Parallel Design Patterns-L14

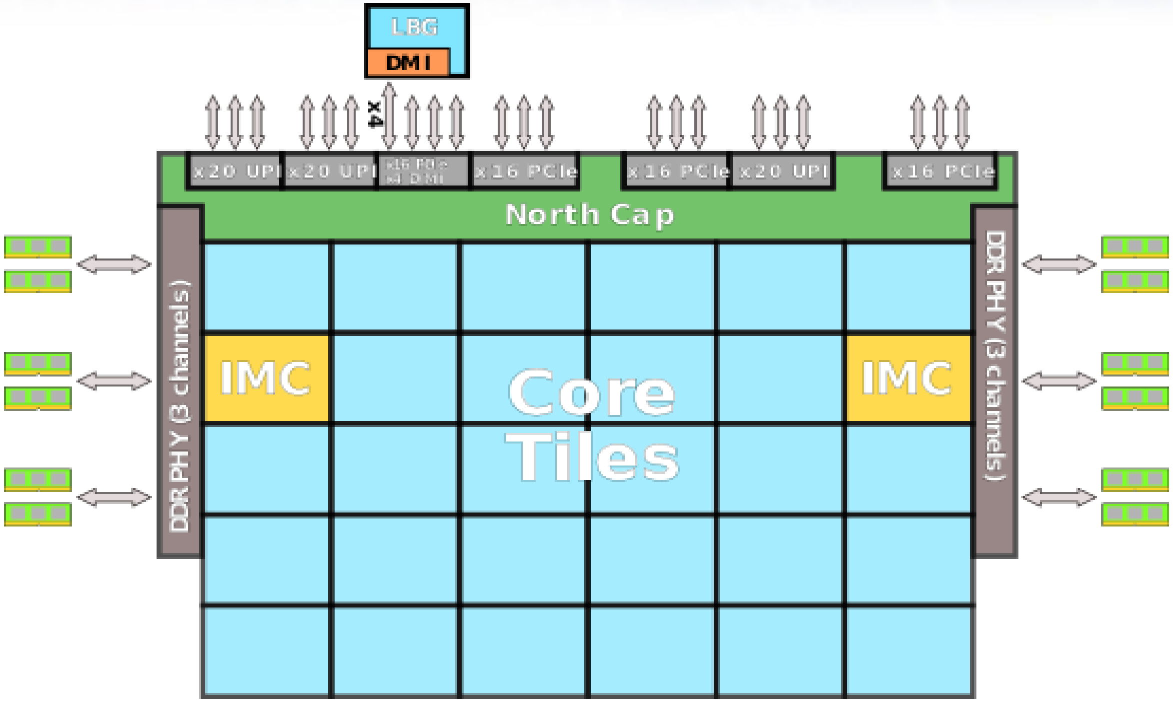
Memory bound codes

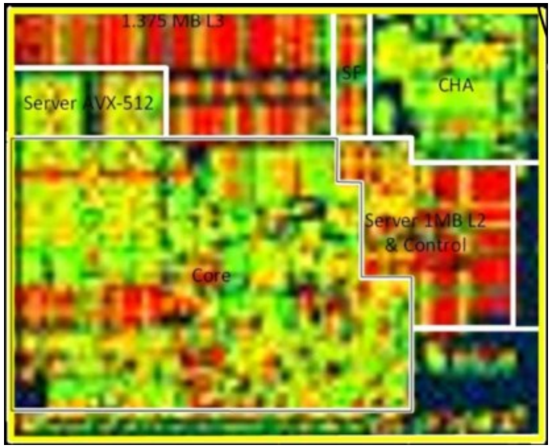
Spatial computing

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes room 2.13

The challenge with modern CPUs

- We have lots of compute
  - Many cores
  - Each core has significant FP performance

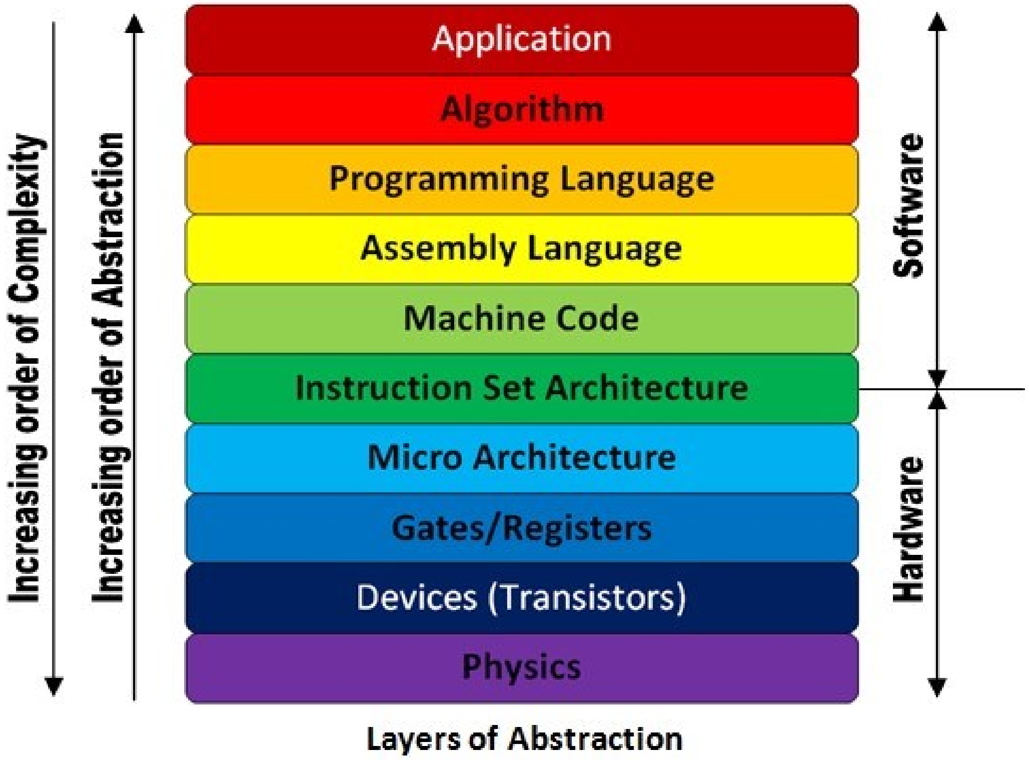




- Crucially the connection to DRAM is much more limited
  - So there is a very large cost in going “off-chip”
  - Caches can help, but there can be overheads with irregular memory access patterns
  - Many HPC codes are thus memory bound

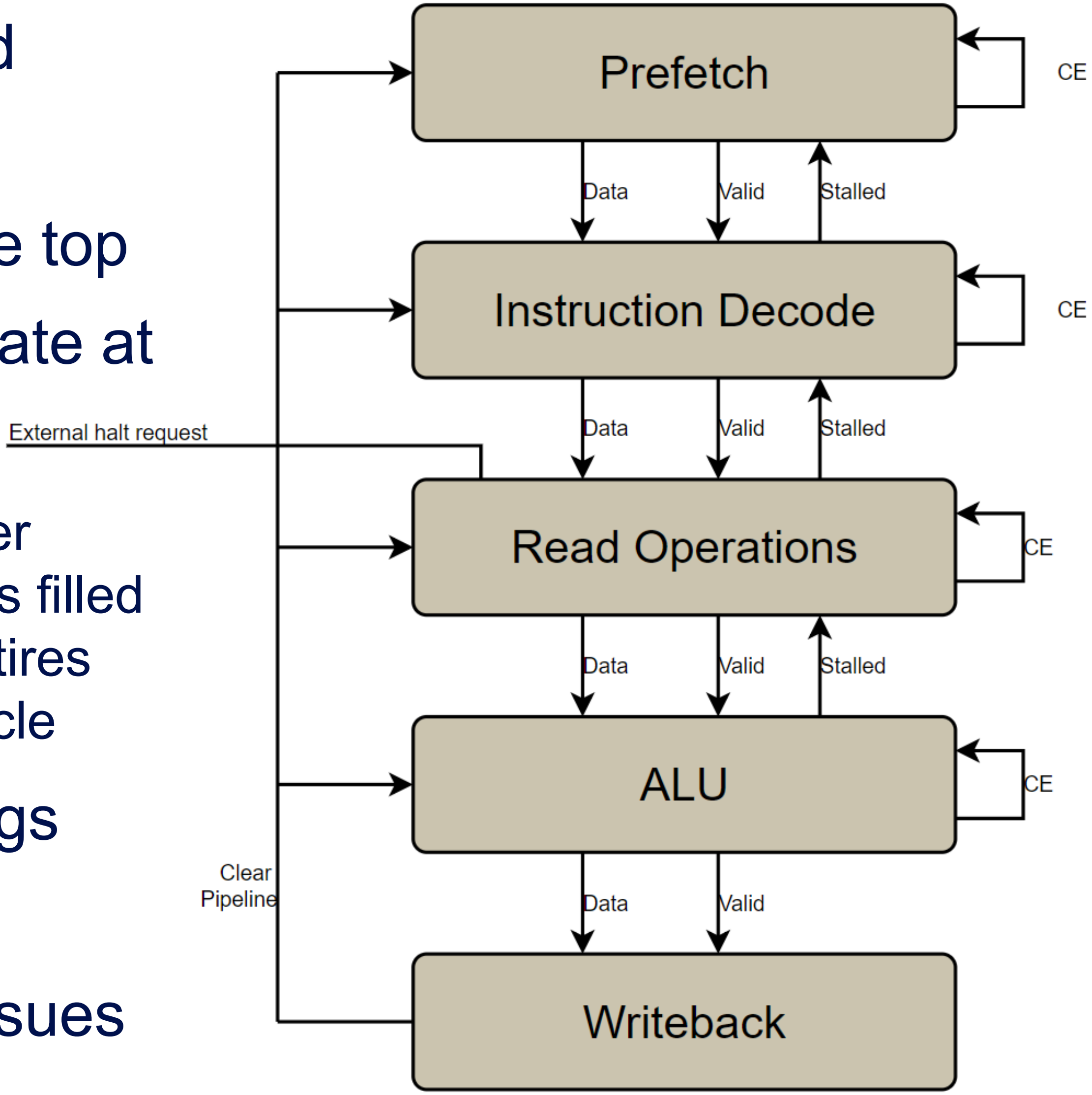
Modern CPU

- CPUs expose themselves via an instruction set architecture (ISA)
  - E.g. x86, x86-64, A32, A64, SPARC, POWER
- A micro-architecture implements the ISA, is largely hidden and can change significantly between generations
  - Useful for the compiler to be aware of this and potentially more advanced programmers

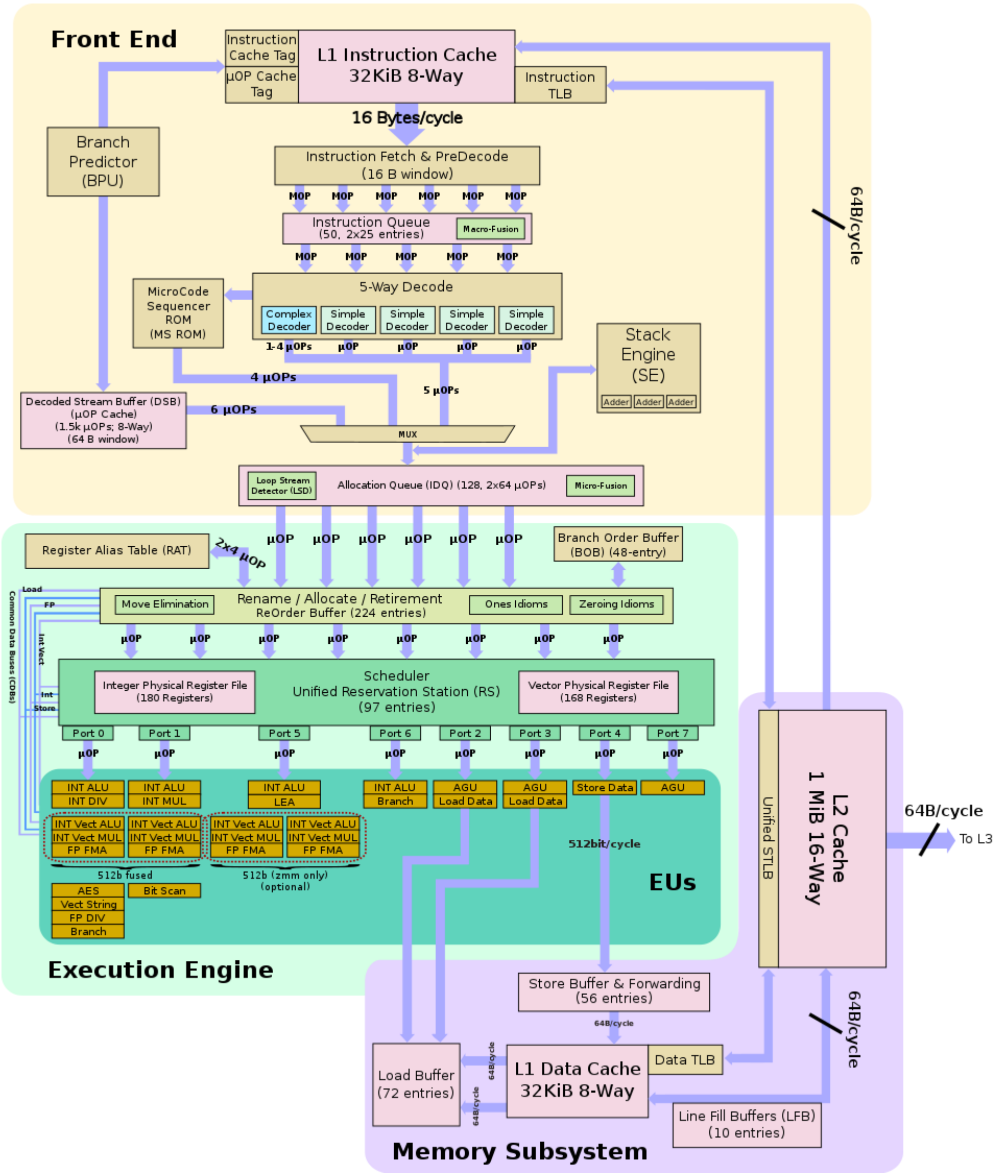


Traditional micro-architecture assumption

- Some sort of pipelined approach where instructions start at the top and are executed a state at a time
  - Assuming one cycle per stage, once pipelined is filled then one instruction retires (i.e. completes) per cycle
- You can see how things like cache misses can cause performance issues here



Out of Order execution

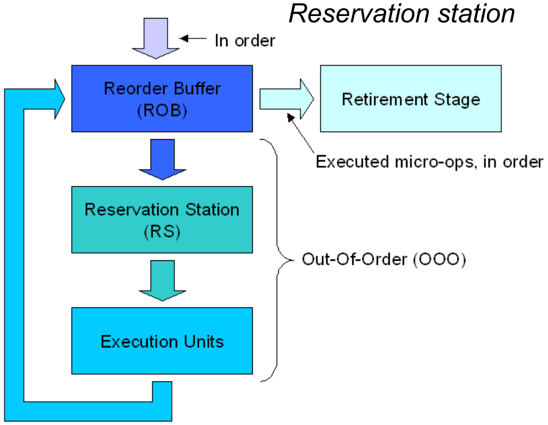


Id	Instruction	Issued	Executed
1	ld F6, 34(R2)	Y	Y
2	ld F2, 45(R3)	Y	
3	multd F0, F2, F4	Y	
4	subd F8, F6, F2	Y	
5	divd F10, F0, F6	Y	
6	addd F6, F8, F2	Y	

Reorder buffer

	EU	Operation	OP1	OP2
	Load1	ld	45(R3)	
✖		multd	(RoB2)	(F4)
✖		subd	(RoB1)	(RoB2)
✖		divd	(RoB3)	(RoB1)
✖		addd	(RoB4)	(RoB2)

Reservation station


In order

Reorder Buffer (ROB)

Retirement Stage

Executed micro-ops, in order

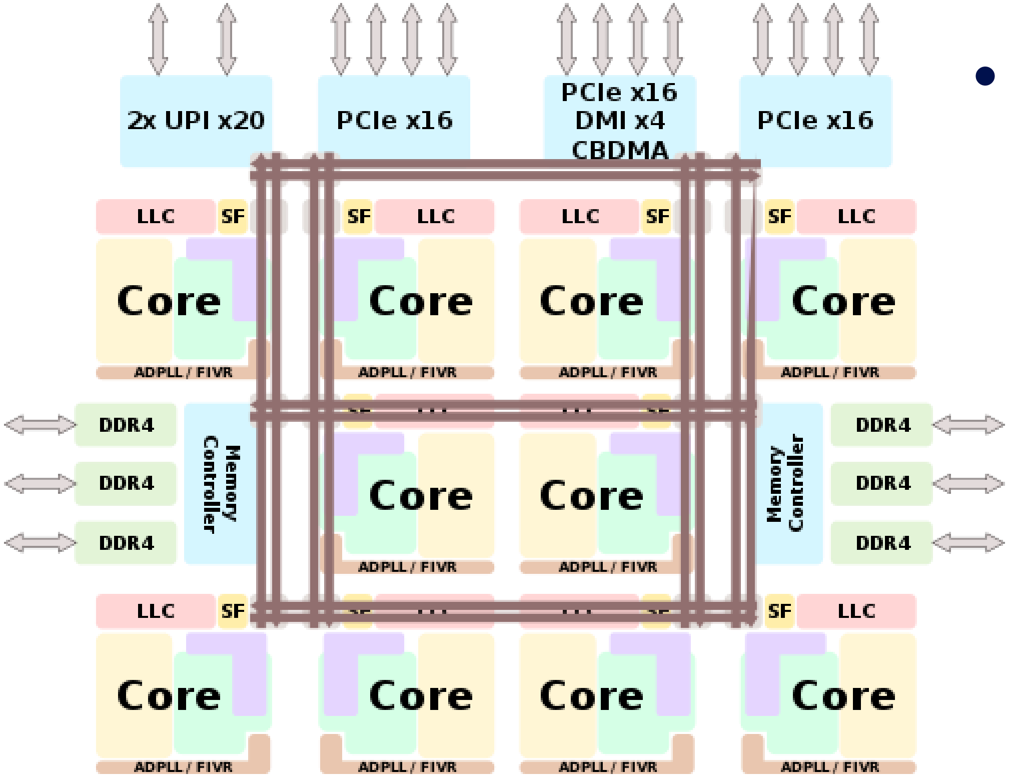
Reservation Station (RS)

Out-Of-Order (OOO)

Execution Units

As programmers, why care?

- The CPU really tries to help you out here
  - CPU caches
  - HW prefetchers that will look ahead and fetch *the next data*
  - Out of order execution to minimise the overhead of memory access



- Therefore missing the cache doesn't always matter as the execution engine might be able to execute other instructions
  - What gets us are delinquent misses, which stall the engine
  - Sometimes limited what we can do about this

Demonstrating this in practice:

- With our practical one serial code (1024 by 1024 to 1e-3)
  - Compiled with GCC, -O3 and run on Skylake-X CPU

Code as provided

```
for(i=0; i<N; i++ ) {
  for (j=0; j<N; j++) {
    u_k[i][j]=u_kp1[i][j]……
  }
}
```

Execution time	15.54 seconds
Number cycles	56.88 billion
Average IPC	2.45
MFLOP/s	5786.58
FPU occupancy	129.32%
Read from DRAM	125999.18 MB
Cache miss reads from DRAM	1.97 MB
% memory ops hit in L1	99.31%
% memory ops hit in L2	0.53%
% memory ops hit in L3	0.03%
% memory ops missed L3	0%
% cycles stalled due to memory	5.98%
Prefetcher hit in the L2 cache	51.39%

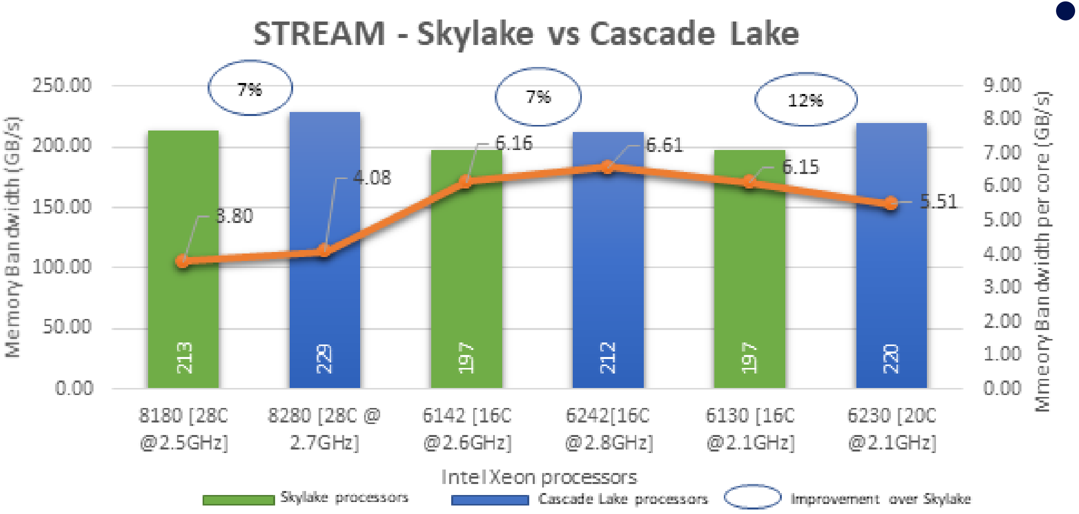
Loop order reversed (bad for cache)

```
for(j=0; j<N; j++ ) {
  for (i=0; i<N; i++) {
    u_k[i][j]=u_kp1[i][j]……
  }
}
```

Execution time	87.72 seconds
Number cycles	322.11 billion
Average IPC	0.59
MFLOP/s	1028.26
FPU occupancy	28%
Read from DRAM	127917.93 MB
Cache miss reads from DRAM	58.35 MB
% memory ops hit in L1	70.5%
% memory ops hit in L2	26.35%
% memory ops hit in L3	1.22%
% memory ops missed L3	2%
% cycles stalled due to memory	58.86%
Prefetcher hit in the L2 cache	29.06%

So what can we do about this?

- The previous example was an extreme one
  - And in this case improving our memory access will significantly help here.
  - But often we can't change our memory access pattern because it is what it is!



- CPU generations and technologies might help
  - But can be heavily dependent on HW improvements, and still leaving performance on the table

- GPUs might help, but still the same fundamental problem
  - Can sometimes hide memory access cost due to swapping warps and faster DDR memory is common-place. But doesn't always help!

Prefetching

- Can compliment or override the HW prefetcher by using SW prefetching via the *_mm_prefetch* function in your code

```
#include <xmmintrin.h>
void _mm_prefetch (char const* p, int i)
```

Pointer to data to prefetch

A hint which will prefetch into L1, L2 or L3

- Some compilers can do this automatically for you
  - I.e. With Intel compiler –opt-prefetch=n , where n is between 1 and 5 (higher being more prefetching)
- E.g. Integer Sort from NASA’s Parallel Benchmark Suite

```
for( i=0; i<NUM_KEYS; i++ ) {
  key = key_array[i];
  key_buff2[bucket_ptrs[key >> shift]++] = key;
}
```

Access to key_array is predictable, but access to bucket_ptrs and key_buff2 is not

Modifying the code to use SW prefetching

```
#include <xmmintrin.h>
………
for( i=0; i<NUM_KEYS+pf1+pf2; i++ ) {
  if (i < NUM_KEYS) {
    key = key_array[i];
    idx=i%tot_distance;
    keys_buffer[idx]=key;
    bucket_idx=key >> shift;
    idx_buffer[idx]=bucket_idx;
    _mm_prefetch(&bucket_ptrs[bucket_idx], _MM_HINT_T0);
  }
```

Prefetch into L1 cache

Work on i-pf1th key

```
  idx=(i-pf1);
  if (idx >= 0 && idx < NUM_KEYS) {
    idx=idx%tot_distance;
    bucket_idx=idx_buffer[idx];
    value=bucket_ptrs[bucket_idx];
    buffer_access[idx]=value++;
    bucket_ptrs[bucket_idx]=value;
    _mm_prefetch(&key_buff2[value], _MM_HINT_T0);
  }
```

Work on i-(pf1+pf2)th key

```
  idx=(i-(pf1+pf2));
  if (idx >= 0) {
    idx=idx % tot_distance;
    key_buff2[buffer_access[idx]]=keys_buffer[idx];
  }
```

Access to key_array is predictable, so that should be HW prefetched. Then do a SW prefetch for bucket_ptrs

pf1 iterations later, use the prefetched value from bucket_ptrs, increment and prefetch key_buff2

pf2 iterations later use the prefetched value from key_buff2 and assign value to this

i
pf1
pf1+pf2

So what should the prefetch distance be?

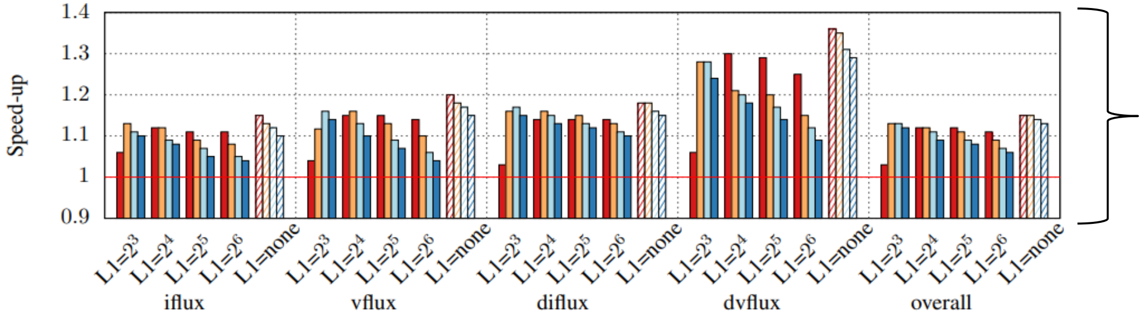
Configuration	Runtime (s)
No prefetching	30.92
Intel compiler flag (n=3)	16.63
Intel compiler flag (n=5)	30.94
Prefetch distance 1	16.63
Prefetch distance 2	14.68
Prefetch distance 4	14.54
Prefetch distance 8	14.55
Prefetch distance 16	14.38
Prefetch distance 24	20.21

Intel compiler 2018, compiled with –O3 and run on Cirrus

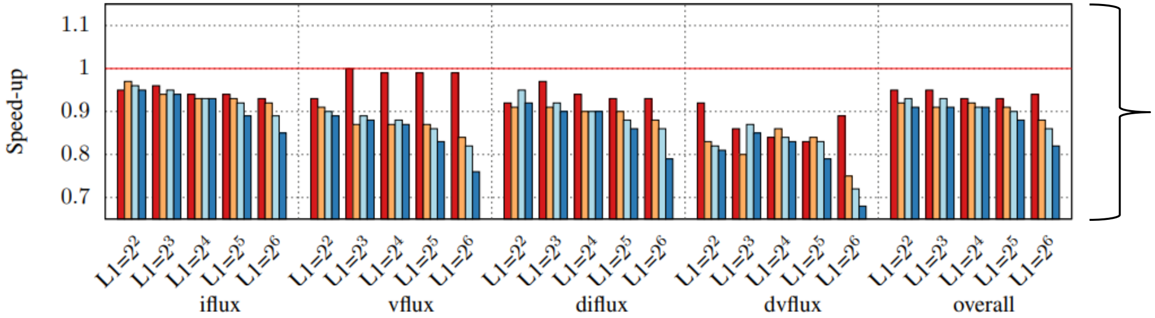
- Compiler based prefetching is reasonable at the correct setting
- But can shave another 2 seconds off with our manual approach
  - But PF distance here is key
  - Adds considerable code complexity and might loose sequential equivalence
- Worth trying, but your results may vary!
  - More overall instructions need to be issued by the CPU
  - Might result in more work for the memory controller

Real-world code example

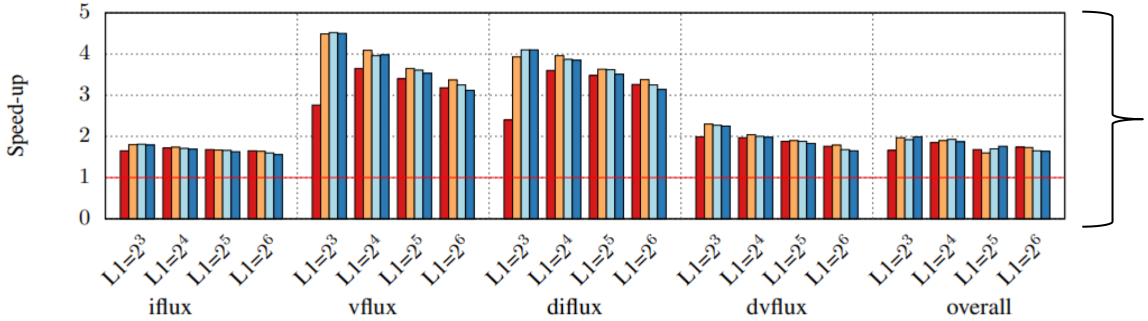
- Prefetching with an unstructured mesh code
  - A paper "*Software prefetching for unstructured mesh applications*" published a couple of years ago by Warwick University



Skylake CPU



Broadwell CPU



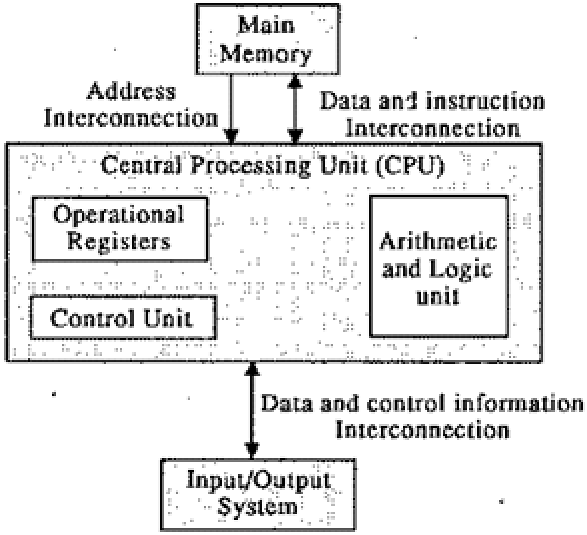
Knights Corner (first version of Xeon Phi)

Advice for us as programmers

- Profile your codes to understand if memory bound
- If they are memory bound then could try a number of things
  - Experiment with different memory layouts (e.g. AoS or SoA)
  - If you have lots of indirect memory accesses then experiment with SW prefetching, first compiler based and then possibly manual prefetching if necessary
  - Distributing to another NUMA region (e.g. another CPU in the same memory space) or across a distributed machine might help
    - Effectively increasing the number of memory controllers and channels you have in the global system
- Or wait until a CPU comes along with more memory bandwidth/lower latency!

But there is (potentially) another way!

- Fundamentally we have lots of transistors in a chip and want these to all be doing something useful all of the time
  - Put quite simply, the Von Neumann architecture isn't how the electronics works
  - Current CPUs (and GPUs) give the illusion of Von Neumann but implement this more efficiently


Main Memory
Address Interconnection
Data and instruction Interconnection
Central Processing Unit (CPU)
Operational Registers
Control Unit
Arithmetic and Logic unit
Data and control information Interconnection
Input/Output System

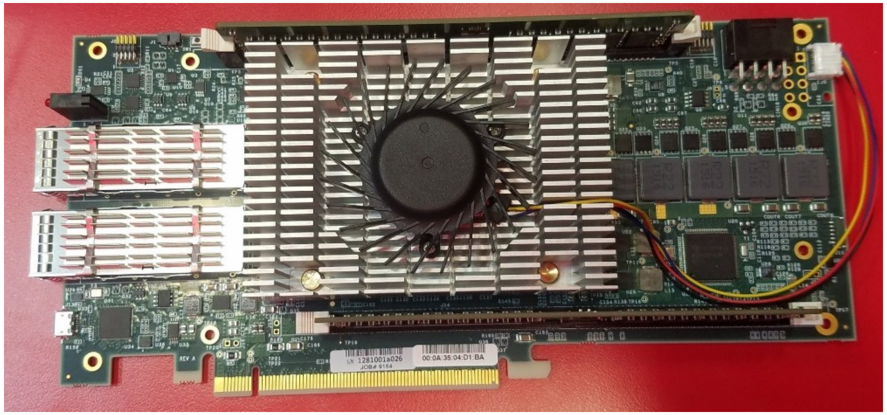
  - Where these two views of the world meet can often be a pain point!

- But there is another way! What if we unify the view exposed to the programmer and how the electronics operates?
  - By designing this around the flow of data we can avoid the memory bound nature of our code?

Step in: Reconfigurable (spatial) computing

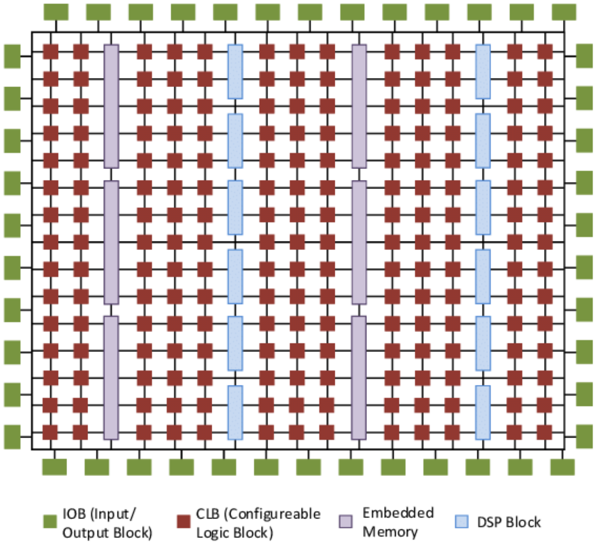
- Creating a custom chip, an ASIC, costs millions of dollars
  - So not realistic to create a chip per application!

- Instead we can use a Field Programmable Gate Array (FPGA) which can be programmed at the electronic level to behave pretty much like anything we want
  - Such as the electronic implementation of our application

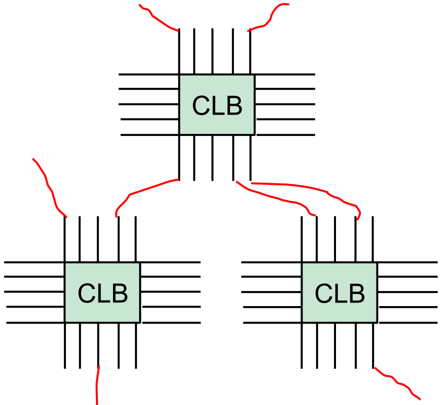


What is an FPGA

- A semiconductor based around a matrix of configurable logic blocks (CLBs) which are themselves connected via programmable interconnects.
  - Allow these blocks to be wired together in many different ways to form lots of different circuits and logic gates.
  - First invented by Xilinx in 1985
  - FPGAs contain lots of interconnect so you can do lots of wiring and specialist bits such as memory and DSP
  - All these logic blocks run in parallel, so massive concurrency at the electronic level

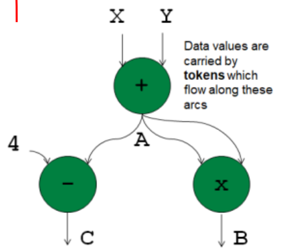


🟩 IOB (Input/ Output Block)
🟥 CLB (Configureable Logic Block)
□ Embedded Memory
□ DSP Block



A^{++} Energy efficient, high density and fast

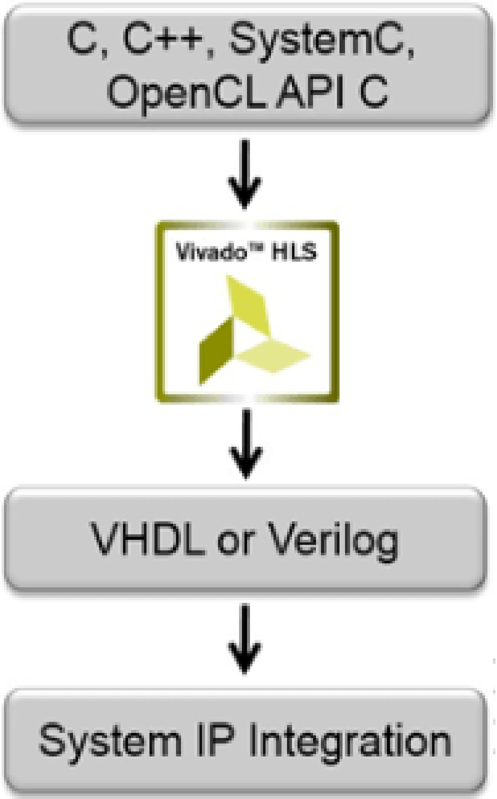
- But very difficult to program!
- Becoming available in public clouds



X
Y
+
A
4
−
x
C
B
Data values are carried by tokens which flow along these arcs

Programming FPGAs

- Was difficult, now is different:



C, C++, SystemC, OpenCL API C

VHDL or Verilog

System IP Integration

1. Program directly using a hardware description language such as VHDL or Verilog (the machine code level)

2. Write code in C or C++ and use high level synthesis (HLS) to synthesise this down to VHDL or Verilog

3. Combine point 2 with OpenCL on the host and automatic integration with the chip

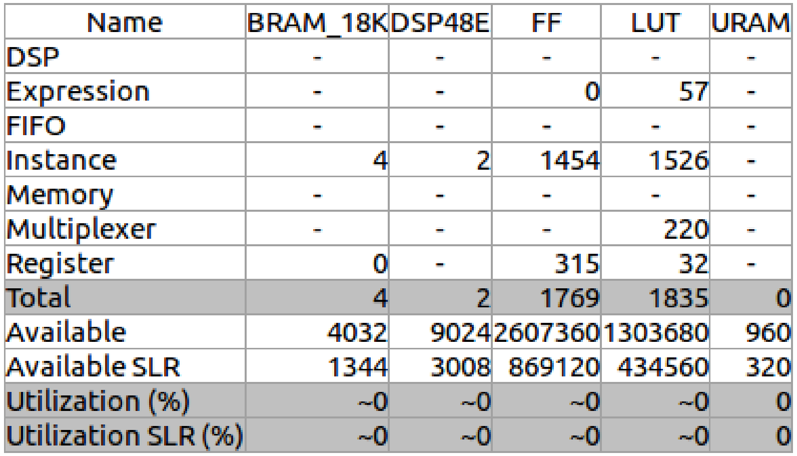
Level of difficulty

A simple example via high level synthesis

```
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
```

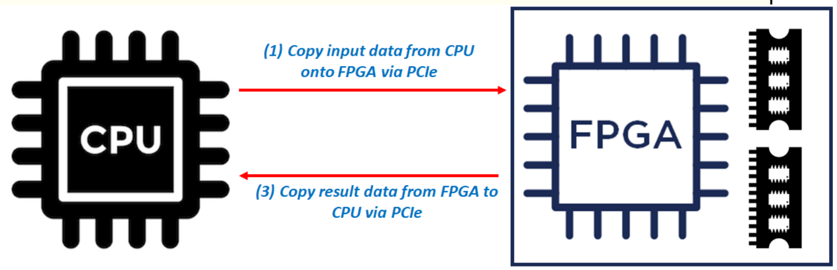
- Compiles this code into the underlying hardware description language
  - Over 20,000 lines of VHDL!
  - On disk these are .xo files which can be thought of similarly to object files



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	57	-
FIFO	-	-	-	-	-
Instance	4	2	1454	1526	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	220	-
Register	0	-	315	32	-
Total	4	2	1769	1835	0
Available	4032	9024	2607360	1303680	960
Available SLR	1344	3008	869120	434560	320
Utilization (%)	~0	~0	~0	~0	0
Utilization SLR (%)	~0	~0	~0	~0	0

Driving the FPGA from the host

```
#define DATA_SIZE 1000000
std::vector<float, aligned_allocator<float>> input_data(DATA_SIZE), result_data(1);

………

cl::Context * context=new cl::Context(device, NULL, NULL, NULL);
// Create the command queue
cl::CommandQueue * command_queue=new cl::CommandQueue(*context, device, CL_QUEUE_PROFILING_ENABLE);
………
// Create a handle to the kernel
cl::Kernel * sum_kernel=new cl::Kernel(*program, "sum_kernel");

unsigned int base_size = DATA_SIZE * sizeof(float);
cl::Buffer * buffer_input=new cl::Buffer(*context, CL_MEM_USE_HOST_PTR, base_size, input_data.data());
cl::Buffer * buffer_result=new cl::Buffer(*context, CL_MEM_USE_HOST_PTR, sizeof(float), result_data.data());

// Set kernel arguments
sum_kernel->setArg(0, *buffer_input);
sum_kernel->setArg(1, *buffer_result);
sum_kernel->setArg(2, (float) 50.0);
sum_kernel->setArg(3, DATA_SIZE);


// Queue migration of memory objects from host to device
command_queue->enqueueMigrateMemObjects({*buffer_input, *buffer_result}, 0);
// Queue kernel execution
command_queue->enqueueTask(*sum_kernel);
// Queue copy result data back from kernel
command_queue->enqueueMigrateMemObjects({*buffer_result}, CL_MIGRATE_MEM_OBJECT_HOST);

// Wait for queue to complete
command_queue->finish();
```



(1) Copy input data from CPU onto FPGA via PCIe

(3) Copy result data from FPGA to CPU via PCIe

(2) Execute HLS kernels on FPGA

```
> g++ -O3 -std=c++11 -o 'host' 'host.cpp' -lpthread -lrt -lstdc++ -lOpenCL
> ./host
Elapsed time: 0.024823 secs, (0.000869 transfer on, 0.023954 execution, 0.000000 transfer off)
```

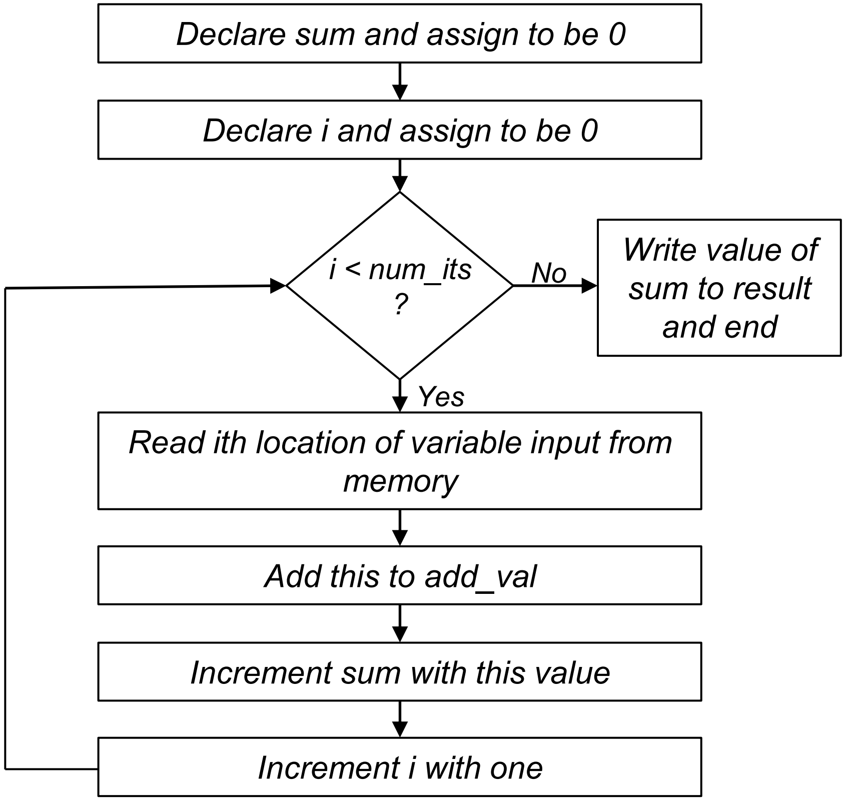
Where’s the spatial computing?

Temporal computing

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

Spatial computing

(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

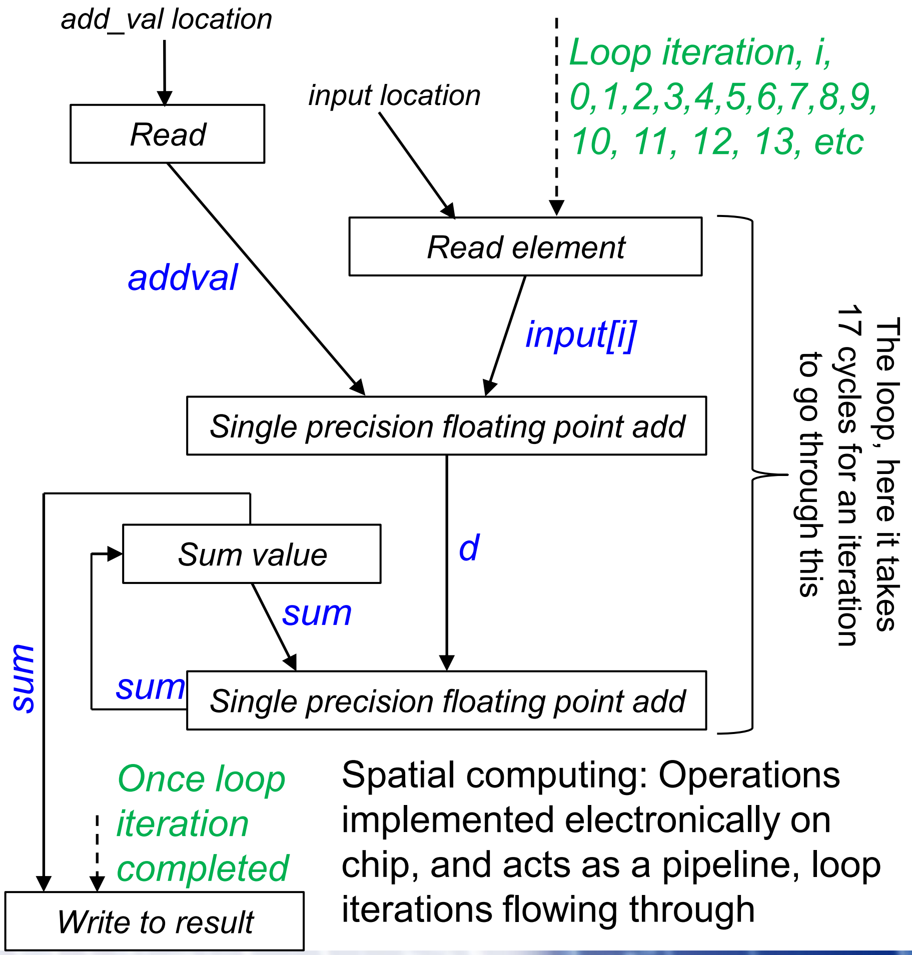
Read ith location of variable input from memory

Add this to add_val

Increment sum with this value

Increment i with one

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



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

d

Sum value

sum

sum

sum

Single precision floating point add

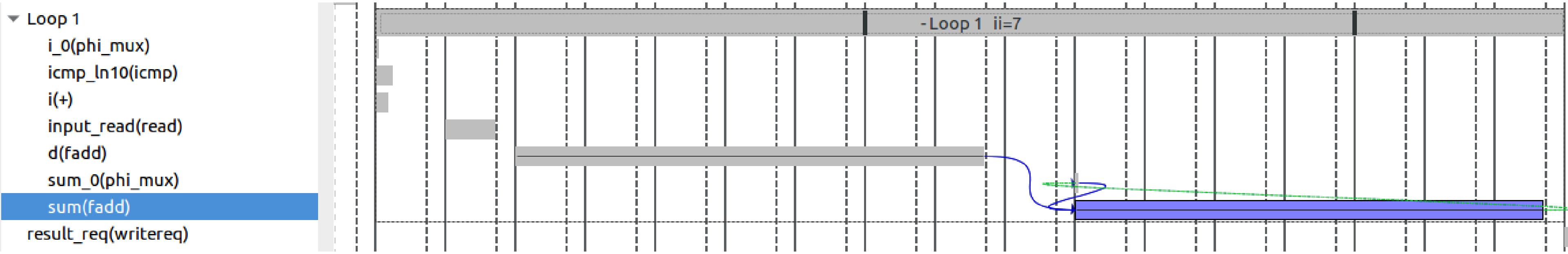
The loop, here it takes 17 cycles for an iteration to go through this

Once loop iteration completed

Write to result

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

This directly impacts our example…



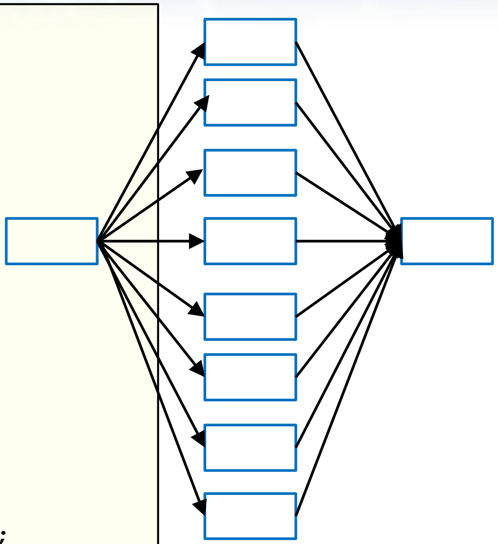
Pipeline depth of 17 cycles
Initiation Interval of 7 cycles

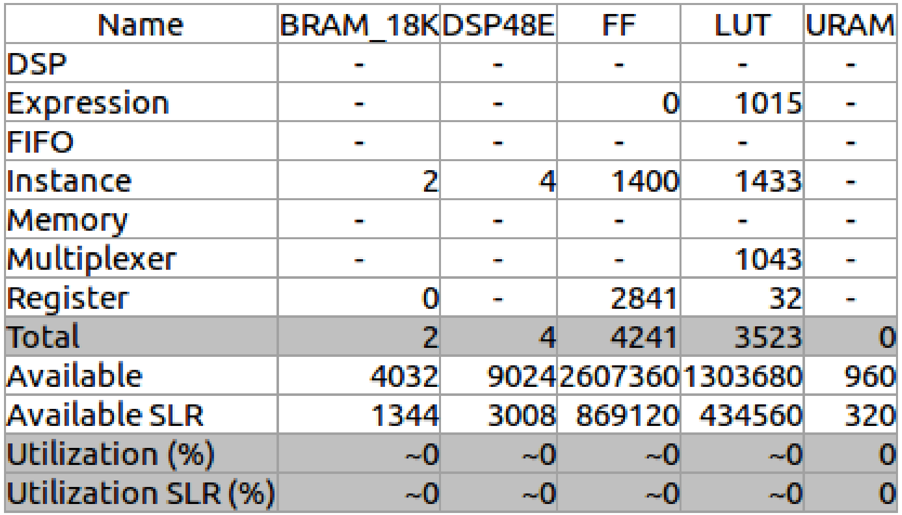
```
float sum=0;
for (int i=0;i<num_its;i++) {
  float d=input[i] + add_val;
  sum+=d;
}
```

- There is a spatial dependency here of 7 cycles
  - Effectively a dependency between the first and last pipeline stage
    - The last state depends directly on the value generated by the first stage and feeds into this
  - So we can only run a loop iteration every seven cycles
    - Instead want an iteration per cycle or else are throwing away lots of potential performance!
  - CPU executes this in 0.004339 seconds (approx. 5 times faster!)

Removing the spatial dependency

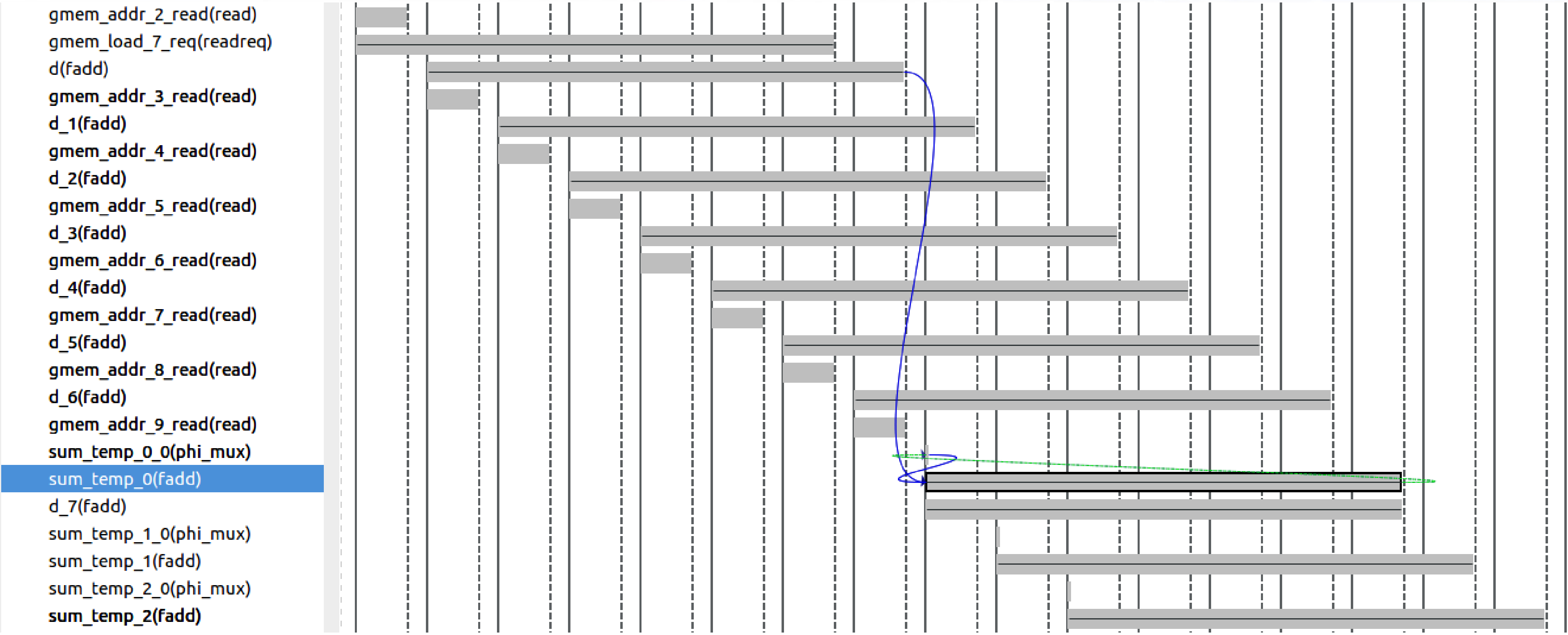
```
void sum_kernel(float * input, float * result, float add_val, int num_its) {
    #pragma HLS INTERFACE m_axi port=input
    #pragma HLS INTERFACE m_axi port=result
    #pragma HLS INTERFACE s_axilite port=add_val bundle=control
    #pragma HLS INTERFACE s_axilite port=num_its bundle=control
    #pragma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

    float sum_temp[8];
    #pragma HLS array_partition variable=sum_temp complete

    for (unsigned int i = 0 ; i<8;i++) {
    #pragma HLS unroll
      sum_temp[i]=0;
    }

    const unsigned int base_its=num_its / 8, remainder_its=num_its - (base_its*8);
    for(unsigned int i=0 ; i<base_its ; i++) {
    #pragma HLS PIPELINE II=8
      for (unsigned int j=0;j<8;j++) {
        float d=input[(i*8)+j] + add_val;
        sum_temp[j]+=d;
      }
    }

    for (unsigned int i=0;i<remainder_its;i++) {
    #pragma HLS unroll
      float d=input[i+(base_its*8)] + add_val;
      sum_temp[i]+=d;
    }
    for (unsigned int i = 0;i<8;i++) {
    #pragma HLS unroll
      *result+=sum_temp[i];
    }
}
```





Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1015	-
FIFO	-	-	-	-	-
Instance	2	4	1400	1433	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	1043	-
Register	0	-	2841	32	-
Total	2	4	4241	3523	0
Available	4032	9024	2607360	1303680	960
Available SLR	1344	3008	869120	434560	320
Utilization (%)	~0	~0	~0	~0	0
Utilization SLR (%)	~0	~0	~0	~0	0

How’s this make a difference?

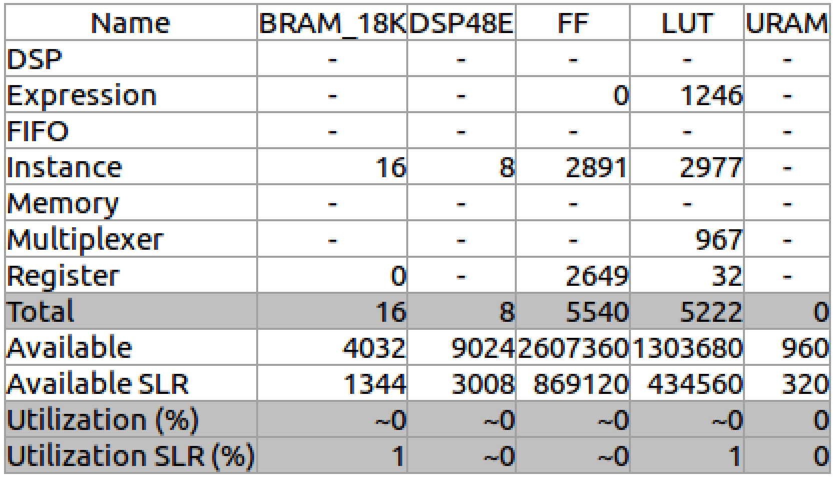


- Unrolling this inner loop and using a temporary variable to avoid the conflict on the 7 cycle floating point addition
  - Now runtime is 0.004722 seconds
    - Very slightly slower than the CPU

```
for(unsigned int i=0 ; i<base_its ; i++) {
#pragma HLS PIPELINE II=8
  for (unsigned int j=0;j<8;j++) {
    float d=input[i+j] + add_val;
    sum_temp[j]+=d;
  }
}
```

Removing read dependencies on “input”

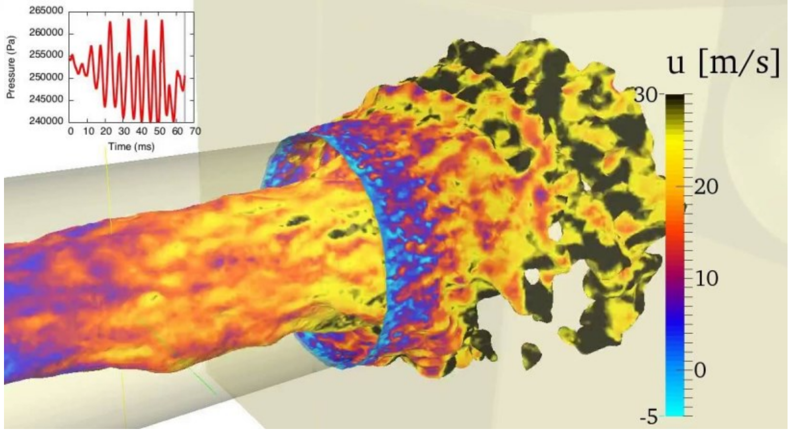
```
struct packed_data { float data[8]; };
void sum_kernel(struct packed_data * input, float * result, float add_val, int num_its) {
    #pragma HLS DATA_PACK variable=input
    #pragma HLS INTERFACE m_axi port=input
    #pragma HLS INTERFACE m_axi port=result
    #pragma HLS INTERFACE s_axilite port=add_val bundle=control
    #pragma HLS INTERFACE s_axilite port=num_its bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control
    float sum_temp[8];
    #pragma HLS array_partition variable=sum_temp complete

    for (unsigned int i = 0 ; i<8;i++) {
    #pragma HLS unroll
      sum_temp[i]=0;
    }
    const unsigned int base_its=num_its / 8, remainder_its=num_its - (base_its*8);
    for(unsigned int i=0 ; i<base_its ; i++) {
    #pragma HLS PIPELINE II=8
      struct packed_data v=input[i];
      for (unsigned int j=0;j<8;j++) {
        float d=v.data[j]+ add_val;
        sum_temp[j]+=d;
      }
    }
    struct packed_data v=input[base_its];
    for (unsigned int i=0;i<remainder_its;i++) {
    #pragma HLS unroll
      float d=v.data[i]+ add_val;
      sum_temp[i]+=d;
    }
    for (unsigned int i = 0;i<8;i++) {
    #pragma HLS unroll
      *result+=sum_temp[i];
    }
}
```



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1246	-
FIFO	-	-	-	-	-
Instance	16	8	2891	2977	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	967	-
Register	0	-	2649	32	-
Total	16	8	5540	5222	0
Available	4032	9024	2607360	1303680	960
Available SLR	1344	3008	869120	434560	320
Utilization (%)	~0	~0	~0	~0	0
Utilization SLR (%)	1	~0	~0	1	0

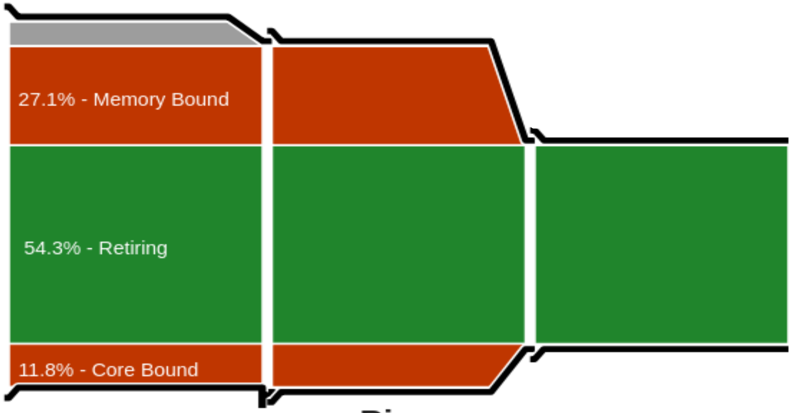
- Runtime is 0.002399 seconds
  - 0.000829 transfer, and 0.001570 execution
  - About twice as fast as the CPU and ten times faster than our initial version!

Alya example

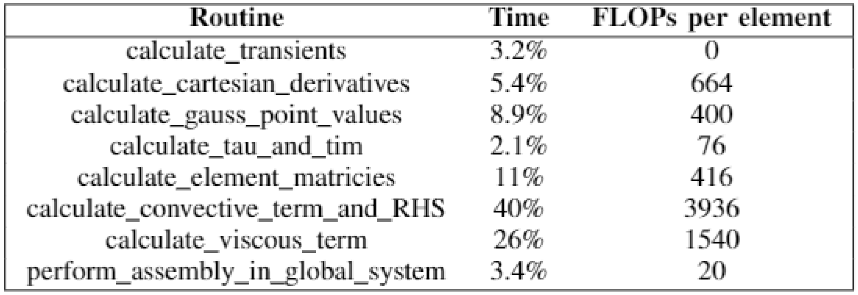


- Developed by BSC for computational mechanics code used to solve complex coupled multi-physics, multi-scale, and multi-domain problems

- Nastin module models incompressible flow and is a key component of many run configurations.
- Building of the matrix, which is used to solve Navier-Stokes equations, represents 64% of the overall model runtime
  - Only 54% of cycles doing useful work, 27% memory bound and 12% core bound

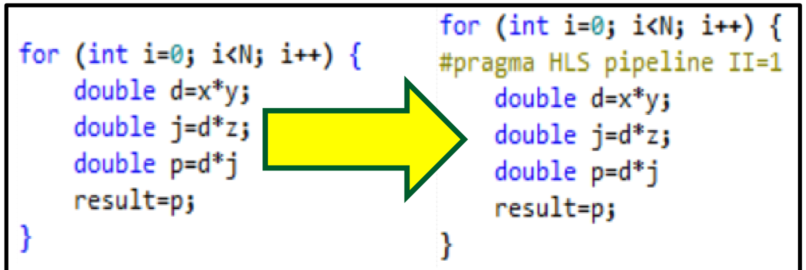


µPipe



Routine	Time	FLOPs per element
calculate_transients	3.2%	0
calculate_cartesian_derivatives	5.4%	664
calculate_gauss_point_values	8.9%	400
calculate_tau_and_tim	2.1%	76
calculate_element_matricies	11%	416
calculate_convective_term_and_RHS	40%	3936
calculate_viscous_term	26%	1540
perform_assembly_in_global_system	3.4%	20

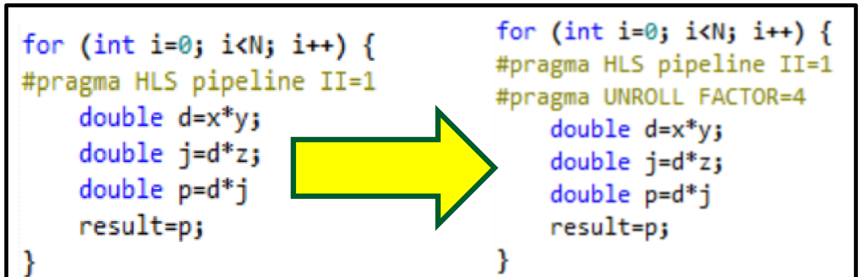
Optimising bottom up



```
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;
}
```

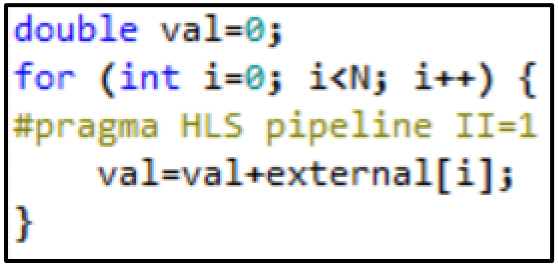
Loop pipelining



```
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;
}
```

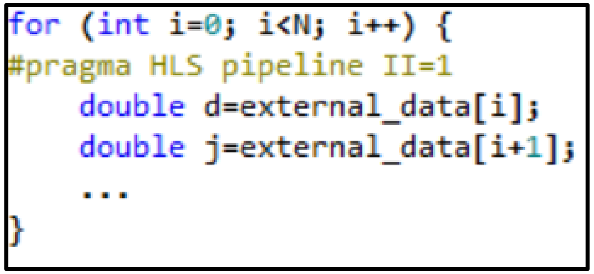
```
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
#pragma UNROLL FACTOR=4
    double d=x*y;
    double j=d*z;
    double p=d*j
    result=p;
}
```

Loop unrolling



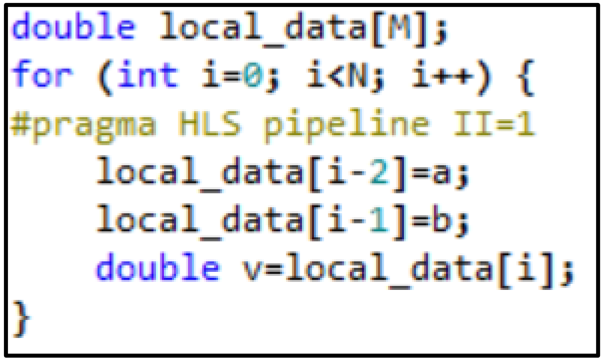
```
double val=0;
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
    val=val+external[i];
}
```

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



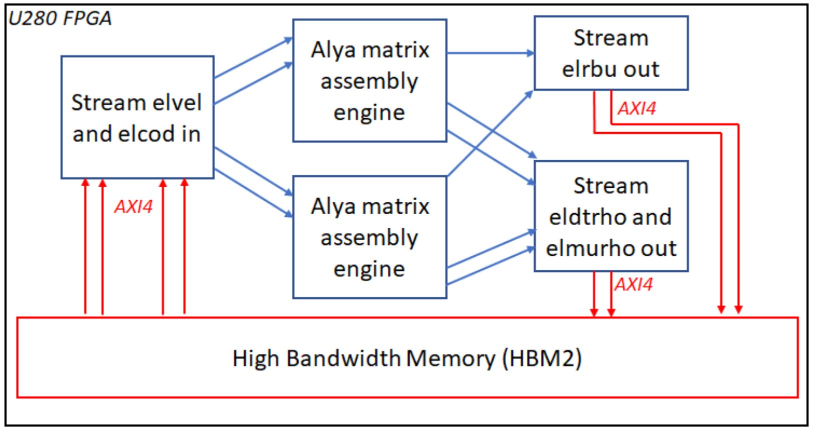
```
double local_data[M];
for (int i=0; i<N; i++) {
#pragma HLS pipeline II=1
    local_data[i-2]=a;
    local_data[i-1]=b;
    double v=local_data[i];
}
```

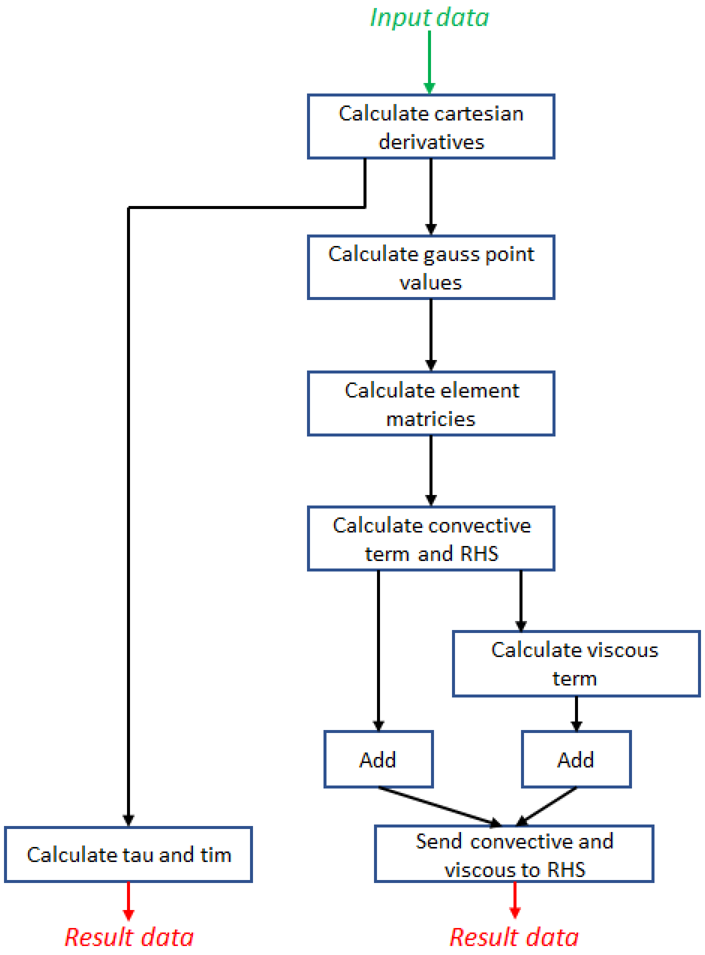
Conflict on (dual-ported) on-chip BRAM memory

Optimising top down too

Description	Runtime (ms)	% of CPU
1 core Xeon CPU	351.05	-
24 cores Xeon CPU	61.72	-
Initial FPGA port	15714.99	0.39%
Optimised top down	1508.60	4.09%
Optimised bottom up	284.04	21.73%
Optimised external memory access	26.67	231.42%

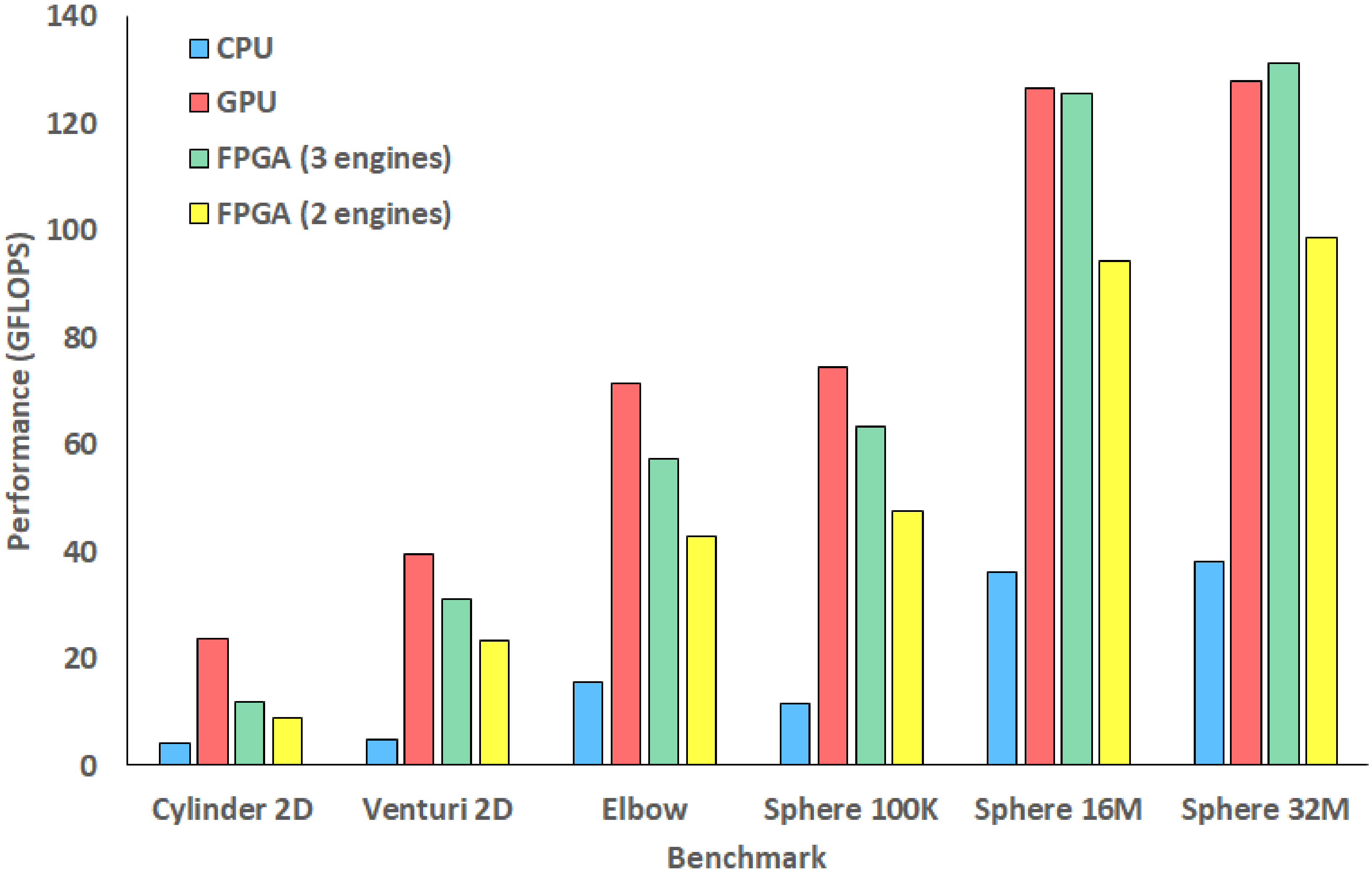
590 times difference in performance


U280 FPGA
Stream elvel and elcod in
Alya matrix assembly engine
Alya matrix assembly engine
Stream elrbu out
Stream eldtrho and elmurho out
AXI4
AXI4
AXI4
High Bandwidth Memory (HBM2)


Input data
Calculate cartesian derivatives
Calculate gauss point values
Calculate element matricies
Calculate convective term and RHS
Calculate viscous term
Add
Add
Calculate tau and tim
Send convective and viscous to RHS
Result data
Result data

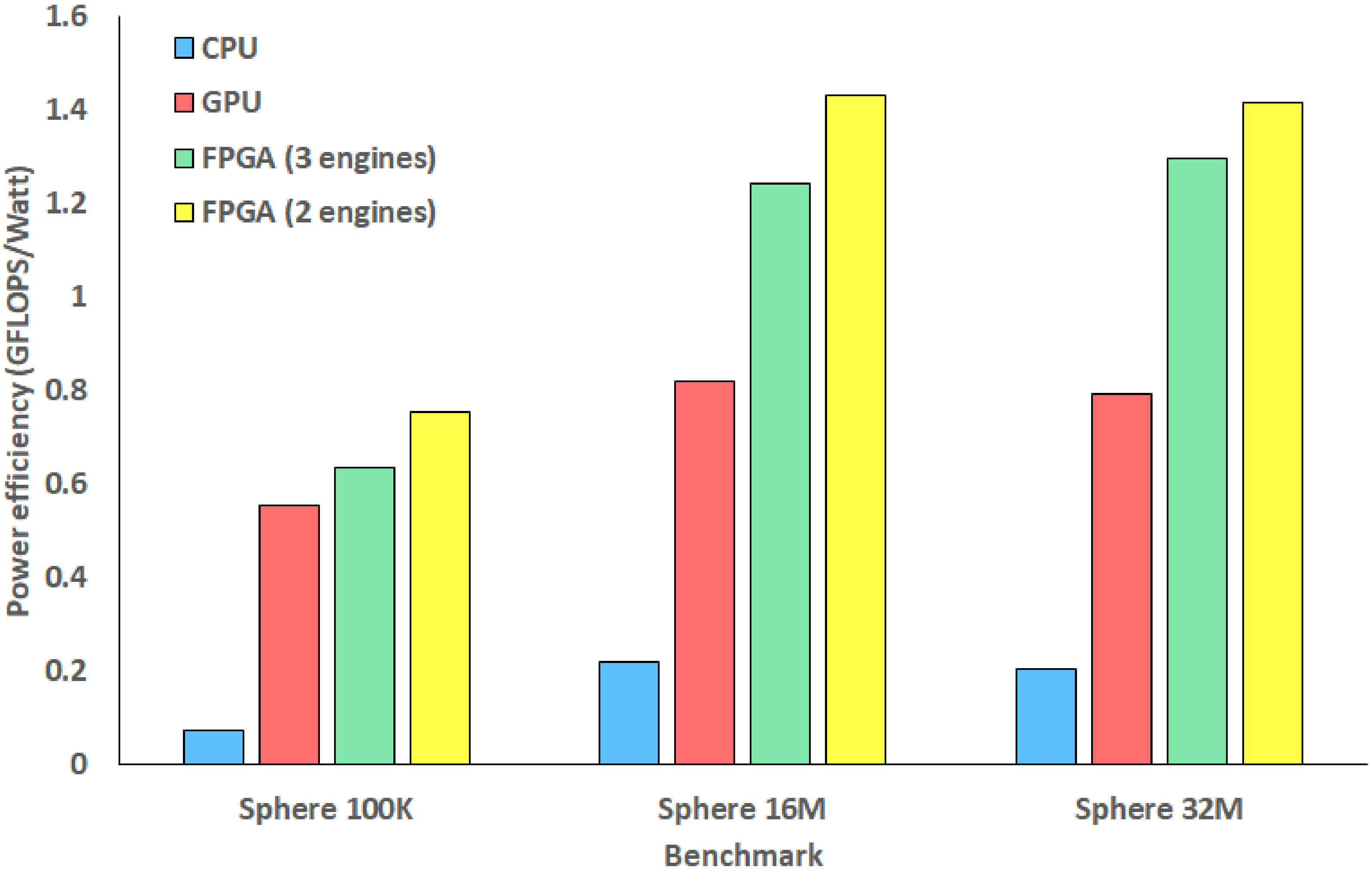
- Idea is to keep our compute fed with data, by running parts of the chip on compute and others on data loading/reordering

Performance against CPU and GPU



- *Xilinx Alveo U280 vs V100 GPU and Xeon Platinum (24 core) Cascade Lake CPU*

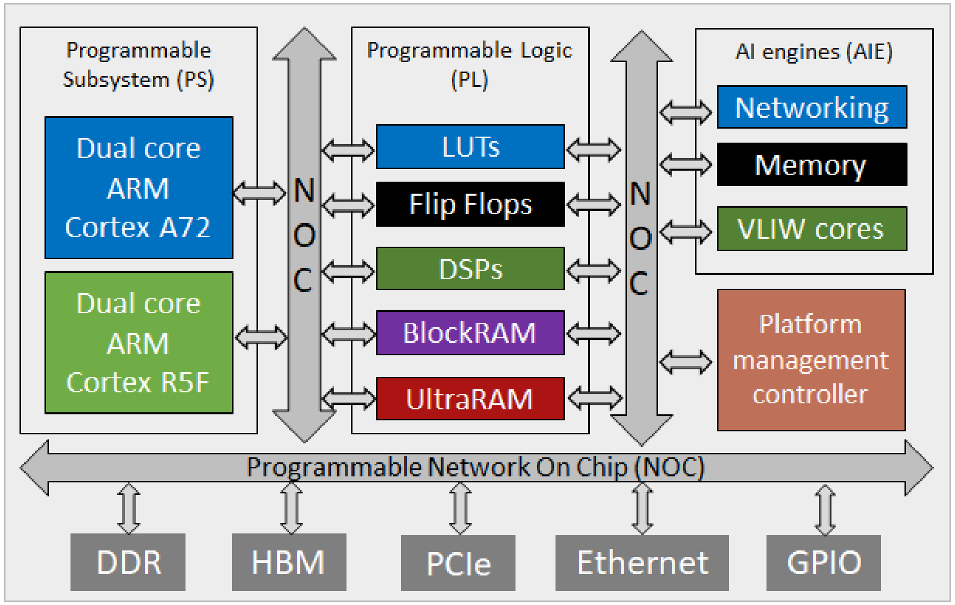
Power efficiency against CPU and GPU

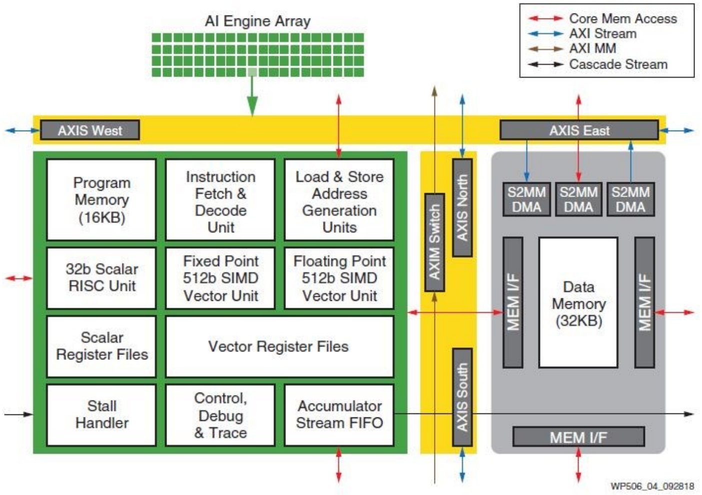


- *Xilinx Alveo U280 vs V100 GPU and Xeon Platinum (24 core) Cascade Lake CPU*

Adaptive Compute Acceleration Platform

- Combines reconfigurable fabric with standard CPUs and AI engines for vectorised acceleration
  - AI engines planned in AMD’s next generation CPUs too

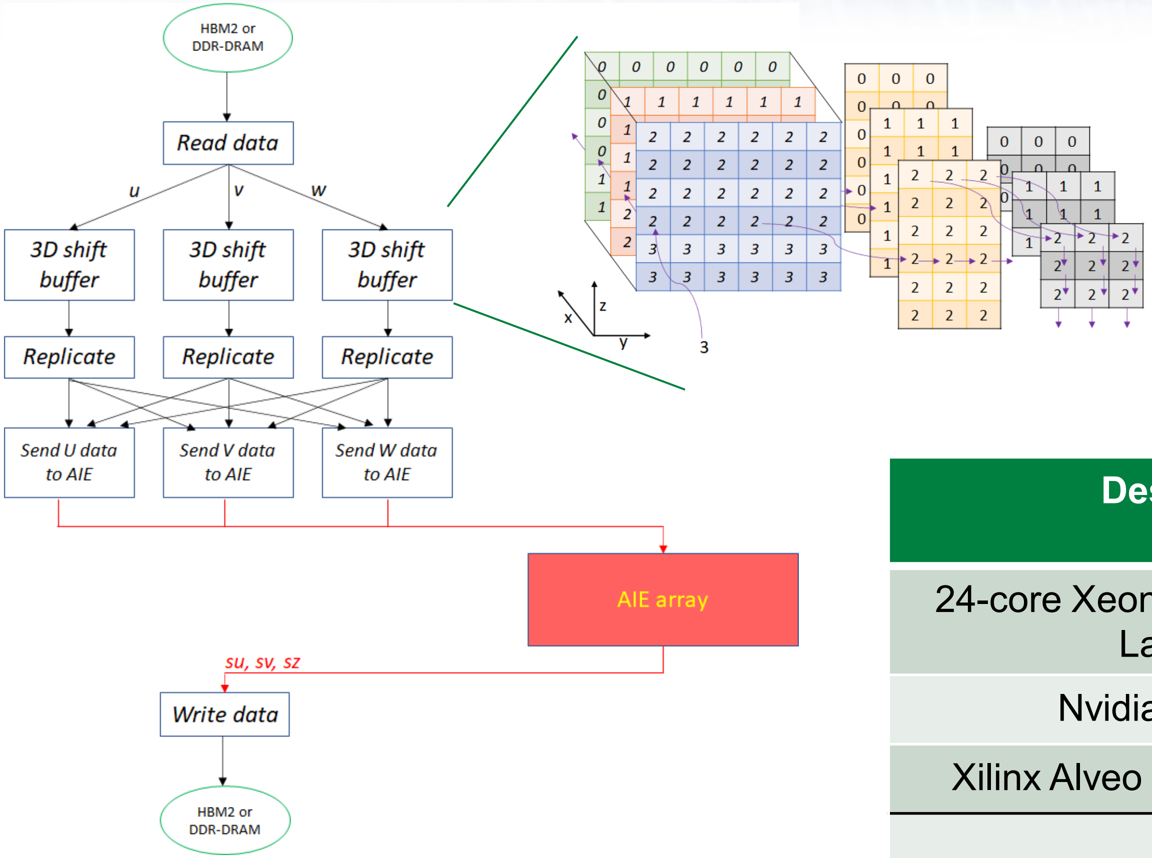


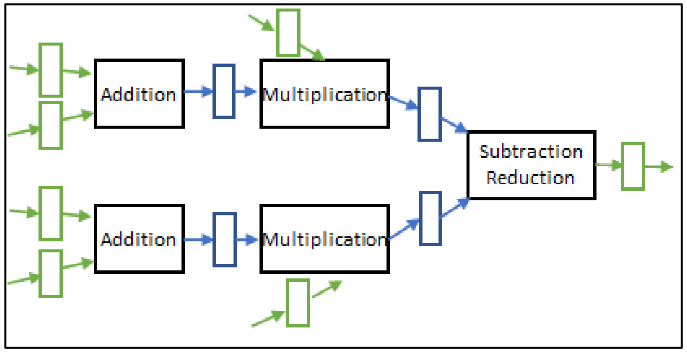


For single precision floating point, 3.8 TFLOPs theoretical peak performance total on the chip

- 400 AI engines running at 1.2GHz
  - 7-way VLIW cores
  - 16KB of program memory and 32KB data
  - Can access neighbouring core’s memory
  - Two 32-bit input and two 32-bit output streams
    - Therefore can read 128-bit every 4 cycles
  - 384-bit cascade stream per core

Accelerating advection on the AIEs

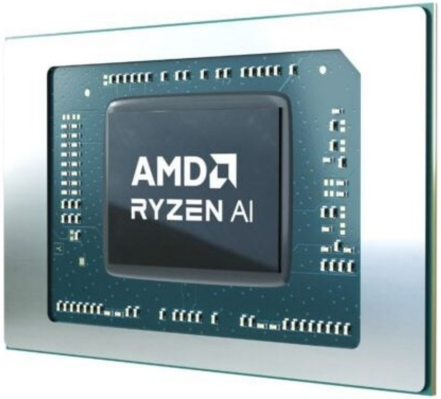




Description	Performance (GFLOPS)
24-core Xeon Platinum Cascade Lake CPU	23.52
Nvidia V100 GPU	227.89
Xilinx Alveo U280 (FPGA only)	72.32
ACAP	145.11

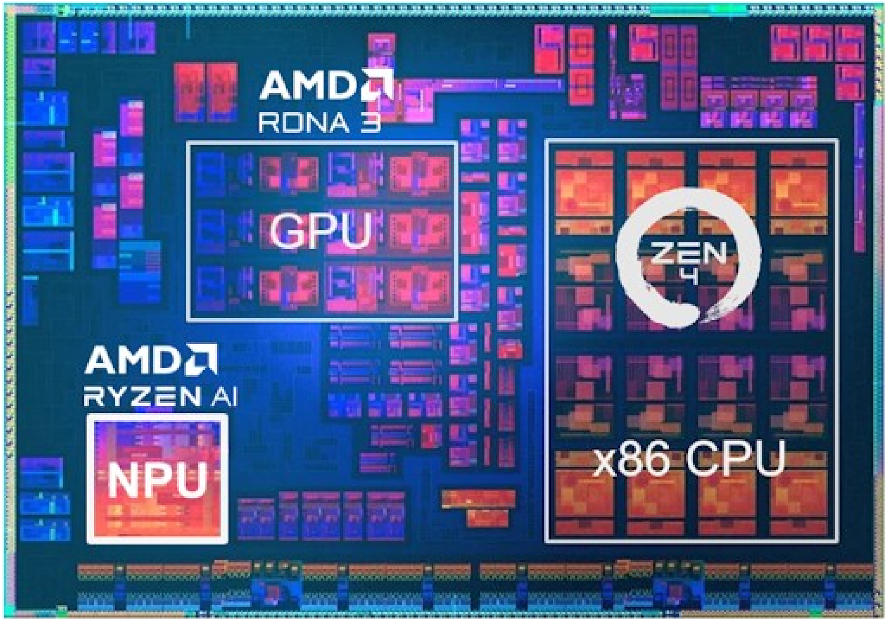
- Accelerating atmospheric advection on the ACAP
  - Idea is to use the reconfigurable fabric for bespoke memory accesses to specialise that aspect
  - Use AIEs for floating point arithmetic

Ryzen AI



- AMD have recently released a CPU with AI engines packaged in the chip
  - Currently aimed at laptop market, but potentially demonstrates direction of travel here
  - And have recently released Riallto which is a framework for programming this chip

- Focus is machine learning
  - Neural Processing Unit (NPU) but this is just a bunch of AIEs
  - Also released lots of compilation tooling too so that people can develop their own frameworks
  - Strong potential for HPC here



Spatial computing summary

- A very interesting approach and coming back in fashion
  - Future HPC architectures
  - Future embedded devices
  - Ways of bringing the computation to the data in challenging environments
    - Could especially help with memory bound codes

- But difficult to program as you also need to think spatially and ensure that the chip is kept as busy as possible
  - Requires thinking about problems in a data-flow manner
  - Compilation of code can be very time consuming
  - Active area of research to address these challenges, but currently there is nothing that solves the issues completely.
  - Jury is still out regarding their utility for HPC codes. Xilinx are currently selling them very heavily for machine learning applications