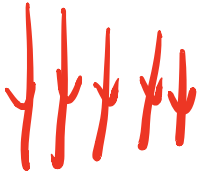
Session 7: Vectorisation and data layout

COMP52315: performance engineering

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk



Current status

So far

Only looked at simple data structures and loop nests

⇒ Loop unrolling, tiling, and judicious alignment sufficient for vectorisation

```
for (i = 0; i < N; i++)
  for (j = 0; j < K; j++)
    C[i] = f(a[i], b[j]);
```

Question

What about *more complicated* data structures or loops?

⇒ Need to consider data layout transformations *in tandem with* loop transformations

Motivating problems

Stencil computations

7-point Laplacian

$$B_{i,j,k} = A_{i-1,j,k} + A_{i,j-1,k} + A_{i,j,k-1} + A_{i+1,j,k} + A_{i,j+1,k} + A_{i,j,k+1} - 6A_{i,j,k} \quad \forall i, j, k$$

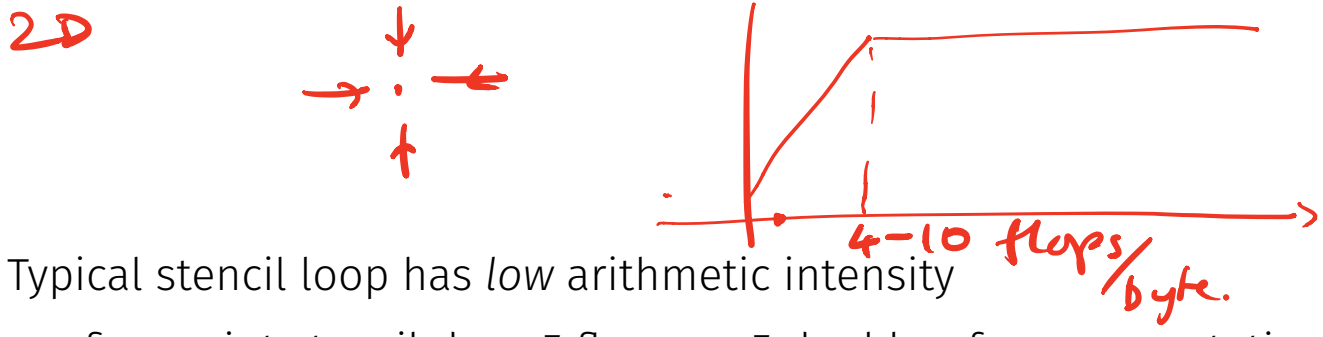
```
for (int i = 1; i < n-1; i++)
  for (int j = 1; j < n-1; j++)
    for (int k = 1; k < n-1; k++)
      B[i, j, k] = (A[i-1, j, k] + A[i, j-1, k] + A[i, j, k-1]
                  + A[i+1, j, k] + A[i, j+1, k] + A[i, j, k+1]
                  - 6*A[i, j, k]);
```

Structs

Computation on 3D points

```
struct point {
  double x, y, z;
};
for (int i = 0; i < n; i++) {
  l1dist[i] = fabs(points[i].x) + fabs(points[i].y) + fabs(points[i].z);
}
```

Observations



- Typical stencil loop has *low* arithmetic intensity
- e.g. five-point stencil does 5 flops on 5 doubles, for a computational intensity of $5/(5*8) = 1/8$ FLOPs/byte.

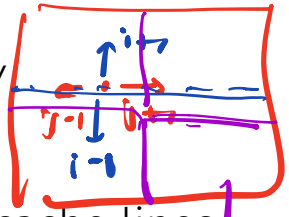
$\Rightarrow$ Relevant machine limit is *memory bandwidth*

- There is some data locality we can exploit, but not just stride-1 streams

$\Rightarrow$ tiling for locality is worthwhile

Complications

Also good for [i, j-1] & [i, j-1]



- Typewriter (standard loop) iteration has low *spatial locality*
- Have perfect access pattern for `[i, j]` indexing
- `[i+/-1, j]` indices miss cache every 8 updates (64 byte cache lines, double precision), similar to matrix-transpose example.

⇒ loop tiling of `i` and `j` to promote cache reuse.

should fit in cache.

```
N = n / b; /* Tile with block size b */
for (ii = 0; ii < N; ii += b)
  for (jj = 0; jj < N; jj += b)
    for (i = ii; i < min(n, i + b); i++)
      for (j = jj; j < min(n, j + b); j++)
        b[i, j] = a[i-1,j] + a[i+1,j] + a[i,j-1] + a[i,j+1] - 4*a[i,j];
```

data dependence

fastest varying index

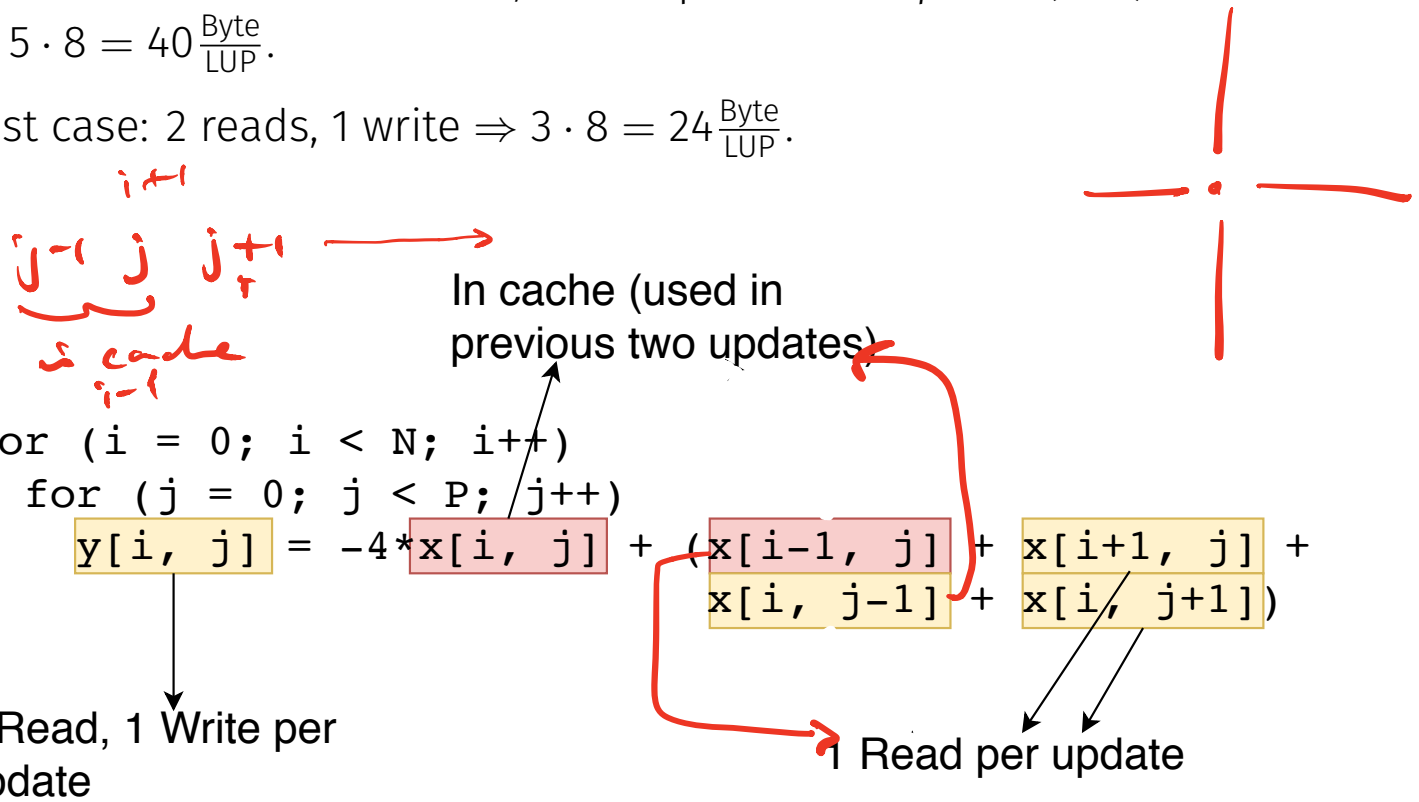
to vectorise, we need some compiler transformation.

- Still leaves us with problematic access patterns for vectorisation.

Layer condition for tile size

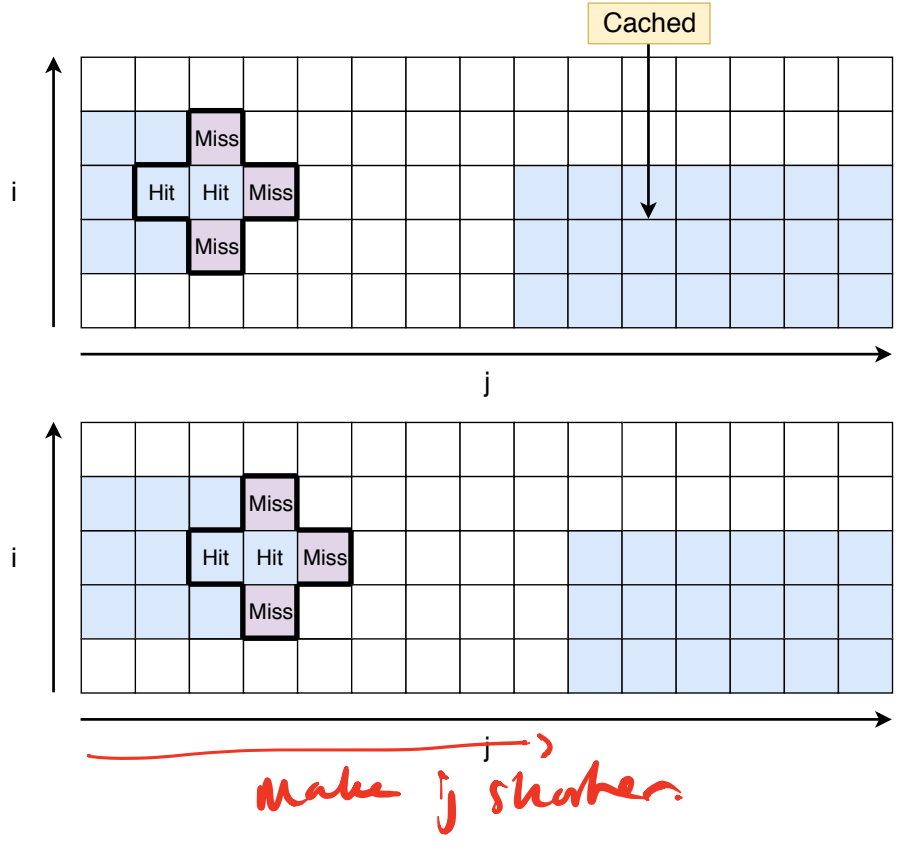
Realistic worst case: 4 reads, 1 write per *lattice update* (LUP)
$\Rightarrow 5 \cdot 8 = 40 \frac{\text{Byte}}{\text{LUP}}$.

Best case: 2 reads, 1 write $\Rightarrow 3 \cdot 8 = 24 \frac{\text{Byte}}{\text{LUP}}$.



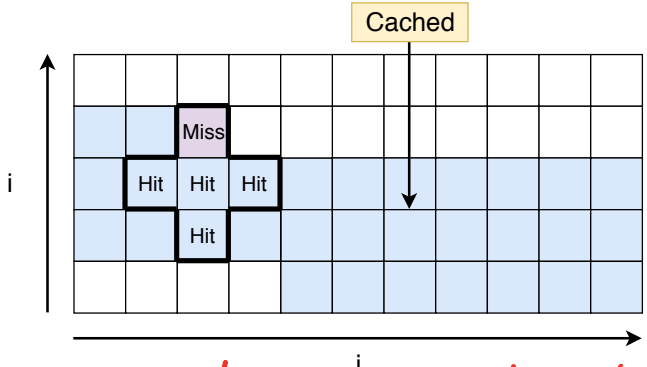
Picture

Worst case: cache *not large enough* to hold three layers (rows) of grid.



Layer condition solution

Tile inner loop dimension until layers fit in cache



2D stencil —!— needs layers in cache

Guideline

Choose blocking factor such that

$$3 \cdot \text{block size} \cdot 8\text{Bytes} < \text{CacheSize}/\underbrace{2}_{\text{Safety factor}}$$

double precision

tiled loop extent

authors of this research found it worked.

Spatial blocking

Before

```
for (i = 0; i < imax; i++)
  for (j = 0; j < jmax; j++)
    y[i, j] = (x[i-1,j] + x[i+1,j] + x[i,j-1]
                + x[i,j+1] - 4*x[i,j]);
```

After

```
for (jb = 0; jb < jmax; jb += jblock)
  for (i = 0; i < imax; i++)
    for (j = jb; j < min(jb + jblock, jmax); j++)
      y[i, j] = (x[i-1,j] + x[i+1,j] + x[i,j-1]
                  + x[i,j+1] - 4*x[i,j]);
```

$$\text{jblock} < \frac{\text{CacheSize}}{48\text{Bytes}}$$

Exercise

So far: good memory access ✓
not yet vectorised ✗

- Measure performance of 5 point stencil. Can you determine when layer condition is not fulfilled. What does using tiling do to the performance?

$\Rightarrow$ Exercise 10.

Loop reordering not enough

Might hope: memory limited ⇒ vectorisation not important ✗

```
for (t = 0; t < T; ++t) {
  for (i = 0; i < N; ++i)
    for (j = 1; j < N+1; ++j)
S1:   C[i][j] = A[i][j] + A[i][j-1];
  for (i = 0; i < N; ++i)
    for (j = 1; j < N+1; ++j)
S2:   A[i][j] = C[i][j] + C[i][j-1];
}
```

Performance:		
	AMD Phenom	1.2 GFlop/s
	Core2	3.5 GFlop/s
	Core i7	4.1 GFlop/s

(a) Stencil code

```
for (t = 0; t < T; ++t) {
  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
S3:   C[i][j] = A[i][j] + B[i][j];
  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
S4:   A[i][j] = B[i][j] + C[i][j];
}
```

Performance:		
	AMD Phenom	**1.9** GFlop/s
	Core2	**6.0** GFlop/s
	Core i7	**6.7** GFlop/s

(b) Non-Stencil code

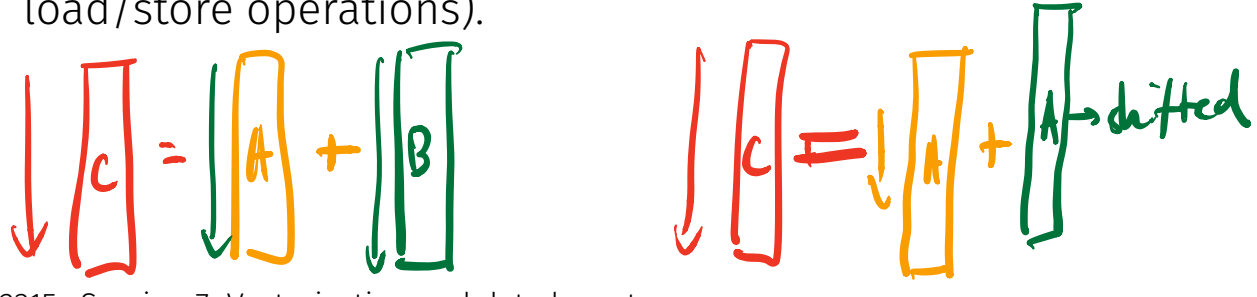
worse throughput.

From *Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures* (2011) `https://web.cs.ucla.edu/~pouchet/doc/cc-article.11.pdf`, doi:10.1007/978-3-642-19861-8_13

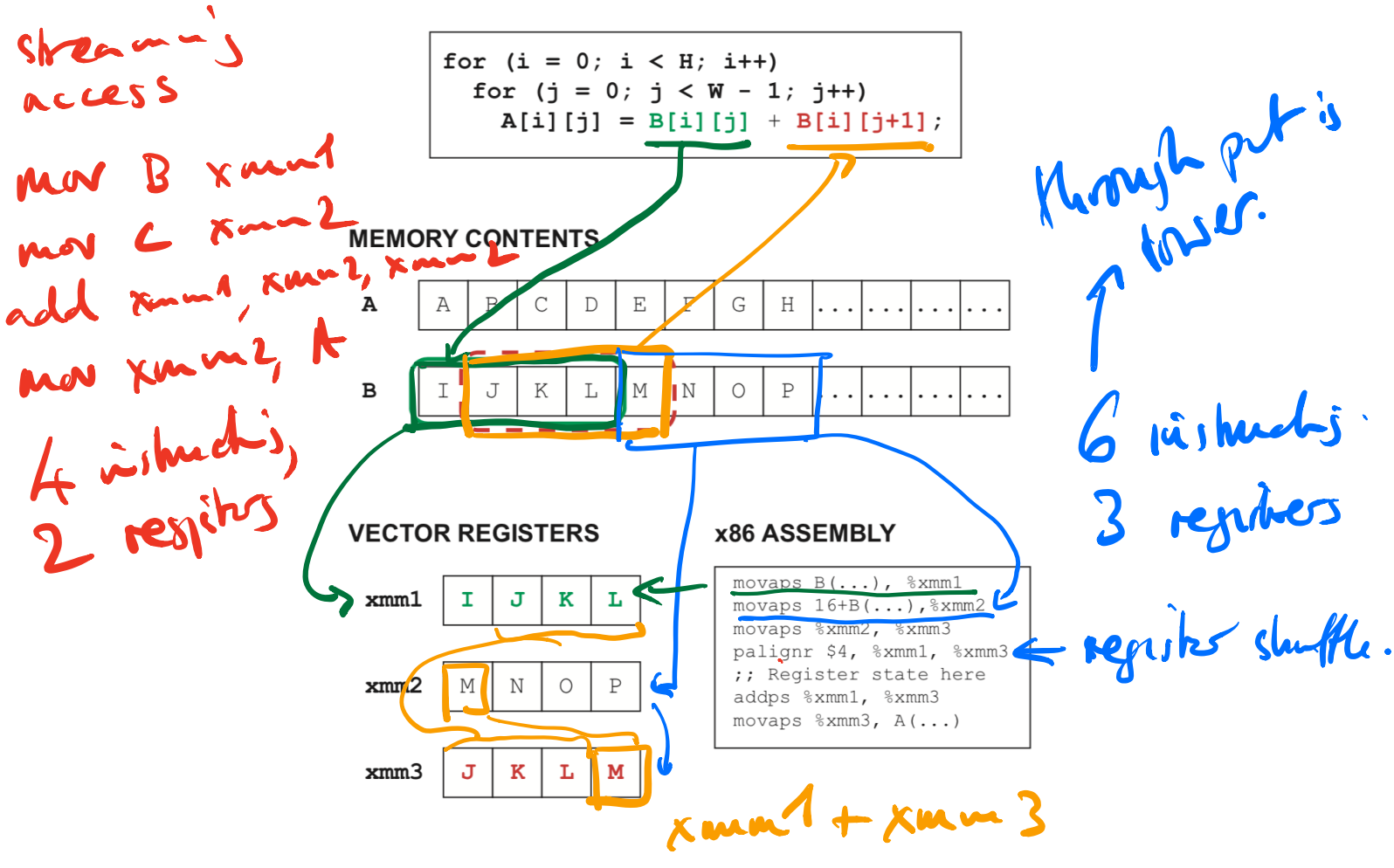
→ how do we fix this?

Loop reordering not enough

- Vectorisation in inner loops ⇔ operations on *streams* of *contiguous* data in memory.
- `C[i, j] = A[i, j] + B[i, j]` adds the stream of data `A[i, 1:N]` to another stream from `B[i, 1:N]`.
- In contrast `C[i, j] = A[i, j] + A[i, j-1]` adds `A[i, 1:N]` and `A[i, 0:N-1]`.
- These shifted streams necessitate inter-register shuffles (or explicit load/store operations).



Loop reordering not enough



A helpful observation

- Empirically, most code executes a small number of "hotspots" that access data over-and-over.
- In stencil codes this might often be an "outer" loop "time" or similar
- Therefore can be worthwhile performing a *data layout transformation* beforehand.
- Sometimes this can be done statically, sometimes only dynamically (if other parts of the program want data in a different format).

```
while (t < T) {
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      b[i, j] = a[i-1, j] + a[i+1, j] + a[i, j-1] + a[i, j+1] - 4*a[i, j];
    }
  }
}
```

Plan

- Two cases
  1. Single access pattern in program
  2. Multiple access patterns in program
- One solution
  - Look for *transformation* of data layout that gives better cache utilisation and vectorisation opportunities
- At one of two different times
  1. Single access pattern ⇒ compile-time (static) layout transformation
  2. Multiple access patterns ⇒ run-time (dynamic) layout transformation
- Latter case has a cost that is *amortized* over the subsequent accesses.

Vectorising on “streams”

- Need successive statements to have same access pattern (may be shifted) for vectorisation.

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S3:       A[i][j] = B[i][j] + C[i][j];
S4:       D[i][j] = A[i][j] + C[i][j];
      }
```

(a) No stream alignment conflict

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S5:       A[i][j] = B[i][j] + C[i][j];
S6:       D[i][j] = A[i][j] + C[i][j+1];
      }
```

(b) Stream alignment conflict exists

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S7:       A[i][j] = B[i][j] + C[i][j];
S8:       C[i][j] = A[i][j] + D[i][j+1];
      }
```

A[i][j+1] = B[i][j+1] + C[i][j+1];

(c) No stream alignment conflict

Vectorising on “streams”

- Need successive statements to have same access pattern (may be shifted) for vectorisation.

```
    for (i = 1; i < N+1; ++i)
      for (j = 0; j < N+1; ++j) {
S9:     A[i][j]     = B[i-1][i+j] +
                      B[ i ][i+j] +
                      B[i+1][i+j];
S10:    A[i+1][j]   = B[ i ][i+j+1] +
                      B[i+1][i+j+1] +
                      B[i+2][i+j+1];
      }
```

(a) No stream alignment conflict

```
    for (i = 1; i < N+1; ++i)
      for (j = 0; j < N+1; ++j) {
S11:    A[i][j]     = B[i-1][i+j+1] +
                      B[ i ][ i+j ] +
                      B[i+1][i+j+1];
S12:    A[i+1][j]   = B[ i ][i+j+2] +
                      B[i+1][i+j+1] +
                      B[i+2][i+j+2];
      }
```

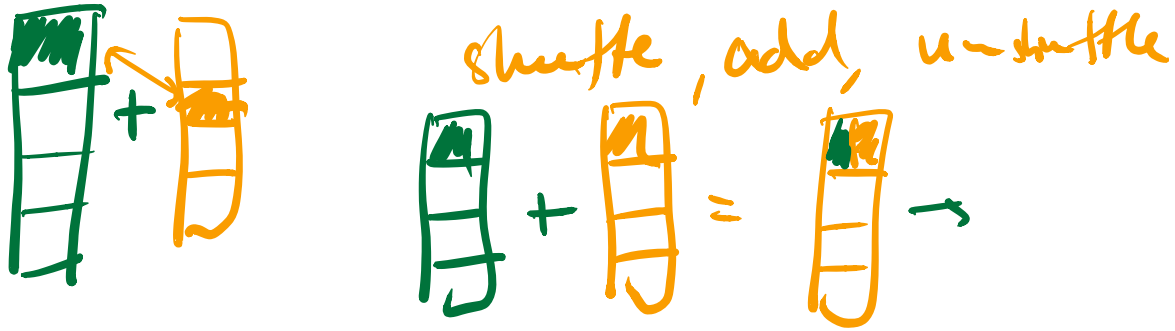
(b) Stream alignment conflict exists

Fig. 4. Illustration of stream alignment conflict with general array indexing expressions

Data layout transformation

Problem

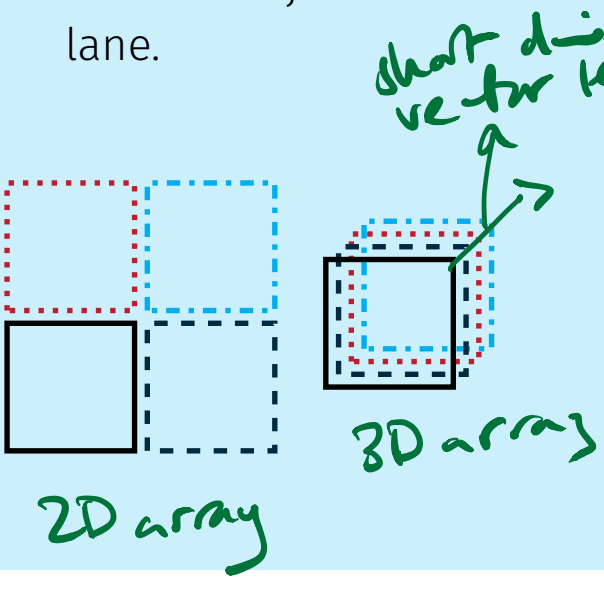
- Adjacent elements in memory map to adjacent slots (or lanes) in the vector registers.
- Vector operations on these adjacent elements therefore require more memory movement, or shuffling in registers.

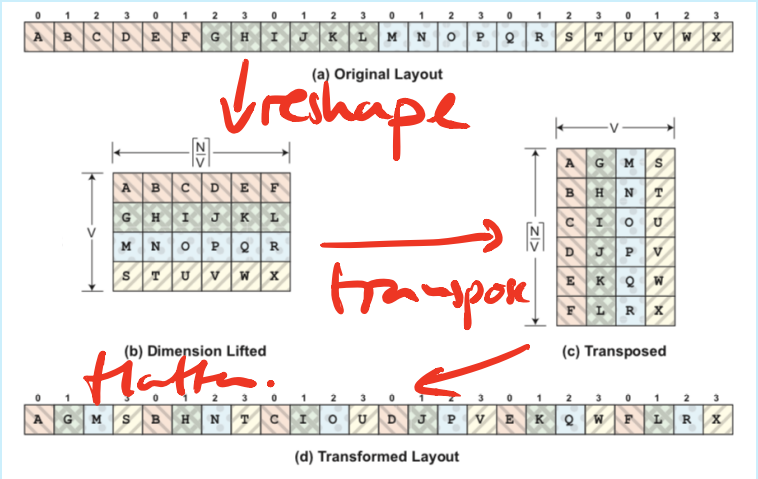


Data layout transformation

Solution

- *relocate* adjacent elements *such that* they can map to the same vector lane.





"Dimension lifted transposition".

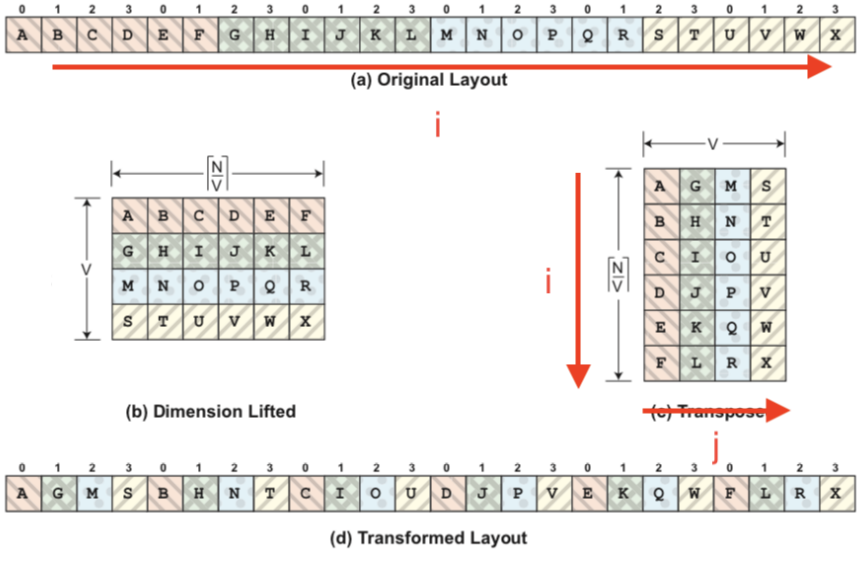
With code

```
for (i = 0; i < 24; i++)
  Z[i] = Y[i-1] + Y[i] + Y[i+1];
```

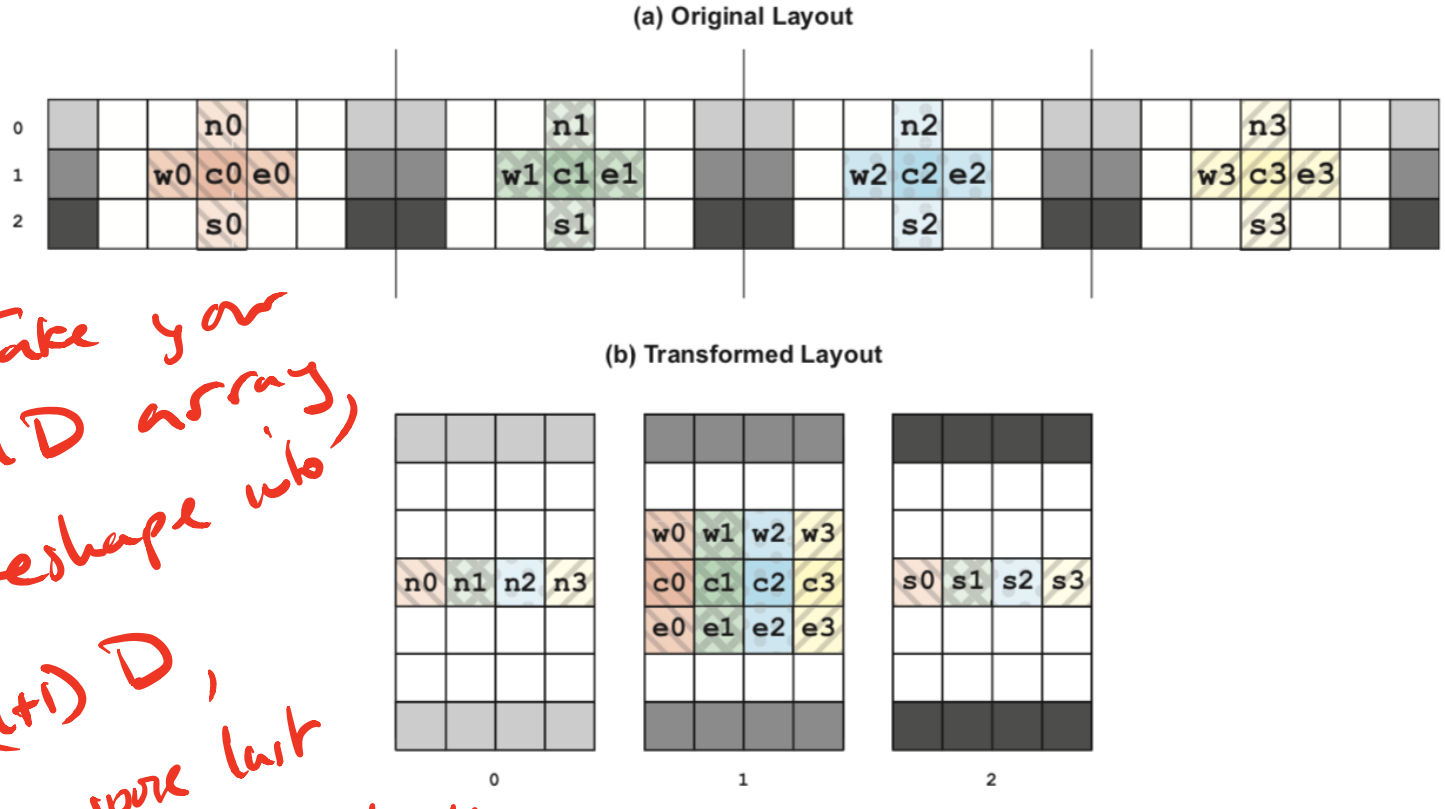
```
for (i = 0; i < 6; i++)
  for (j = 0; j < 4; j++)
    Zt[i, j] = Yt[i-1, j] + Y[i, j] + Y[i+1, j];
```

Bad for vectorisation

Good for vectorisation



Same idea in higher dimensions



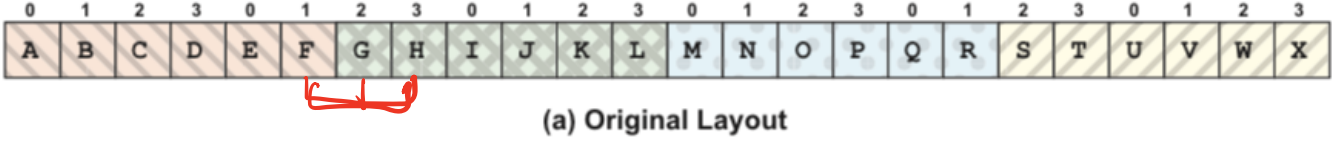
Take your nD array, reshape into (n+1)D, transpose last dimension, and flatten.

What about the boundaries?

Problem

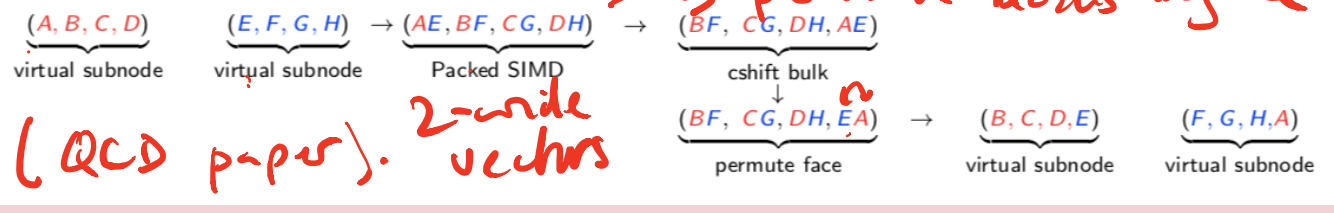
- This scheme works perfectly in the bulk of the domain
- At the edges, it is a little trickier.

$\Rightarrow$ We need to add $F + G + H$, not $F + A + B$.



(a) Original Layout

(d) Transformed Layout



What about the boundaries?

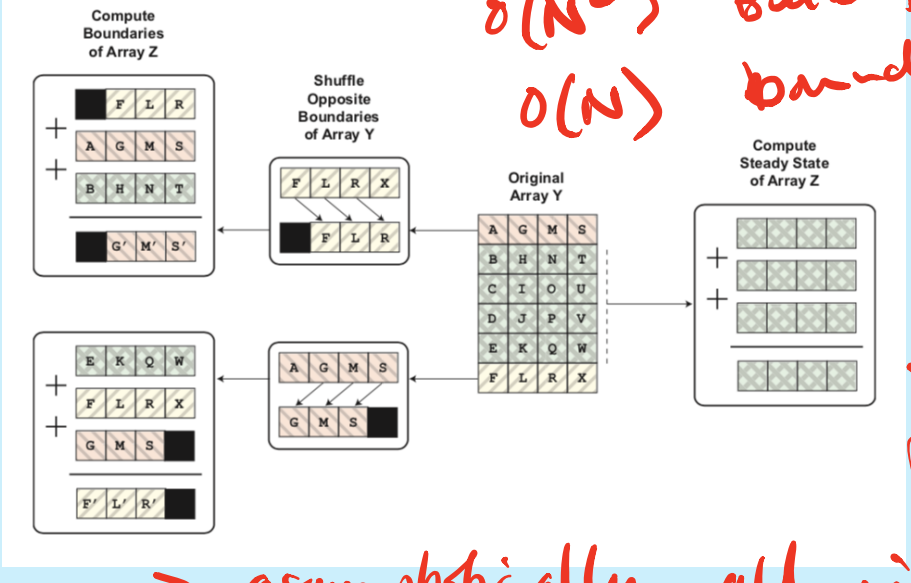
Solution

- Use *vector shuffles* and masking to handle (literal) edge cases.
- Performance for these updates is *lower* but suppressed by surface-to-volume ratio.

2D. N×N grid

$O(N^2)$ Bulk parts ✓

$O(N)$ boundary parts ✗



$\frac{N}{N^2} = \frac{1}{N}$

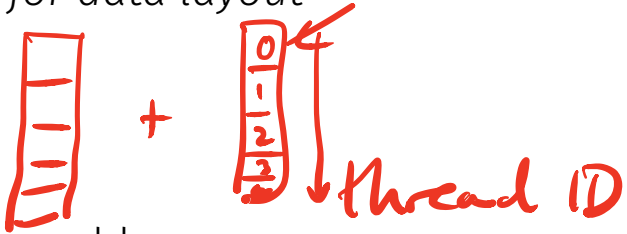
→ 0 for large N.

→ asymptotically: all work is perfectly vectorised.

AoS vs. SoA

Use struct of arrays, not array of structs for data layout (NVidia)

vector lanes



- Why? "Coalesced memory access"

⇔ adjacent threads access adjacent memory addresses

- Same principle applies for vectorisation

Array of structs

```
struct Point {
  double x, y, z;
};

struct Point *points = ...;
```

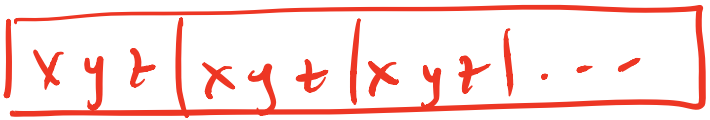
✗ Good for cache

Struct of arrays

```
struct Points {
  double *x, *y, *z;
};

struct Points points = ...;
```

✓ Good for vectors



Pros and cons

AoS

- ✓ most obvious (each record has all its fields together)
- ✓ good cache usage for streaming access to all fields
- ✓ Minimal memory footprint (e.g. need one cache line for each 3D point)
- ✗ bad for vectorising (unless # fields matches vector width), since natural inner loop is over fields.

SoA

- ✗ data structure a little harder to manage
- ✓ good cache usage for streaming access to each field
- ✗ higher memory pressure for accessing all fields (e.g. need three cache lines for each 3D point)
- ✓ good for vectorising, since natural inner loop is over data items.

Best of both worlds

- Dimension-lifted transposition ticks all boxes (presuming your cache is big enough).
- Can apply to AoS vs. SoA argument by instead having AoSoA (Array of struct of (short) arrays)

$\Rightarrow$ add extra dimension to data layout and transpose the data.

```
#define N 4 /* vector width */
struct pointN {
  double x[N], y[N], z[N];
}

struct pointN *points = ...;
```

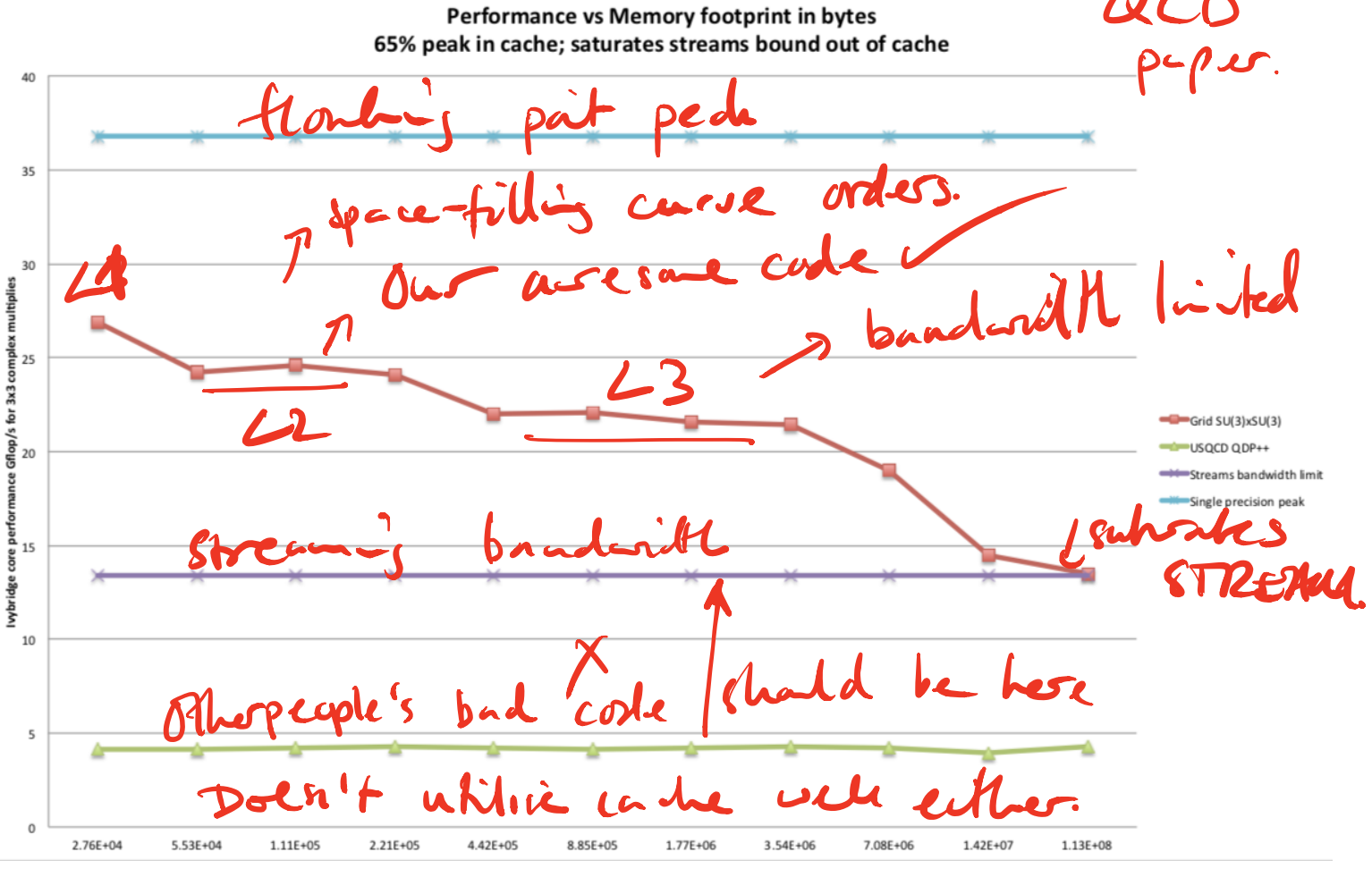
"Array of structs of short arrays"

Evaluation of performance

		Phenom				Core2 Quad				Core i7			
		SP		DP		SP		DP		SP		DP	
		GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.
J-1D	Ref.	4.27	1.00×	3.08	1.00×	3.71	1.00×	2.46	1.00×	8.67	1.00×	3.86	1.00×
	DLT	7.68	1.80×	3.79	1.23×	9.42	2.54×	2.83	1.15×	10.55	1.22×	4.01	1.04×
	DLTi	11.38	2.67×	5.71	1.85×	13.95	3.76×	7.01	2.85×	15.35	1.77×	7.57	1.96×
J-2D-5pt	Ref.	6.96	1.00×	2.71	1.00×	3.33	1.00×	2.94	1.00×	8.98	1.00×	4.54	1.00×
	DLT	9.00	1.29×	3.75	1.38×	8.86	2.66×	4.58	1.56×	10.20	1.14×	5.18	1.14×
	DLTi	11.31	1.63×	5.67	2.09×	11.58	3.48×	5.85	1.99×	13.12	1.46×	6.58	1.45×
J-2D-9pt	Ref.	4.48	1.00×	3.21	1.00×	4.21	1.00×	2.72	1.00×	8.30	1.00×	4.11	1.00×
	DLT	7.71	1.72×	3.81	1.18×	8.04	1.91×	4.08	1.50×	10.23	1.23×	5.23	1.27×
	DLTi	12.26	2.74×	6.11	1.90×	12.01	2.85×	6.03	2.22×	13.62	1.64×	6.80	1.65×
J-3D	Ref.	6.01	1.00×	2.90	1.00×	6.07	1.00×	3.04	1.00×	9.04	1.00×	4.64	1.00×
	DLT	6.84	1.14×	3.73	1.29×	8.07	1.33×	4.25	1.40×	9.46	1.05×	5.02	1.08×
	DLTi	10.08	1.68×	5.36	1.85×	10.36	1.71×	5.31	1.75×	12.02	1.33×	6.04	1.30×
Heatttut-3D	Ref.	6.06	1.00×	3.02	1.00×	6.64	1.00×	3.29	1.00×	8.75	1.00×	4.55	1.00×
	DLT	7.12	1.18×	3.36	1.11×	8.71	1.31×	4.45	1.35×	9.99	1.14×	4.91	1.08×
	DLTi	9.59	1.58×	5.12	1.70×	8.86	1.33×	4.45	1.35×	11.99	1.37×	6.05	1.33×
FDTD-2D	Ref.	5.86	1.00×	3.26	1.00×	6.42	1.00×	3.35	1.00×	8.72	1.00×	4.34	1.00×
	DLT	6.89	1.18×	3.65	1.12×	7.71	1.20×	4.03	1.20×	8.91	1.02×	4.73	1.09×
	DLTi	6.64	1.13×	3.41	1.05×	8.03	1.25×	4.03	1.20×	9.74	1.12×	4.82	1.11×
Rician-2D	Ref.	3.29	1.00×	1.93	1.00×	1.87	1.00×	1.27	1.00×	3.98	1.00×	2.16	1.00×
	DLT	3.46	1.05×	2.40	1.25×	2.59	1.39×	1.27	1.00×	4.13	1.04×	2.23	1.03×
	DLTi	8.09	2.46×	2.56	1.33×	8.50	4.55×	1.27	1.00×	11.31	2.84×	2.23	1.03×

Don't present your data like this!

Evaluation of performance



Summary

- For regular (stencil-like) array access patterns, can obtain basically perfect cache usage and vectorisation through data layout transformations.
- Approach of "dimension-lifting" is generic
  - Applies on both CPUs and GPUs (just use different inner dimension lengths)
  - Exact details of loop structure not hugely important
- Same ideas can also be applied to *unstructured* computations with local stencils