

# Session 5: Cache blocking/tiling

COMP52315: performance engineering

---

Lawrence Mitchell\*

\*[lawrence.mitchell@durham.ac.uk](mailto:lawrence.mitchell@durham.ac.uk)

What can we do to  
fix this?

→ Fort code is hard.  
→ "standard techniques"  
→ scientific code is "easy".

# An exemplar problem

Dense linear algebra.

## Matrix transpose

$$B_{ij} \leftarrow A_{ji} \quad A, B \in \mathbb{R}^{n \times n}$$

```
double *a, *b;
```

```
...
```

```
for (int i = 0; i < N; i++)
```

```
    for (int j = 0; j < N; j++)
```

```
        b[i*N + j] = a[j*N + i];
```

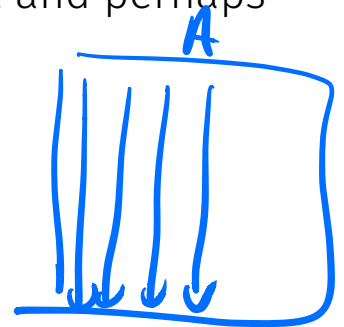
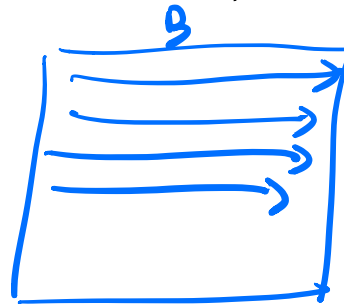
Model for performance

→ measure  
⇒ bad performance :C

→ fix it.

So far, we've talked about how to measure performance, and perhaps determine that it is bad.

⇒ what can we do about it?



# Matrix transpose: simple performance model

## Set up our expectation

- $N^2$  loads,  $N^2$  stores, no compute
- ⇒ all we're doing is copying data
- Hence we might expect to see performance close to that of the streaming memory bandwidth, independent of matrix size.

ideally, all our algorithms will run at performance independent of  $N^2$ .  
want this!

# Matrix transpose: simple performance model

## Set up our expectation

- $N^2$  loads,  $N^2$  stores, no compute
- ⇒ all we're doing is copying data
- Hence we might expect to see performance close to that of the streaming memory bandwidth, independent of matrix size.

Matrix size	BW [GByte/s]
128 × 128	22
256 × 256	13
512 × 512	13
1024 × 1024	5
2048 × 2048	1.6
4096 × 4096	0.9

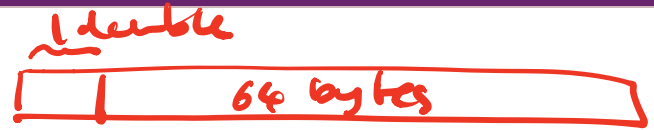
512 × 512 × 2 × 8  
4 MB.  
16 MB  
= cache

had 0

Work with  
N increasing.

out of cache.

# What went wrong?



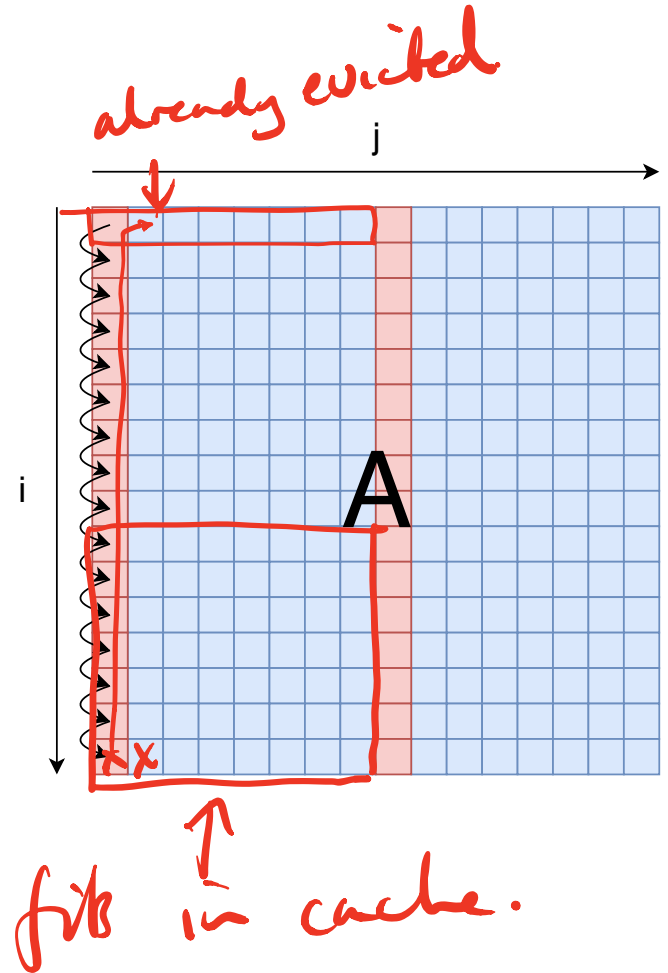
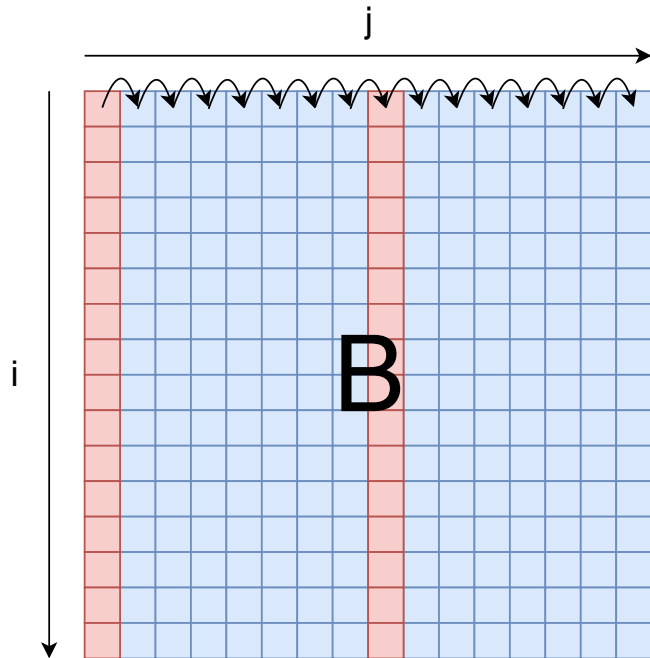
```
double *a, *b;
```

```
...
```

```
for (int i = 0; i < N; i++)  
    for (int j = 0; j < N; j++)  
        b[i*N + j] = a[j*N + i];
```

- We have streaming access to **b**, but stride- $N$  access to **a**.
- If both matrices fit in cache, this is OK, and a reasonable model of time is  $T_{\text{cache}} = N^2(t_{\text{read}} + t_{\text{write}})$ .
- Note that the reads of **a** load a full cache line, but use only 8 bytes of it.
- Better model  $T_{\text{mem}} = N^2(8t_{\text{read}} + t_{\text{write}})$

# A picture



# Cache locality

- Since we have strided access to  $\mathbf{a}$ , we need to hold  $LN$  bytes in the cache to get any reuse, where  $L$  is the cache line size in. This is not possible for large matrices.
- A mechanism to fix this is to *reorder* the loop iterations to preserve spatial locality.

<sup>T</sup>  
doubles

## Idea

- Break loop iteration space into blocks
  - *strip-mining*
  - *loop reordering*

# Strip mining

- Break a loop into blocks of consecutive elements

## Before

```
for ( int i = 0; i < N; i++ )  
    a[i] = f(i);
```

## After

```
for ( int ii = 0; ii < N; ii += stride )  
    for ( int i = ii; i < min(N, ii + stride); i++ )  
        a[i] = f(i);
```

- Not that useful for just a single loop, although there are circumstances where one might use it

*↳ compiler cost model is wrong.*



# Strip mining multiple loops

- Let's do the same for both loops of the transpose:

Before

```
for (int i = 0; i < N; i++)  
  for (int j = 0; j < N; j++)  
    a[i*N + j] = a[j*N + i];
```

*strip - stride i*  
*strip - stride j*

After

```
for (int ii = 0; ii < N; ii += stridei)  
  for (int i = ii; i < min(N, ii+stridei); i++)  
    for (int jj = 0; jj < N; jj += stridej)  
      for (int j = jj; j < min(N, jj+stridej); j++)  
        b[i*N + j] = a[j*N + i];
```

- Haven't yet made any change to the performance

*still has same traversal order,  
=> same problem.*

# Reorder loops

After permuting i and jj loops

```
for (int ii = 0; ii < N; ii += stridei)  
  for (int jj = 0; jj < N; jj += stridej)  
    for (int i = ii; i < min(N, ii+stridei); i++)  
      for (int j = jj; j < min(N, jj+stridej); j++)  
        b[i*N + j] = a[j*N + i];
```

*loop over  
blocks  
cache -  
block.*

- Two free parameters `stridei` and `stridej`
- Need to choose these appropriately to levels in the cache hierarchy
- Ideally block for L1, L2, L3, etc... *← most people block just for L2.*
- The extra logic adds some overhead

# Why is it “tiling”?

*Slightly detrimental for B.*

Iteration over B.

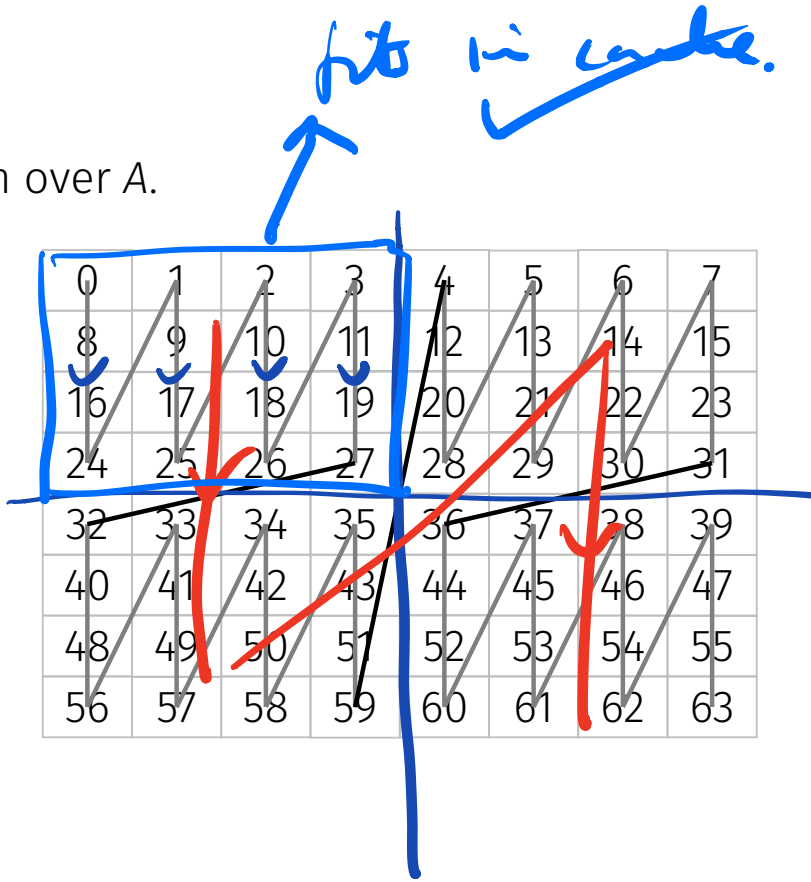
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

# Why is it "tiling"?

Iteration over A.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63



# Does it work?

- Have a go, I provide some sample code for which you can tune the blocking parameters.

⇒ Exercise 7.

`https:  
//teaching.wence.uk/comp52315/exercises/exercise07/`

## A second problem

still achieve research cre.   
 Data-orientation + flops.

### Matrix-Matrix multiplication

$$C_{ij} \leftarrow C_{ij} + \sum_k A_{ik} B_{kj} \quad A, B, C \in \mathbb{R}^{n \times n}$$

```
for (int i = 0; i < n; i++)  
  for (int j = 0; j < n; j++)  
    for (int k = 0; k < n; k++)  
      C[i*n + j] += A[i*n + k] * B[k*n + j];
```

FMA

Peak throughput  
CPU →  
flops FMA

Same story here (or at least it was in the 90s!).

poster-child for flop-limited.  $\alpha \times \beta$

# (Another) simple model for computation

- Simple model of memory, two levels: “fast” and “slow”

- Initially all data in slow memory

*“cache” “RAM”*

$m$  number of data elements moved between fast and slow memory

$t_m$  time per slow memory operation

$f$  number of flops

*as fast → slow.*

$t_f \ll t_m$  time per flop *← fast memory*

$q =: f/m$  average flops per slow memory access

- Minimum time to solution (all data in fast memory)

- Typical time

$t_{ff}$

$$ft_f + mt_m = ft_f \left( 1 + \frac{t_m}{t_f} \frac{1}{q} \right)$$

*> 1*  
*property of alg.*

- $t_m/t_f$  property of hardware,  $q$  property of algorithm

*$\frac{1}{q} \rightarrow 0$*   
 *$1 + \frac{t_m}{t_f} \frac{1}{q} \rightarrow 1$*

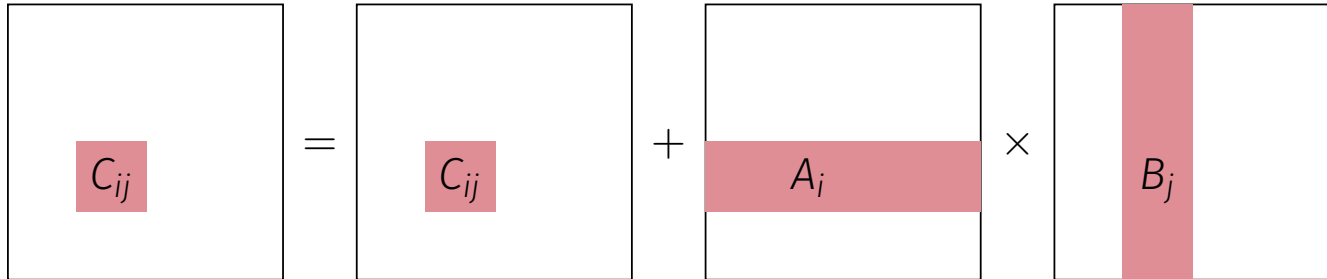
# Naïve matrix-multiply

```
for (int i = 0; i < n; i++)  
  for (int j = 0; j < n; j++)  
    for (int k = 0; k < n; k++)  
      C[i*n + j] = C[i*n + j] + A[i*n + k] * B[k*n + j];
```

2 flops  
(or 4 FMA) per it  
 $N^3$  it

- Algorithm does  $2n^3 = \mathcal{O}(n^3)$  flops and touches  $3 \cdot 8n^2$  bytes of memory
- $q$  potentially  $\mathcal{O}(n)$ , arbitrarily large for large  $n$ .

$26n^2$  bytes  
 $3n^2$  elements.



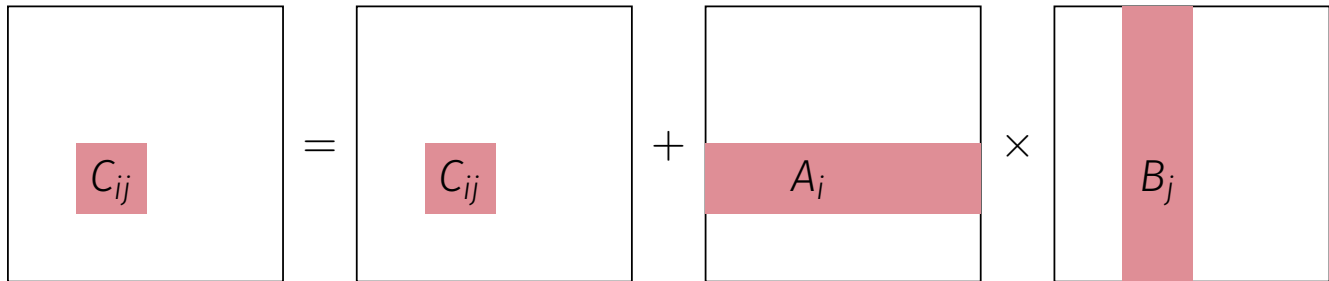
$q = \frac{2n^3}{3n^2} = \frac{2}{3}n$   
 $n \rightarrow \infty$



# Naïve matrix-multiply

```
for (int i = 0; i < n; i++)  
  // Read row i of A into fast memory  
  for (int j = 0; j < n; j++)  
    // Read Cij into fast memory  
    // Read column j of B into fast memory  
    for (int k = 0; k < n; k++)  
      C[i*n + j] = C[i*n + j] + A[i*n + k] * B[k*n + j];  
    // Write Cij back to slow memory
```

*n* reads of size *n*  
*n*<sup>2</sup> reads of size 1  
*n*<sup>2</sup> reads of size *n*  
*n*<sup>2</sup> writes of size 1.



# Naïve matrix-multiply

## Number of slow memory references

$m = n^3$  each column of  $B$  is read  $n$  times

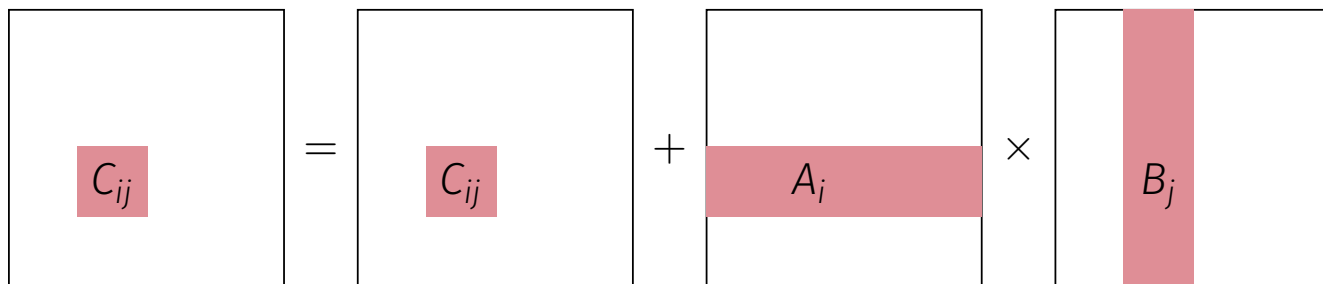
$+ n^2$  each row of  $A$  is read ~~1~~ once

$+ 2n^2$  each entry of  $C$  is read once and written once

$$= (n^3 + 3n^2)$$

Hence

$$\lim_{n \rightarrow \infty} q = \frac{f}{m} = \frac{2n^3}{(n^3 + 3n^2)} = 2$$



# From model to prediction

- So for a triply-nested loop structure, the *best* time to solution our model predicts is:

*N* large.

$$T = t_{ff} \left( 1 + \frac{t_m}{2t_f} \right)$$

- Recall that on modern hardware, memory *latency* is around 200 cycles per cache line. So let's approximate  $t_m \approx \underline{200/8 = 25}$ , and say  $t_f = 1$ .

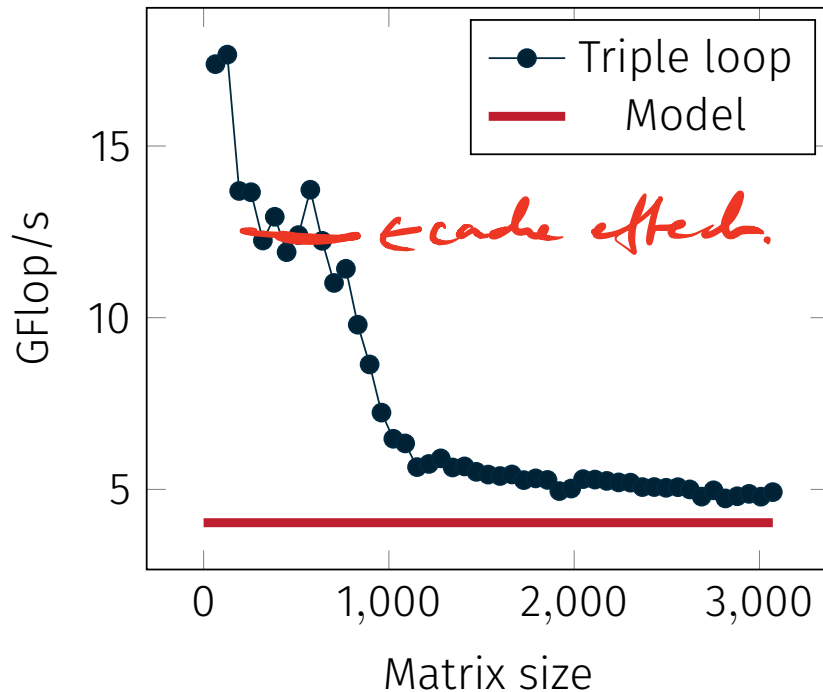
$$T = t_{ff}(1 + 25/2) = 13.5t_{ff}$$

- Maximally 7% peak. *flaps*
- This is *only* an estimate.

*estimate*  
*estimate*  
*estimate*

# Measurement

- Single core Intel i5-8259U. *@ 3.6 GHz*
  - 2 4-wide FMAs per cycle  $\Rightarrow$  16 DP FLOPs/cycle.
- $\Rightarrow$  Peak is  $3.6 \cdot 16 = 57.6$  GFLOPs/s, model predicts 4.03 GFLOPs/s.



# How to improve reuse?

- Problem is that we move rows and columns into fast memory, and then evict them
- Need way of keeping the loaded data in fast memory as long as possible.

⇒ tile iterations

*block matrix*  $\times$   $\mathbb{R}^{n \times n}$  where each entry is scalar

```
// Treat  $A, B, C \in (\mathbb{R}^{b \times b})^{N \times N}$  ✓  
// that is,  $N \times N$  matrices where each entry is a  $b \times b$  matrix.
```

```
for (int i = 0; i < N; i++)  
  for (int j = 0; j < N; j++)  
    // Read block  $C_{ij}$  into fast memory  
    for (int k = 0; k < n; k++)  
      // Read block  $A_{ik}$  into fast memory  
      // Read block  $B_{kj}$  into fast memory  
      // Do matrix multiply on the blocks  
      C[i*N + j] = C[i*N + j] + A[i*N + k] * B[k*N + j];  
    // Write block  $C_{ij}$  back to slow memory
```

*triple loop.*

$N^2$  reads size  $b^2$

$N^3$  reads  $b^2$

$N^3$  reads  $b^2$

$\rightarrow N^2$  writes  $b^2$

*Block matrix multiply.*

*worst "scalar code"*

# How to improve reuse?

- Problem is that we move rows and columns into fast memory, and then evict them
- Need way of keeping the loaded data in fast memory as long as possible.

⇒ tile iterations

```
// Treat  $A, B, C \in (\mathbb{R}^{b \times b})^{N \times N}$   
// that is,  $N \times N$  matrices where each entry is a  $b \times b$  matrix.
```

```
for (int ii = 0; ii < N; ii++)  
  for (int jj = 0; jj < N; jj++)  
    for (int kk = 0; kk < N; kk++)  
      for (int i_ = 0; i_ < b; i_++)  
        for (int j_ = 0; j_ < b; j_++)  
          for (int k_ = 0; k_ < b; k_++) {  
            const int i = ii*b + i_  
            const int j = jj*b + j_  
            const int k = kk*b + k_  
            C[i*n + j] = C[i*n + j] + A[i*n + k] * B[k*n + j];  
          }  
    }
```

} blocks

} multiply on block

→ every thing is fast memory.

# What did that do to the data movement?

$$\begin{aligned} m &= Nn^2 && \text{each block of } B \text{ is read } N^3 \text{ times} \Rightarrow N^3 b^2 = N^3 (n/N)^2 = Nn^2 \\ &+ Nn^2 && \text{each block of } A \text{ is read } N^3 \text{ times} \\ &+ 2n^2 && \text{each block of } C \text{ is read once and written once} \\ &= 2n^2(N + 1) \end{aligned}$$

Hence

$$\lim_{n \rightarrow \infty} q = \frac{f}{m} = \frac{2n^3}{2n^2(N + 1)} = \frac{n}{N} = b$$

- $b \gg 2$  so much better than previously. Can improve performance by increasing  $b$  *as long as blocks still fit in fast memory!*
- Detailed analysis of blocked algorithms in Lam, Rothberg, and Wolf *The Cache Performance and Optimization of Blocked Algorithms* (1991)

# From model to machine characteristics

- Arbitrarily choose a “fast” algorithm to be  $\geq 50\%$  peak, this requires

$$ft_f \left( 1 + \frac{t_m}{t_f} \frac{1}{q} \right) \leq 2t_f f \Leftrightarrow \frac{t_m}{t_f} \frac{1}{q} \leq 1 \Leftrightarrow q \geq \frac{t_m}{t_f}$$

← solve for  $q$ .

- Again, approximate  $t_m = 25, t_f = 1$

$$\Rightarrow b \approx q \geq 25.$$

- Need to hold all three  $b \times b$  matrices in cache

$$\Rightarrow \text{Need space for } 3b^2 = 3 \cdot 25^2 = 1875 \text{ matrix entries, approximately } 14.6 \text{KB of fast memory } M_{\text{fast}}. \quad 1875 \times 8 = 14.6 \text{ kB.}$$

- This is smaller than L1, but larger than fits in registers.

L1  $\downarrow$  typically 16kB.



# Is this the best we can do?

## Theorem

Hong and Kung (1981) Any reorganization of this algorithm that only exploits associativity has

$$q = \mathcal{O}(\sqrt{M_{fast}})$$

and the number of data elements moved between slow and fast memory is

$$\Omega\left(\frac{n^3}{\sqrt{M_{fast}}}\right)$$

useful - practice

Strassen  $N^{2.6}$   
Coppersmith Winograd

$N^{2.3}$  ...

huge constant factors

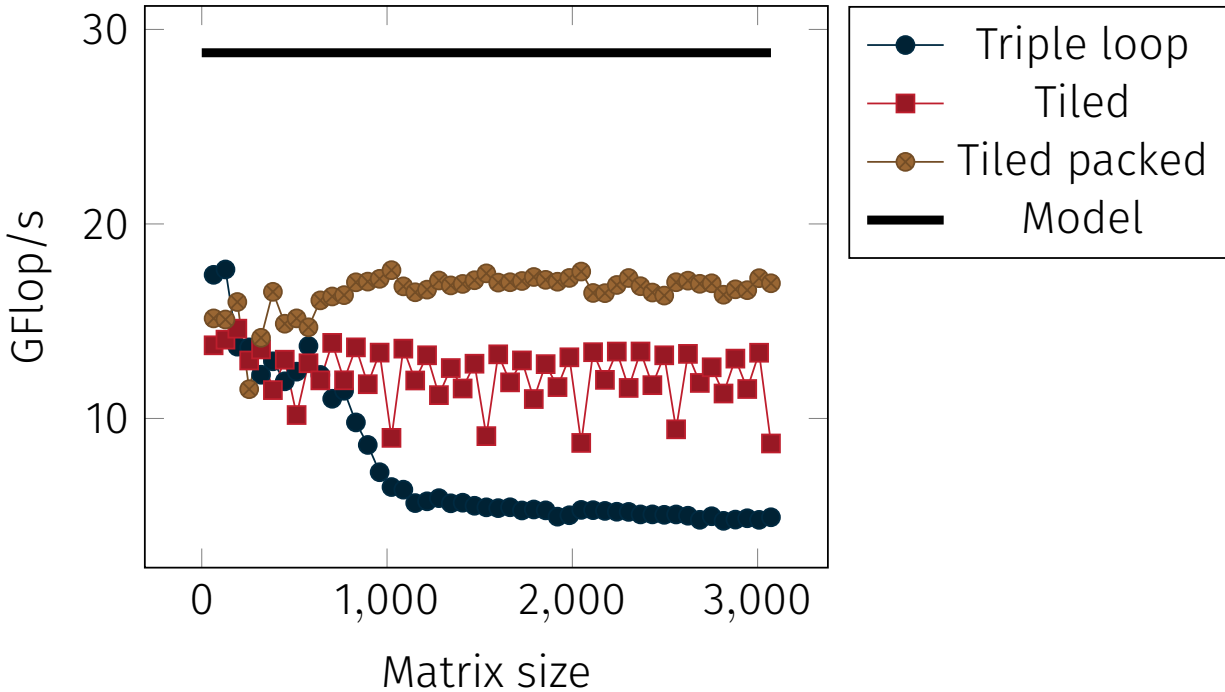
- Exact values for the bounds are not known, the best bounds are provided by Smith and van de Geijn (2017) [arXiv: 1702.02017](https://arxiv.org/abs/1702.02017) [cs.CC]
- The GotoBLAS/OpenBLAS approach approaches these bounds.

# Matching reality with models

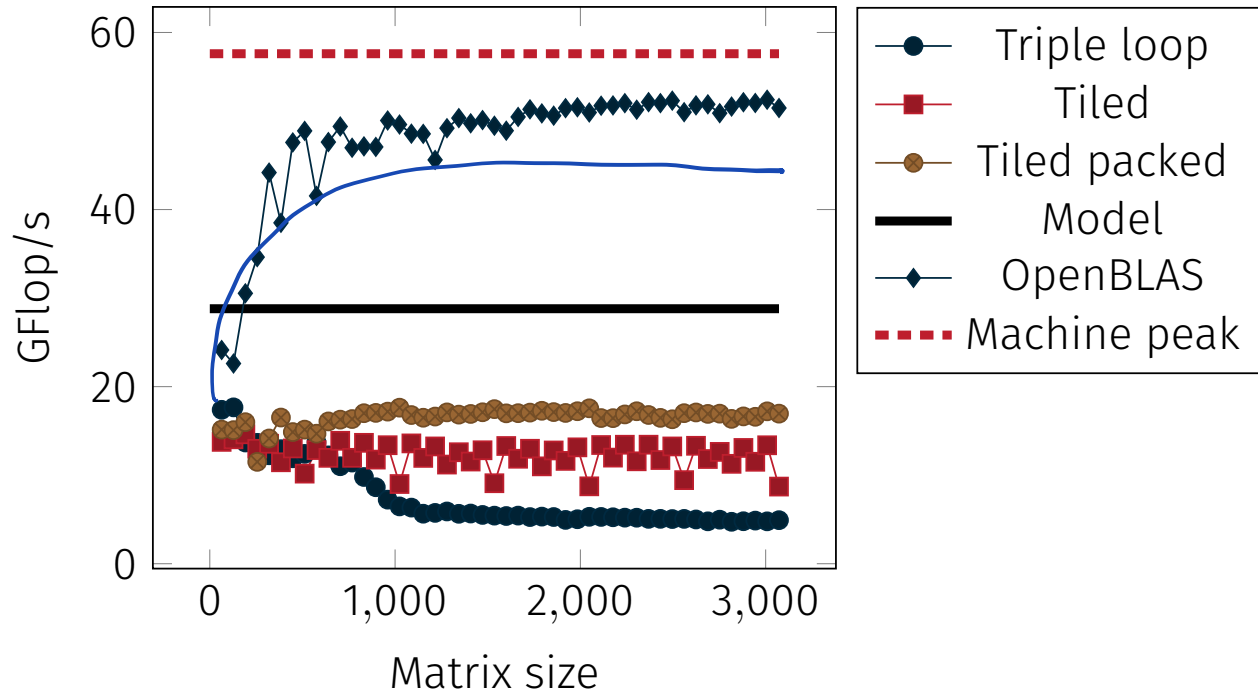
- I provide some sample code that implements this scheme  
⇒ Exercise 8.

`https:  
//teaching.wence.uk/comp52315/exercises/exercise08/`

# Is this the best we can do?



# Is this the best we can do?



# What accounts for this difference?

- Managed to get big matrices to behave like small ones with naive code.
- ⇒ reaching in-cache performance of the starting point.
- For better results, need to
    1. Block for registers and all levels of cache
    2. Perform data-layout transformation to promote (better) vectorisation
  - Will look more at data layout transforms next time.

# Summary

- Loop tiling can *significantly* improve performance of nested loops.
- Particularly important to exploit data reuse.
- For the “last mile” we have to do more. Mostly the same idea, but thinking hard about data layout and explicit vectorisation.
- Simple models can be used to motivate whether things are worth trying.