IN3200/IN4200: Chapter 3
Data access optimization

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Objectives

- Balance analysis and “lightspeed” estimates
- Data access optimization techniques

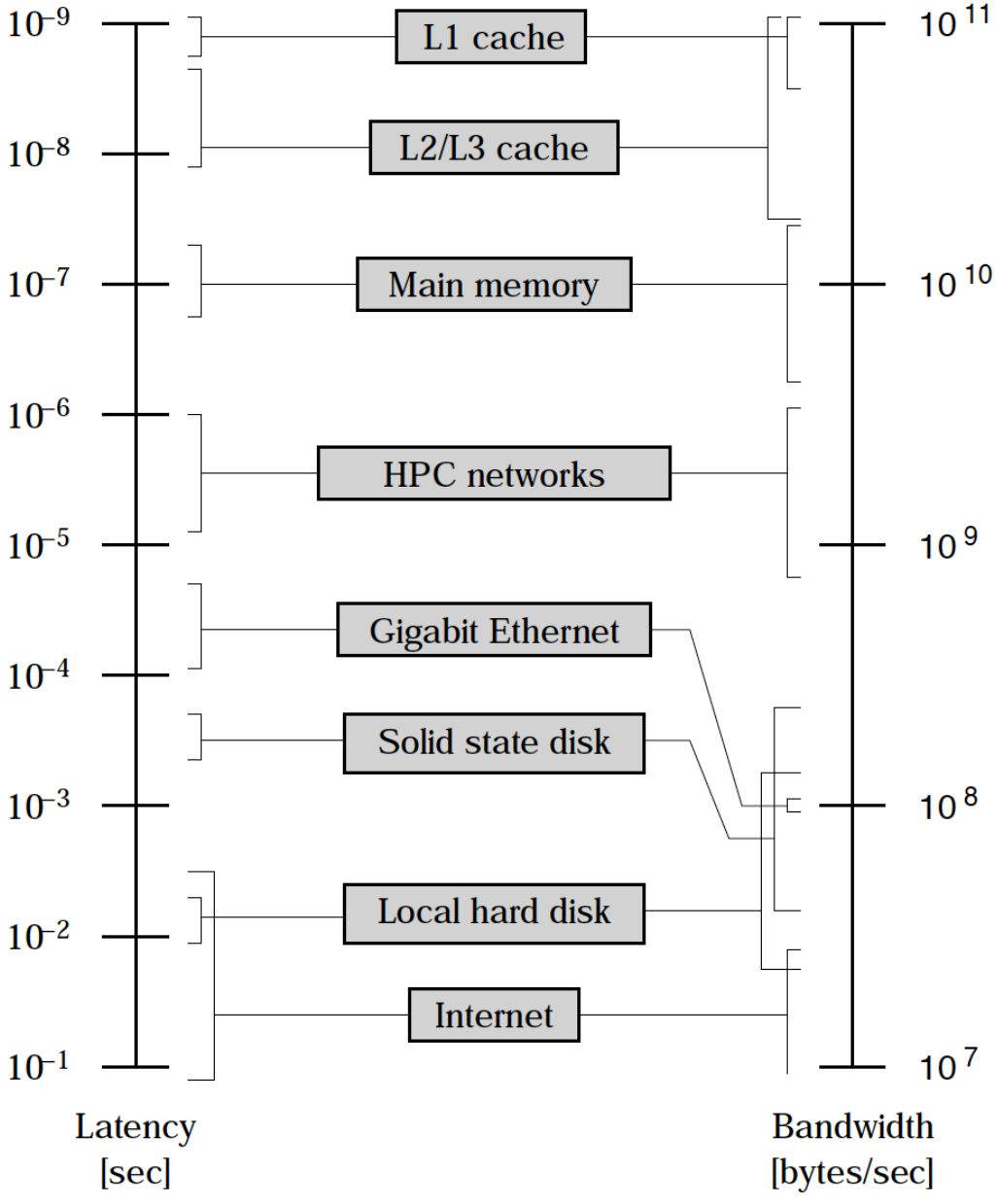
Importance of data access

Applications in science and engineering mostly consist of **loop-based** code that moves large amounts of data in and out the CPU.

Accessing data in the memory hierarchy (from L1 cache to main memory) is often the most prominent performance limiter.

Modern microprocessors have a very impressive theoretical peak performance (in number of FP operations executable per second), but the memory system is "**too slow**".

Typical latency and bandwidth numbers



Rule of the thumb

Any optimization attempt, with respect to data access, should first aim at reducing traffic over slow data paths, or, making the data transfer as efficient as possible.

Understanding the "limitation"

Bandwidth-based performance modeling—to get a rough idea about the maximum performance for a code.

One can *estimate* the theoretically achievable performance of loop-based code, if it is bound by bandwidth limitations.

The concept of "machine balance"

Machine balance, B_{m}, of a processor is the ratio between the maximum memory bandwidth and the peak FP performance:

$$B_{\mathrm{m}} = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak FP performance [GFlops/sec]}} = \frac{b_{\mathsf{max}}}{P_{\mathsf{max}}}$$

Access latency is assumed to be hidden completely.

"Memory bandwidth" could also be substituted by the bandwidth to caches or even network bandwidth.

Example values of machine balance

data path	**balance** [W/F]
cache	0.5–1.0
machine (memory)	0.03–0.5
interconnect (high speed)	0.001–0.02
interconnect (GBit ethernet)	0.0001–0.0007
disk (or disk subsystem)	0.0001–0.01

Table 3.1: Typical balance values for operations limited by different transfer paths. In case of network and disk connections, the peak performance of typical dual-socket compute nodes was taken as a basis.

The above values are somewhat outdated.

The increase of memory bandwidth typically falls behind the increase of FP performance—the ever-increasing **DRAM gap**.

A new example of machine balance

Intel Xeon Skylake Platinum 28-core CPU (model 8180)

- Peak memory bandwidth:
  6 memory channels $\times$ 2.666 GT/sec $\times$ 1 word/T $=$
  16 GWords/sec
- Peak double-precision FP performance:
  28 cores $\times$ 2.3 GHz AVX-512 clock rate $\times$ 32 Flops/cycle $=$
  2061 GFlops/sec
- So the machine balance is only $\frac{16}{2061} = \mathbf{0.00776}$!

The concept of "code balance"

To characterize a loop, we can calculate the **code balance** B_c:

$$B_\mathrm{c} = \frac{\text{data traffic [Words]}}{\text{floating-point operations [Flops]}}$$

That is, you should count the number of FP operations (easy), and also count (or estimate) the amount of data transfered over the performance-limiting data path (can be difficult).

Note: $\frac{1}{B_\mathrm{c}}$ is called **computational intensity**.

What is the expected maximum performance of a loop?

When you know the machine balance B_m of a CPU, and you want to run a loop that has B_c as its code balance.

What will be the maximum achievable performance P (in Flops/sec)?

$$P = \min\left(P_\mathrm{max}, \frac{b_\mathrm{max}}{B_\mathrm{c}}\right)$$

Recall: P_max denotes the maximum FP performance, b_max denotes the maximum bandwidth of the performance-limiting data path.

Comparing P with P_{max}

- In case $P \approx P_{\text{max}}$: the achievable performance is not limited by bandwidth (so data access optimization is **not** needed).
- In case $P \ll P_{\text{max}}$: more analysis is needed to find out whether the code balance B_{c} can be improved, that is, making B_{c} smaller by data access optimization. (Note: smaller $B_{\text{c}} \rightarrow$ higher $P = \frac{b_{\text{max}}}{B_{\text{c}}}$)

“Lightspeed” of a loop

$$\frac{P}{P_{\text{max}}} = \min\left(1, \frac{B_{\text{m}}}{B_{\text{c}}}\right)$$

is the maximum achievable fraction of peak performance for a code with balance B_{c} on a machine with balance B_{m}—also called the **lightspeed** of a loop.

Example of "balance analysis"

```
for (i=0; i<N; i++)
  A[i] = B[i] + C[i]*D[i];
```

- Each iteration has three loads (`B[i]`,`C[i]`,`D[i]`), one store (`A[i]`) and two floating-point operations
- Code balance: $B_\mathrm{c} = \frac{3+1}{2} = 2$
- If a CPU has machine balance $B_\mathrm{m} = 0.1$, then the maximumly achievable performance is $\frac{B_\mathrm{m}}{B_\mathrm{c}} P_\mathrm{max}$, that is, 5% of the peak FP performance
- On cache-based microprocessors, each store miss may incur a *cache line write allocate*, if non-temporal stores are not used. In that case, each store of `A[i]` in effect must be counted as a load plus a store, B_c thus becomes 2.5 $\rightarrow$ only 4% of P_max is maximumly achievable.

Read about "non-temporal stores" at `https://vgatherps.github.io/2018-09-02-nontemporal/`

How realistic is $b_{\max}$?

In reality, even the simplest memory-intensive loops are not able to achieve the theoretical hardware maximum memory bandwidth $b_{\max}$.

The well-known stream micro-benchmarks can be used to measure the realistically achievable maximum memory bandwidth.

STREAM micro-benchmarks

Four micro-benchmarks (https://www.cs.virginia.edu/stream/)

type	kernel	DP words	flops	B_c
COPY	`A(:)=B(:)`	2 (3)	0	N/A
SCALE	`A(:)=s*B(:)`	2 (3)	1	2.0 (3.0)
ADD	`A(:)=B(:)+C(:)`	3 (4)	1	3.0 (4.0)
TRIAD	`A(:)=B(:)+s*C(:)`	3 (4)	2	1.5 (2.0)

Table 3.2: The STREAM benchmark kernels with their respective data transfer volumes (third column) and floating-point operations (fourth column) per iteration. Numbers in brackets take write allocates into account.

More realistic "balance analysis"

We will from now on use the realistically achievable memory bandwidth, b_{S}, which is measured by STREAM.

Then, the realistically achievable maximum FP performance is estimated as

$$P = \min\left(P_{\mathsf{max}}, \frac{b_{\mathrm{S}}}{B_{\mathrm{c}}}\right)$$

Storage order of multi-dimensional arrays

Multi-dimensional arrays normally have an underlying contiguous 1D storage.

C program typically adopts a **row-major** storage order.

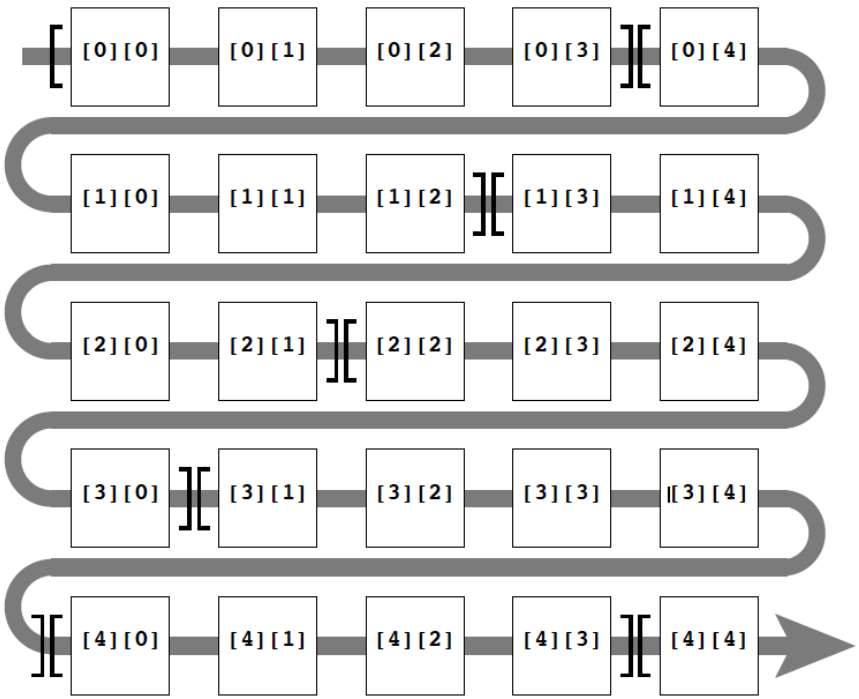


Figure 3.3: Row major order matrix storage scheme, as used by the C programming language. Matrix rows are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.

Storage order of multi-dimensional arrays (cont'd)

Fortran program typically adopts a **column-major** storage order.

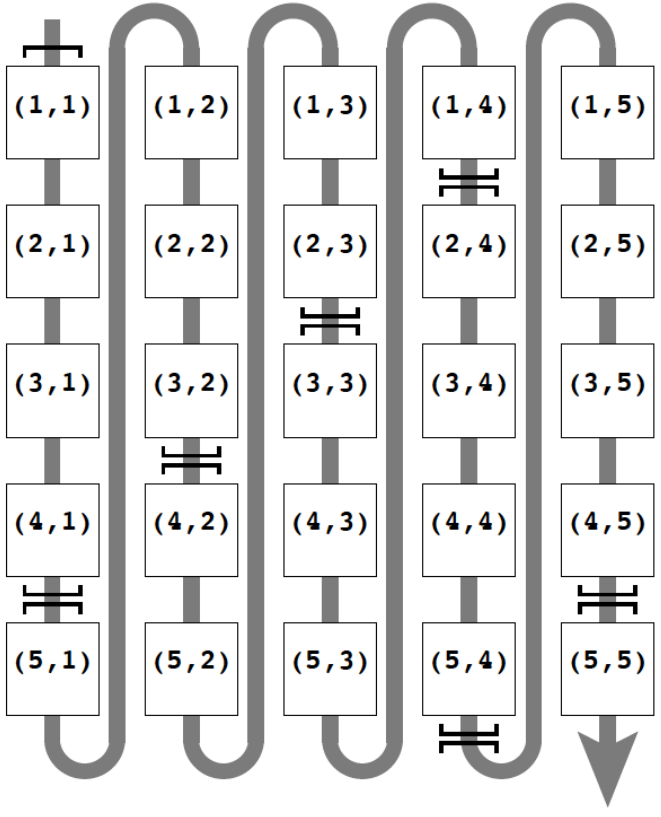


Figure 3.4: Column major order matrix storage scheme, as used by the Fortran programming language. Matrix columns are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.

(Read the textbook with care, because most coding examples are in Fortran.)

Use unit-stride to access arrays, if possible

Assume that 2D array `A` has row-major storage order.

```
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    A[i][j] = i*j;   // stride-1 access, good
```

```
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    A[j][i] = i*j;   // stride-N access, bad!!!
```

Case study: The 2D Jacobi algorithm

Skipping the mathematical and numerical details (given in the textbook), let us focus on the following computation:

```
for (it=0; it<itmax; it++) {
  for (k=1; k<kmax-1; k++)
    for (i=1; i<imax-1; i++)
      phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                       +phi[k][i+1]+phi[k+1][i])*0.25;
  /* pointer swapping */
  temp_ptr = phi_new;
  phi_new = phi;
  phi = temp_ptr;
}
```

Note: both `phi_new` and `phi` are 2D arrays (row-major storage)

2D Jacobi: performance prediction

Balance analysis applied to 2D Jacobi:

- 4 floating-point operations per (k, i)
- 1 store to memory per (k, i)
- **How many loads from memory per (k, i)?**
  (*It depends on the cache size.*)

2D Jacobi: performance prediction (cont'd)

Suppose the cache is very small, that is, not enough to even store one row of `phi`. Then, memory load traffic needed for computing `phi_new[k][i]` is as follows:

- The `phi[k-1][i]` value has to be loaded from memory again (although it was loaded from memory twice already);
- The `phi[k][i-1]` value is already in cache (it was latest loaded from memory for computing `phi_new[k][i-2]`);
- The `phi[k][i+1]` has to be loaded from memory (will be immediately reused for computing `phi_new[k][i+2]`);
- The `phi[k+1][i]` value has to be loaded from memory (and it will be evicted from the cache before needed again);

Therefore, 3 memory loads per $(k, i) \rightarrow B_c = \frac{3 \text{ loads} + 1 \text{ store}}{4 \text{ FPs}}$

2D Jacobi: performance prediction (cont'd)

Suppose the cache can store at least two rows of `phi`, but not enough to store the entire array `phi`. Then, memory load traffic needed for computing `phi_new[k][i]` is as follows:

- The `phi[k-1][i]` value is still in cache (it was first loaded from memory for computing `phi_new[k-2][i]`);
- The `phi[k][i-1]` value is in cache;
- The `phi[k][i+1]` value is also in cache;
- The `phi[k+1][i]` value has to be loaded from memory (and it will be reused during computation on rows `k+1` and `k+2`);

In effect, 1 memory load per $(k, i) \rightarrow B_c = \frac{1 \text{ load}_{+1 \text{ store}}}{4 \text{ FPs}}$

The case of 2 rows fit in cache

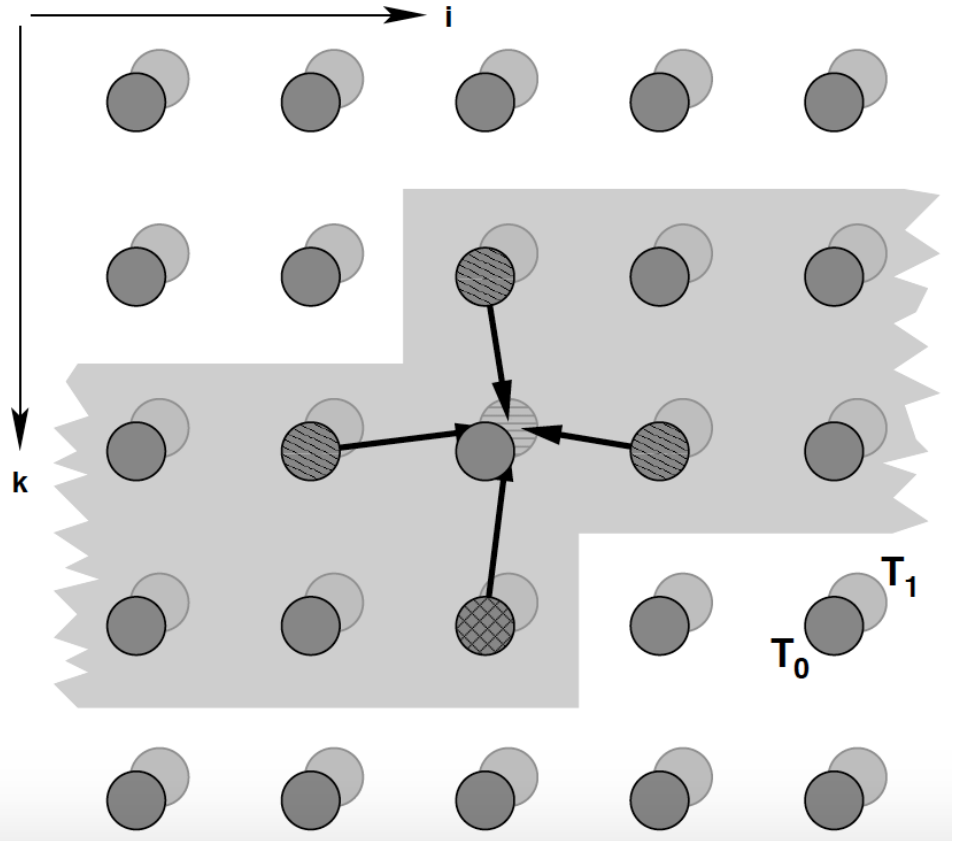


Figure 3.5: Stencil update for the plain 2D Jacobi algorithm. If at least two successive rows can be kept in the cache (shaded area), only one T_0 site per update has to be fetched from memory (cross-hatched site).

Access optimization for algorithm class $O(N)/O(N)$

Algorithm class $O(N)/O(N)$

- 1D loops (N: loop length)
- 1D arrays (N: array length)

Normally not much room for data access optimization, but *loop fusion* can **sometimes** help.

Example of loop fusion

Two loops after each other:

```
for (i=0; i<N; i++) {
  A[i] = B[i] + C[i];
}

for (i=0; i<N; i++) {
  Z[i] = B[i] + E[i];
}
```

- Number of floating-point operations: $2N$
- Number of memory loads & stores: $4N + 2N$

Code balance: $B_c = \frac{6}{2}$, can we improve?

Example of loop fusion (cont'd)

Loop fusion:

```
for (i=0; i<N; i++) {
  A[i] = B[i] + C[i];
  Z[i] = B[i] + E[i];
}
```

- Now each $B[i]$ value is only loaded once instead of twice!
- New code balance: $B_{\mathrm{C}} = \frac{5}{2}$
- Loop fusion will also reduce looping overhead
- Beware of the limited register resources: The code body of each iteration shouldn't be too large. (Otherwise, *register spilling* can lead to performance degradation.)

Access optimization for algorithm class $O(N^2)/O(N^2)$

Algorithm class $O(N^2)/O(N^2)$

- Two-level loop nests (N: loop length on each level)
- Number of floating-point operations: $O(N^2)$
- Number of memory loads & stores: $O(N^2)$

There is more room for data access optimization (than the class of $O(N)/O(N)$)

Example of data access optimization for $O(N^2)/O(N^2)$

Dense matrix-vector multiply

```
for (i=0; i<N; i++) {
  double tmp = C[i];
  for (j=0; j<N; j++)
    tmp += A[i][j]*B[j];
  C[i] = tmp;
}
```

- Number of FP: $2N^2$
- Number of loads: N^2 for 2D array `A`, N for 1D array `C`
- But, how many loads are associated with 1D array `B`?
  - Small cache $\rightarrow$ array `B` is loaded N times $\rightarrow$ N^2 memory loads
  - Large cache $\rightarrow$ array `B` is loaded only once $\rightarrow$ N memory loads

Illustration of array B being loaded N times

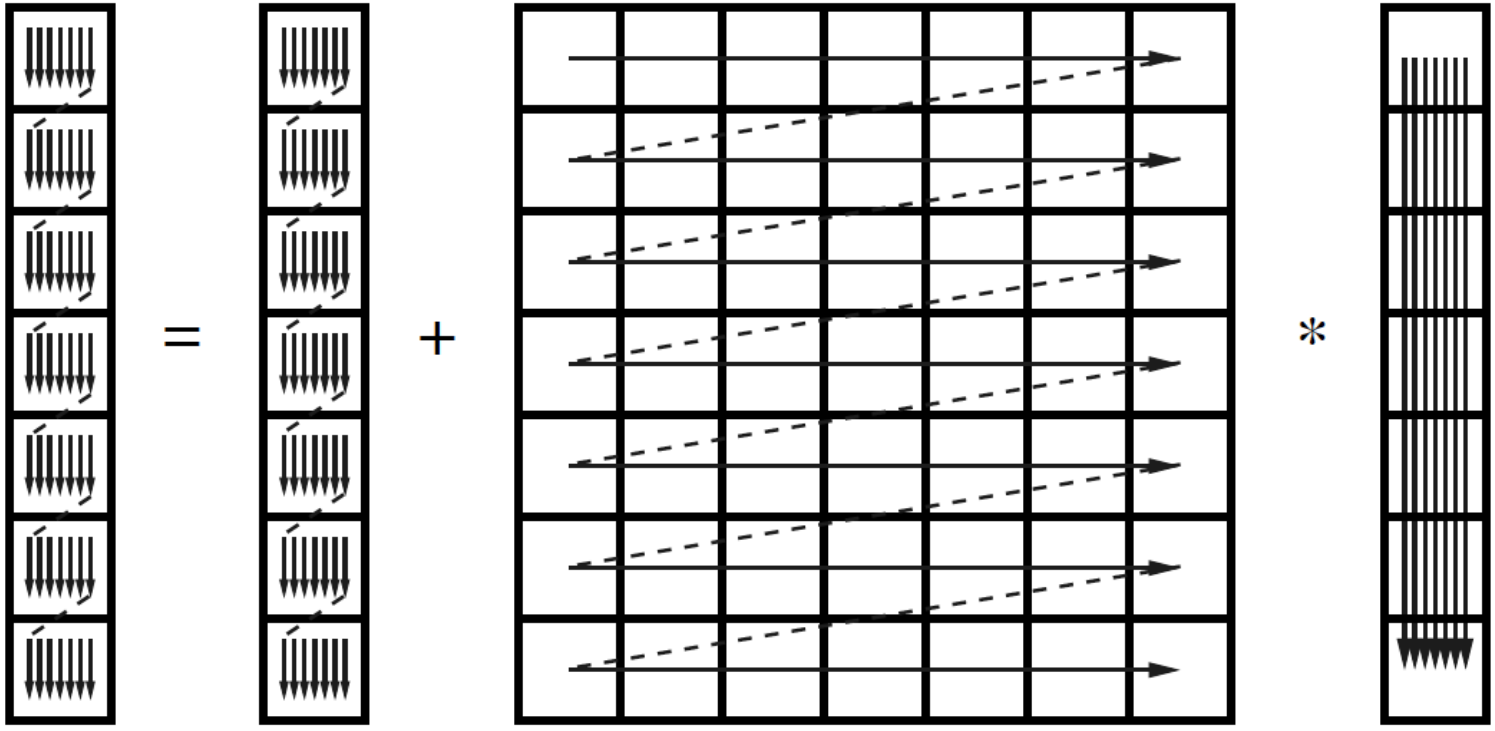


Figure 3.11: Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded N times.

Loop *unrolling*

m-way unroll and jam:

```
for (i=0; i<N; i+=m) {
  for (j=0; j<N; j++) {
    C[i+0] += A[i+0][j]*B[j];
    C[i+1] += A[i+1][j]*B[j];
    // ...
    C[i+m-1] += A[i+m-1][j]*B[j];
  }
}
// remainder code in case (N%m)>0 ....
```

- *m*-fold reuse of each `B[j]` from register
- Total number of memory loads: $N^2 + N^2/m + N$ (for small cache size)
- Size of *m* shouldn't be too large, to avoid too high *register pressure*

Illustration of the effect of unrolling

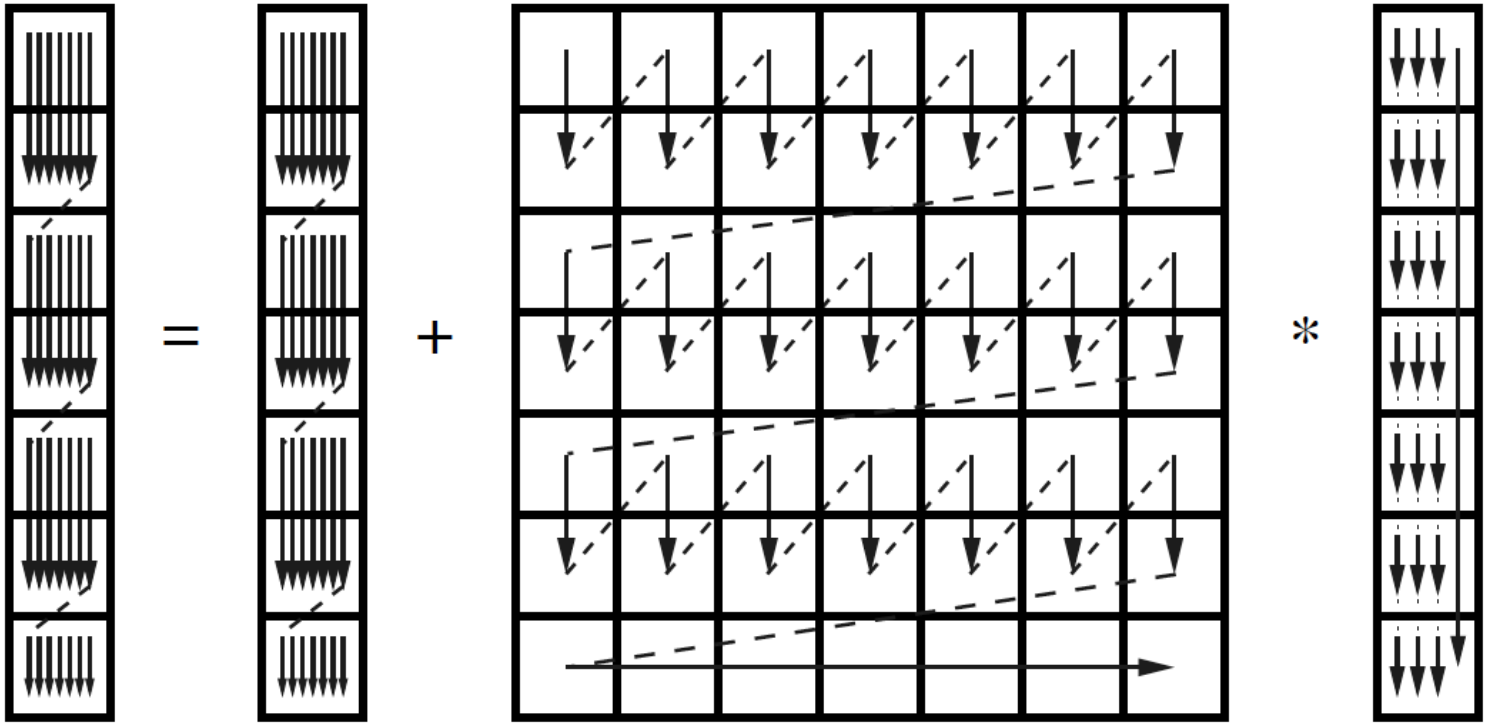


Figure 3.12: Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

Another $O(N^2)/O(N^2)$ algorithm: matrix transpose

$$A = B^T$$

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    A[j][i] = B[i][j];
```

Both `A` and `B` are assumed to be 2D arrays with row-major storage.

Very large jumps in memory associated with loading `B[i][j]` $\rightarrow$ very bad cache line utilization.

Loop unrolling applied to matrix transpose

```
for (j=0; j<N; j+=m)
  for (i=0; i<N; i++) {
    A[j+0][i] = B[i][j+0];
    A[j+1][i] = B[i][j+1];
    // ....
    A[j+m-1][i] = B[i][j+m-1];
  }
```

Illustration

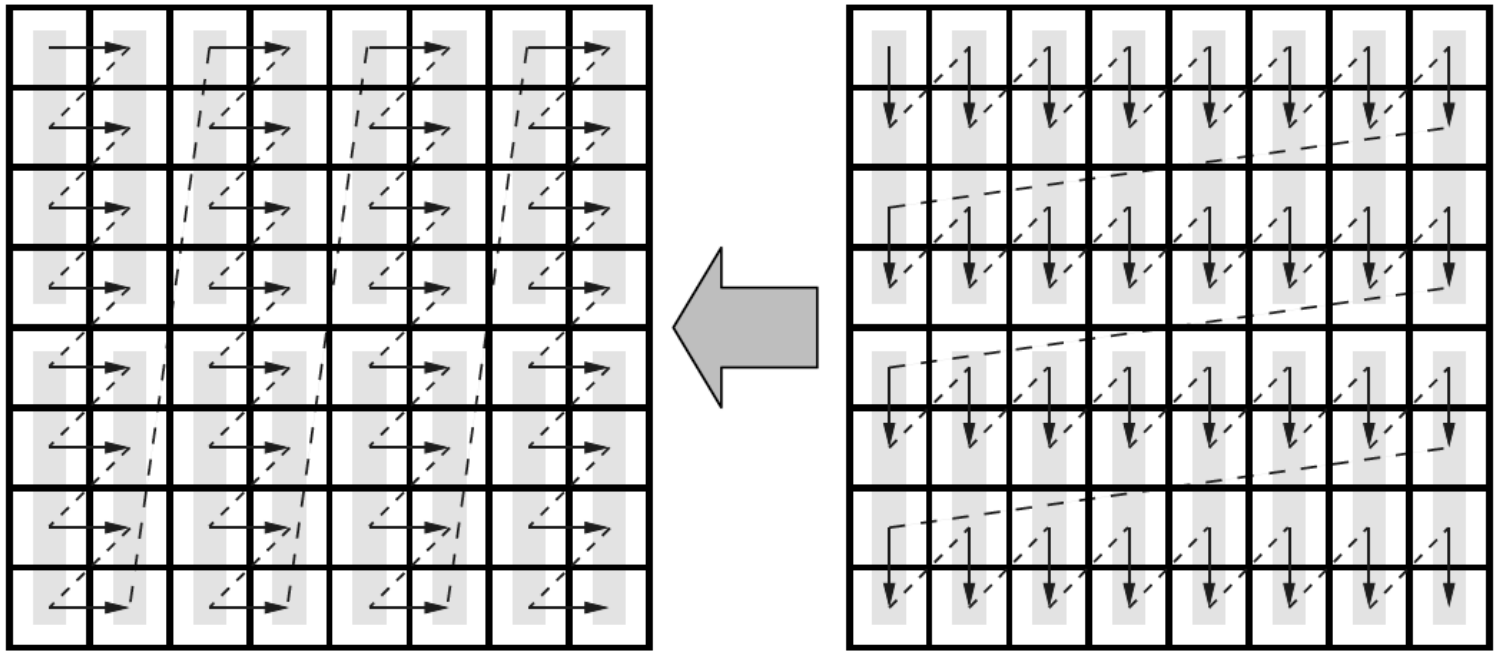


Figure 3.13: Two-way unrolled "flipped" matrix transpose (i.e., with strided load in the original version).

B *A*

Loop blocking + unrolling

```
for (jj=0; jj<N; jj+=b) {
  jstart = jj; jstop = jj+b-1;
  for (ii=0; ii<N; ii+=b) {
    istart = ii; istop = ii+b-1;

    for (j=jstart; j<=jstop; j+=m)
      for (i=istart; i<=istop; i++) {
        A[j+0][i] = B[i][j+0];
        A[j+1][i] = B[i][j+1];
        // ....
        A[j+m-1][i] = B[i][j+m-1];
      }
  }
}
```

Illustration

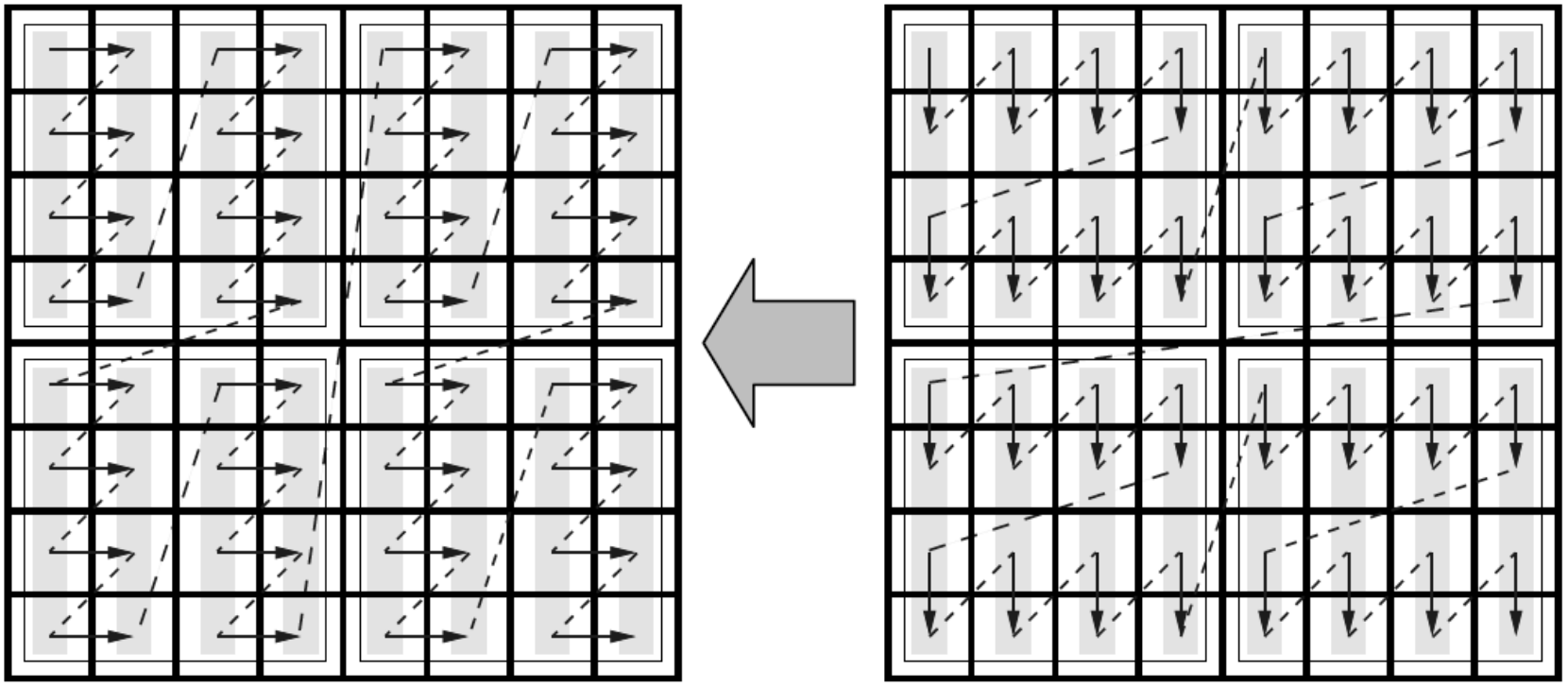


Figure 3.14: 4×4 blocked and two-way unrolled "flipped" matrix transpose.

B A

Access optimization for algorithm class $O(N^2)/O(N)$

Example:

```
double sum = 0.;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++)
    sum = sum + foo(A[i],B[j])
}
```

- Array `B` has the risk of being loaded N times (when N is large)
- Total number of memory loads: $N + N^2$

Applying loop blocking

```
double sum = 0.;

for (jj=0; jj<N; jj+=b) {
  jstart = jj; jstop = jj+b-1;

  for (i=0; i<N; i++) {
    for (j=jstart; j<=jstop; j++)
      sum = sum + foo(A[i],B[j])
  }
}
```

- Appropriate choice of b will allow array B to be loaded from memory only once.
- Array `A` will now be loaded N/b times.
- Total number of memory loads: $N + N^2/b$