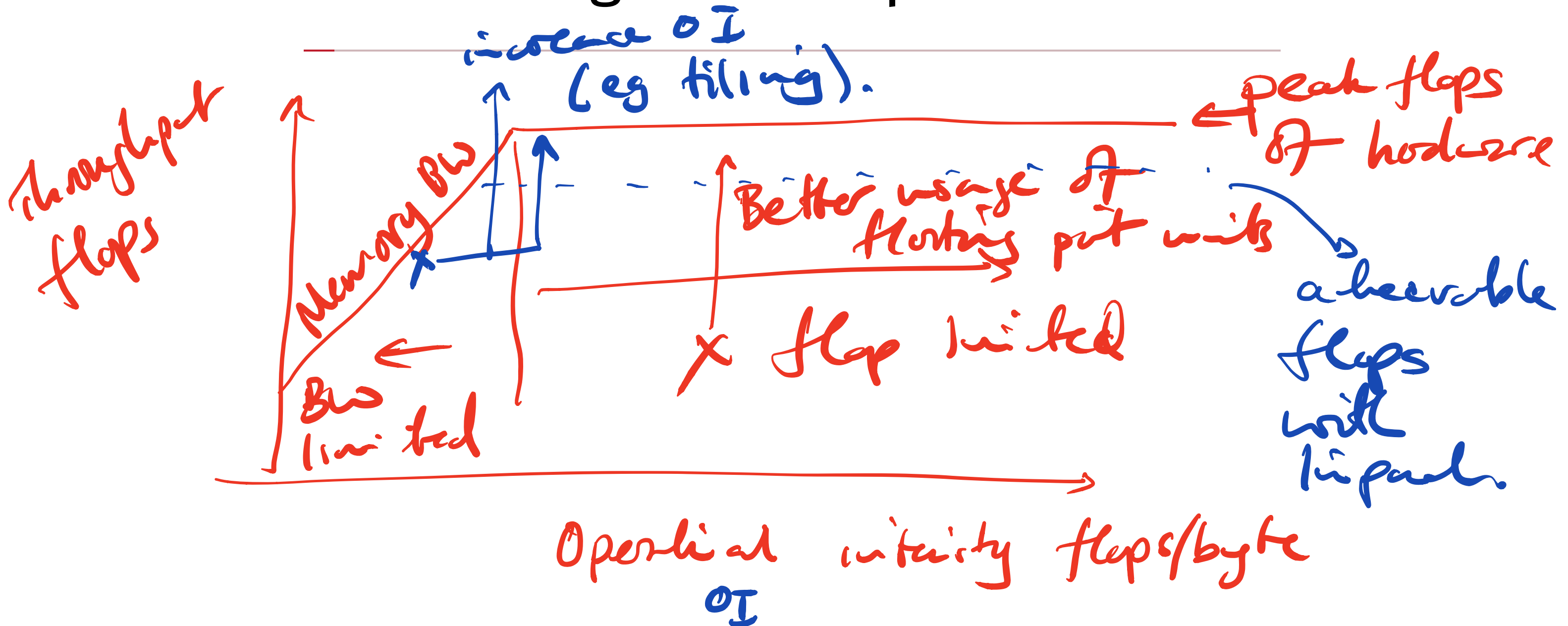
# Session 8: Roofline walkthrough + real-world code

## COMP52315: performance engineering

Lawrence Mitchell[*]

[*]lawrence.mitchell@durham.ac.uk

# Generating roofline plots



# Starting point

## Determine machine characteristics

1. Measure streaming memory bandwidth using `likwid-bench triad` benchmark.

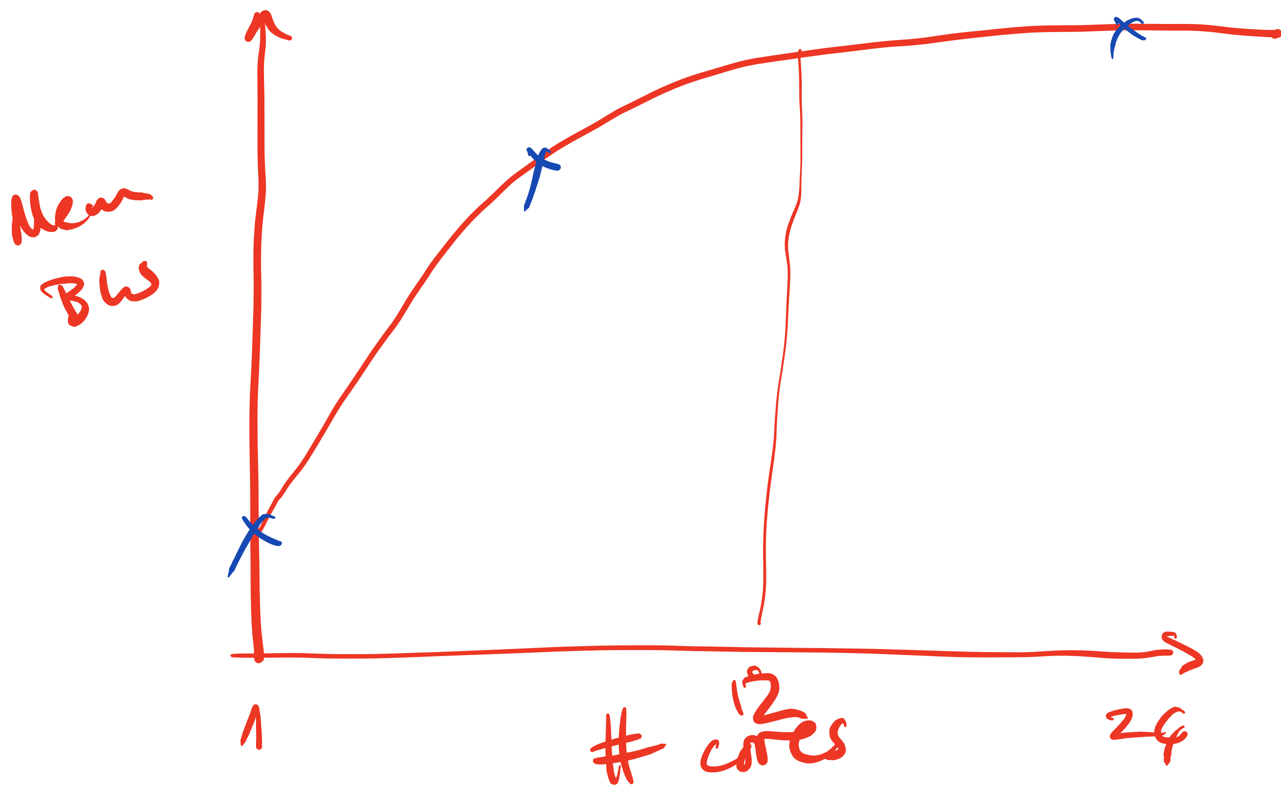
$\Rightarrow$ e.g. for single thread `likwid-bench -t triad -w S0:1GB:1`

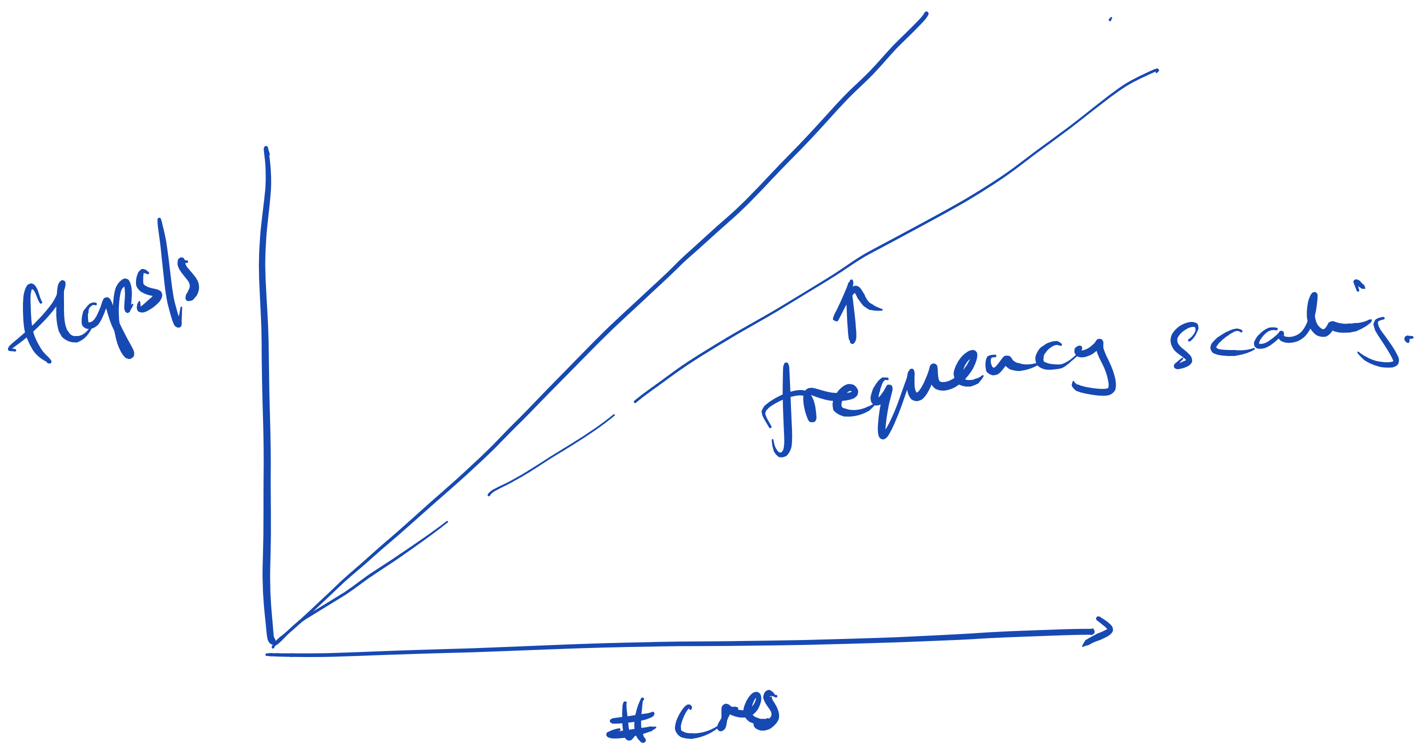
2. Calculate peak floating point performance

$$\text{Clock speed} \times \text{Vector width} \times \text{Issue width} \times \text{FMA}$$

3. On Hamilton nodes, for double precision, the vector width is 4, it is dual issue, and does support FMA. So we have

$$\text{Clock speed} \times 4 \times 2 \times 2 = \text{Clock speed} \times 16$$


Mem
BW
1
12
# cores
24


flops/s
frequency scaling.
# cores

# Machine characteristics: clock speed

- Peak clock speed is a variable, rather than fixed number
- For single-thread code, on Hamilton the cores can *turbo boost* to 2.9GHz. But if they run for too long, they will be clocked down.

$\Rightarrow$ can put multiple lines on roofline, for base frequency of 2.2GHz and turbo boost of 2.9GHz

# Application characteristics

## Just measure it

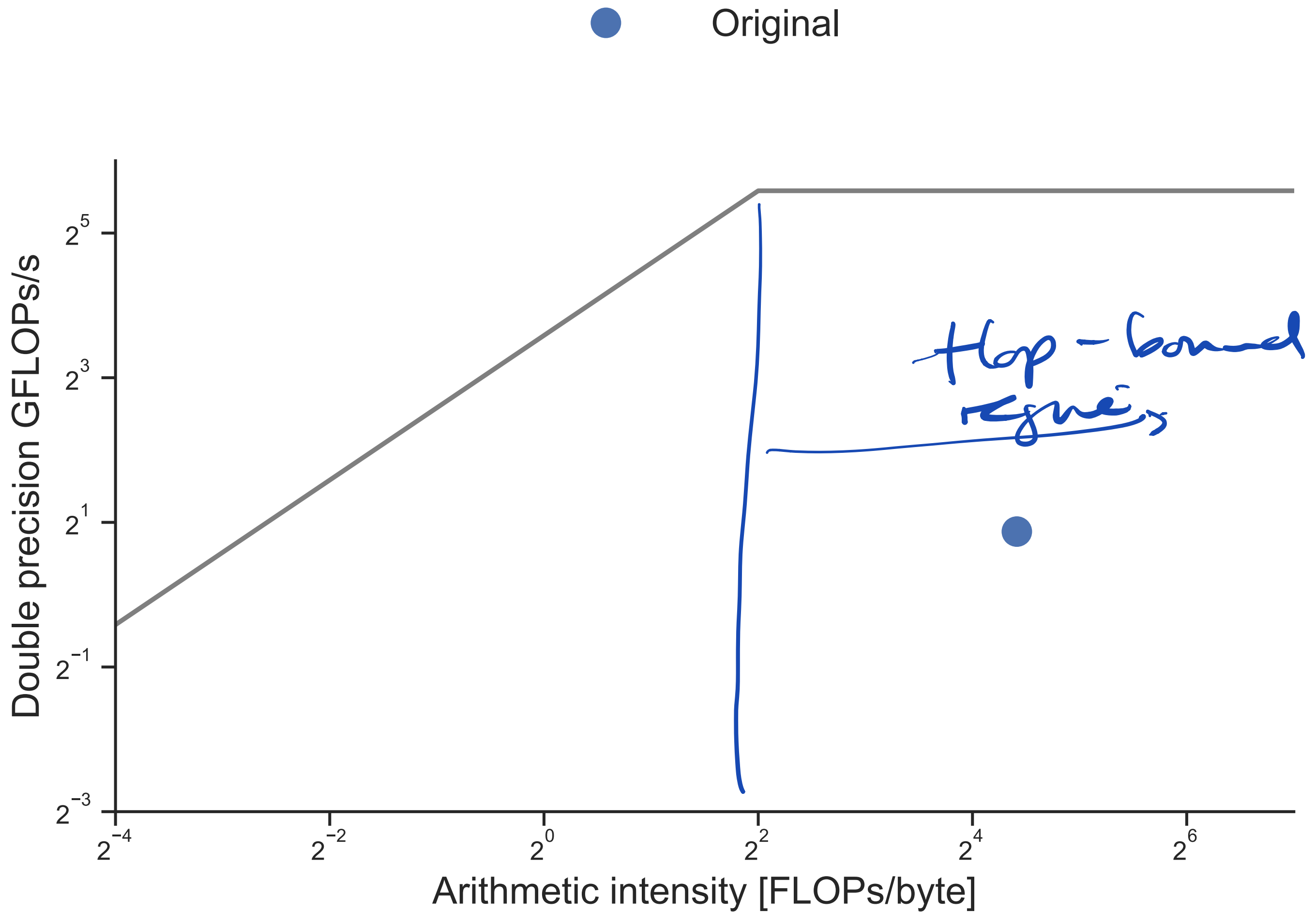
If you know the hotspot, `likwid-perfctr` can *measure* the arithmetic intensity.

```
likwid-perfctr -g MEM_DP ...
```

```
...
|          DP MFLOP/s            | 1831.3403 |
|      AVX DP MFLOP/s            |         0 |
|      Packed MUOPS/s            |         0 |
|      Scalar MUOPS/s            | 1831.3403 |
...
|  Memory bandwidth [MBytes/s]   |   85.8695 |
|  Memory data volume [GBytes]   |    0.2813 |
|     Operational intensity      |   21.3270 |
```

Now we can plot.

# Roofline plot

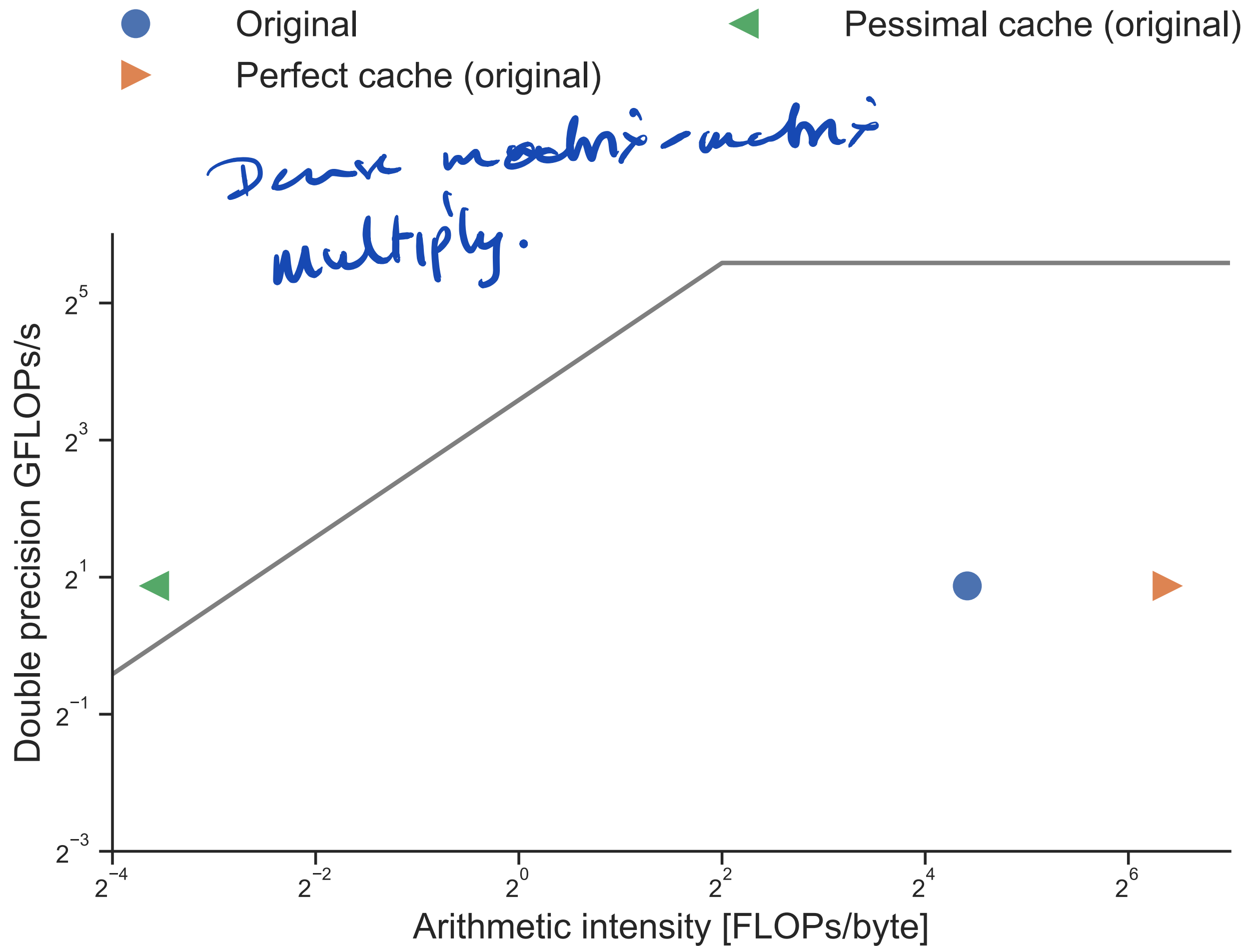


# Application characteristics: model

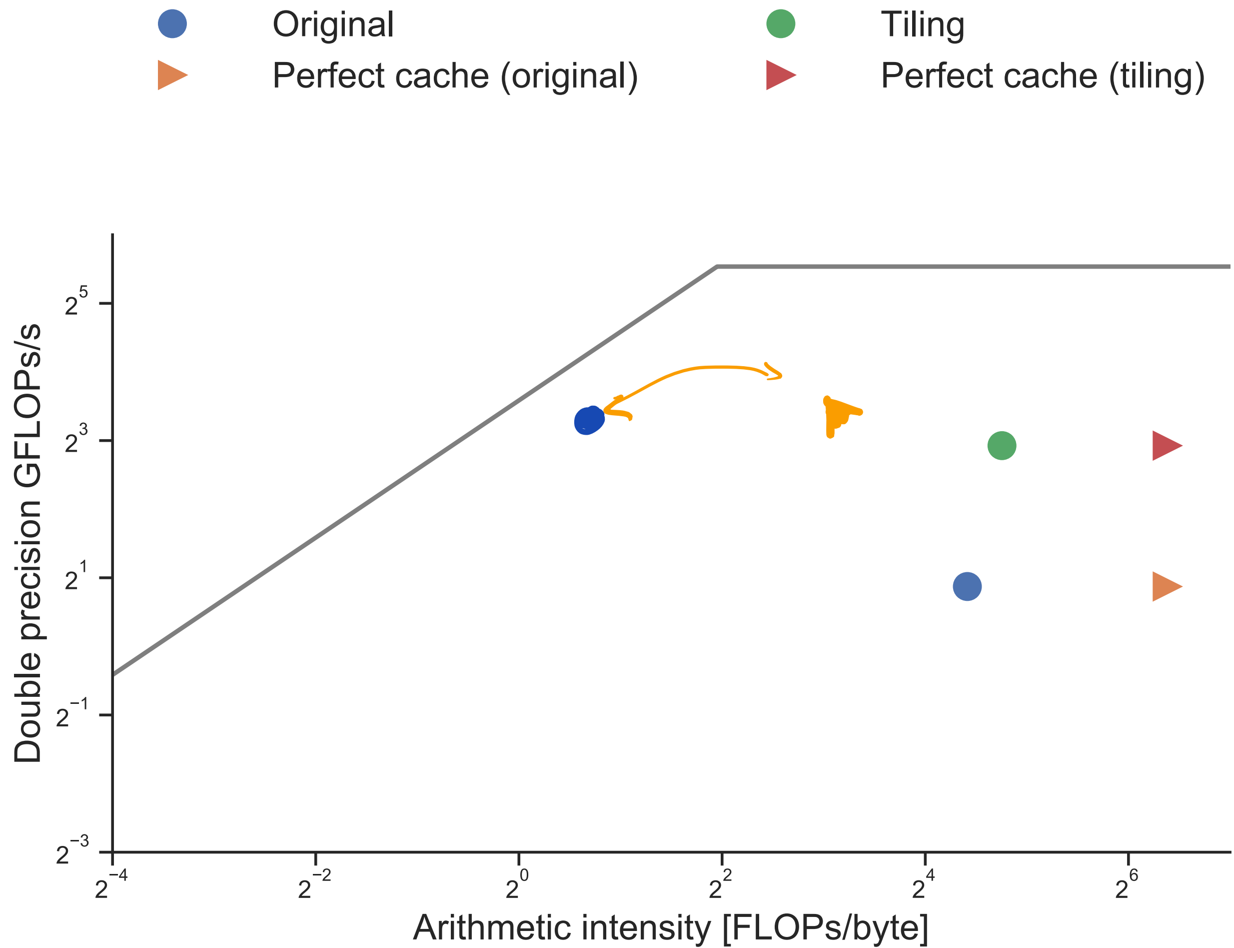
- Here it is clear that we should attempt to optimise for floating point throughput.
- We can use bounds on the data movement to bound the possible arithmetic intensity.
- Recall "perfect cache": each piece of data only loaded once
- And "pessimal cache": each piece of data loaded every time

$\Rightarrow$ Read code to count data access and floating point operations.

# Roofline with bounds

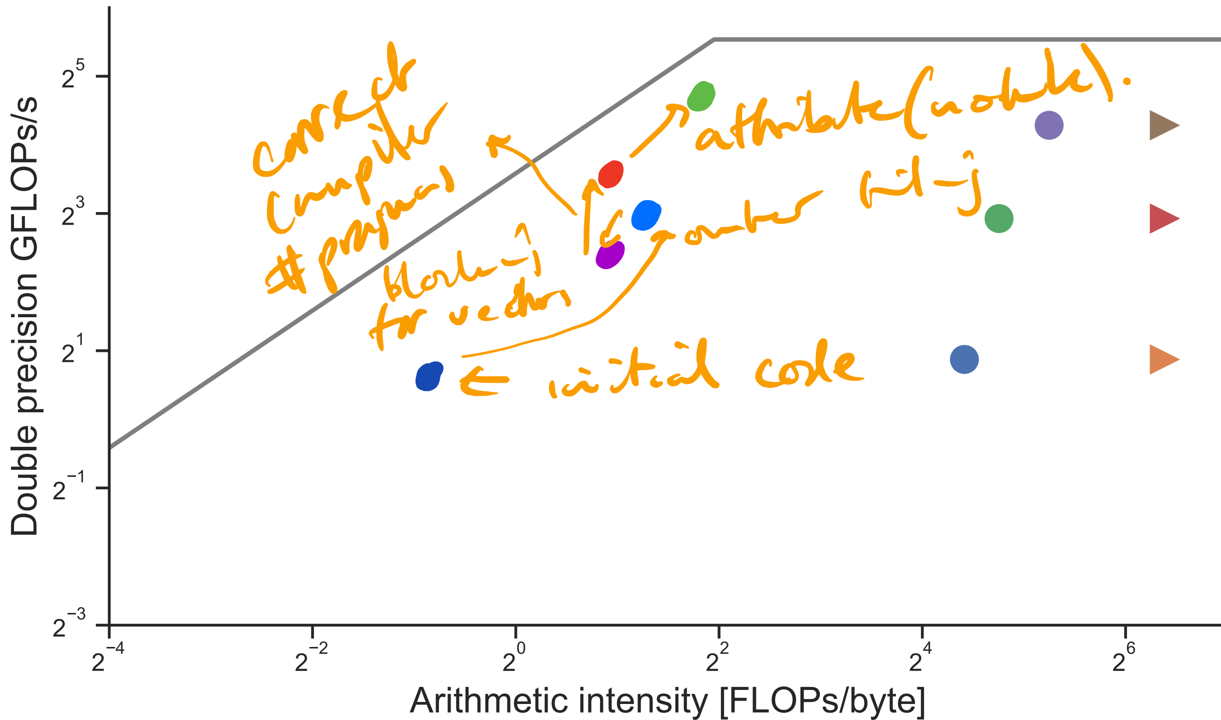


# Tiling

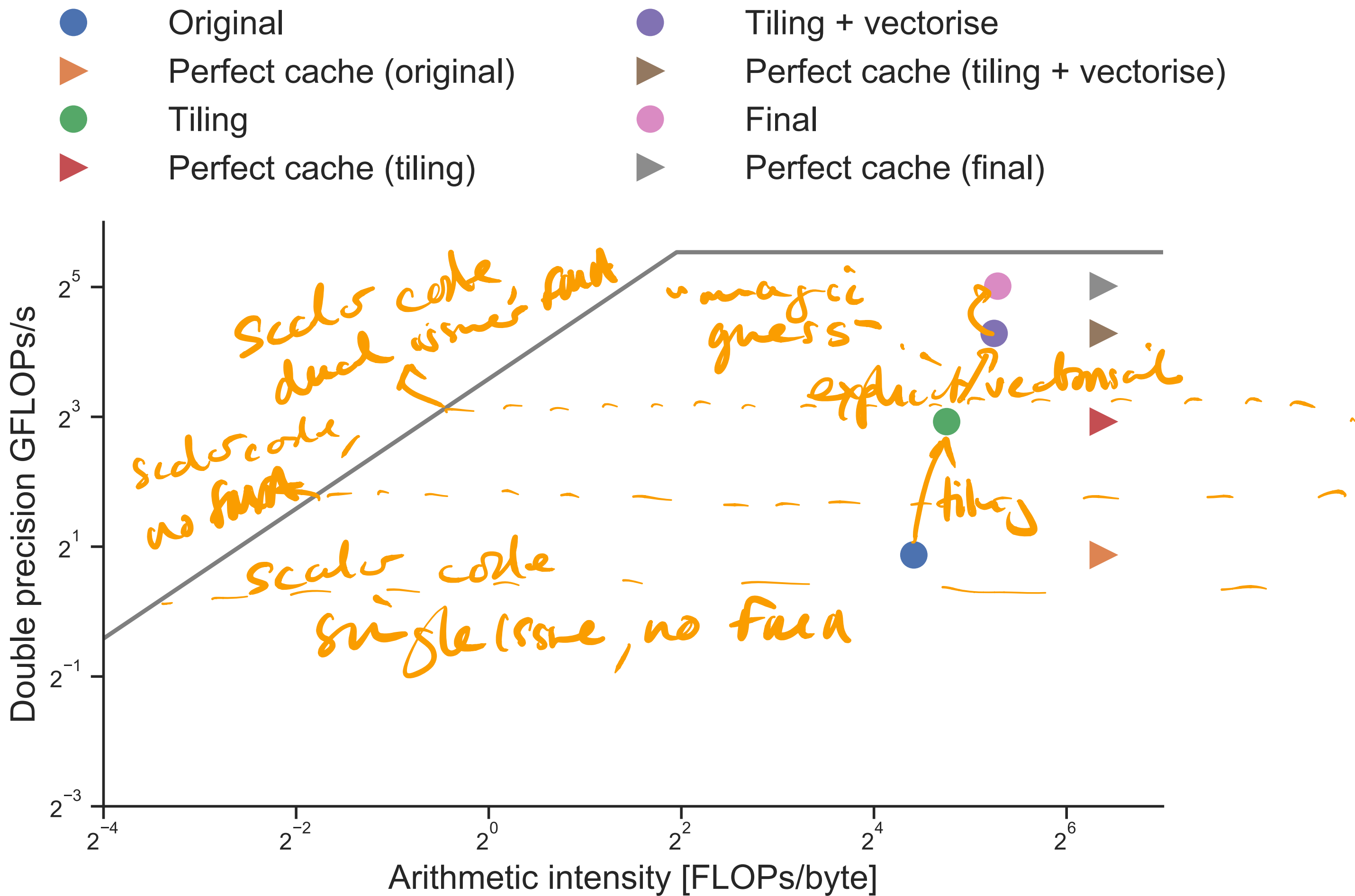


# Tiling + vectorisation

- Original
- Perfect cache (original)
- Tiling
- Perfect cache (tiling)
- Tiling + vectorise
- Perfect cache (tiling + vectorise)



# Tiling + vectorisation + compiler fiddling



# Application: high performance finite elements

# Application area

## Numerical solution of partial differential equations

Physical phenomena can be *modelled* using differential equations.

Example: viscous flow

`https://www.youtube.com/watch?v=p08_KlTKP50`

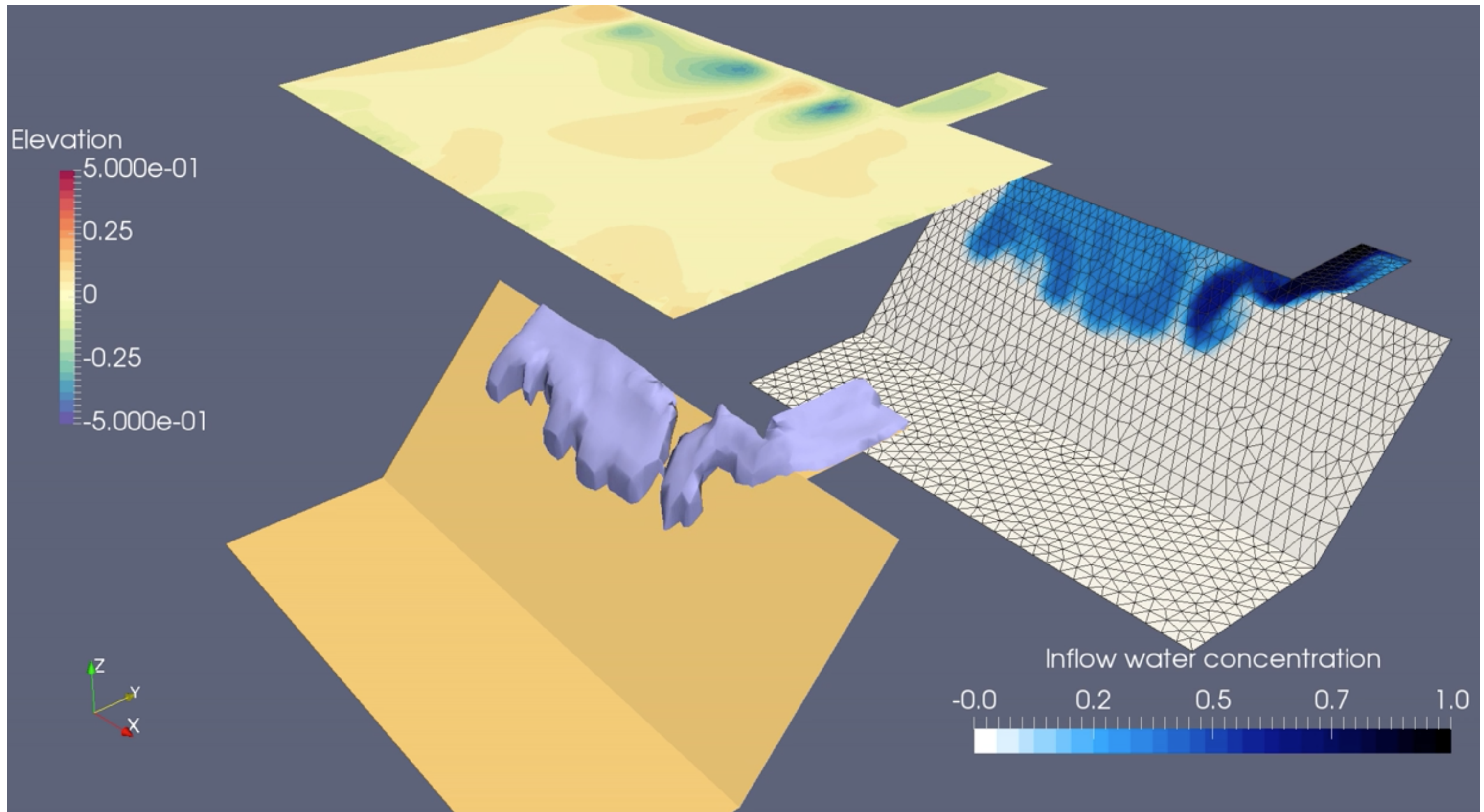
Modelled by Stokes' equations

$$\frac{\partial \mathbf{u}}{\partial t} - \nu \nabla^2 \mathbf{u} + \nabla p = f$$

$$\nabla \cdot \mathbf{u} = 0$$

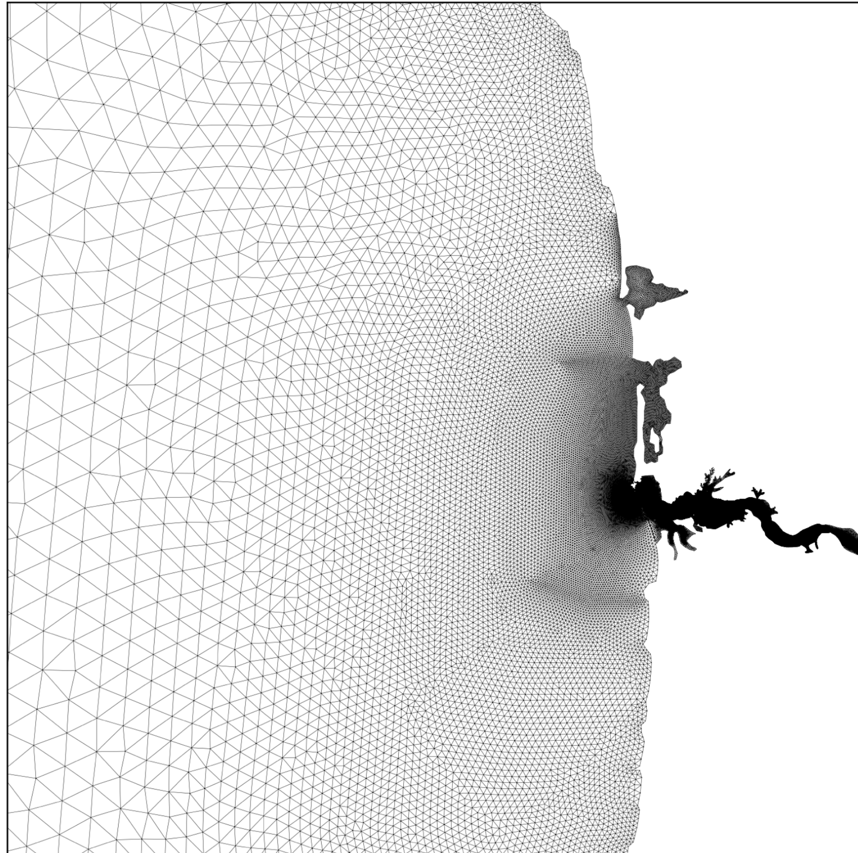
For complicated geometries, not possible to solve with pen and paper $\Rightarrow$ numerical solution.

Freshwater mixing

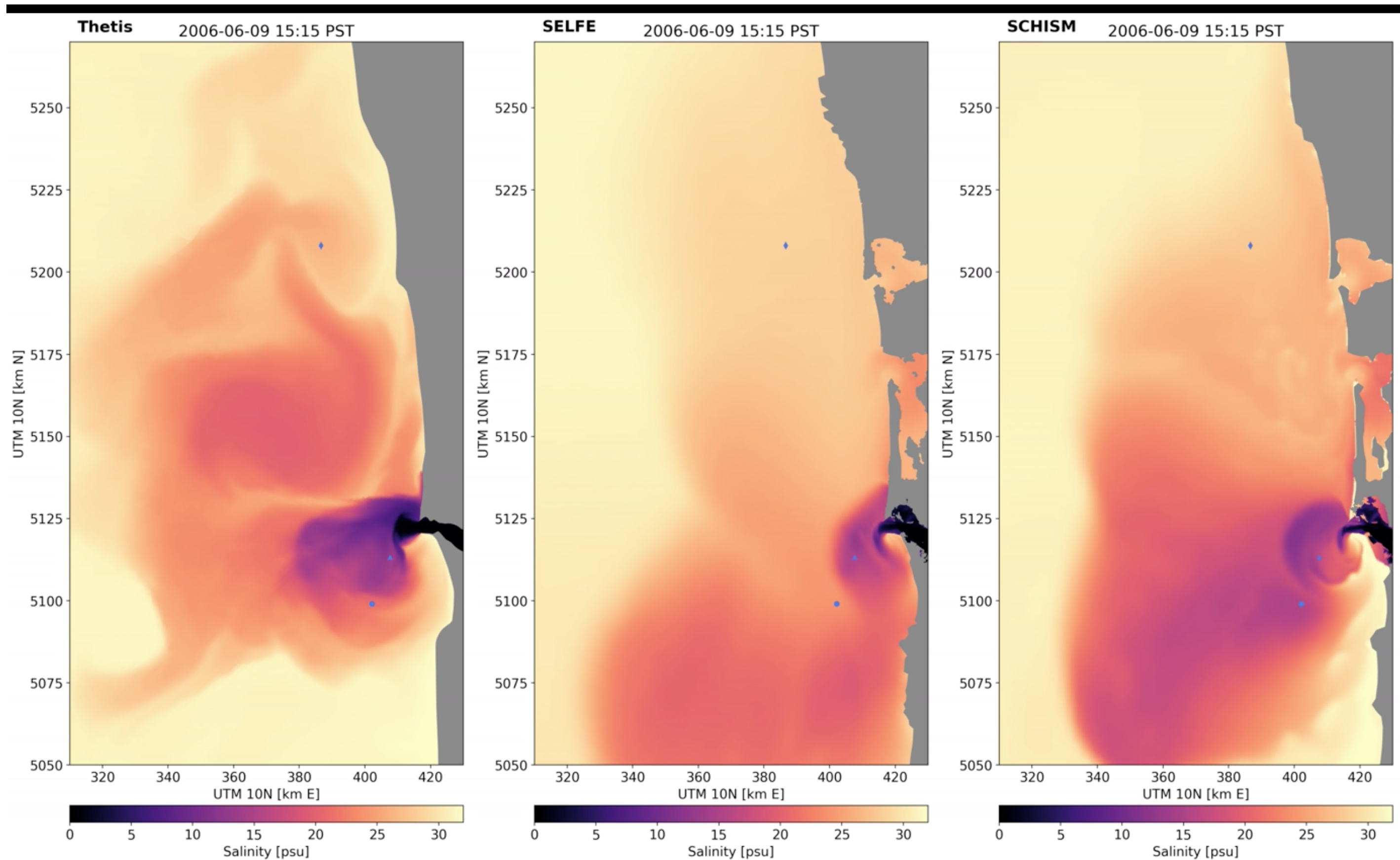


# Pictures

Freshwater mixing

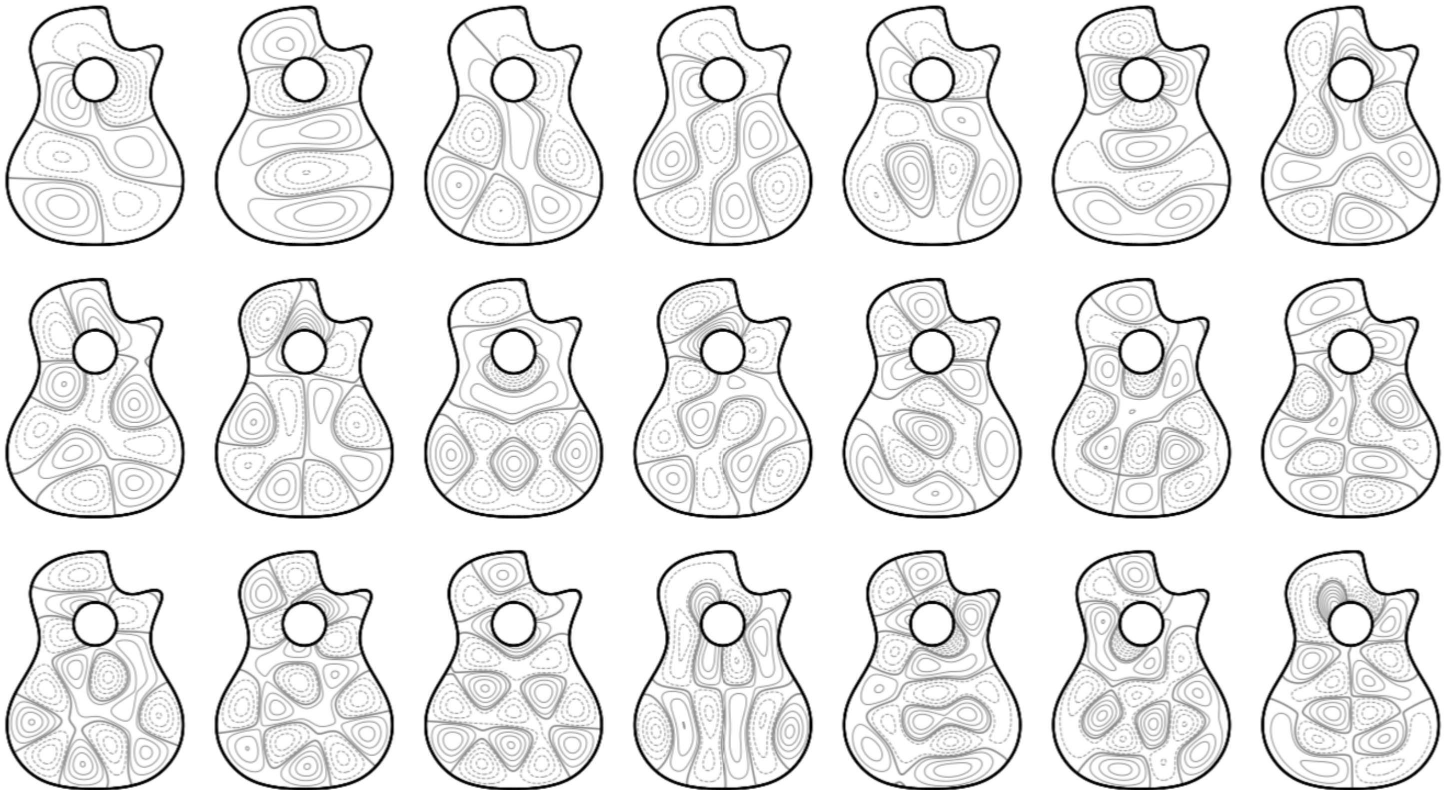


Freshwater mixing



# Pictures

Chladni plates `https://www.youtube.com/watch?v=wvJAgrUBF4w`



# Automated finite elements

**Challenge**

- Simulation software needs to exploit *fine-grained* parallelism.
- Most code intimately intertwines the numerical algorithm with its *implementation*.
- To apply program transformations, we have to unpick, understand, and reimplement.
- *Every time* the hardware changes.

⇒ Develop domain-specific language and *optimising* compilers for this language.

# Firedrake `www.firedrakeproject.org`

> [...] *an automated system for the solution of partial differential equations using the finite element method.*

- Written in Python.
- Finite element problems specified with embedded domain specific language.
- Domain-specific optimising compiler.
- Runtime compilation to low-level (C) code.

# Finite elements I

$$F(u) = 0 \text{ in } \Omega$$
$$+ \text{ boundary conditions}$$

Seek weak solution in some space of functions $V(\Omega)$.

Now we need to solve the (infinite dimensional) problem, find $u \in V$ s.t.

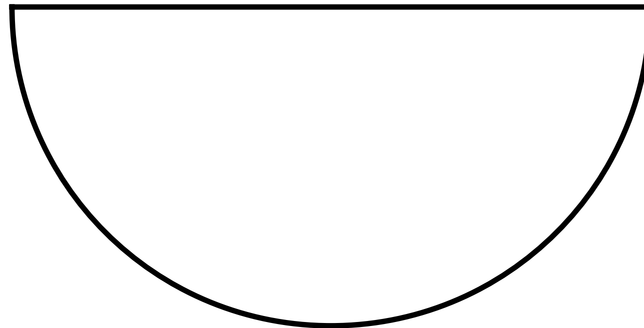
$$\int_\Omega F(u)v \,\mathrm{d}x = 0 \quad \forall\, v \in V$$

Choose finite dimensional subspace $V_h \subset V$, find $u_h \in V_h$ s.t.

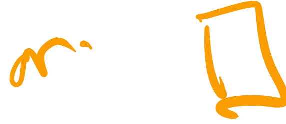
$$\int_\Omega F(u_h)v_h \,\mathrm{d}x = 0 \quad \forall\, v_h \in V_h$$

# Finite elements II

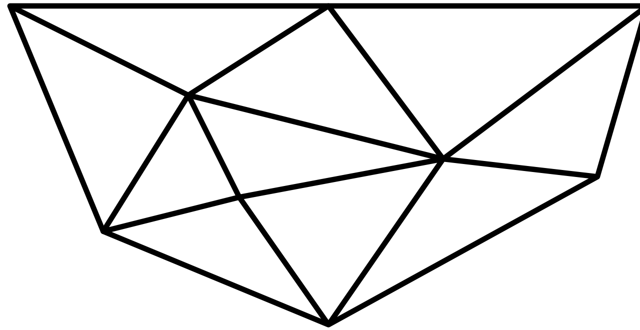
Divide domain $\Omega$...



# Finite elements II

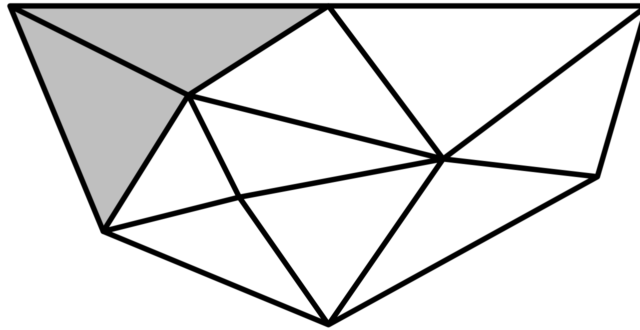


…into triangulation $\mathcal{T}$…



# Finite elements II

…and choose basis with finite support.



# Finite elements III

Integrals become sum over element integrals

$$\int_\Omega F(u_h) v_h \,\mathrm{dx} = \sum_{e \in \mathcal{T}} \int_e F(u_h) v_h \,\mathrm{dx}$$

$$\int_0^1 f(x)\, dx \approx f(\tfrac{1}{2})$$

(Usually) perform element integrals with numerical quadrature

$$\int_e F(u_h) v_h \,\mathrm{dx} = \sum_q w_q F(u_h(q)) v_h(q) \,\mathrm{dx}$$

Replace $u_h(q), v_h(q)$ with expansion in finite element basis

$$u_h(q) = \sum_i u_h^i \phi_i(q)$$

$$v_h(q) = \phi_j(q)$$

# Challenges

## Global

Meshes are *unstructured*, can't just use dense arrays.

Use sparse representation, needs graph algorithms for ordering and partitioning of work.

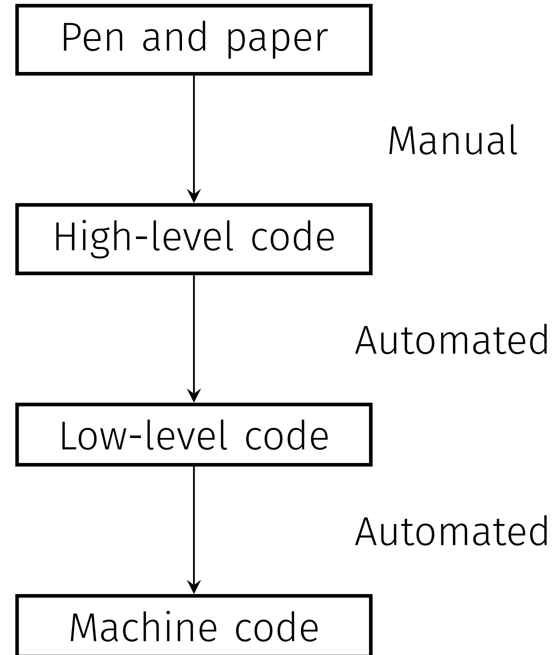
## Local

Computational *kernel* more complicated than simple stencil code.

# A domain-specific compiler

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x \quad \forall v \in V$$

```
V = FiniteElement("Lagrange", triangle, 1)
u = TrialFunction(V)
v = TestFunction(V)
F = dot(grad(u), grad(v))*dx
```



# Typical generated code

Flops on local data

```c
static inline void kernel(double A[6],
    const double *restrict coords,
    const double *restrict w_0) {
  static const double t3[12] = {...};
  static const double t4[12][6] = {...};
  double t0 = -1 * coords[0];
  double t1 = -1 * coords[1];
  double t2 =
    fabs((t0 + coords[2]) * (t1 + coords[5]) -
         (t0 + coords[4]) * (t1 + coords[3]));
  for (int ip = 0; ip < 12; ip += 1) {
    double t5 = 0.0;

    for (int i_0 = 0; i_0 < 6; i_0 += 1) {
      t5 += t4[ip][i_0] * w_0[i_0];
    }
    double t6 = t3[ip] * t2 * pow(t5, 2);

    for (int j = 0; j < 6; j += 1) {
      A[j] += t4[ip][j] * t6;
    }
  }
}
```

execute on global data

```c
int execute_kernel(int start, int end,
    double *A, int32_t *Amap,
    double *coords, int32_t *coordsmap,
    double *w_0, int32_t *w_0map) {
  for (int i = start; i < end; i++) {
    double lA[6] = {0};
    double lcoords[3, 2];
    double lw_0[6];
    /* Pack into contiguous element data */
    for (int i_0 = 0; i_0 < 3; ++i_0) {
      lcoords[i_0, 0] = coords[coordsmap1[i, i_0], 0];
      lcoords[i_0, 1] = coords[coordsmap1[i, i_0], 1];
    }
    for (int i_0 = 0; i_0 < 6; ++i_0) {
      lw_0[i_0] = w_0[w_0map2[i, i_0]];
    }
    /* Execute kernel on contiguous data */
    kernel(lA, lcoords, lw_0);
    /* Scatter result */
    for (int i_0 = 0; i_0 < 6; ++i_0) {
      A[Amap[i, i_0]] += lA[i_0];
    }
  }
  return 0;
}
```

← loop over elements

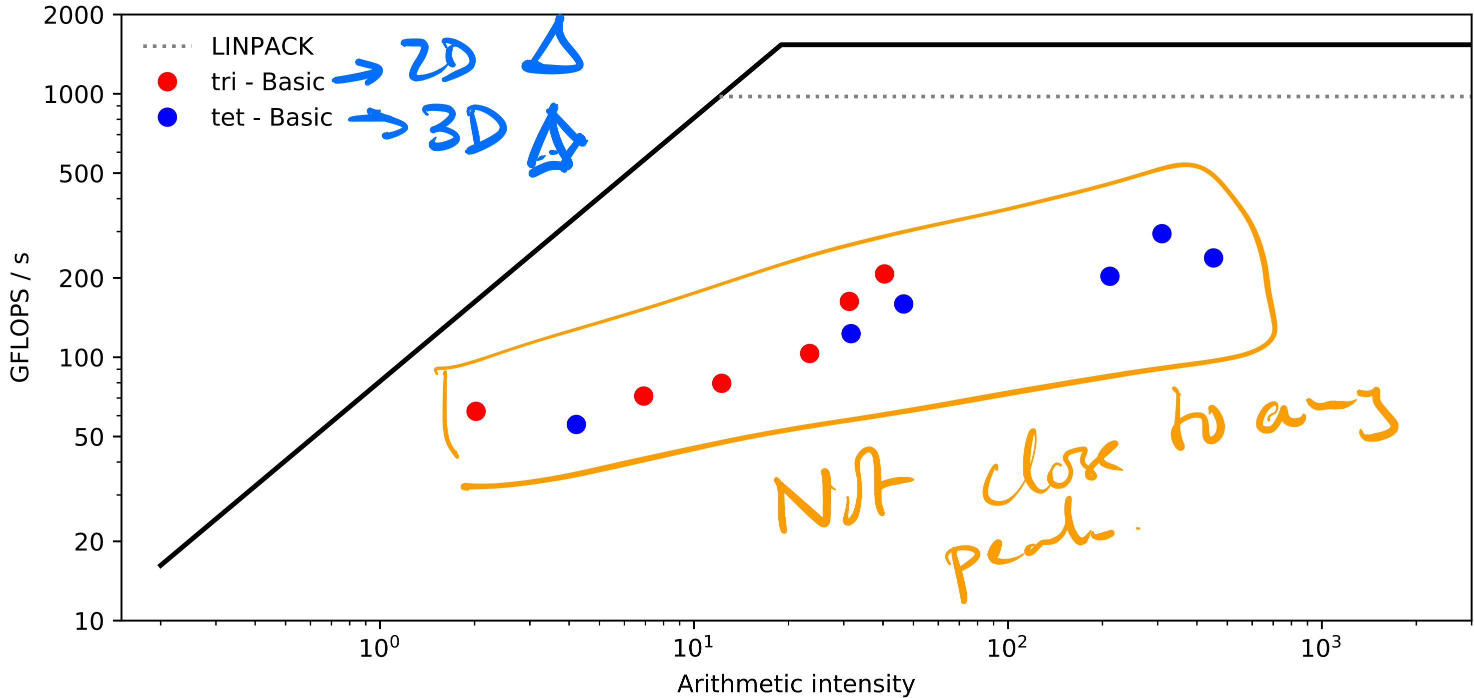
gather global data

execute kernel

produce result

# What does the performance look like?

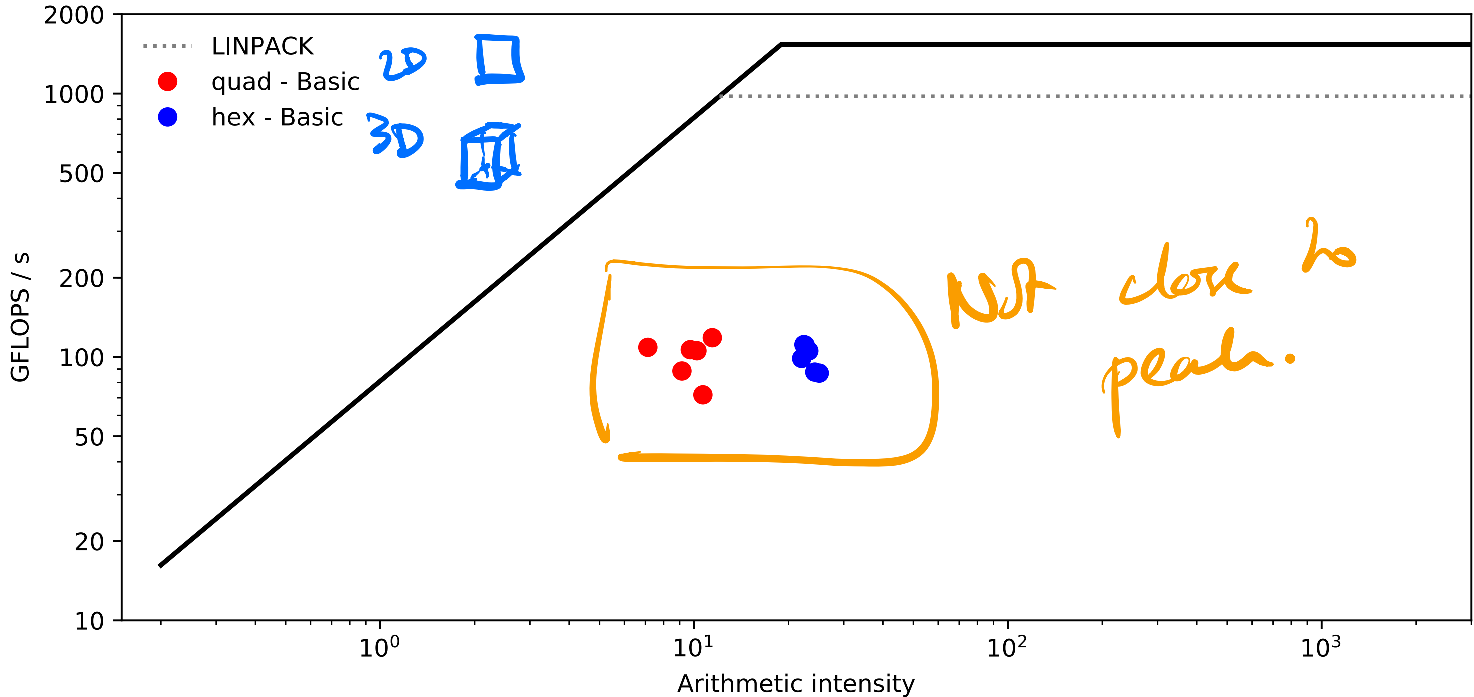
Full node of Xeon Gold (Skylake) with AVX512 instruction set.




Not great.

# What does the performance look like?

Full node of Xeon Gold (Skylake) with AVX512 instruction set.



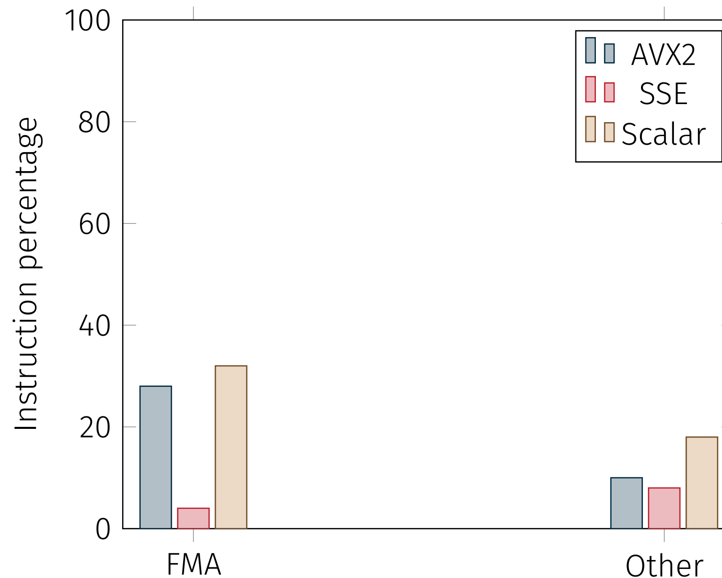

Not great.

# Cause

- No data point gets close to roofline limits

⇒ Hypothesis: compilers do a bad job of vectorising this code.

- Confirmed by measuring instruction mix.




⇒ Need to help compiler to generate better code.

- Local data layout transformation.

# Idea

- Enough *local* work for vectorisation to be effective
- ...but loop trip counts are unfriendly;

*inner loops have length 3 or 7, or ...*

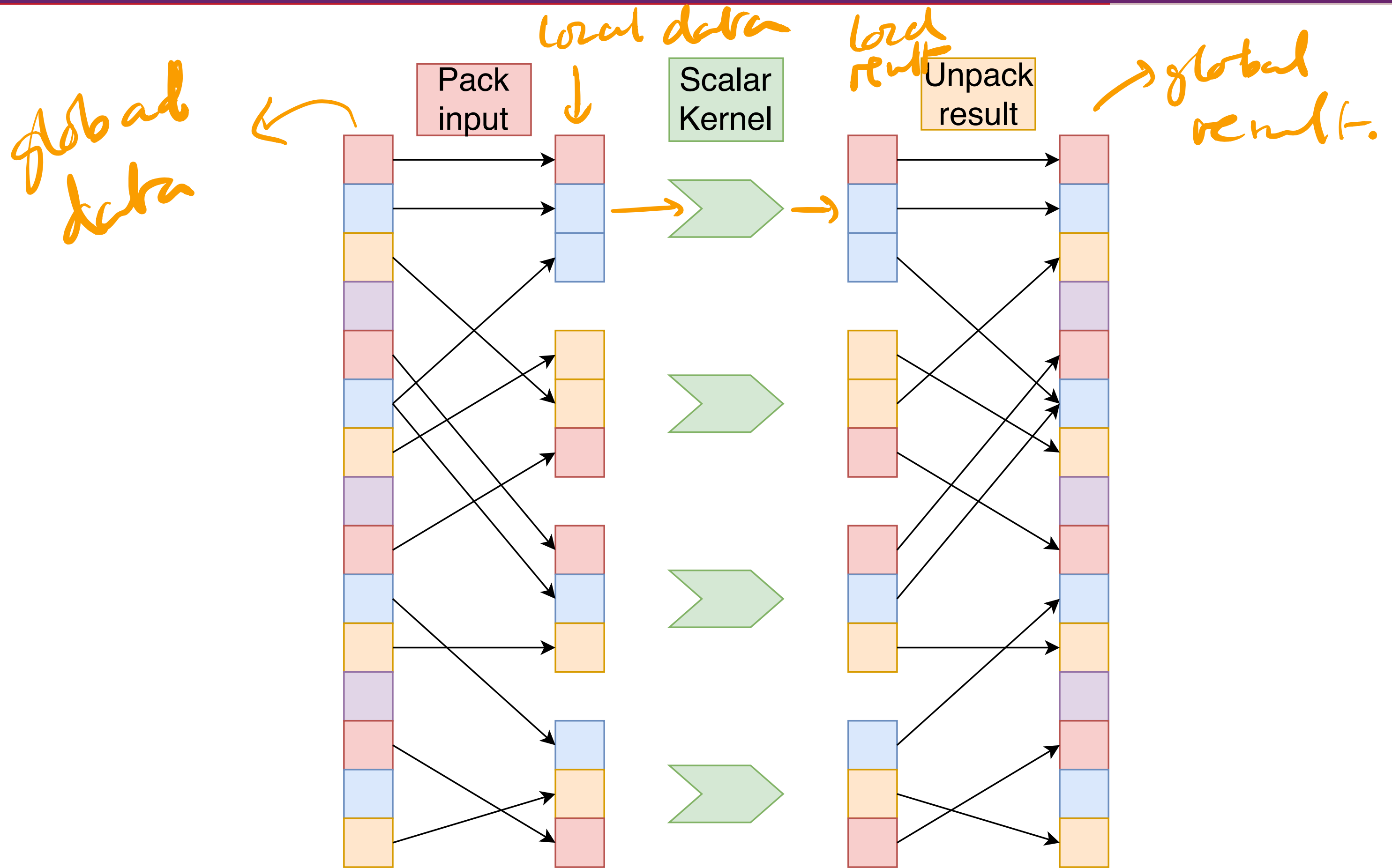
- ...and loop nests are hard to analyse.
- To obtain nice loops, need *outer loop* vectorisation and layout transformation

⇒ must do this by hand

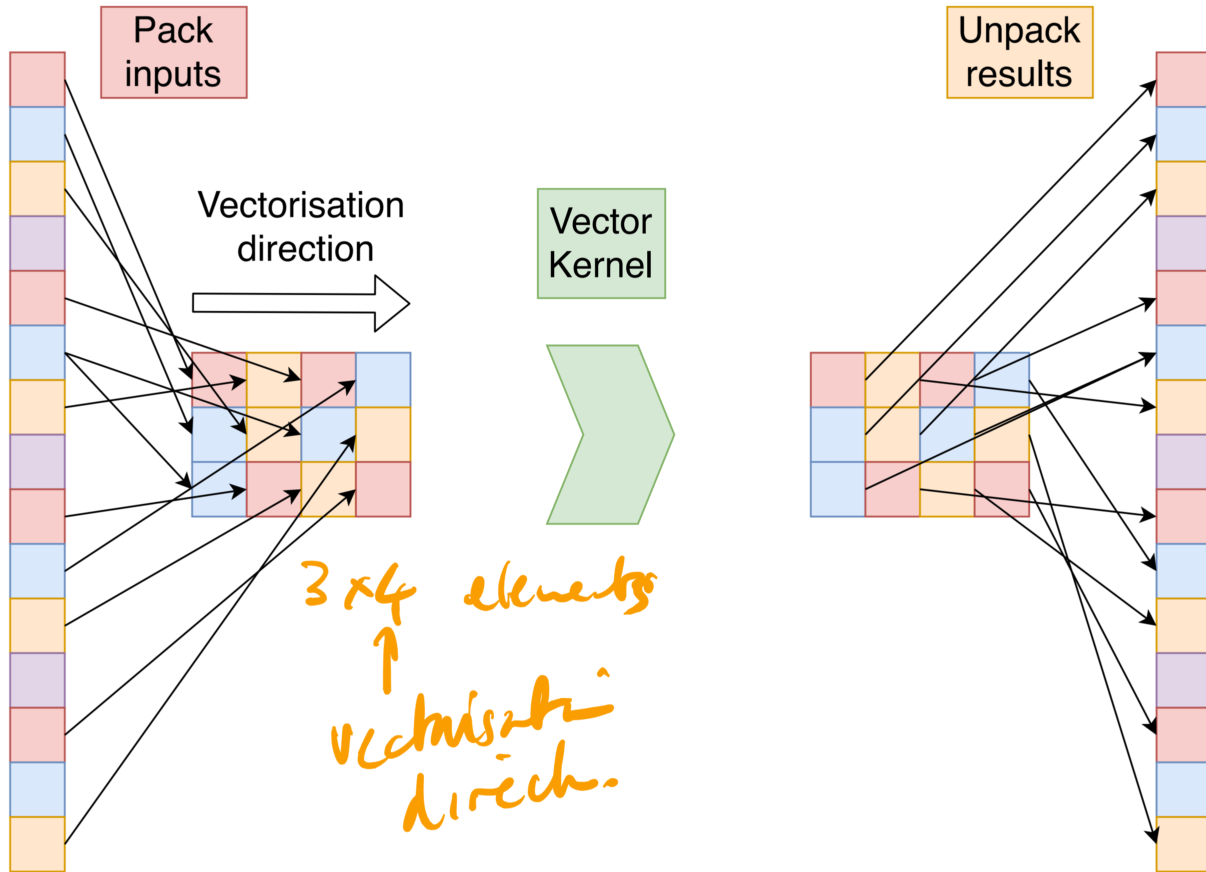
- Strategy: operate on `SIMD_WIDTH` elements at once.
- Dimension-lifted transposition approach, but for unstructured data

⇒ don't expect perfect speedup.

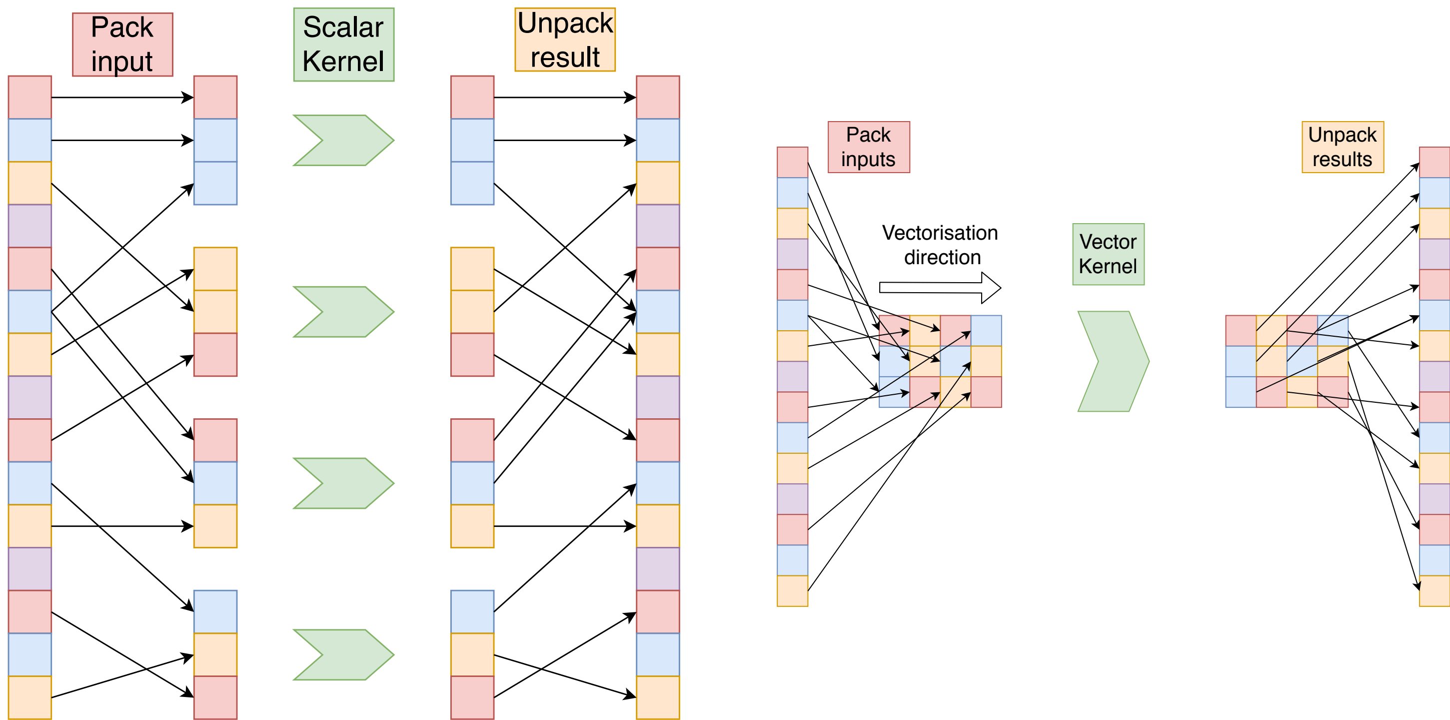
# Schematic: before



# Schematic: after



# Schematic: both



# Promoting vectorisation: kernel

## Hard to vectorise

```c
static inline void kernel(double A[6],
    const double *restrict coords,
    const double *restrict w_0) {
  static const double t3[12] = {...};
  static const double t4[12][6] = {...};
  double t0 = -1 * coords[0];
  double t1 = -1 * coords[1];
  double t2 =
    fabs((t0 + coords[2]) * (t1 + coords[5]) -
         (t0 + coords[4]) * (t1 + coords[3]));
  for (int ip = 0; ip < 12; ip += 1) {
    double t5 = 0.0;

    for (int i_0 = 0; i_0 < 6; i_0 += 1) {
      t5 += t4[ip][i_0] * w_0[i_0];
    }
    double t6 = t3[ip] * t2 * pow(t5, 2);

    for (int j = 0; j < 6; j += 1) {
      A[j] += t4[ip][j] * t6;
    }
  }
}
```

## Easy to vectorise

```c
static inline void kernel4(double A[6, 4],
    const double *restrict coords[4],
    const double *restrict w_0[4]) {
  static const double t3[12] = {...};
  static const double t4[12][6] = {...};
  for (int b = 0; b < 4; b++) {
    double t0 = -coords[0, b];
    double t1 = -coords[1, b];
    double t2 =
      fabs((t0+coords[2, b])*(t1+coords[5, b]) -
           (t0+coords[4, b])*(t1+coords[3, b]));
    for (int ip = 0; ip < 12; ip += 1) {
      double t5 = 0.0;

      for (int i_0 = 0; i_0 < 6; i_0 += 1) {
        t5 += t4[ip][i_0] * w_0[i_0, b];
      }
      double t6 = t3[ip] * t2 * pow(t5, 2);

      for (int j = 0; j < 6; j += 1) {
        A[j, b] += t4[ip][j] * t6;
      }
    }
  }
}
```

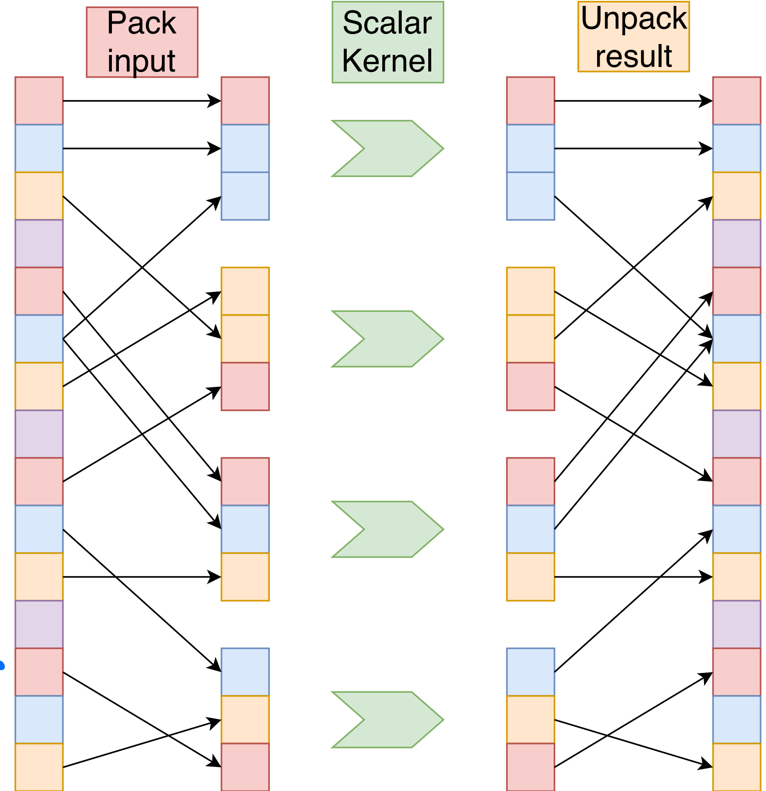
# Execution wrapper: before

```
int execute_kernel(int start, int end,
    double *arg0, int32_t *map0,
    double *arg1, int32_t *map1,
    double *arg2, int32_t *map2) {
  for (int i = start; i < end; i++) {
    double buffer0[6] = {0};
    double buffer1[3, 2];
    double buffer2[6];
      /* Pack into contiguous element data */
      for (int i_0 = 0; i_0 < 3; ++i_0) {
        buffer1[i_0, 0] = arg1[map1[i, i_0], 0];
        buffer1[i_0, 1] = arg1[map1[i, i_0], 1];
      }
      for (int i_0 = 0; i_0 < 6; ++i_0) {
        buffer2[i_0] = arg2[map2[i, i_0]];
      }
      /* Execute kernel on contiguous data */
      kernel(buffer0, buffer1, buffer2);
      /* Scatter result */
      for (int i_0 = 0; i_0 < 6; ++i_0) {
        arg0[map0[i, i_0]] += buffer0[i_0];
      }
  }
}
```

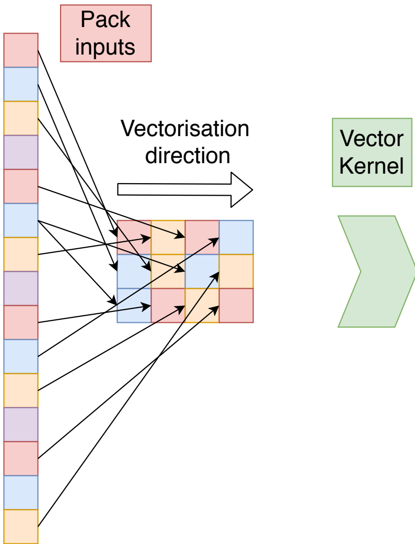
pack

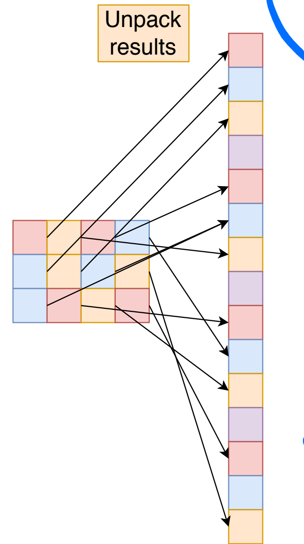
work

unpack



# Execution wrapper: after




```
int execute_kernel(int start, int end,
    double *arg0, int32_t *map0,
    double *arg1, int32_t *map1,
    double *arg2, int32_t *map2) {
  /* Stride by SIMD width */
  for (int i = start; i < end; i+=4) {
    double buffer0[6, 4] = {0.0};
    double buffer1[3, 2, 4];
    double buffer2[6, 4];
    /* Pack four elements into contiguous data */
    for (int i_0 = 0; i_0 < 3; ++i_0) {
      for (int b = 0; b < 4; b++) {
        buffer1[i_0, 0, b] = arg1[map1[i + b, i_0], 0];
        buffer1[i_0, 1, b] = arg1[map1[i + b, i_0], 1];
      }
    }
    for (int i_0 = 0; i_0 < 6; ++i_0) {
      for (int b = 0; b < 4; b++) {
        buffer2[i_0, b] = arg2[map2[i + b, i_0]];
      }
    }
    /* Execute kernel on contiguous data */
    kernel4(buffer0, buffer1, buffer2);
    /* Scatter result */
    for (int i_0 = 0; i_0 < 6; ++i_0) {
      for (int b = 0; b < 4; b++) {
        arg0[map0[i + b, i_0]] += buffer0[i_0, b];
      }
    }
  }
}
```

pack

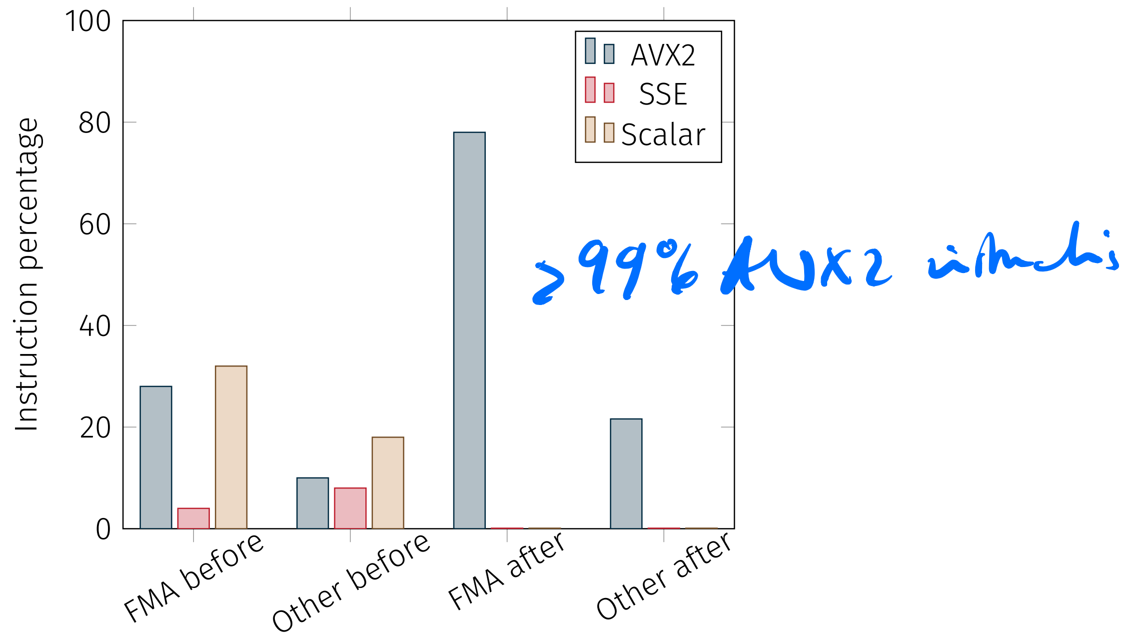
unpack

# New instruction mix

- Inner kernel annotated with `#pragma omp simd` appropriately

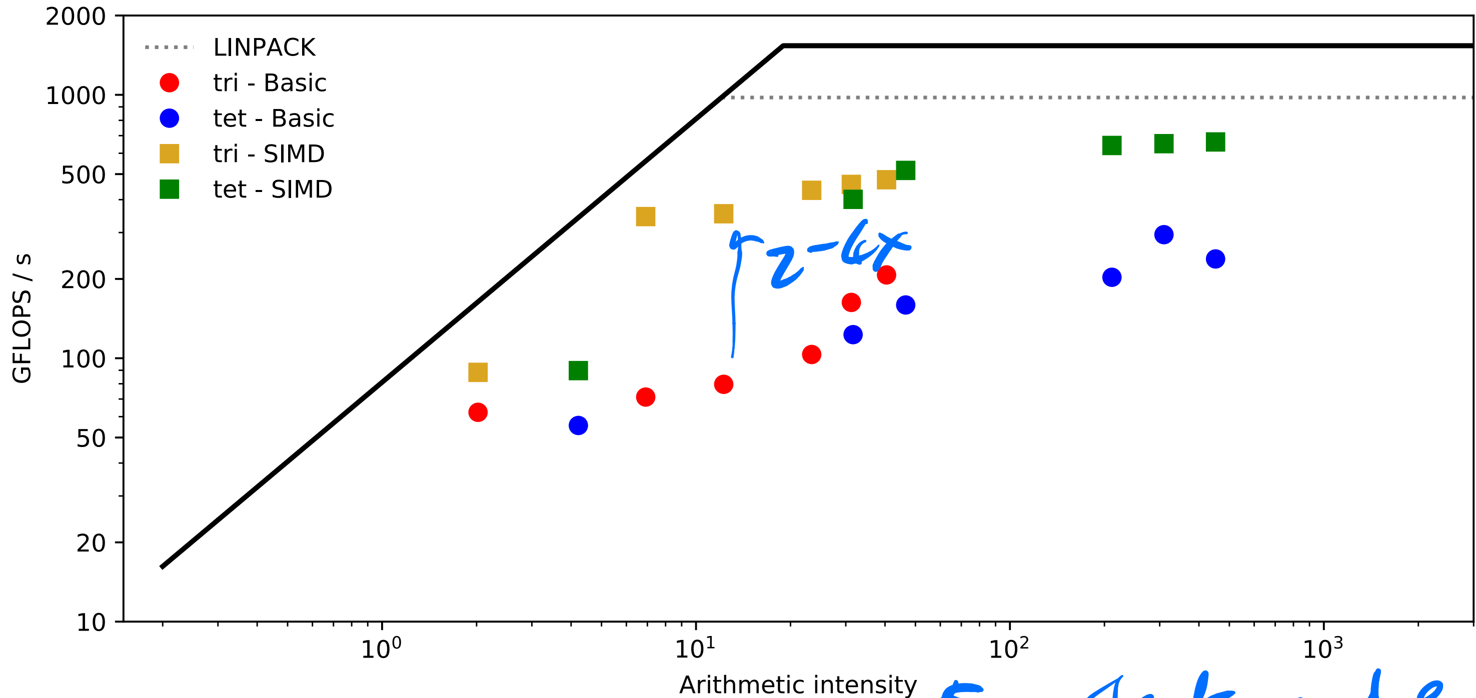
⇒ Compilers should do a good job vectorising this code (stride-1 inner loop, perfect data layout)

- Confirmed by measuring instruction mix.



# Performance significantly better

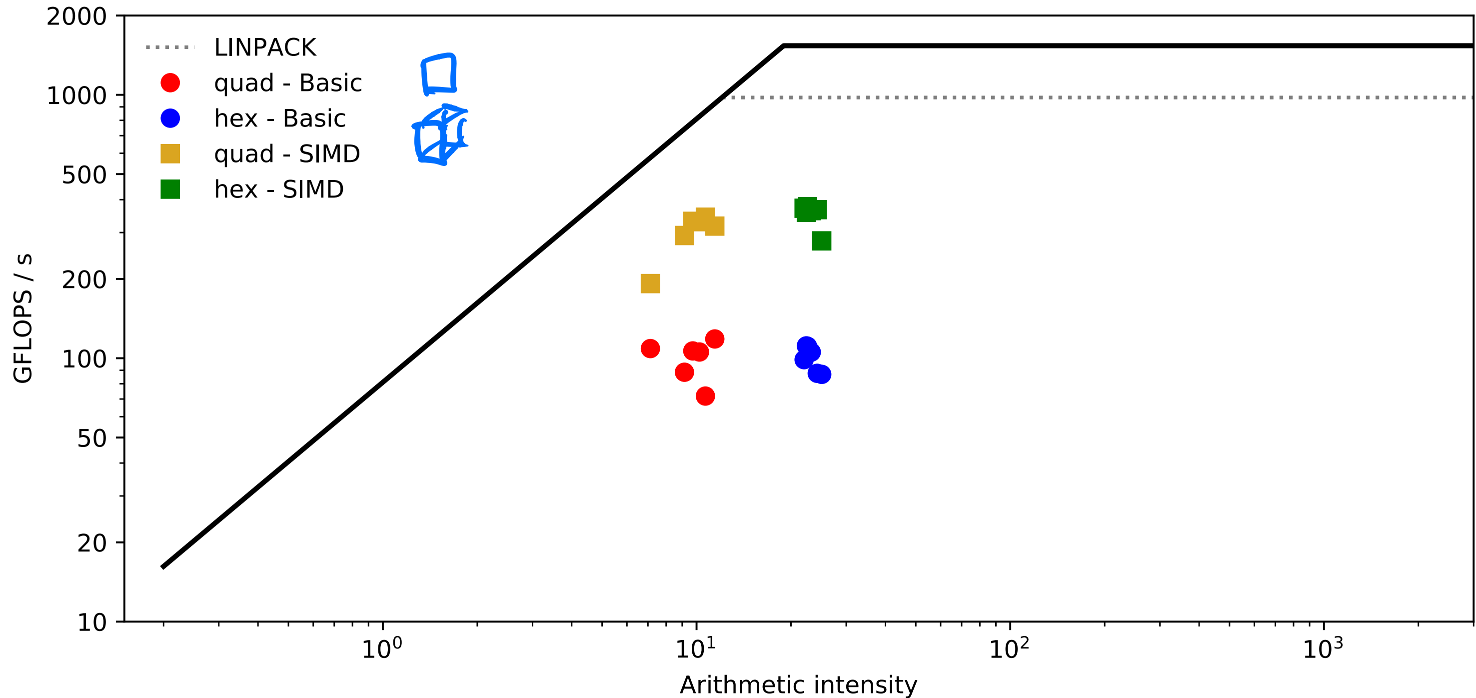
Full node of Xeon Gold (Skylake) with AVX512 instruction sets.




Significantly better.

# Performance significantly better

Full node of Xeon Gold (Skylake) with AVX512 instruction sets.




Significantly better.

# Conclusions

- Performance now limited by combination of unstructured data movement and instruction throughput.

$\Rightarrow$ needs more work for further gains.

- More details in `arXiv: 1903.08243 [cs.MS]`

**Projects using Firedrake**

If you're interested in either numerics-based or performance-based projects using numerical software, come and speak to me!