Exploring the acceleration of the Met Office NERC Cloud model using FPGAs

Nick Brown, EPCC

n.brown@epcc.ed.ac.uk

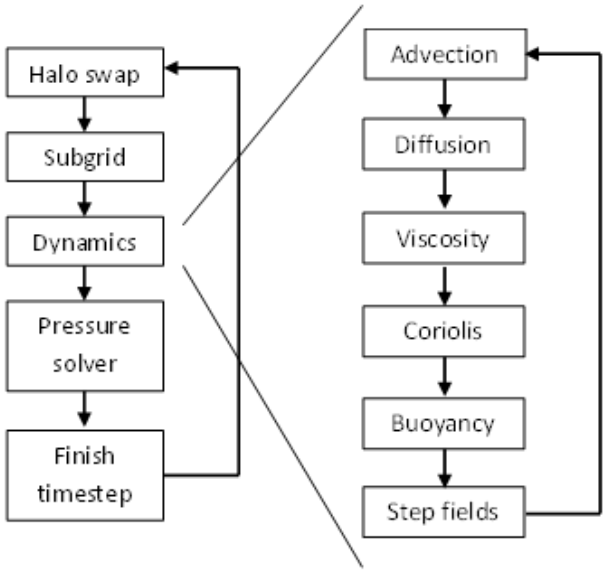
Met Office NERC Cloud (MONC) model

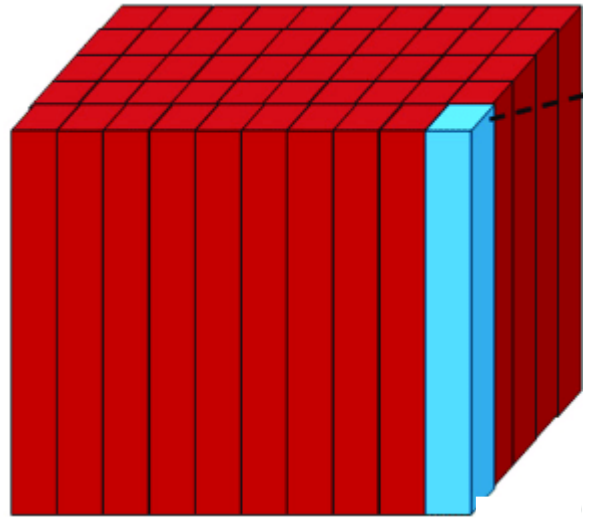
- MONC is a model we developed with the Met Office for simulating clouds and atmospheric flows
  - Written in Fortran 2003 and oriented around the concept of plug-ins.
  - A model core is provided which contains general utility functionality but all science and parallelism is provided by independent, separate components

- Runs on much larger domains (billions of grid points) than previous generations of models
  - Unlocks the potential to explore the atmosphere at scales that were unobtainable before

MONC acceleration on GPUs

- Advection is the most computationally intensive part of the code at around 40% runtime
  - Part of the dynamics group of components
  - Stencil based code

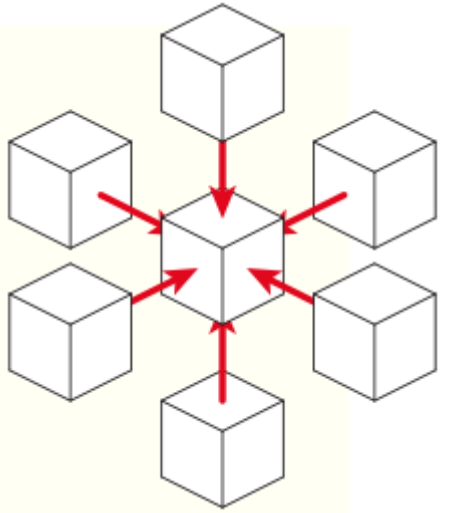




- Components in the dynamics group contribute their calculations to source terms
  - Columns are tightly coupled
  - However within a timestep each column is independent from every other column
  - Each activity generates source terms

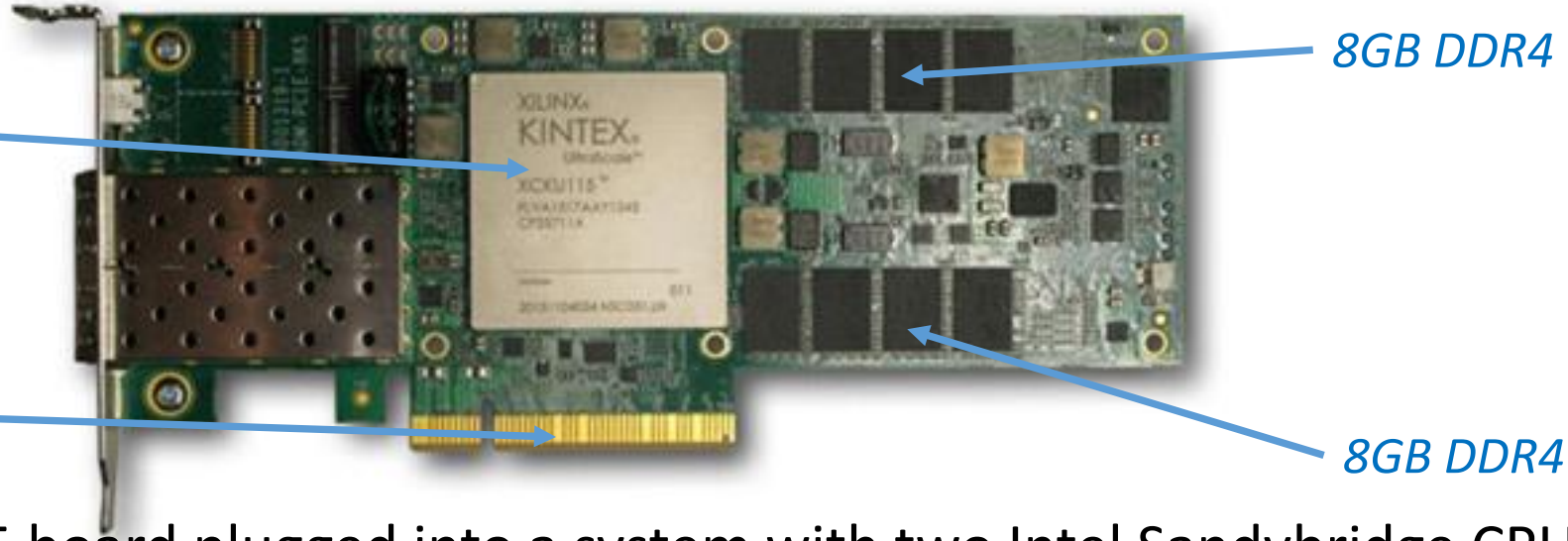
Advection code sketch

```
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
```



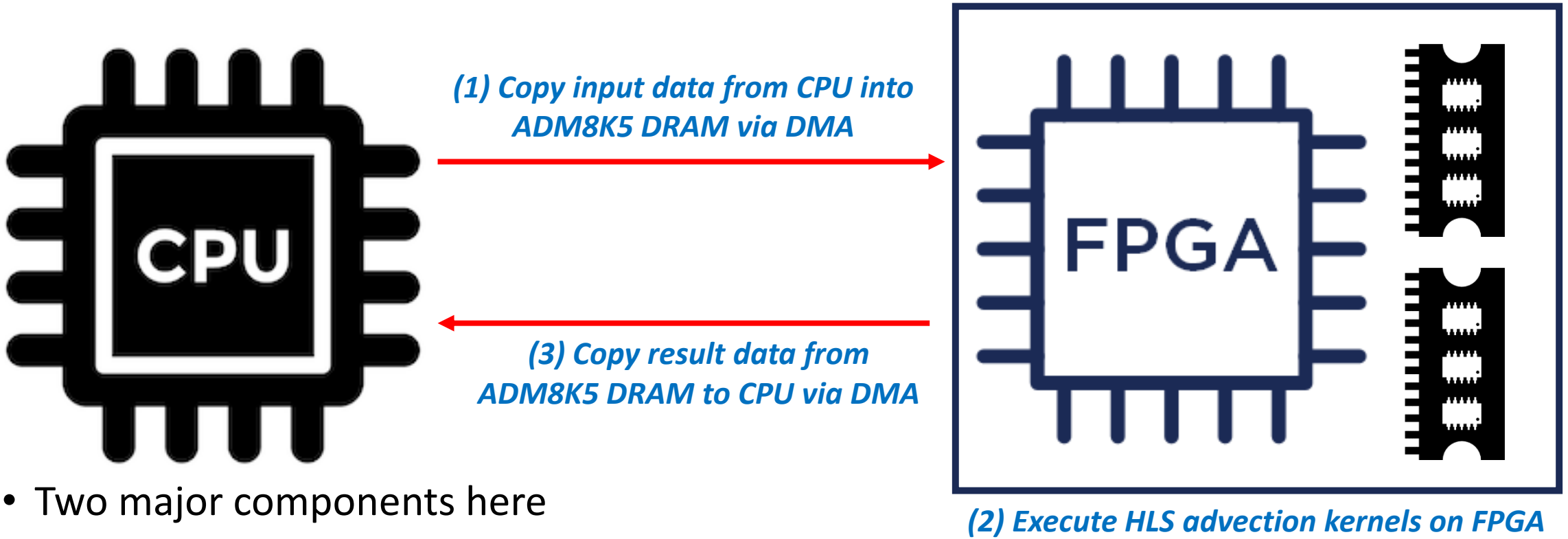
- In all, 53 double precision floating point operations over advecting u v, and w fields
  - 32 double precision floating point multiplications, 21 floating point additions or subtractions

Experiment set-up



- AlphaData's ADM8K5 board plugged into a system with two Intel Sandybridge CPUs (4 cores each)
  - Intel CPU system using a Xilinx FPGA
- Following Xilinx's UltraFast High-Level Productivity Design Methodology
  - Write the kernel(s) using HLS which generates the RTL
  - Use the block design in Vivado to hook all components together

High level workflow



- Two major components here
  - HLS kernels
  - DMA transfers

High Level Synthesis advection kernel

- Using the High Level Synthesis tool synthesise the kernel and export IP block

Description	Runtime (ms)	LUT usage	DSP48E usage	BRAM-18k usage
On Sandybridge CPU	676.4	N/A	N/A	N/A
Initial port	51498	9743	85	0
Pipeline directive on inner loop	14130	11356	58	64
Local BRAM for column data	3213.2	27598	267	130
Local BRAM batches columns in Y	1513.2	37474	393	453
Extract all variables	1301.6	38393	469	312
Burst mode on port	1097.2	40913	469	324
Re-order X and Y loops	621.3	41151	469	324
Replace memcpy with explicit loops	568.1	40638	466	324
Tune double precision cores and clock to 310Mhz	514.9	27601	406	324

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
  - Runtime of 51 seconds ☹
  - Vs 0.676 seconds on CPU!

- Pipeline the inner loop with initiation interval of one
  - Decreases runtime five times 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 (int i=start_x;i<end_x;i++) {
  for (int j=start_y;j<end_y;j++) {
    for (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)));
      ……..
    }
  }
}
```

High Level Synthesis kernel

```
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
      .....
    }
  }
}
```

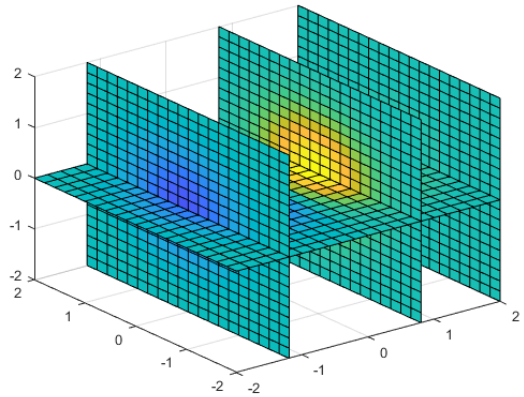
- Use local BRAM to hold data required for working with a single column
  - In all twenty two arrays created
  - Copy all data required for a column from the external data ports, then process the column
  - MAX_VERTICAL_SIZE is required as an not dynamically size these in HLS
  - Either single or dual ported, but more than 2 accesses can be needed at any time – hence duplicate these out (e.g. u_vals and u_vals2)

- Sped up by a further four times (3.2 s), trebling LUT usage, five times usage of DSP slices and over doubling BRAM usage
  - 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 57 of these cycles (28%) is the pipeline full utilised ☹

High Level Synthesis kernel

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

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

- Feed the pipeline by Y_BATCH_SIZE of columns now
- HLS also reported was able to reduce the II down to 1
- Assuming a Y BATCH SIZE of 64 and column size of 64, the pipeline now runs for 4167 cycles, 97% of which the pipeline is fully filled

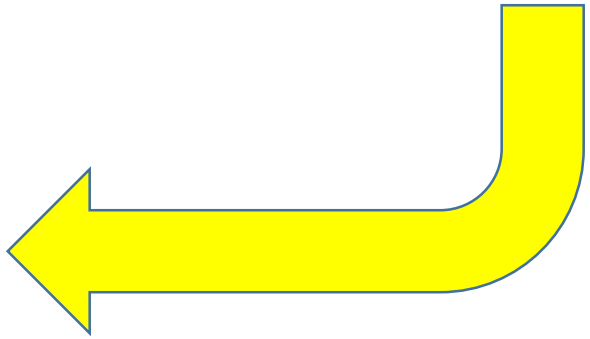


- Halved the runtime, doubled BRAM usage and increased LUT and DSP count
  - 1.5 seconds, vs 0.67 seconds on Sandybridge

High Level Synthesis kernel

```
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);
```



- Found that HLS does a fairly poor job of identifying which variables are shared and hence can be reused between calculations.
  - So extracted them all out manually into temporary variables and reusing these where possible
  - Reduced pipeline depth from 71 to 65 cycles
  - Most importantly reduced number of reads on local arrays, reducing number of local arrays by 30%
    - 1.3 seconds runtime (0.2 seconds improvement)

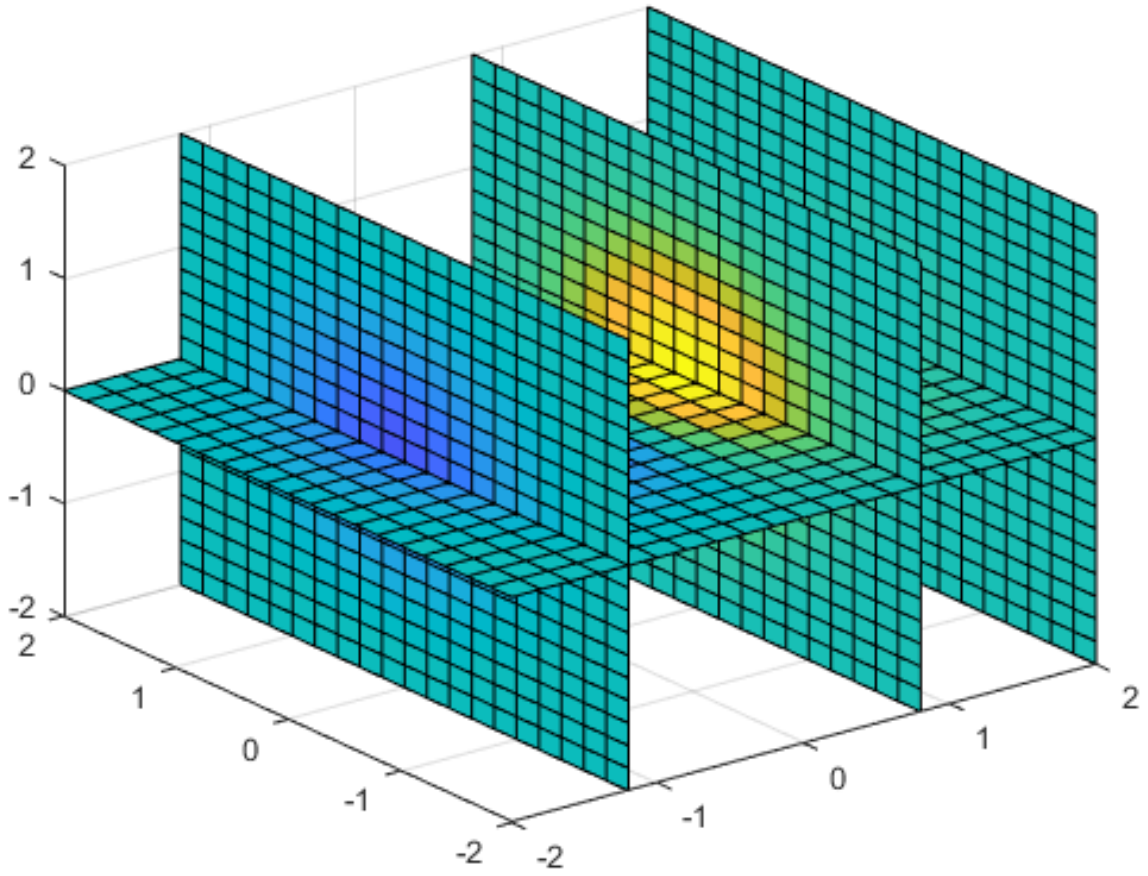
High Level Synthesis kernel


```
int pw_advection(double * u, double * su, ..., int size_x, int size_y, ...) {

  #pragma HLS INTERFACE m_axi latency=60 port=su offset=direct num_read_outstanding=2 num_write_outstanding=2
        max_read_burst_length=256 max_write_burst_length=256

  #pragma HLS INTERFACE m_axi latency=60 port=su offset=direct num_read_outstanding=2 num_write_outstanding=2
        max_read_burst_length=256 max_write_burst_length=256

  .....
}
```

- Instead of issueing a single access for each individual data access, retrieve data in bursts
  - *max_read_burst_length* being the amount of data to retrieve in each burst
  - *num_read_outstanding* being the number of bursts that can be in progress/stored at a point in time
  - *latency* is the number of cycles before data is needed that it is fetched (effectively pre-fetching)

- All these bursts are stored in BRAM, slight increase in BRAM usage but runtime down to 1.1 seconds

High Level Synthesis kernel



- Looping through the outer loop (*i*) is very expensive as data must be fetched for the *i*, *i-1*, *i+1* location
- But crucially the *i-1* and *i* data has been fetched for the previously iteration of the outer loop
  - So we are naively fetching much more data that's needed, actually if we recast the algorithm then on each iteration of the outer loop we only need to fetch the *i+1* iteration

High Level Synthesis kernel

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

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

- Looping through the outer loop (*i*) is very expensive as data must be fetched for the *i*, *i-1, i+1* location
- But crucially the *i-1* and *i* data has been fetched for the previously iteration of the outer loop
  - So we are naively fetching much more data that's needed, actually if we recast the algorithm then on each iteration of the outer loop we only need to fetch the *i+1* iteration

High Level Synthesis kernel

```
for (unsigned int m=start_y;m<end_y;m+= Y_BATCH_SIZE) {
  memcpy(up1_vals, &u(start_x,m,0), sizeof(double) * size_z * number_in_y);
  ....
  for (unsigned int i=start_x;i<end_x;i++) {
    memcpy(u_vals, up1_vals, sizeof(double) * size_z * number_in_y);
    memcpy(up1_vals, &u(i+1,m,0), sizeof(double) * size_z * number_in_y);
    ....
    for (unsigned int j=0;j<number_in_y;j++) {
      ....
    }
  }
}
```

- Swap outer and second loop to remove two redundant memory accesses for each *i* iteration by pipelining them
  - Halves runtime of kernel to 0.62 seconds – first time we have beaten the Sandybridge CPU ☺

- Replace *memcpy* call with explicit loop, allows us to run these concurrently
  - Can pop multiple assignments in same loop
  - Slight runtime decrease to 0.57 seconds and slight LUT usage decrease

```
for (unsigned int cpy_idx=0;cpy_idx<size_z*number_in_y;cpy_idx++) {
  #pragma HLS PIPELINE II=1
  u_vals[cpy_idx]=up1_vals[cpy_idx];
}
```

High Level Synthesis kernel

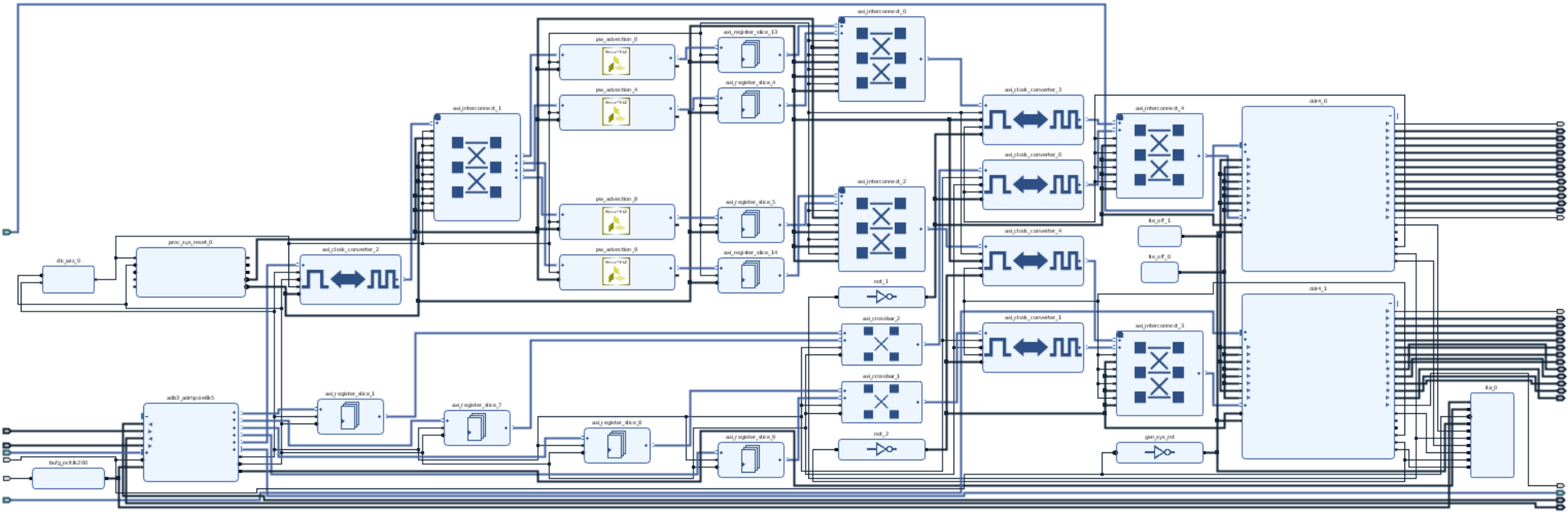
```
unsigned int jk_index=(size_z * j) + k;
double u_data=u_vals[jk_index];
double um1_data=um1_vals[jk_index];
double up1_data=up1_vals[jk_index];

#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
  - This increase of pipeline also reduced the LUT usage
  - 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.
  - 0.51 seconds runtime

High Level Synthesis advection kernel

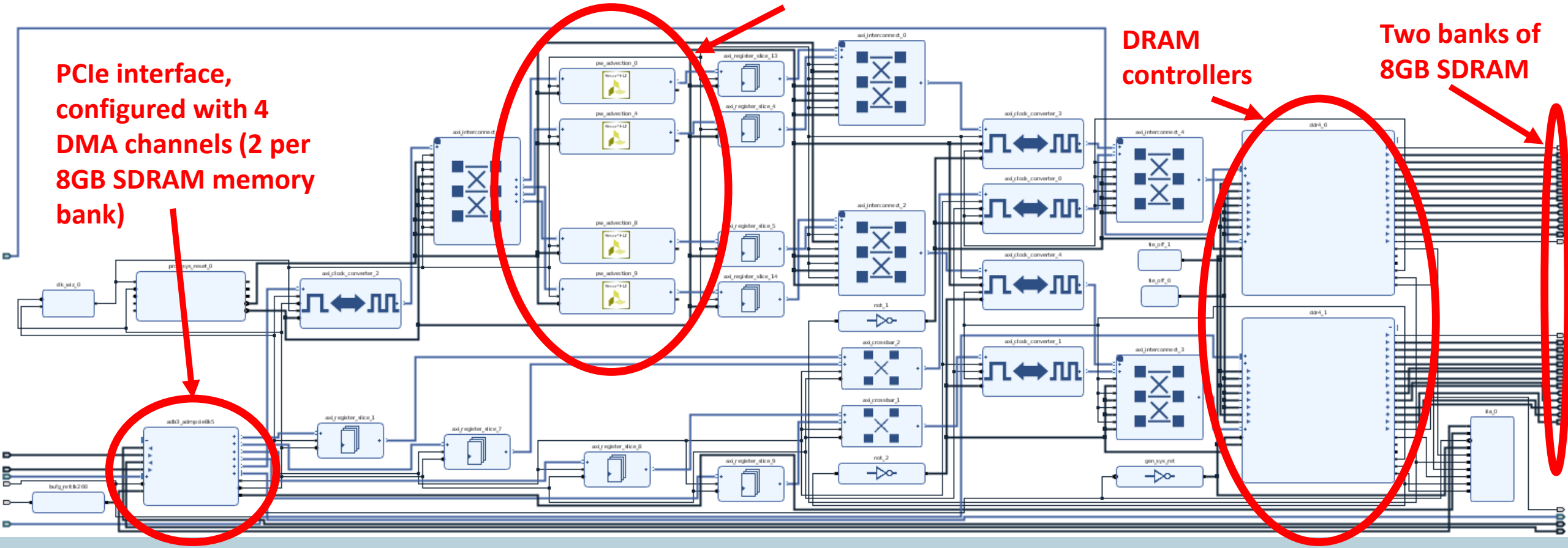
Description	Runtime (ms)	LUT usage	DSP48E usage	BRAM-18k usage
On Sandybridge CPU	676.4	N/A	N/A	N/A
Initial port	51498	9743	85	0
Pipeline directive on inner loop	14130	11356	58	64
Local BRAM for column data	3213.2	27598	267	130
Local BRAM batches columns in Y	1513.2	37474	393	453
Extract all variables	1301.6	38393	469	312
Burst mode on port	1097.2	40913	469	324
Re-order X and Y loops	621.3	41151	469	324
Replace memcpy with explicit loops	568.1	40638	466	324
Tune double precision cores and clock to 310Mhz	514.9	27601	406	324

The block design



- With 12 HLS kernels we use 78.5% of LUTs, 84.2% of BRAM-18k blocks and 89% of DSP48E slices

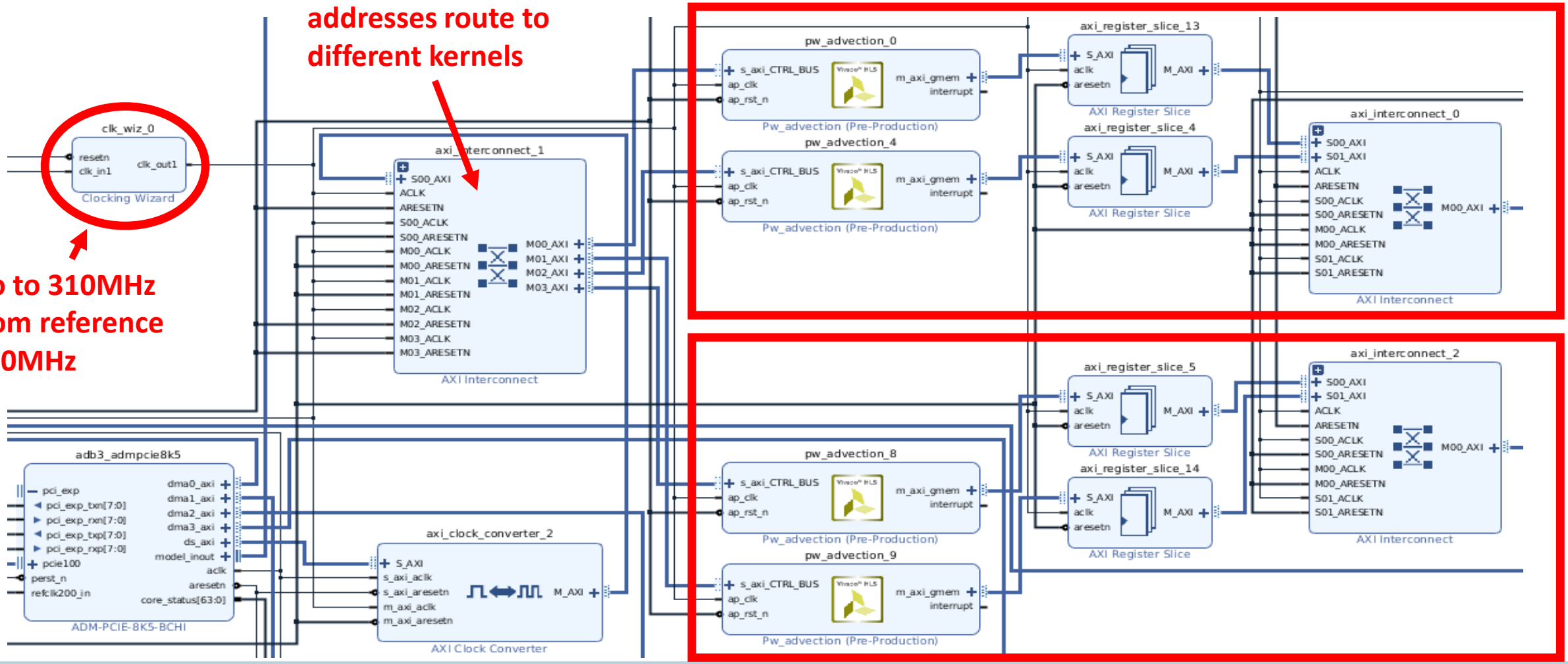
The block design



Zooming in……..

Separate memory addresses route to different kernels

Go to 310MHz from reference 250MHz

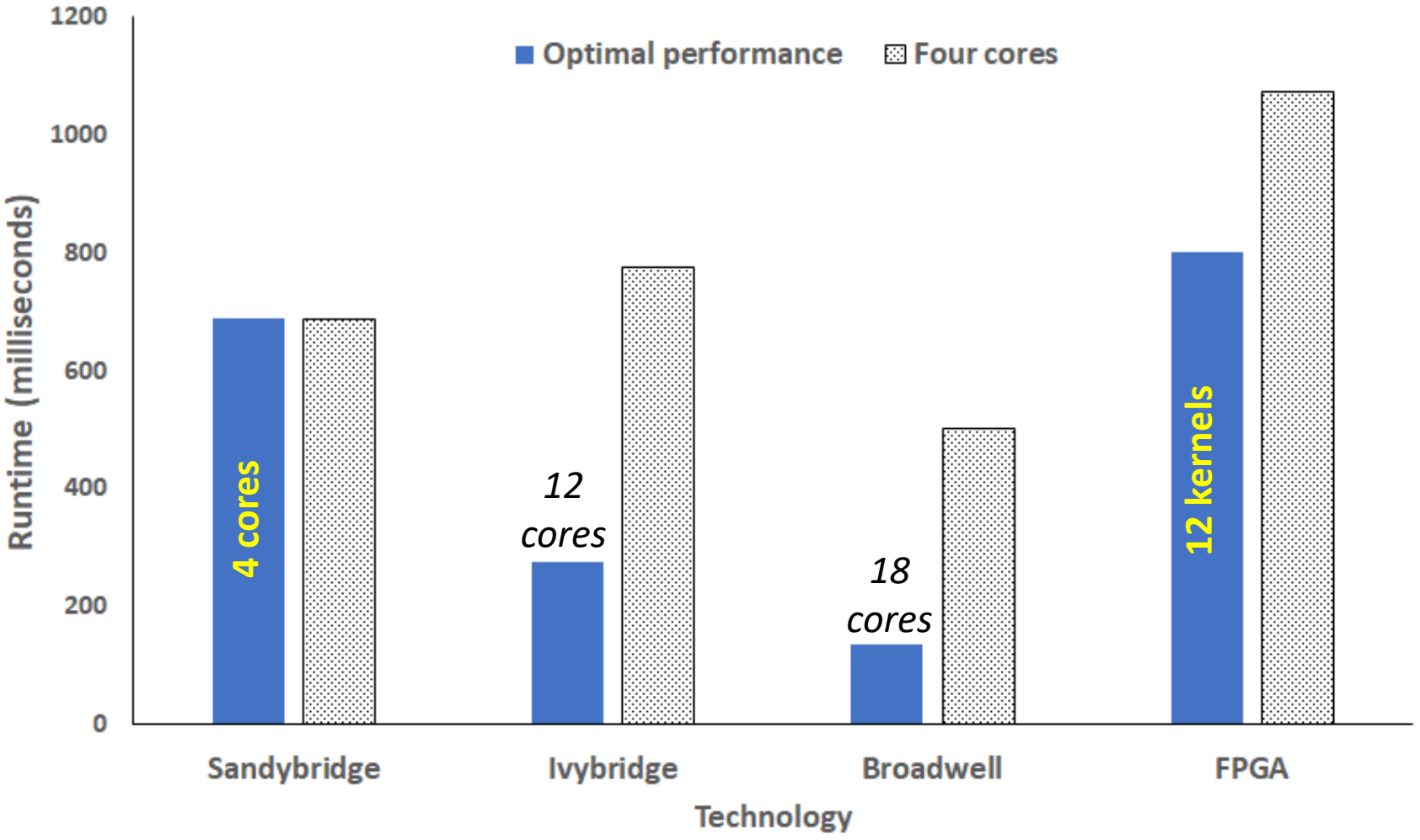


To tie the DRAM together or not…….

Description	DMA transfer time for 1.6 GB (ms)
Our design (two memory controllers, split apart)	232
One memory controller only	280
Two memory controllers unified memory space	239
One DMA channel per memory controller	242

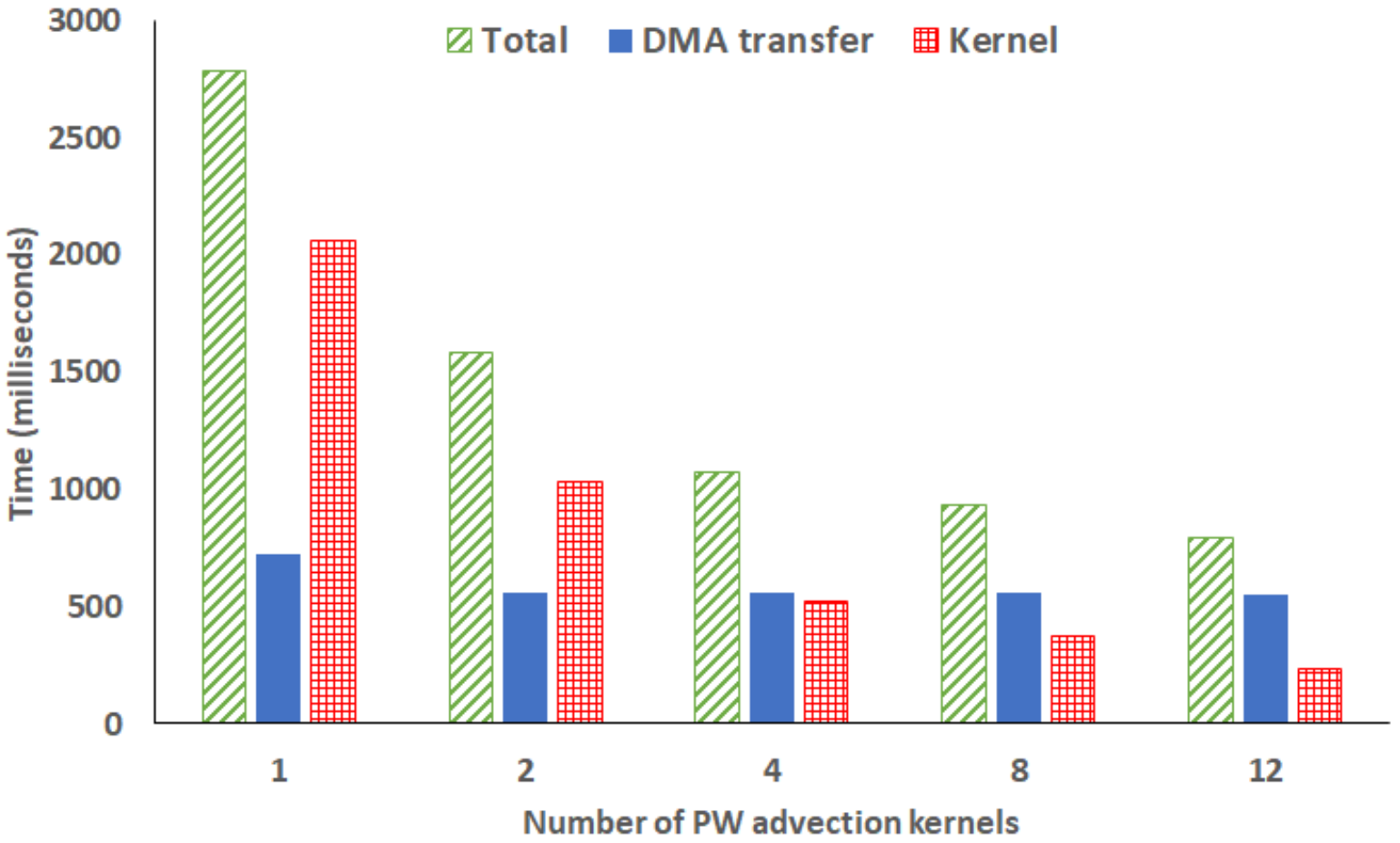
- Have two banks of 8GB, so have a choice whether expose all in a single unified memory space or split up into two banks
- Also have four DMA channels, currently connect two to each bank and run transfers concurrently
  - But these route into a single memory controller, so does this make any difference?

Performance of FPGA advection kernel



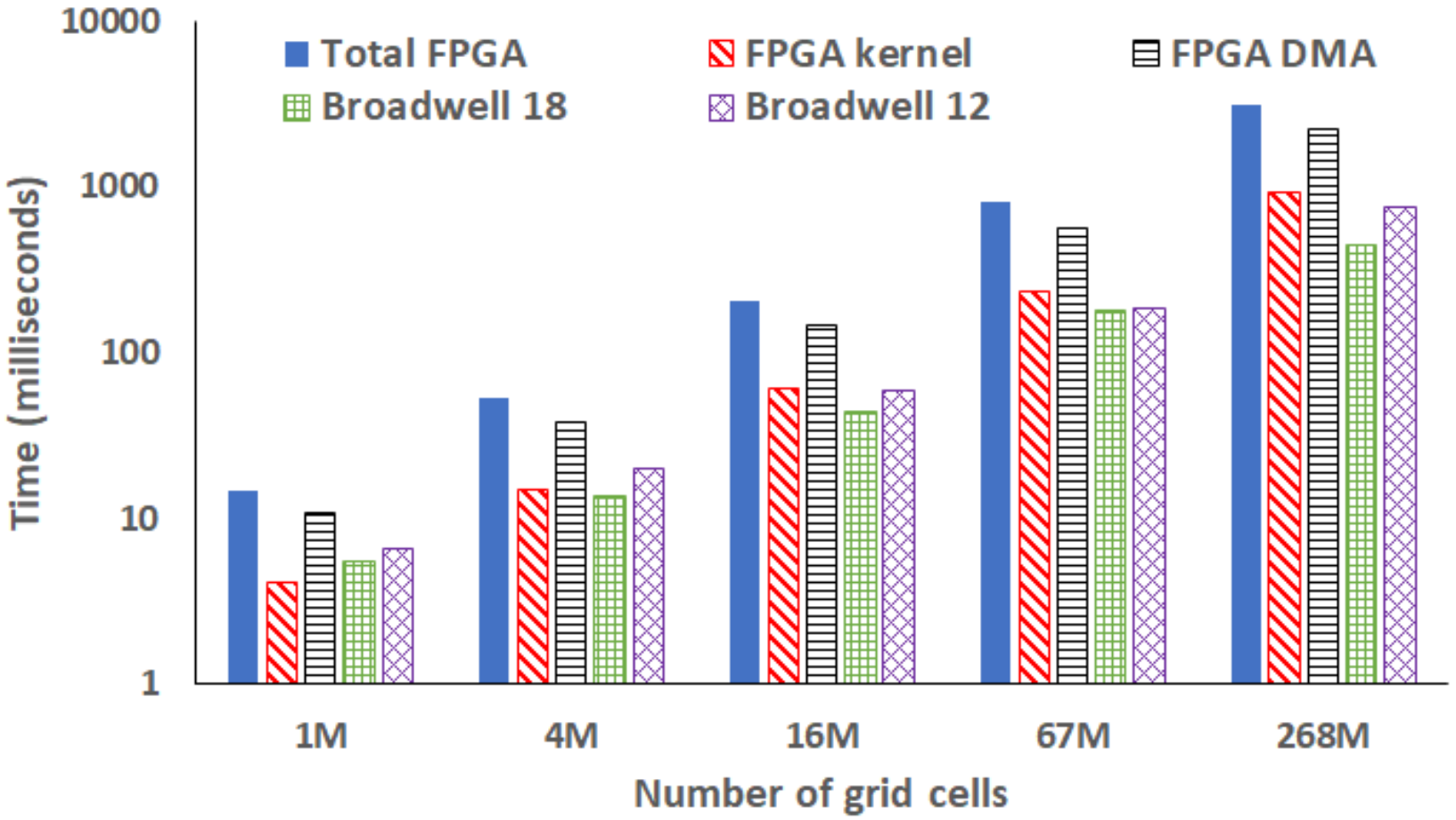
- Four core Sandybridge, 12 core Ivybridge, 18 core Broadwell and 12 advection kernel FPGA
- Standard status cloud test-case 67 million grid points (x=1012, y=1024, z=64)
- Two components to the overall runtime
  - HLS kernel execution time
  - DMA transfer time (input data on, results off) required each timestep

Performance of FPGA advection kernel



- Standard status cloud test-case 67 million grid points (x=1012, y=1024, z=64)
- 4 kernels and beyond, the dominant cost is that of data transfer, not calculation!
  - Infact, if we removed the cost of DMA at 12 kernels then the execution time would be less than a third

Scaling of grid (problem) size



- At 268 million grid cells:
  - FPGA kernel alone: 14.36 GFLOP/s
  - Full FPGA: 4.2 GFLOP/s
  - 12-core Broadwell: 17.75 GFLOP/s
  - 12.88GB being transferred, takes 2.2 seconds rate of 5.85 GB/s

Conclusions and further work

- FPGAs are very promising, but the devil is in the detail!
  - Algorithm requires significant refactoring to get good performance via HLS & code looks very different from its sequential counterpart.
  - Currently Intel CPU systems outperform our FPGA implementation, but there are plenty of opportunities for optimisation!

- DMA is clearly a target for optimisation
  - Can not stream directly to HLS kernel, but we believe that we could stream into an IP block and automatically start kernels when the required data has arrived.
- HLS profiling needed
  - We should be getting better performance from our IP block than we are. We believe that this is due to memory bottlenecks, but there isn't the mechanisms in place to collect hardware counter information
  - Hooking this up to an AXI timer at the moment as a basis for this.
- Moving away from FP64
  - Single or half precision (less data, more kernels, higher clock rate), fixed point instead of floating point