid size	FPGA Kernel GFLOP/s	Total FPGA GFLOP/s	Broadwell GFLOP/s
1M	25.2	24.7	9.5
4M	26.5	23.6	15.4
16M	42.4	18.8	19.6
67M	39.4	22.9	18.8
268M	38.1	24.4	30.2

- FPGA draws 28.9 Watts idle and 35.7 Watts under load
  - Vivado estimates power draw to be 23 Watts
- Don’t have power measurement fitted to the Broadwell, but TDP is 120 Watts

Conclusions and further work

- Data movement is another example of *having to think dataflow*
  - Tempting to focus on precision of operations, but if the computation is only responsible for a small amount of the overall runtime then that's going to have limited impact.
  - Critically important for us to have a rich profiling environment enabling detailed performance analysis of kernels.

- High Bandwidth Memory (HBM) would be very interesting to explore to see if we can increase our 85% of time in compute even further
- Further developing our DMA streaming approach to be driven more by the FPGA rather than the host explicitly starting kernels
- Detailed power analysis and comparison on the CPU