Parallel Design Patterns-L13

Vectorisation

Active messaging

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

Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

Algorithm Structure

- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures

- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms

- UE Management, Synchronisation, Communication, …

Supporting structures is also known as implementation strategy and we will use these terms interchangeably

Program structures

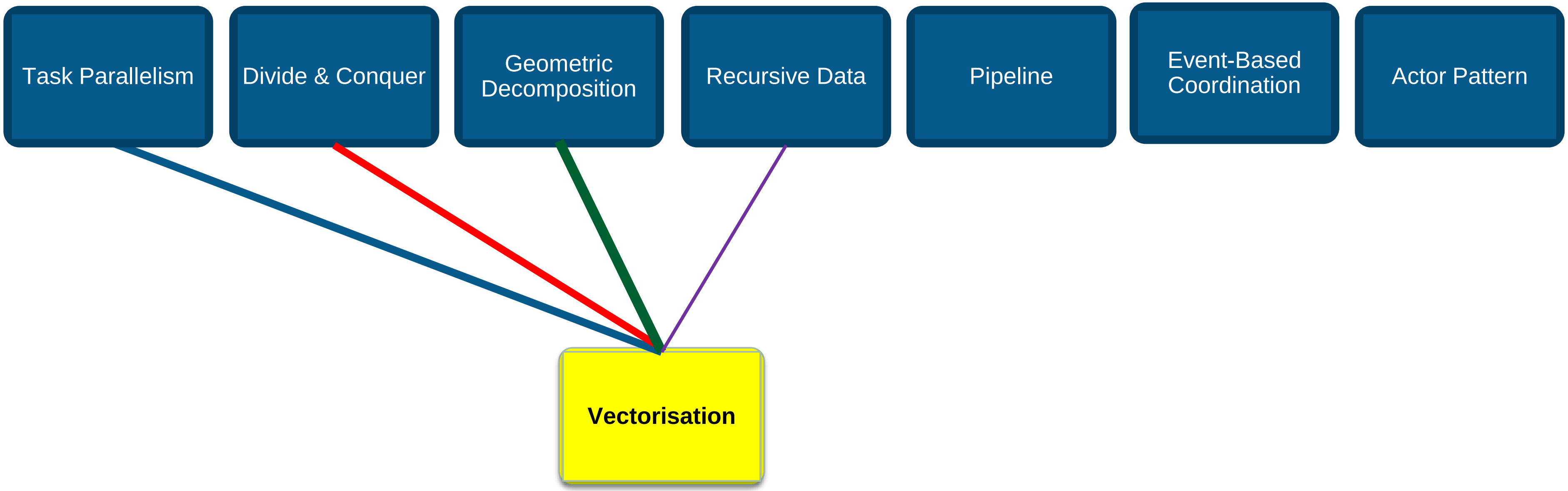
- SPMD
- Master/worker
- Loop parallelism
- Fork/join
- Active messaging
- Vectorisation

Data structures

- Shared data
- Shared queue
- Distributed array

Supporting structures

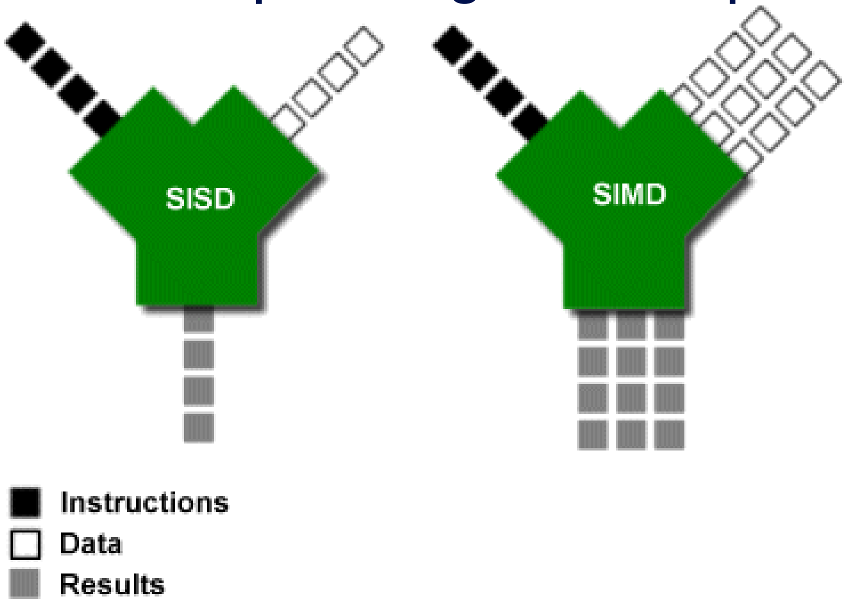
Vectorisation: The Problem



- Vectorisation is an *Implementation Strategy*
- The Problem: Given a program whose run time is dominated by a set of calculations, how can this be translated into a parallel program?
- Also known as SIMD

Single Instruction Multiple Data

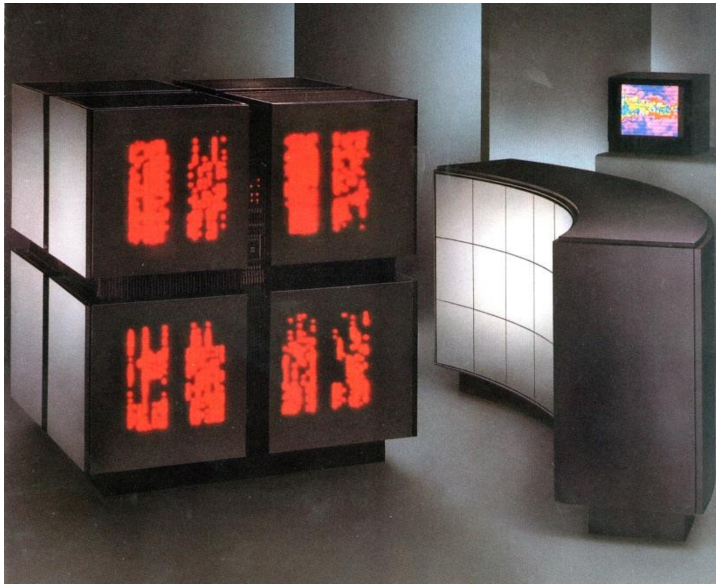
- Single stream of instructions operating on multiple data streams



- The problem is typically defined in terms of arrays that can be updated concurrently using the same instructions
- Create a single stream of instructions
  - Can have a mask to allow for some selection based on data
- Can work well when your problem is truly data parallel

Trying to force SIMD through code

```
int inputNumbers[1000];

int i,finalSum;

finalSum=0;
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

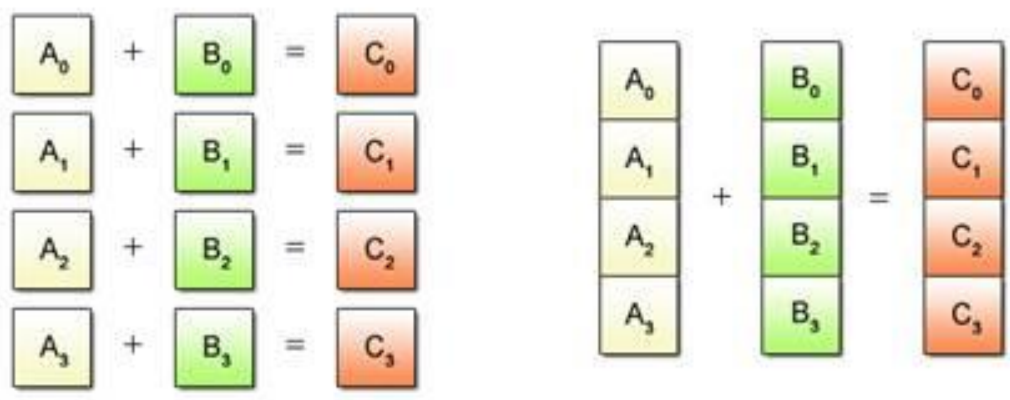


```
int inputNumbers[1000];
int results[4];

int i,j, finalSum;

for (i=0;i<=3;i++) {
  results[i]=0;
  for (j=0;j<=249;j++) {
    results[i] += inputNumbers[i + j*4];
  }
}

finalSum=0;
for (i=0;i<=3;i++) {
  finalSum+=results[i];
}
```

Streaming SIMD Extensions (SSE)

- SIMD instruction set added to Intel CPUs in 1999
  - SSE1 added eight 128 bit registers where data can be packed into and operated on concurrently with associated instructions

(a) Scalar Operation (b) SIMD Operation




```
result.x = v1.x + v2.x;
result.y = v1.y + v2.y;
result.z = v1.z + v2.z;
result.w = v1.w + v2.w;
```

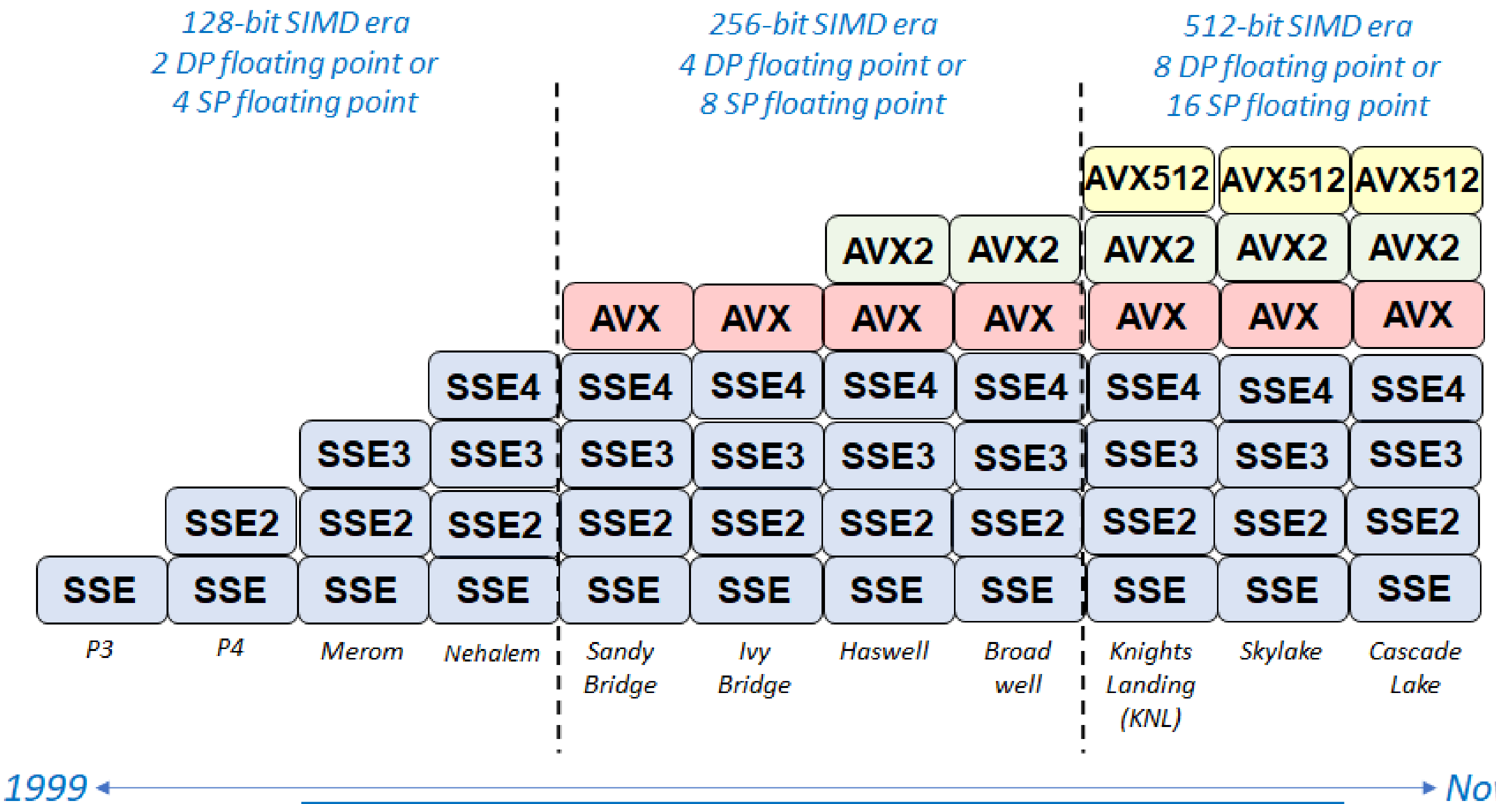
movaps **xmm0**, **[v1]**

v1.x	v1.y	v1.z	v1.w

addps **xmm0**, **[v2]**

v1.x+v2.x	v1.y+v2.y	v1.z+v2.z	v1.w+v2.w

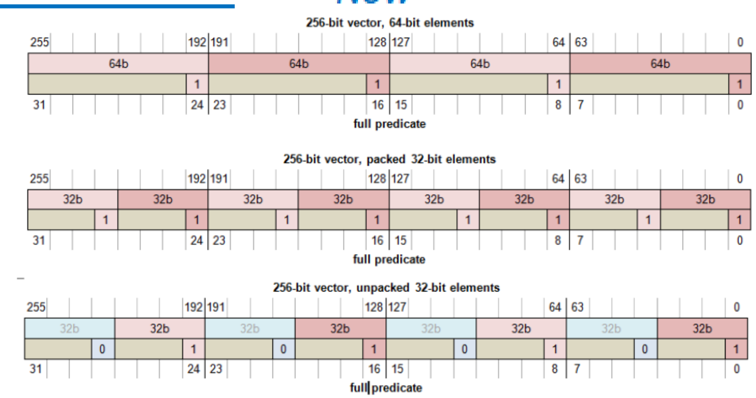
SIMD technologies



AMD Zen2 in ARCHER2 has AVX2 (AMD's Zen4 released late 2022 is the first AMD CPU supporting AVX512)

ARM has NEON which provides 128 bit registers and has standardised Scalar Vector Extensions (SVE) which provide up to 2048 width

- Currently implemented by A64FX which provides up to 512 bit width



Automatic vectorisation

- Compilers will attempt to automatically vectorise your code when compiled with optimisation enabled (–O3 on GCC)
  - With GCC you can get feedback on this using the -ftree-vectorizer-verbose=n flag, where n is 1 to 6 (the higher = more information)

- For single and double precision floating point can instruct the compiler to do this via SSE
  - With gcc using the flags -msse2, -mfpmath=sse
  - Can involve lots of memory to register movements so work experimenting with this flag to see if it is worth it

Manual vectorisation through GCC

- Compiler intrinsics support SSE

Vector is 16 bytes wide, which is 4 integers

The base type

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si v1, v2, result;

result = v1 + v2;
```

Each of these variables contains 4 integer elements

Each element of v1 added to corresponding element of v2 and the result stored in result

Compiler intrinsics with sum example

```
int inputNumbers[1000];

int i,finalSum;

finalSum=0;
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

```
#include <emmintrin.h>
…
int inputNumbers[1000] ;
int j, finalSum=0;

__m128i s, v =_mm_set_epi32(0,0,0,0);

for (j=0;j<=999;j+=4) {
  s=_mm_set_epi32(inputNumbers[j],
           inputNumbers[j+1], inputNumbers[j+2],
           inputNumbers[j+3]);
  v+=s;
}

for (j=0;j<=3;j++) {
  finalSum+=((int*)&v)[j];
}
```

Compile with flags -msse, -msse2, -march=native

OpenMP 4.0 SIMD

```
double inputNumbers[1000], finalSum=0;
int i;
#pragma omp simd reduction(+:finalSum)
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

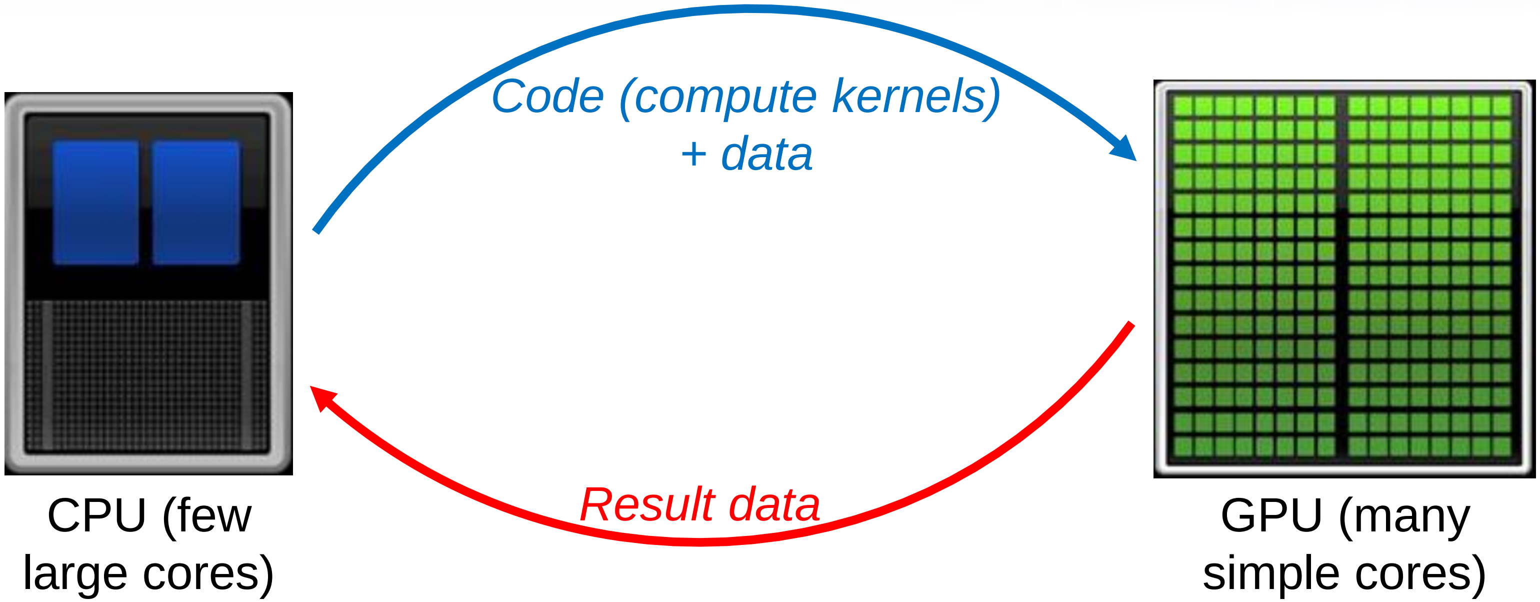
- The SIMD directive means that iterations of the loop can be executed by the SIMD lanes available to the thread.

- Can combine with the *for* directive to split iterations across threads and then across SIMD lanes
  - *The schedule should be a multiple of the SIMD length*

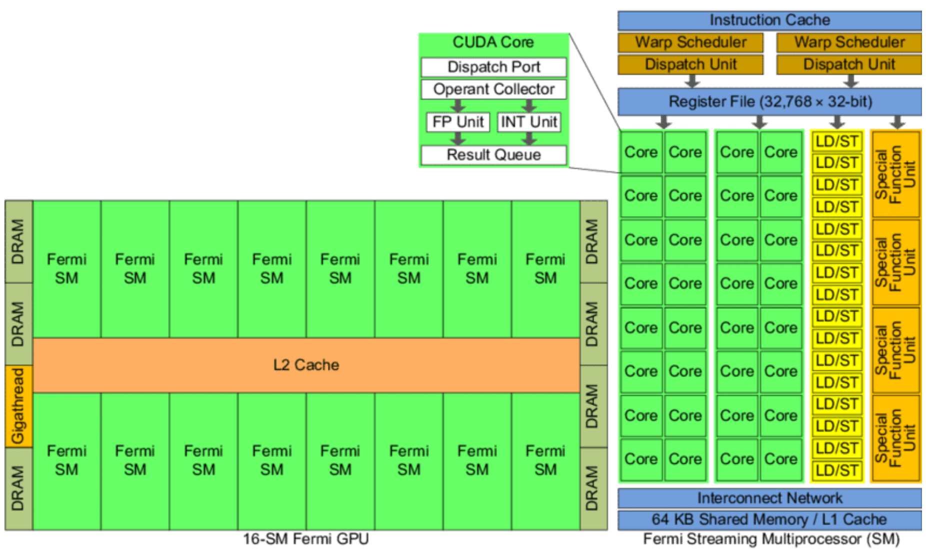
```
double inputNumbers[1000], finalSum=0;
int i;
#pragma omp for simd \
reduction(+:finalSum) schedule (static, 4)
for (i=0;i<=999;i++) {
  finalSum+=inputNumbers[i];
}
```

https://doc.itc.rwth-aachen.de/download/attachments/28344675/SIMD+Vectorization+with+OpenMP.PDF

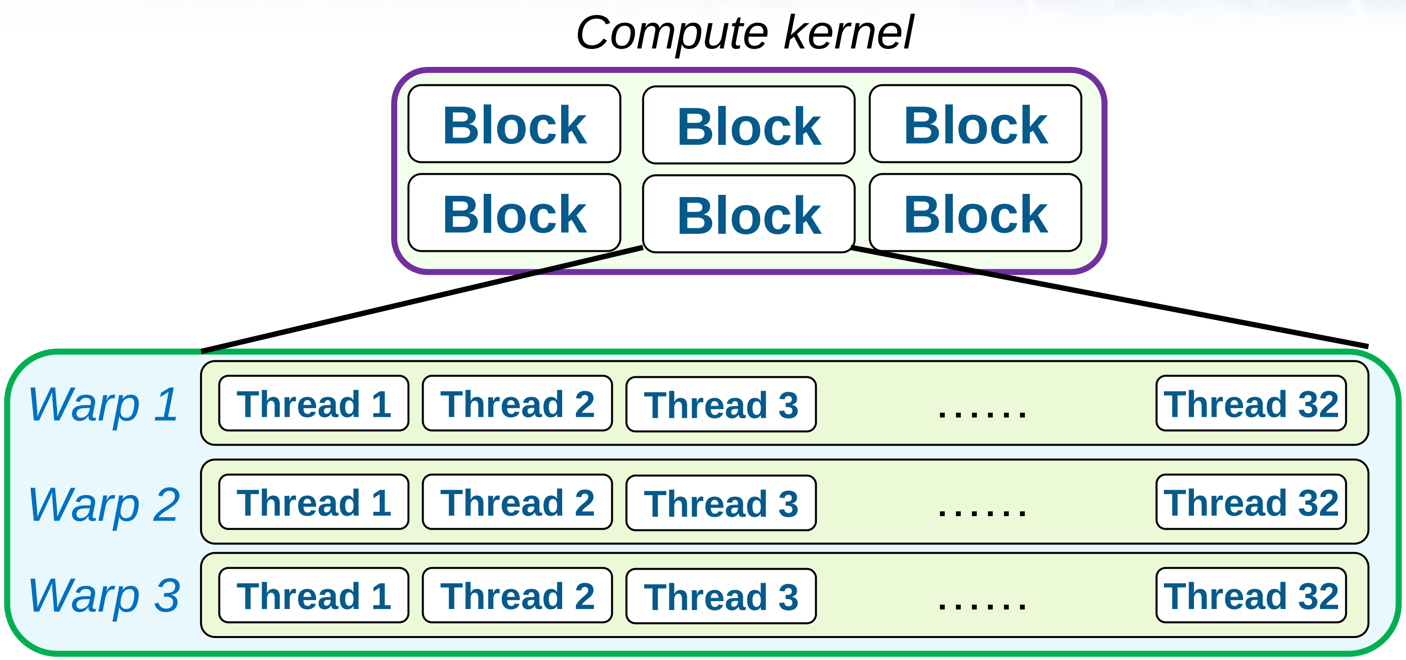
GPUs as a big vector machine



- Use GPU for floating point intensive calculations
- Use CPU for everything else
- Single Instruction Multiple Thread (SIMT)



Kernels, Blocks, Warps and Threads



- 32 threads per warp which are mapped to SMs for execution
  - Each thread executes on a CUDA core which are themselves pipelined
- Each thread of the warp executing on a CUDA core must be doing the same instruction, just on different data
  - Keeps electronics simple, warps can be paused and interleaved

Key performance factors

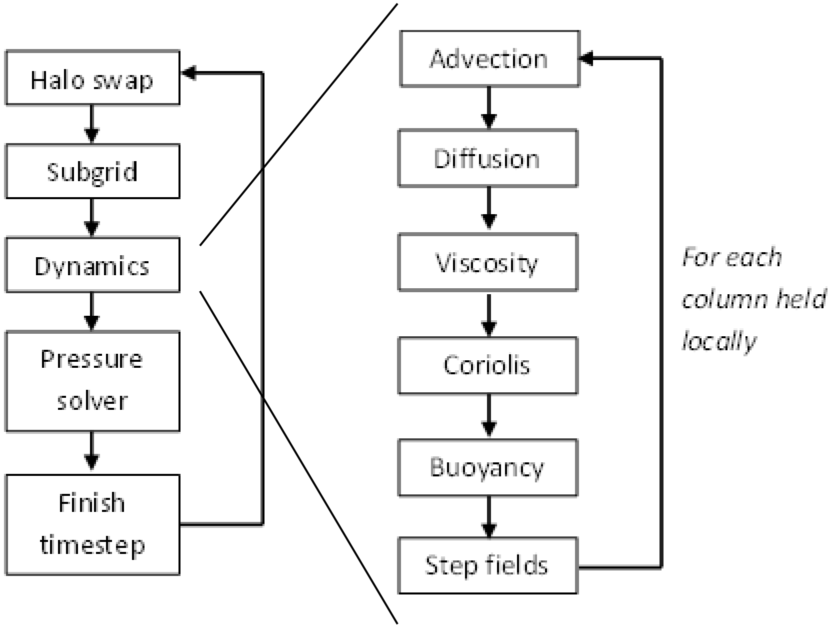
1. How quickly you can transfer data to & from the GPU
   – Parallel overhead
2. The amount of time the CPU and/or GPU will be idle
   – Wasted resources/load imbalance
3. How well your code takes advantage of the GPU architecture
   – Keeping the floating point engine busy!

Porting step	Million pairs/s
Initial MPI+OpenMP	250
Initial OpenACC	37
Optimised data transfer	61
Lattice data kept on GPU	839
Memory access pattern optimised for GPU	1190
Concurrency with streams	1270
Vectorised halo data movement	1812

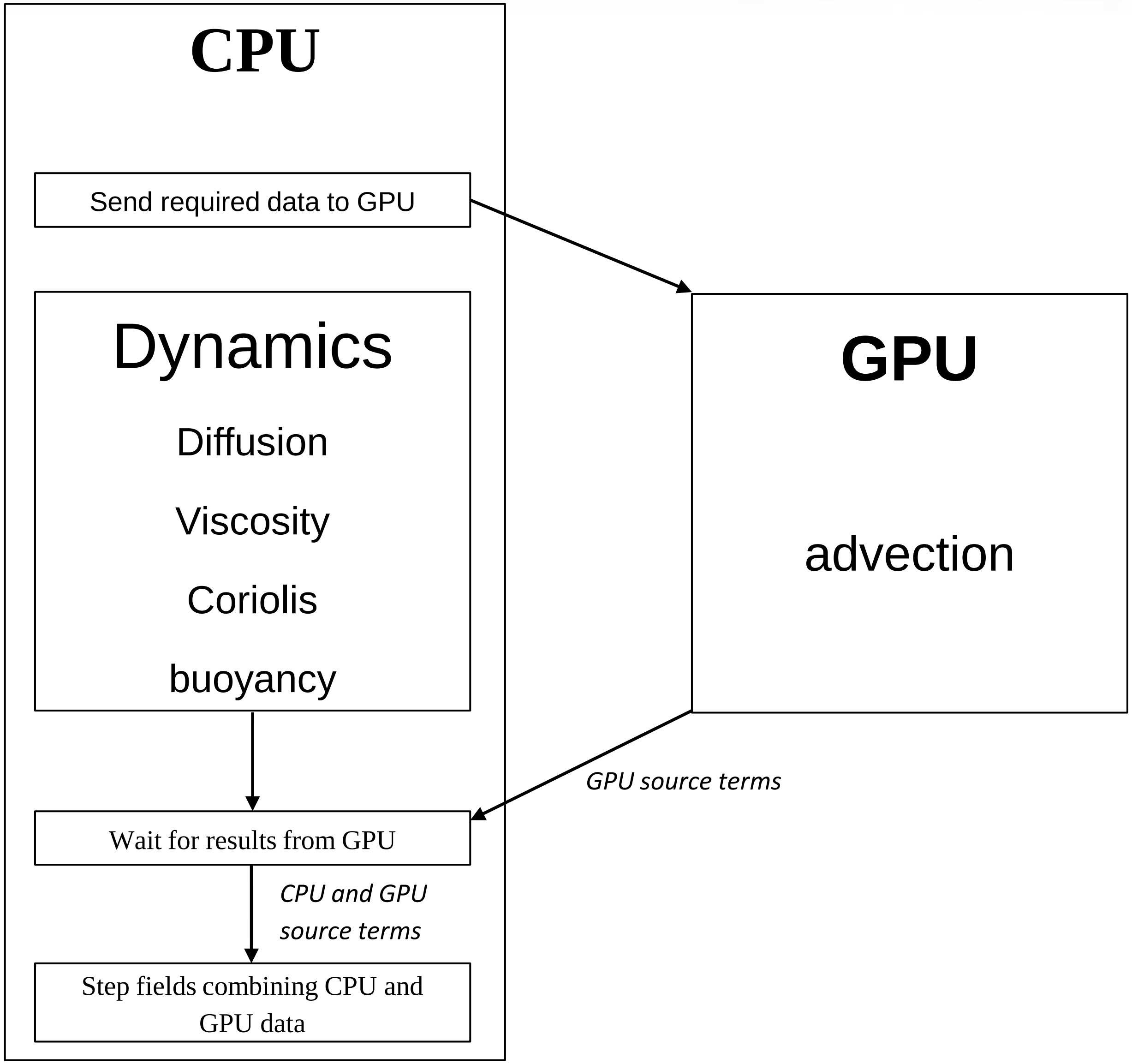
Tesla Products	Tesla P100	Tesla K80	Tesla K40	Tesla M40
GPU	GP100 (Pascal)	2 x GK210 (Kepler)	GK110 (Kepler)	GM200 (Maxwell)
SMs	56	26 (13 per GPU)	15	24
CUDA cores	3840	4992 (2 x 2496)	2880	3072
Base Clock	1328 MHz	560 MHz	745 MHz	948 MHz
GPU Boost Clock	1480 MHz	875 MHz	810/875 MHz	1114 MHz
Peak Double Precision	5.3 TFLOPS	2.91 TFLOPS	1.68 TFLOPS	.2 TFLOPS
Peak Single Precision	10.6 TFLOPS	8.73 TFLOPS	5.04 TFLOPS	7 TFLOPS
Memory Interface	4096-bit HBM2	2 x 384-bit GDDR5	384-bit GDDR5	384-bit GDDR5
Memory Size	16 GB	24GB (12GB per GPU)	12 GB	24 GB
Peak Bandwidth	720 GB/s	480 GB/s (240 GB/s per GPU)	288 GB/s	288 GB/sec
TDP	300 Watts	300 Watts	235 Watts	250 Watts
Transistors	15.3 billion	2 x 7.1 billion	7.1 billion	8 billion
GPU Die Size	610 mm²	2 x 561mm²	551 mm²	601 mm²
Manufacturing Process	16-nm	28-nm	28-nm	28-nm

MONC on GPUs

- The Met Office NERC Cloud model is an atmospheric model we developed with the Met Office
- The majority of the calculation for each timestep is in the dynamics functionality (dynamical core)
  - These add their contributions to a source term which is integrated at the end of the timestep
  - This addition is associative and commutative – i.e. can be executed in any order without impacting the results
  - So let’s offload bits of the dynamical core to GPUs



Asynchronous data transfer



- Data transfer is asynchronous
- Constants are only copied across once (model initialisation)
- Allow kernels to share data

Kernel illustration

```
!$acc update device(u, v, w) async(1)
………
!$acc update device(theta) async(2)
………
```

Kick off (asynchronous) data transfers to the GPU as early as possible

```
!$acc parallel loop collapse(3) wait(1,2) async(30)
default(none) present(u, v, w, theta, stheta, start_index,
end_index, tcx, tcy, tzc1, tzc2, tzd1, tzd2)

do x = start_index(X_INDEX), end_index(X_INDEX)
    do y = start_index(Y_INDEX), end_index(Y_INDEX)
        do k = start_index(Z_INDEX), end_index(Z_INDEX)
                ……
        end do
    end do
end do
```

Schedule the kernel to run on the GPU once data transfers have completed.

Each iteration corresponds to a thread

```
!$acc update host(stheta) wait(30) async(31)
```

Once kernel completed copy results back to CPU (asynchronously)

```
!$acc wait(31)
```

Wait (on the CPU) for it all to complete

Array of structs or struct of arrays

- Different data layouts suit specific access patterns which might change as we port codes from CPUs to GPUs

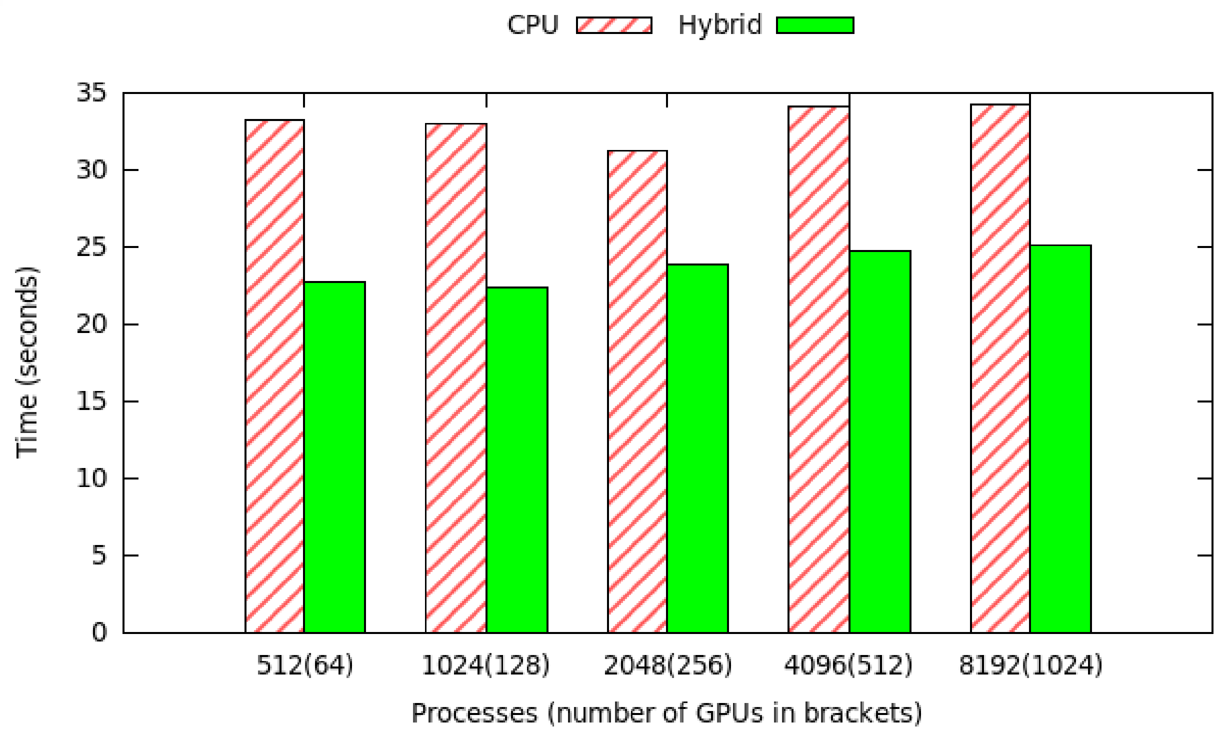
```
struct data_point {
  double u, v, w, theta
}
struct data_point pts[100], res[100];
for (i=1;i<99;i++) {
  res.u[i]=pts.u[i-1]+pts.u[i+1];
  res.v[i]=pts.v[i-1]+pts.v[i+1];
  res.w[i]=pts.w[i-1]+pts.w[i+1];
  res.theta[i]=pts.theta[i-1]+
                   pts.theta[i+1];}
```

```
u[0],v[0],w[0],theta[0],u[1],v[1],w[1],theta[1],
u[2],v[2],w[2],theta[2],u[3],v[3],w[3],theta[3],
u[4],v[4],w[4],theta[4],u[5],v[5],w[5],theta[5],
u[6],v[6],w[6],theta[6],u[7],v[7],w[7],theta[7],
u[8],v[8],w[8],theta[8],u[9],v[9],w[9],theta[9],
...
```

```
struct array_container {
  double u[100], v[100], w[100], theta[100];
}
struct array_ container data, res;
for (i=1;i<99;i++) {
  res.u[i]=data.u[i-1]+data.u[i+1];
}
............
```

```
u[0],u[1],u[2],u[3],u[4],u[5],u[6],u[7],u[8],u[9], ...
v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9], ...
w[0],w[1],w[2],w[3],w[4],w[5],w[6],w[7],w[8],w[9], ...
theta[0], theta[1], theta[2], theta[3], theta[4],
theta[5], theta[6], theta[7], theta[8], theta[9], ...
```

Code structure changes



- The structure of the code has needed to be changed in order to offload the computational kernels
  - Constants at the start of the model
  - Data transfer for each timestep
  - The kernel itself
  - Waiting for results and integrating back (which is problem specific)

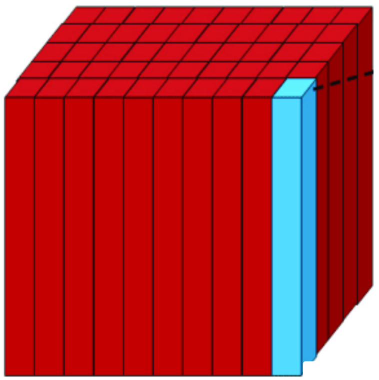
More complex GPU code

- There is a performance improvement with advection on GPUs, but it isn't great
  - Not the performance improvement promised by FLOPS figures!
  - When profiling we see two things:
    1. Data transfer takes up a significant amount of GPU time (around 40%)
    2. The GPU finishes work far sooner than the CPU completes the rest of the dynamics
  - Hence the GPU is underloaded and whilst advection is around 50% of the runtime of MONC, it isn't enough for GPUs really.

- Cloud AeroSol Interactions Microphysics (CASIM) model for modelling moisture interactions was then integrated
  - Doubled or even trebled the runtime of MONC!

CASIM – a choice

- The challenge is that CASIM has a number of computationally intensive kernels
  - These are called frequently in the code but with lots of non-computational work done before and after each hotspot
  - Such as conditionals, loops etc
- The scheme is operating on many Q (moisture) fields in vertical columns
- Tightly coupled in the vertical but not in other two dimensions
- Per timestep columns are independent



```
subroutine CASIM()
   do i = i_start, i_end
      do j = j_start, j_end
         ...
         call hotspot1()
         ...
         call hotspot2()
         ...
         call hotspot3()
         ...
      end do
   end do
end subroutine CASIM
```

CASIM – a choice

```
do i = i_start, i_end
   do j = j_start, j_end
      call before()
      call hotspot()
      call after()
   end do
end do
```

```
do i = i_start, i_end
   do j = j_start, j_end
      call before()
   end do
end do
do i = i_start, i_end
   do j = j_start, j_end
      call hotspot()
   end do
end do
do i = i_start, i_end
   do j = j_start, j_end
      call after()
   end do
end do
```

- So we can either refactor the code like so
- But this will result in lots of data movement and the CPU will still be busy (negate our hybrid approach)
- Or we can offload the entirety of CASIM to the GPU

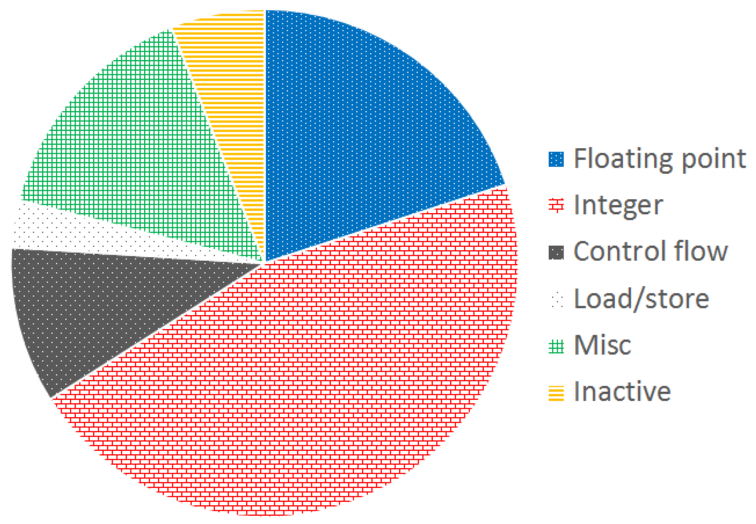
Offloading the entirety of CASIM

```
subroutine CASIM()
   !$acc parallel
   !$acc loop collapse(2) gang worker vector
   do i = is, ie
      do j = js, je
          ...
          call microphysics_common(i,j)
          ...
      end do
   end do
   !$acc end loop
   !$acc end parallel
end subroutine CASIM

subroutine microphysics_common(i,j)
   !$acc routine seq
   ...
end subroutine microphysics_common
```

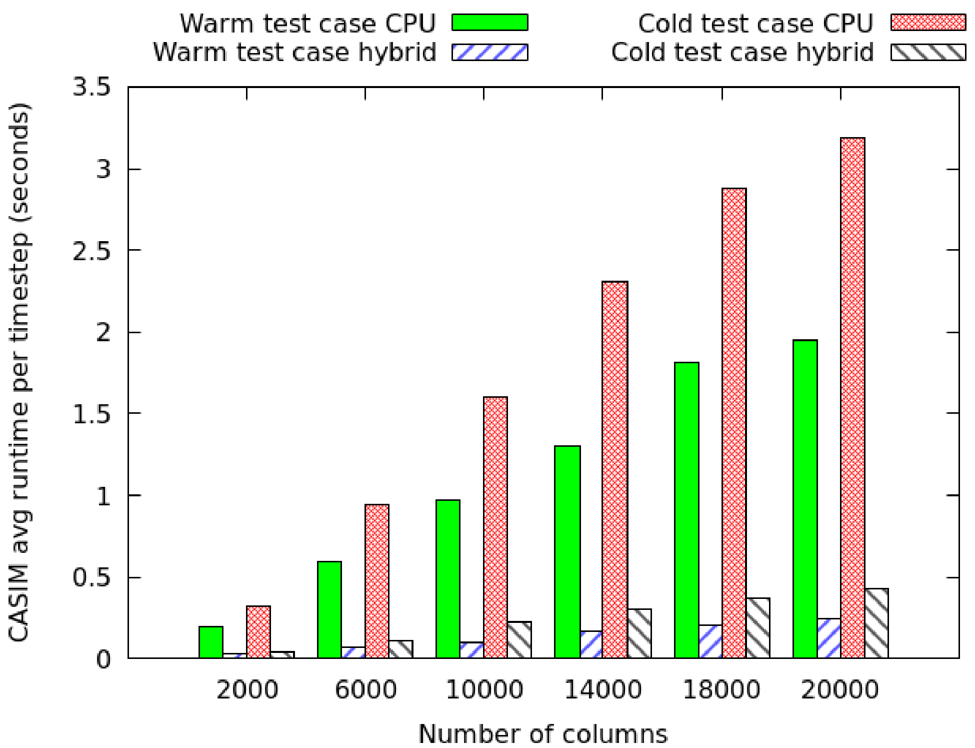
- OpenACC support offloading subroutines
  - We allocate a thread per column in the domain
  - Hence the seq in each subroutine
- In total 50 Fortran modules and 123 subroutines offloaded

- CPU code contains lots of intermediate temporary variables
  - Which have to be duplicated for each thread as they are all running concurrently on separate columns in the domain

CASIM profiling



Column size	Config	To GPU	Kernel	From GPU
2000	Warm	1.5ms 5%	29ms 93%	0.8ms 3%
2000	Cold	1.82ms 4%	39ms 94%	0.74ms 4%
10000	Warm	7ms 7%	88ms 88%	4.6ms 5%
10000	Cold	11.2ms 5%	214ms 93%	4.6ms 2%
20000	Warm	18ms 7%	224ms 90%	8ms 3%
20000	Cold	22.56ms 5%	395ms 93%	8.1ms 2%



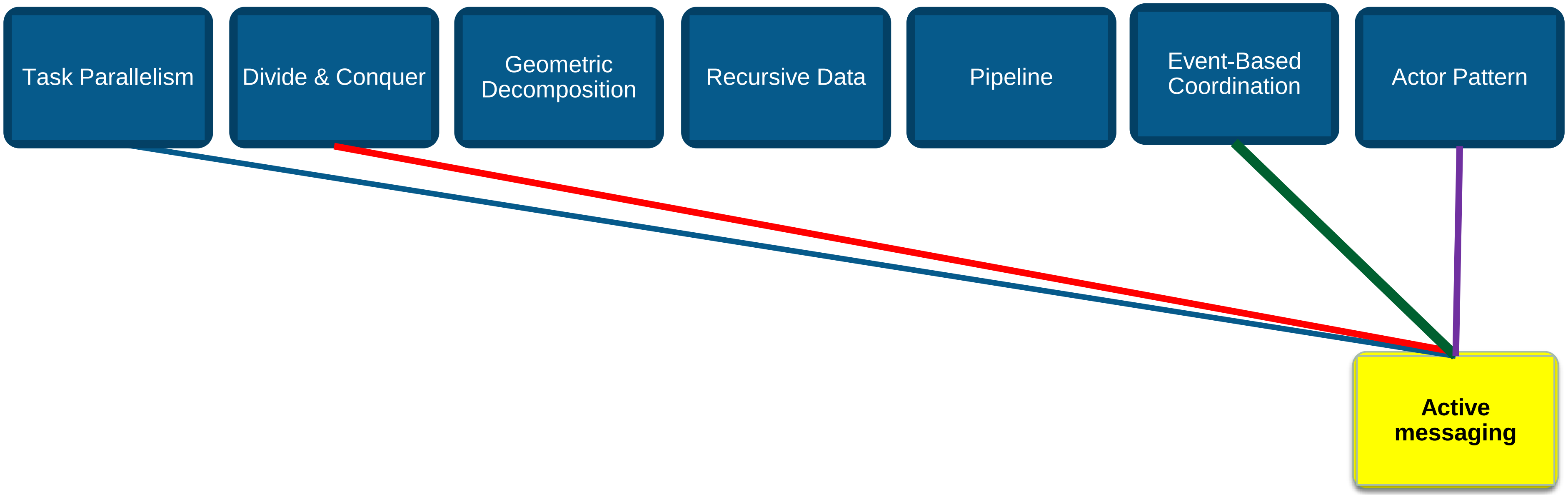
- But the option for offloading the entirety of the model means we are dominated by integer operations

Vectorisation - summary

- Parallelism at multiple levels
  - Instruction level, core level, processor level, node level
  - Significant performance improvements can be obtained by leveraging vectorisation correctly
  - Many compilers will do this automatically for you, but not all compilers are created equally!
  - Technologies such as OpenMP and OpenACC (for GPUs) make this look similar to loop parallelism

- Viewing GPUs as SIMD engines
  - Need to keep them feed with calculations to work on
  - They work best doing floating point arithmetic
  - Need to consider how to keep the CPU and GPU busy at the same time

Active messaging: The Problem



- Active messaging is an *Implementation Strategy*
- The Problem: We want to run multiple tasks, which are driven by irregular interactions, on a UE. How can we best structure our code to support this?

Example problem

- I am running a code with lots of tasks per UE
  - There are lots of tasks (e.g. function calls) that I have available to run on the UE and-so don't want to block for communications. However my communications are irregular and I need to work with values I receive.

```
a=receive(1);
calculate(a);
```

```
handle=nonblocking_receive(1);
while (!test(handle)) {
  Do some other work
}
calculate(a);
```

- This is OK but relies on being able to find some other work to do and carry lots of request handles around
  - Might not be possible, or with irregular & unpredictable communications might be difficult to structure code generally to support this

Active messaging

- The arrival of a message will activate some handling block of code on the target UE (also known as a *callback*)

```
send(data, target rank, unique identifier);
register_recv(callback, source rank, unique identifier);
```

- *The unique identifier* (UUID) is used to match the message with a specific handler
- The callback function will typically receive the data and metadata (such as amount of data, type etc.)
- Sending is either blocking or non-blocking
- The receive call is non-blocking

```
send(data, 1, "hello");
```

UE 0

UE 1

```
register_recv(calculate, 0, "hello");


void calculate(data, metadata) {
    .........
}
```

Active messaging

- Called *active messaging* as messages explicitly activate the block of code which will handle them
  - Some or all of the code will be structured around these handlers
  - Callback handlers might persist (i.e. can be called for many different messages) or transitory (once called they are deregistered.)

- Implementation choice between running handlers concurrently or sequentially
  - When a message arrives do we kick a UE off (i.e. a thread from a pool) which calls the handler
  - Or are messages queued up and processed one at a time?
- If you run handlers concurrently you will need to protect shared data shared between them (*shared data pattern*.)

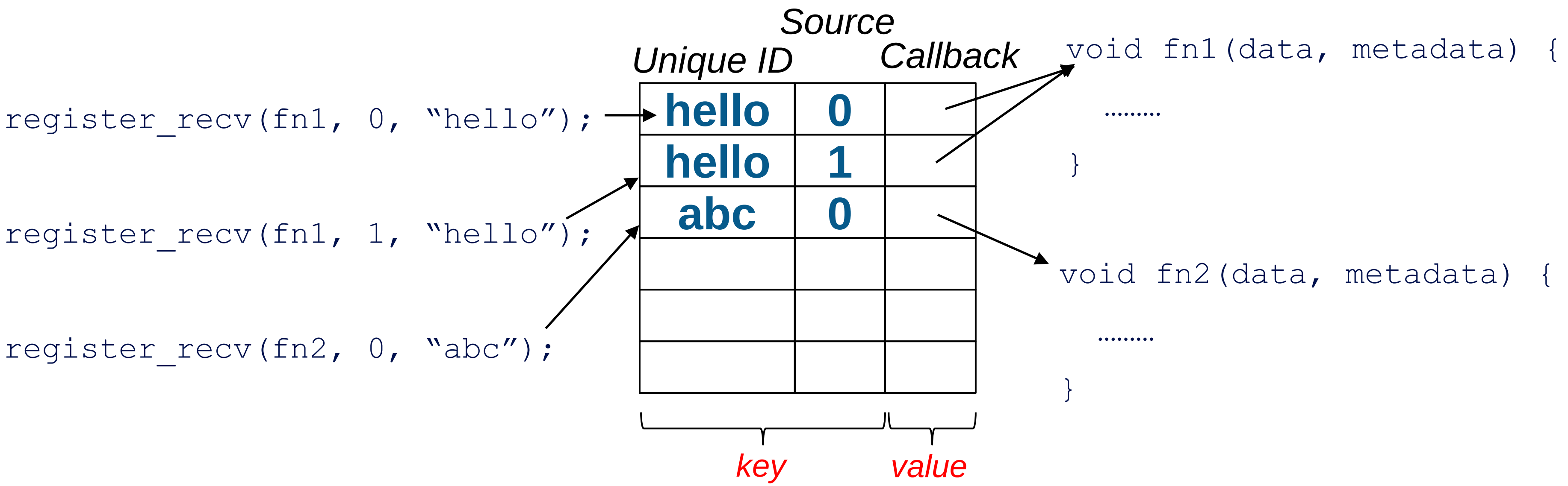
Supports collective messaging too

The callback routine — *Value on each process to use* — *Operation* — *Root* — *Unique ID*

```
register_reduce(my_handler, my_value, "sum", 0, "my_reduction1");


void my_handler(data, metadata) {

   .........

}
```

- In this case each process issues a reduction, *my_handler* is then executed on process 0 with the resulting value
  - Callback is only executed on process 0 once every single process has issued this call and the reduction is completed
  - The callback routine could be NULL on other processes
- Crucially the UUIDs determine what collective messages match rather than the issue order
  - This provides greater flexibility for irregular applications where codes might issue collective messages in different orders.

Active messaging - implementation



- Have a map style structure where they key is a combination of the unique identifier and the source rank, the value is a pointer to the appropriate callback function
- Behind the scenes you poll for a messages, from this extract the unique ID and use this in combination with the source rank to find the appropriate callback handler function to execute
  - The rest of the message is then split up to extract the data and any other metadata

Active messaging - implementation

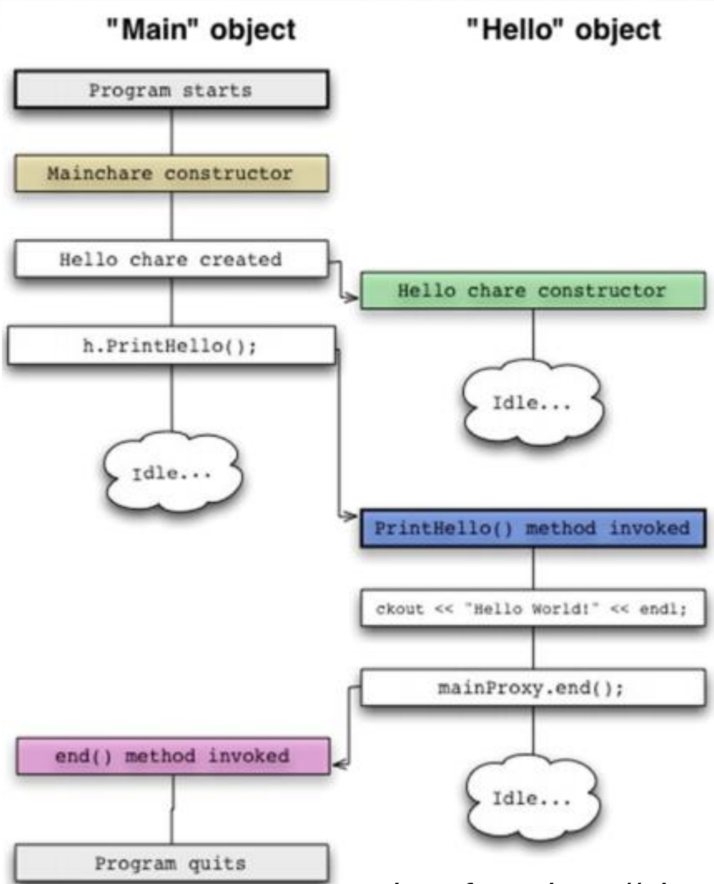
- Can build this on top of communication technologies like MPI
  - When sending package the data and metadata (unique ID etc) up and send as type MPI_BYTE
  - On the receiver side can probe for a messages and extract the message size (and source) from the status, allocate memory and then physical receive data (via MPI_Recv.)
    - Might be driven by a thread continually polling for incoming data

- Some implementation challenges
  - What if we have not yet registered a receive handler for a specific message but this message has arrived? – Need to store unmatched messages
  - When should we terminate? –when all UEs are idle, there is no data in flight and no messages are outstanding

Active messaging technologies

- In other fields active messaging is fairly popular
  - Remote Procedure Call (RPC) is a concrete example of this such as Java's Remote Method Invocation (RMI)
- Not so much in HPC but Charm++ is one example technology
  - Built on C++, the programmer expresses their program components as parallel objects called *chares*
  - The programmer can call methods on these chares held on other processes, which is effectively an active message to execute that method remotely with the provided arguments in a thread
  - As methods in a chare can share object data, by default only one method can be active at any one time (*one at a time concurrency protection – see shared data lecture.)*
  - NAMD, a popular molecular dynamics package is written in Charm++

Charm++ example




```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;

class Main : public Chare {
public:
  Main(CkArgMsg* m) {
    delete m;
    mainProxy = thishandle;

    CProxy_Hello h = CProxy_Hello::ckNew();
    h.PrintHello();
  }

  void end() {
    CkExit();
  }
};

class Hello : public CBase_Hello {
public:
  Hello() {}

  void PrintHello(void) {
    ckout << "Hello World!" << endl;
    mainProxy.end();
  }
};

#include "hello.def.h"
```

Taken from http://charm.cs.illinois.edu/research/charm

- Programmer must rewrite their code in C++ and this chares approach
  - An additional .ci file must be written that defines a proxy for each object and feeds into their compiler
- One at a time concurrently is limiting, can disable this but then is entirely up to the programmer to manage concurrency

Active messaging - Summary

- This way of structuring the communications can provide additional flexibility
  - Can be helpful when you have very many, asynchronous and different messages which you want to process in different ways
  - Using the unique identifier to match against handling logic means you can kick off lots of communications without worrying too much about the ordering in which they will arrive
- Structuring the code in this manner can help organise the concurrency
  - Especially if you allow for multiple handlers to execute concurrently
  - Each handler can be viewed as a task, driven by the arrival of data. But it gets more challenging when these handlers need to interact or work with some shared data
  - There are existing programming technologies, but none are mature