Session 5: Cache blocking/tiling

COMP52315: performance engineering

Lawrence Mitchell[*]

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

An exemplar problem

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];
```

$$\boxed{\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}}$$

$$\boxed{\begin{matrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{matrix}}$$

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

0 floating point operations.
1 load of an entry of A
1 store of B every loop iteration.

⇒ what can we do about it?

→ expect memory BW-limited.

Matrix transpose: simple performance model

Set up our expectation

- N^2 loads, N^2 stores, no compute

$\Rightarrow$ 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 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

L2
in cache

L3
cache

main memory

On laptop
with
stream BW
A ~ 10 GB/s.

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];
```

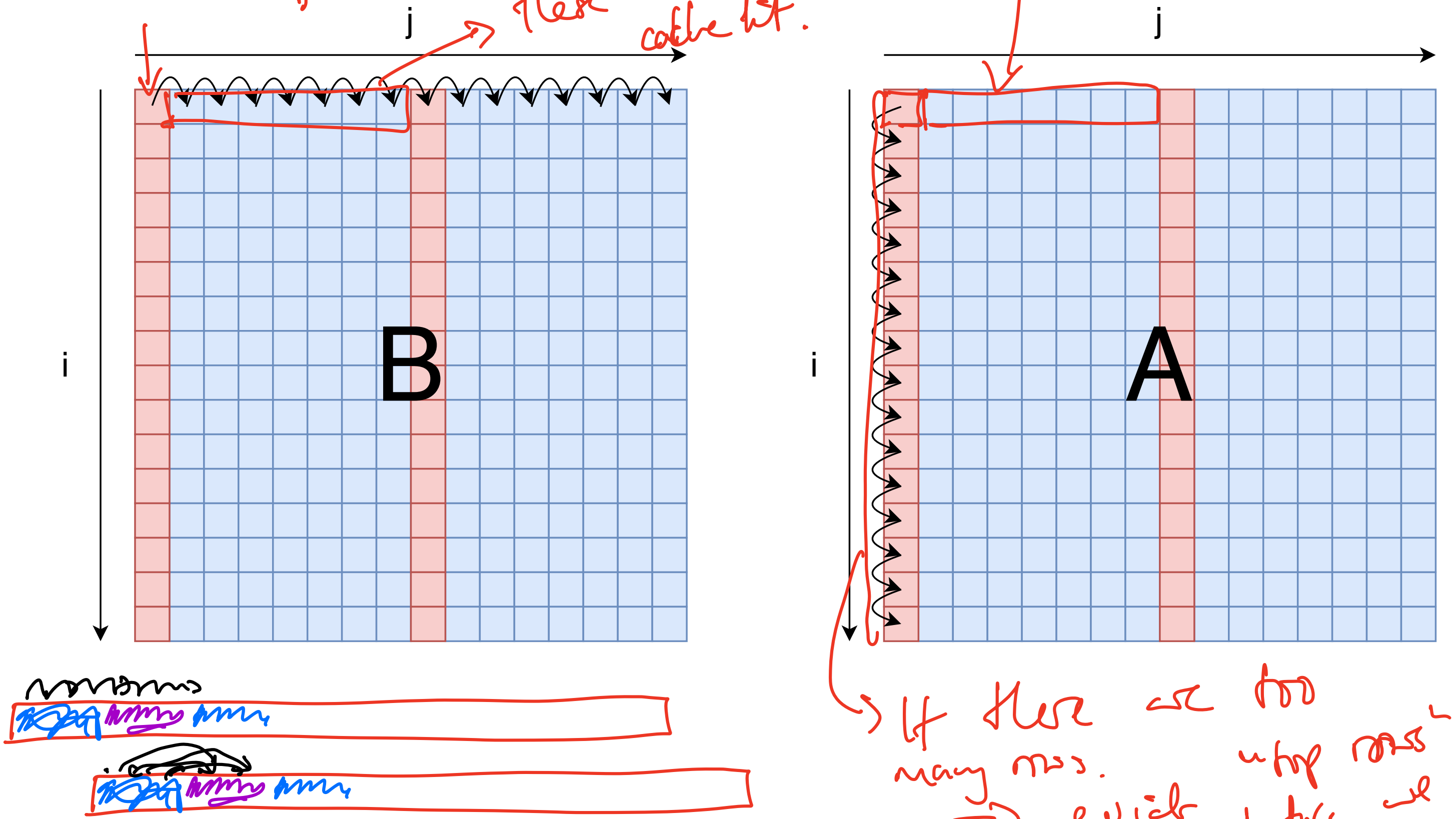
- We have streaming access to **b**, but stride-N access to **a**.
- If both matrices fit in cache, this is OK, and a reasonable model of time is $T_{\text{cache}} = N^2(t_{\text{read}} + t_{\text{write}})$.
- Note that the reads of **a** load a full cache line, but use only 8 bytes of it.
- Better model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

A picture

Load that promotes a new cache line to be brought into cache

These are a cache hit.

These are not accessed "contiguously".



If there are too many rows. ⇒ evict "top rows" before we reach.

Cache locality

- Since we have strided access to **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.

  # of doubles.

- A mechanism to fix this is to *reorder* the loop iterations to preserve spatial locality.

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 → compilers do this for vectorisation.

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];
```

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

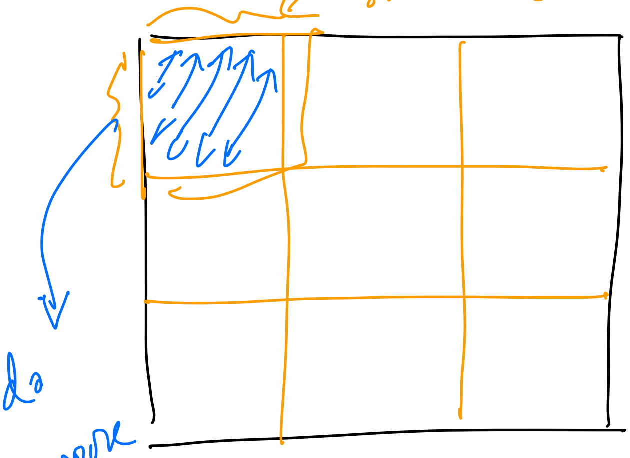
Reorder loops

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];
```

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

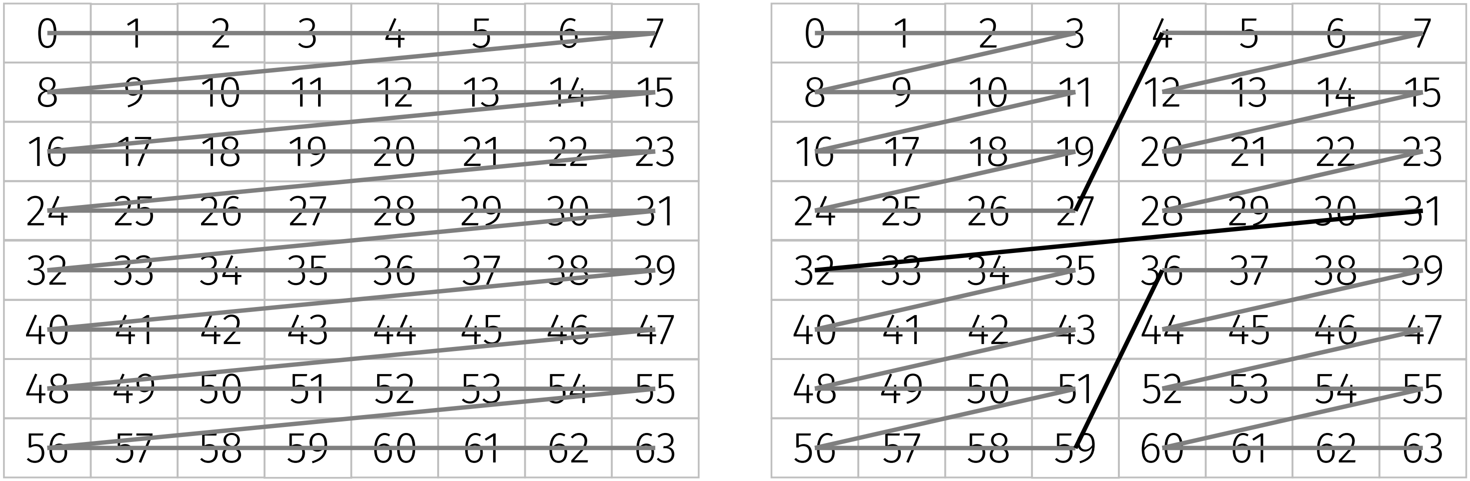
small enough to fit in cache.



do transpose on each block.

Why is it “tiling”?

Iteration over *B*.

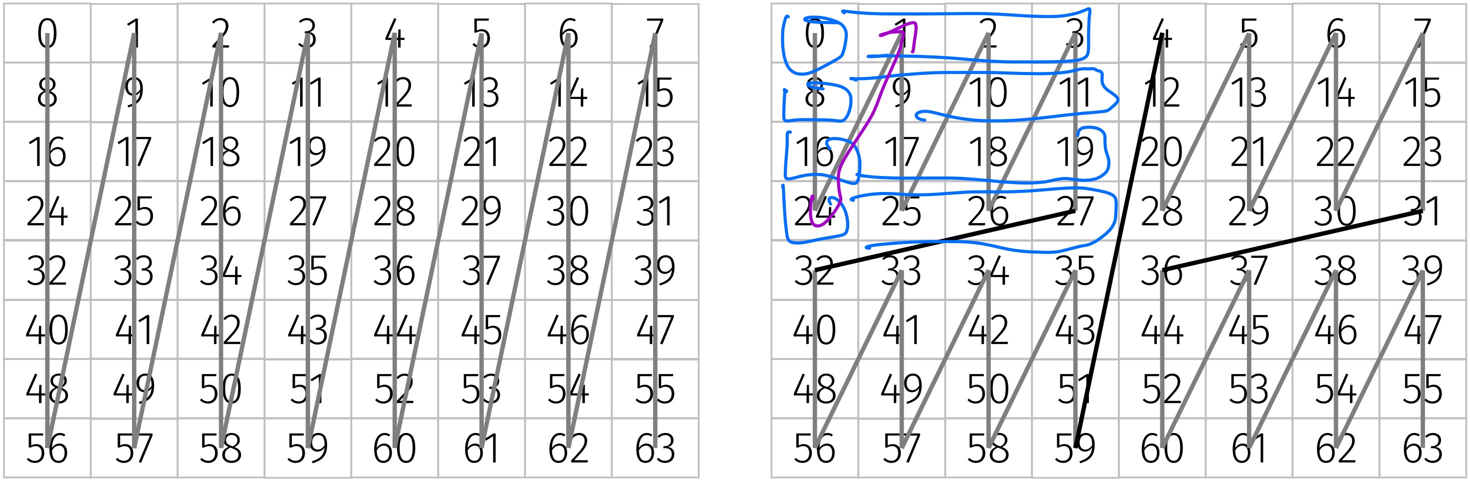


0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

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

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.

. . . . 1100

. . . 1101

. . . 1110

. 1111

4 bit ~~tag~~ cache line index:

=> 16 cache lines.

A second problem

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];
```

Perfect for modern hardware

Perfect instruct: balance

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

N^3 muls

N^3 adds

always in a pair ⇒ FMA.

⇒ Touch $O(N^2)$ data.

⇒ should be flop limited.

(Another) simple model for computation

- Simple model of memory, two levels: “fast” and “slow”
- Initially all data in slow memory

m	number of data elements moved between fast and slow memory
t_m	time per slow memory operation
f	number of flops
$t_f \ll t_m$	time per flop
$q =: f/m$	average flops per slow memory access

Inspired by a 1981 paper.

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

$$t_f f$$

- Typical time

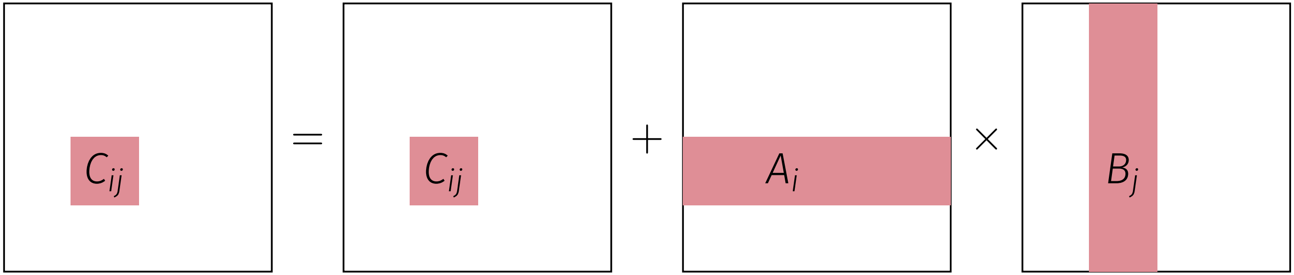
$$\overset{\text{compute}}{f t_f} + \overset{\text{memory movement}}{m t_m} = \underbrace{f t_f \left(1 + \overset{\text{hardware}}{\frac{t_m}{t_f}} \frac{1}{q}\right)} \leftarrow \text{algorithm.}$$

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

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];
```

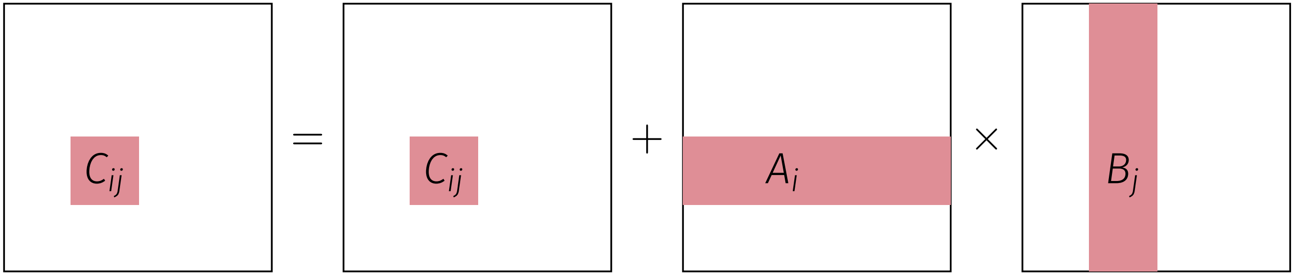
- Algorithm does $2n^3 = \mathcal{O}(n^3)$ flops and touches $3 \cdot 8n^2$ bytes of memory
- q potentially $\mathcal{O}(n)$, arbitrarily large for large n.



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 C_ij 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 C_ij back to slow memory
```

everything we need in fast memory.



Naïve matrix-multiply

Number of slow memory references

$$\begin{aligned} m = n^3 &\quad \text{each column of } B \text{ is read } n \text{ times} \\ + n^2 &\quad \text{each row of } A \text{ is read } \cancel{n} \text{ once} \\ + 2n^2 &\quad \text{each entry of } C \text{ is read once and written once} \\ = (n^3 + 3n^2) & \end{aligned}$$

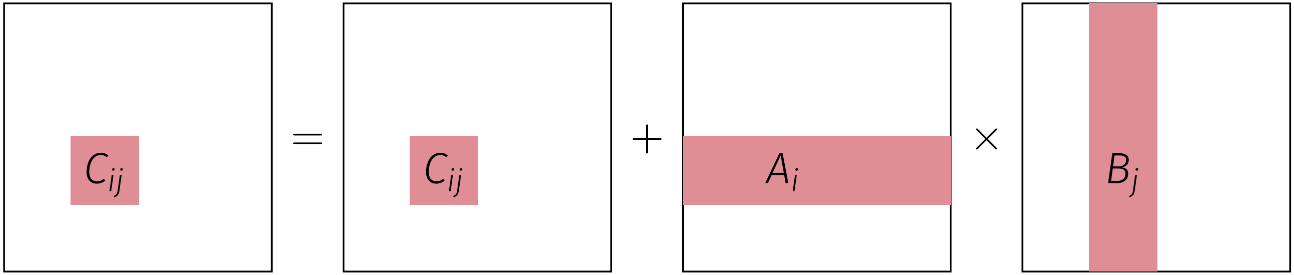
← n rows of A ⇒ n^2 data

So this algorithm is bad.

Hence

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

$$t_f f\left(1 + \frac{t_m}{t_f}\frac{1}{2}\right)$$



From model to prediction

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

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

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

$$T = t_f f(1 + 25/2) = 13.5 t_f f$$

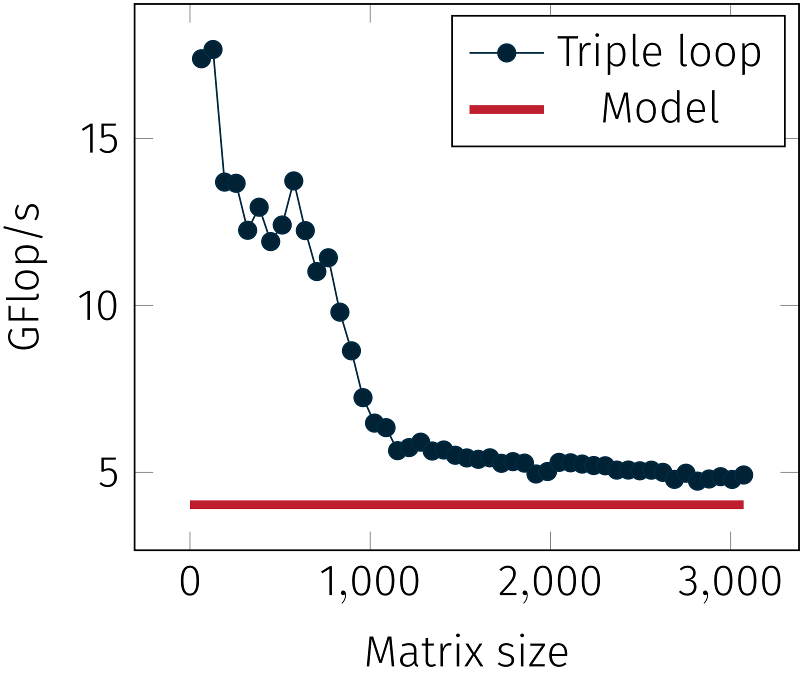
Large N limit takes ~ 13.5 times as long as peak flop rate.

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

Measurement

- Single core Intel i5-8259U.
- 2 4-wide FMAs per cycle $\Rightarrow$ 16 DP FLOPs/cycle.

$\Rightarrow$ Peak is $3.6 \cdot 16 = 57.6$ GFLOPs/s, model predicts 4.03GFLOPs/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

```
// Treat $A, B, C \in \left(\mathbb{R}^{b \times b}\right)^{N \times N}$
// 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
```

"block multiply"

→ hiding three more loops.

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 $A, B, C \in \left(\mathbb{R}^{b\times b}\right)^{N\times N}$
// 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];
          }
```

What did that do to the data movement?

$N < n$.

before this was n^3

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

→ before this was n^2

→ Before it was $n^3 +$ ~~[illegible]~~ n^2.

Hence

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

- $b \gg 2$ 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

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

- 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

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

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

$\Rightarrow$ Should be achievable.

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

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

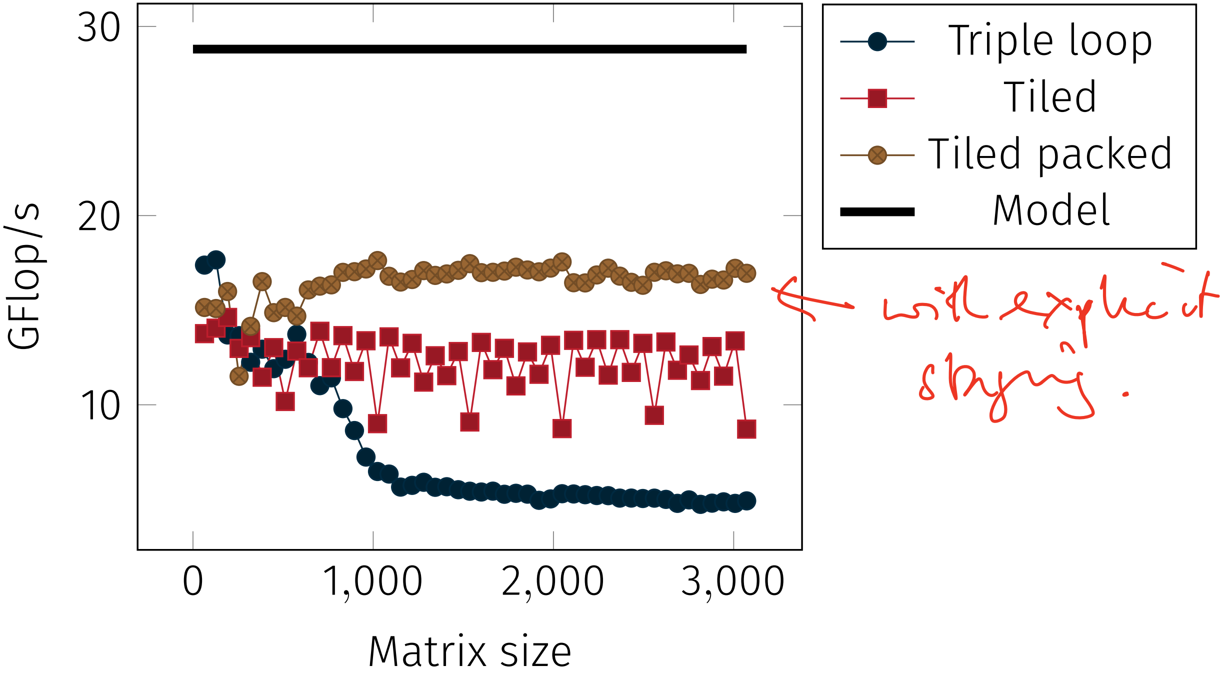
- Exact values for the bounds are not known, the best bounds are provided by Smith and van de Geijn (2017) `arXiv: 1702.02017 [cs.CC]`
- The GotoBLAS/OpenBLAS approach approaches these bounds.

Matching reality with models

- I provide some sample code that implements this scheme

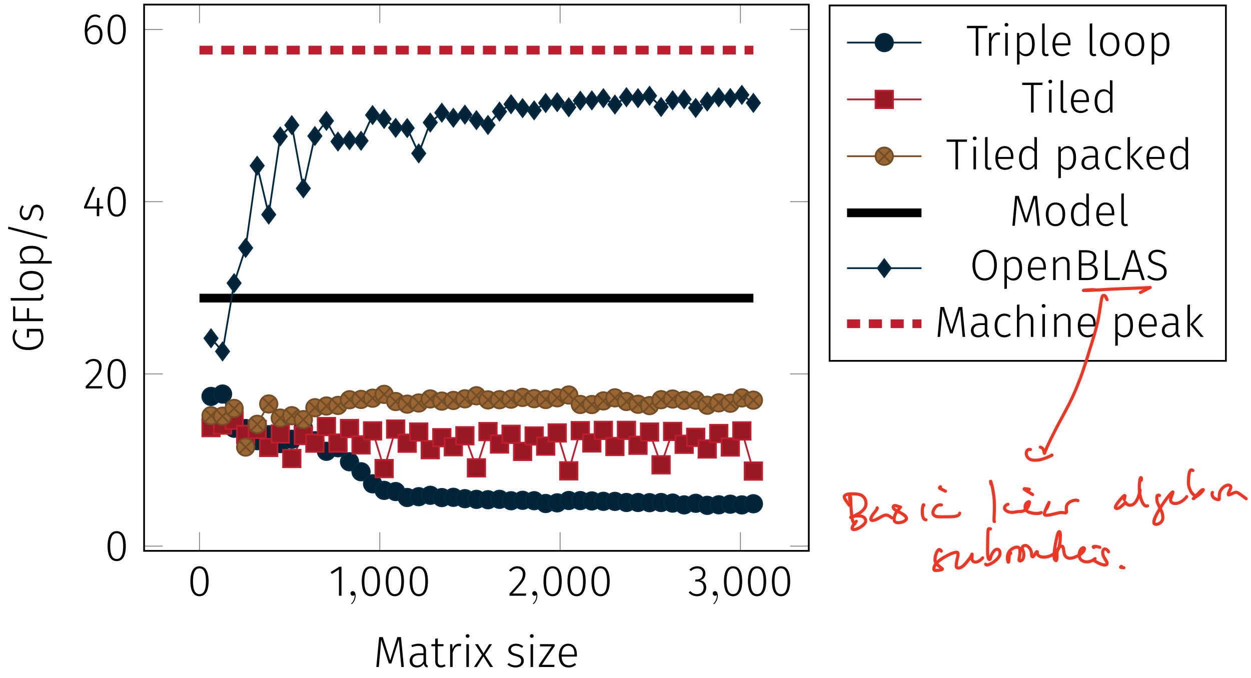
⇒ Exercise 8.

Is this the best we can do?



← with explicit striding.

Is this the best we can do?



Basic linear algebra subroutines.

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

     4 levels of memory sytm.

  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.