Do less work; example 2

```
for (i=0; i<500; i++)
  for (j=0; j<80; j++)
    for (k=0; k<4; k++)
      a[i][j][k] = a[i][j][k] + b[i][j][k]*c[i][j][k];
```

How many times is the `k`-indexed loop executed? And how many times for the `j`-indexed loop?

Do less work; example 2 (cont'd)

If the 3D arrays `a`, `b` and `c` have **contiguous** memory storage for all their values, then we can re-code as follows:

```
double *a_ptr = a[0][0];
double *b_ptr = b[0][0];
double *c_ptr = c[0][0];

for (i=0; i<(500*80*4); i++)
  a_ptr[i] = a_ptr[i] + b_ptr[i]*c_ptr[i];
```

This technique is called *loop collapsing*. The main motivation is to reduce loop overhead, may also help other (compiler-supported) optimizations.

Do less work; example 3

```
for (i=0; i<ARRAY_SIZE; i++) {
  a[i] = 0.;
  for (j=0; j<ARRAY_SIZE; j++)
    a[i] = a[i] + b[j]*d[j]*c[i];
}
```

Observation: `c[i]` is independent of the `j`-indexed loop.

Do less work; example 3 (further simplification)

There is a common factor:
`b[0]*d[0]+b[1]*d[1]+...+b[ARRAY_SIZE-1]*d[ARRAY_SIZE-1]`
which is unnecessarily re-computed in every `i` iteration!

```
t = 0.;
for (j=0; j<ARRAY_SIZE; j++)
  t = t + b[j]*d[j];

for (i=0; i<ARRAY_SIZE; i++)
  a[i] = t*c[i];
```

This technique is called *loop factoring* or *elimination of common subexpressions*.

Another example of common subexpression elimination

```
for (i=0; i<N; i++)
  A[i] = A[i] + s + r*sin(x);
```

```
tmp = s + r*sin(x);
for (i=0; i<N; i++)
  A[i] = A[i] + tmp;
```

Avoid expensive operations!

Special math functions (such as trigonometric, exponential and logarithmic functions) are usually very costly to compute.

An example from simulating non-equilibrium spins:

```
for (i=1; i<Nx-1; i++)
  for (j=1; j<Ny-1; j++)
     for (k=1; k<Nz-1; k++) {
       iL = spin_orientation[i-1][j][k];
       iR = spin_orientation[i+1][j][k];
       iS = spin_orientation[i][j-1][k];
       iN = spin_orientation[i][j+1][k];
       iO = spin_orientation[i][j][k-1];
       iU = spin_orientation[i][j][k+1];
       edelz = iL+iR+iS+iN+iO+iU;
       body_force[i][j][k] = 0.5*(1.0+tanh(edelz/tt));
     }
```

Example continued

If the values of `iL, iR, iS, iN, iO, iU` can only be -1 or $+1$, then the value of `edelz` (which is the sum of `iL, iR, iS, iN, iO, iU`) can only be $-6, -4, -2, 0, 2, 4, 6$.

If `tt` is a constant, then we can create a lookup table:

```
double tanh_table[13];
for (i=0; i<=12; i+=2)
  tanh_table[i] = 0.5*(1.0+tanh((i-6)/tt));
```

```
for (i=1; i<Nx-1; i++)
  for (j=1; j<Ny-1; j++)
     for (k=1; k<Nz-1; k++) {
       ....
       edelz = iL+iR+iS+iN+iO+iU;
       body_force[i][j][k] = tanh_table[edelz+6];
     }
```

Strength reduction

```
for (i=0; i<N; i++)
  y[i] = pow(x[i],3)/s;
```

```
double inverse_s = 1.0/s;
for (i=0; i<N; i++)
  y[i] = x[i]*x[i]*x[i]*inverse_s;
```

Strength reduction (another example)

```
for (i=0; i<N; i++)
  y[i] = a*pow(x[i],4)+b*pow(x[i],3)+c*pow(x[i],2)
       +d*pow(x[i],1)+e;
```

```
for (i=0; i<N; i++)
  y[i] = (((a*x[i]+b)*x[i]+c)*x[i]+d)*x[i]+e;
```

Use of Horner's rule of polynomial evaluation:

$$ax^4 + bx^3 + cx^2 + dx + e = (((ax + b)x + c)x + d)x + e$$

Shrinking the work set!

The *work set* of a code is the amount of memory it uses (or touches), also called *memory footprint*.

In general, shrinking the work set (if possible) is a good thing for performance, because it raises the probability of cache hit.

One example: The `spin_orientation` array should store values of type `char` instead of type `int`. (A factor of 4 in the difference of memory footprint.)

Avoiding branches

"Tight" loops: few operations per iteration, typically optimized by compiler using some form of pipelining. In case of conditional branches in the loop body, the compiler optimization will easily fail.

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++) {
    if (i>j)
      sign = 1.0;
    else if (i<j)
      sign = -1.0;
    else
      sign = 0.0;

    C[j] = C[j] + sign * A[j][i] * B[i];
  }
```

Avoiding branches (cont'd)

```
for (j=0; j<N-1; j++)
  for (i=j+1; i<N; i++)
    C[j] = C[j] + A[j][i] * B[i];

for (j=1; j<N; j++)
  for (i=0; i<j; i++)
    C[j] = C[j] - A[j][i] * B[i];
  }
```

We have got rid of the if-tests completely!

Another example of avoiding branches

```
for (i=0; i<n; i++) {
  if (i==0)
    a[i] = b[i+1]-b[i];
  else if (i==n-1)
    a[i] = b[i]-b[i-1];
  else
    a[i] = b[i+1]-b[i-1];
}
```

Another example of avoid branches (cont'd)

Using the technique of *loop peeling*, we can re-code as follows:

```
a[0] = b[1]-b[0];
for (i=1; i<n-1; i++)
  a[i] = b[i+1]-b[i-1];
a[n-1] = b[n-1]-b[n-2];
```

Yet anothe example of avoiding branches

```
for (i=0; i<n; i++) {
  if (j>0)
    x[i] = x[i] + 1;
  else
    x[i] = 0;
}
```

```
if (j>0)
  for (i=0; i<n; i++)
    x[i] = x[i] + 1;
else
  for (i=0; i<n; i++)
    x[i] = 0;
```

Using SIMD instructions

A “vectorizable” loop can potentially run faster if multiple operations can be performed with a single instruction.

Using SIMD instructions, register-to-register operations will be greatly accelerated.

Warning: if the code is strongly limited by memory bandwidth, no SIMD technique can bridge this gap.

Ideal scenario for applying SIMD to a loop

- All iterations are independent
- There is no branch in the loop body
- The arrays are accessed with a stride of one

Example:

```
for (i=0; i<N; i++)
  r[i] = x[i] + y[i];
```

(We assume here that the memory regions pointed by `r`, `x`, `y` do not overlap—no aliasing)

An example of applying SIMD

Pseudocode of applying SIMD (assuming that each SIMD register can store 4 values):

```
int i, rest = N%4;
for (i=0; i<N-rest; i+=4) {
  load R1 = [x[i],x[i+1],x[i+2],x[i+3]];
  load R2 = [y[i],y[i+1],y[i+2],y[i+3]];
  R3 = ADD(R1,R2);
  store [r[i],r[i+1],r[i+2],r[i+3]] = R3;
}
for (i=N-rest; i<N; i++)
  r[i] = x[i] + y[i];
```

Beware of loop dependency!

If a loop iteration depends on the result of another iteration—**loop-carried dependency**

```
for (i=start; i<end; i++)
  A[i] = 10.0*A[i+offset];
```

If `offset<0` $\rightarrow$ **real** dependency (read-after-write hazard)
If `offset>0` $\rightarrow$ pseudo dependency (write-after-read harzard)

When there is loop-carried dependency...

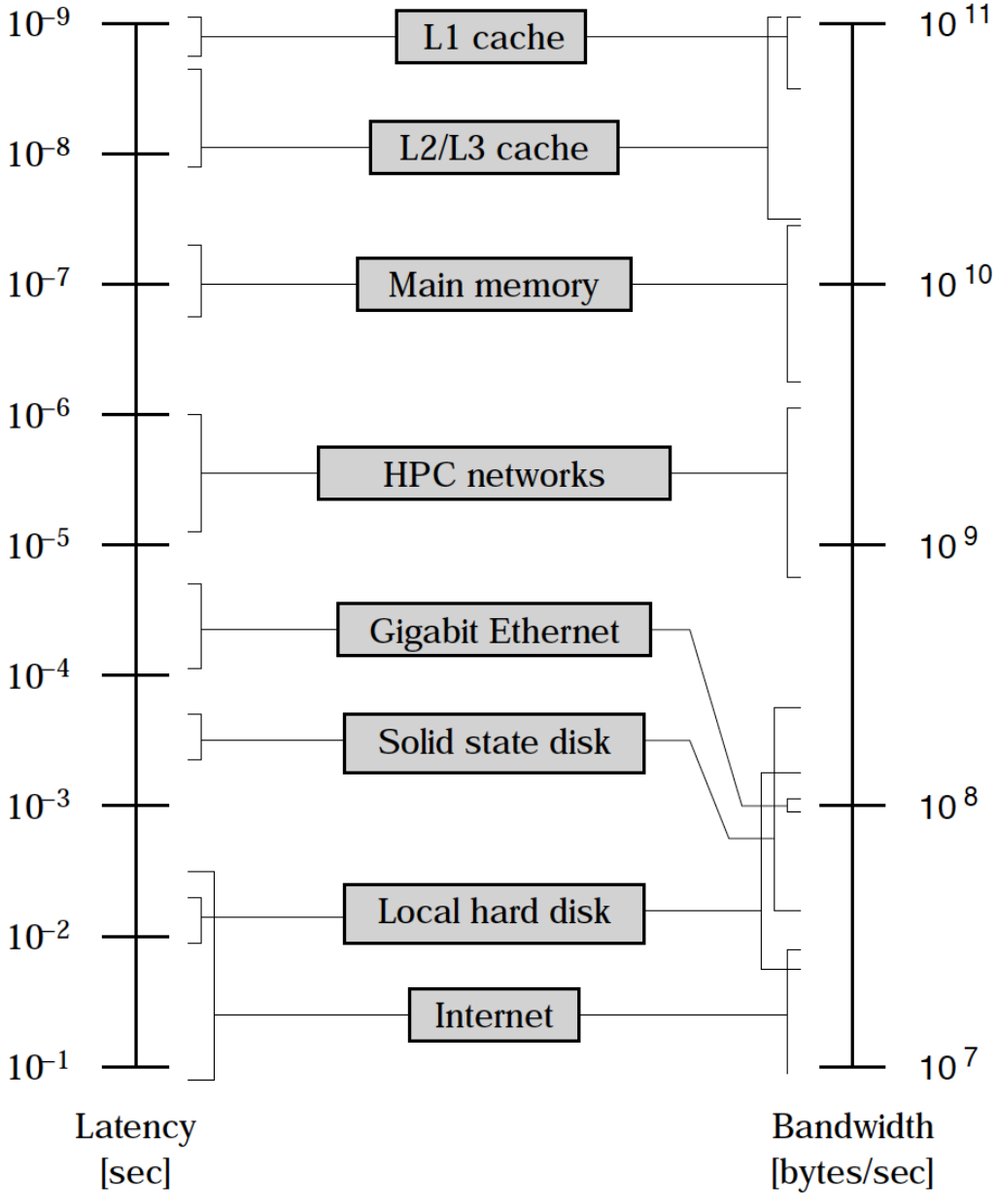
In case of real dependency, SIMD cannot be applied if the negative `offset` size is smaller than the SIMD width. For example,

```
for (i=start; i<end; i++)
  A[i] = 10.0*A[i-1];
```

In case of pseudo dependency, SIMD can be applied. For example when `offset`>0,

```
for (i=start; i<end; i++)
  A[i] = 10.0*A[i+offset];
```

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.

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 (for example thanks to prefetch).

"Word" = one DP value (8 bytes)

"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**.

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/`

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.

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 Section 3.3 of 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, different from the Fortran code example used in the textbook!)

2D Jacobi: performance prediction

Balance analysis applied to 2D Jacobi:

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

2D Jacobi: performance prediction (cont'd)

Suppose the (last-level) 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 guaranteed to be already in cache (it was recently loaded again from memory for computing `phi_new[k][i-2]`);
- The `phi[k][i+1]` has to be loaded again from memory for computing `phi_new[k][i]` (and 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 still in cache;
- The `phi[k][i+1]` value is also still 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

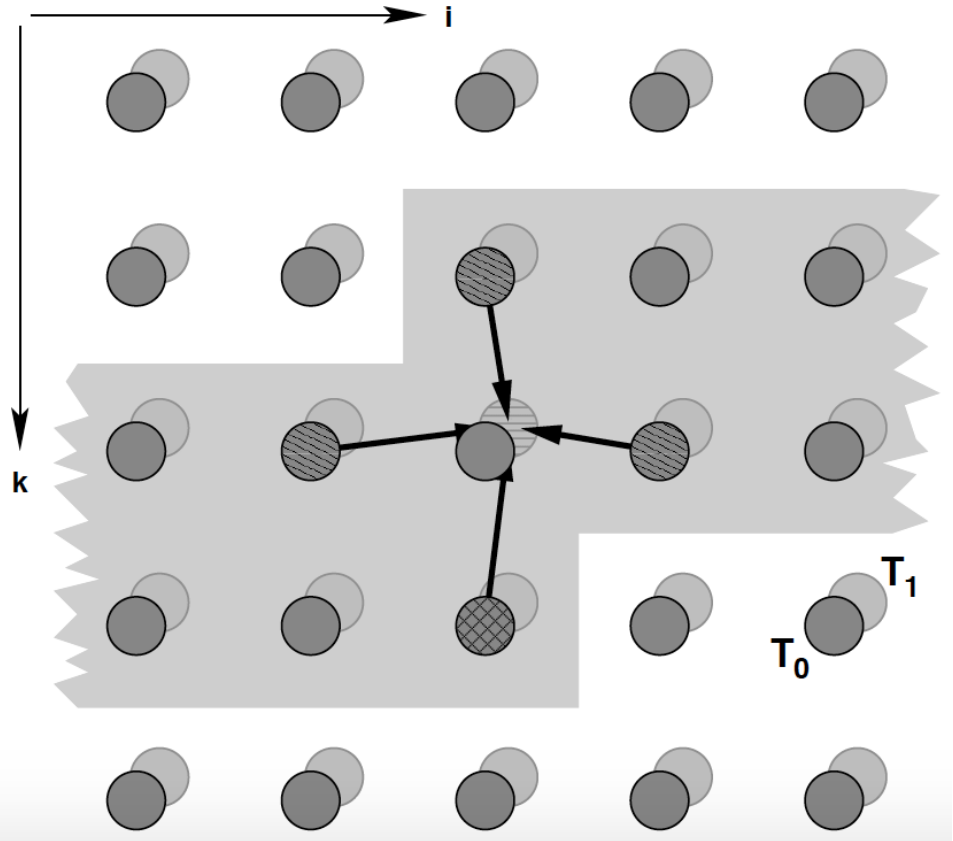


Figure 3.5: Stencil update for the plain 2D Jacobi algorithm. If at least two successive rows can be kept in the cache (shaded area), only one T_0 site per update has to be fetched from memory (cross-hatched site).

Question to consider

What will be the code balance, if one row of `phi` fits in the (last-level) cache, but not two rows?

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

Original code: 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 & stores: N^2 for 2D array `A`, $2N$ 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

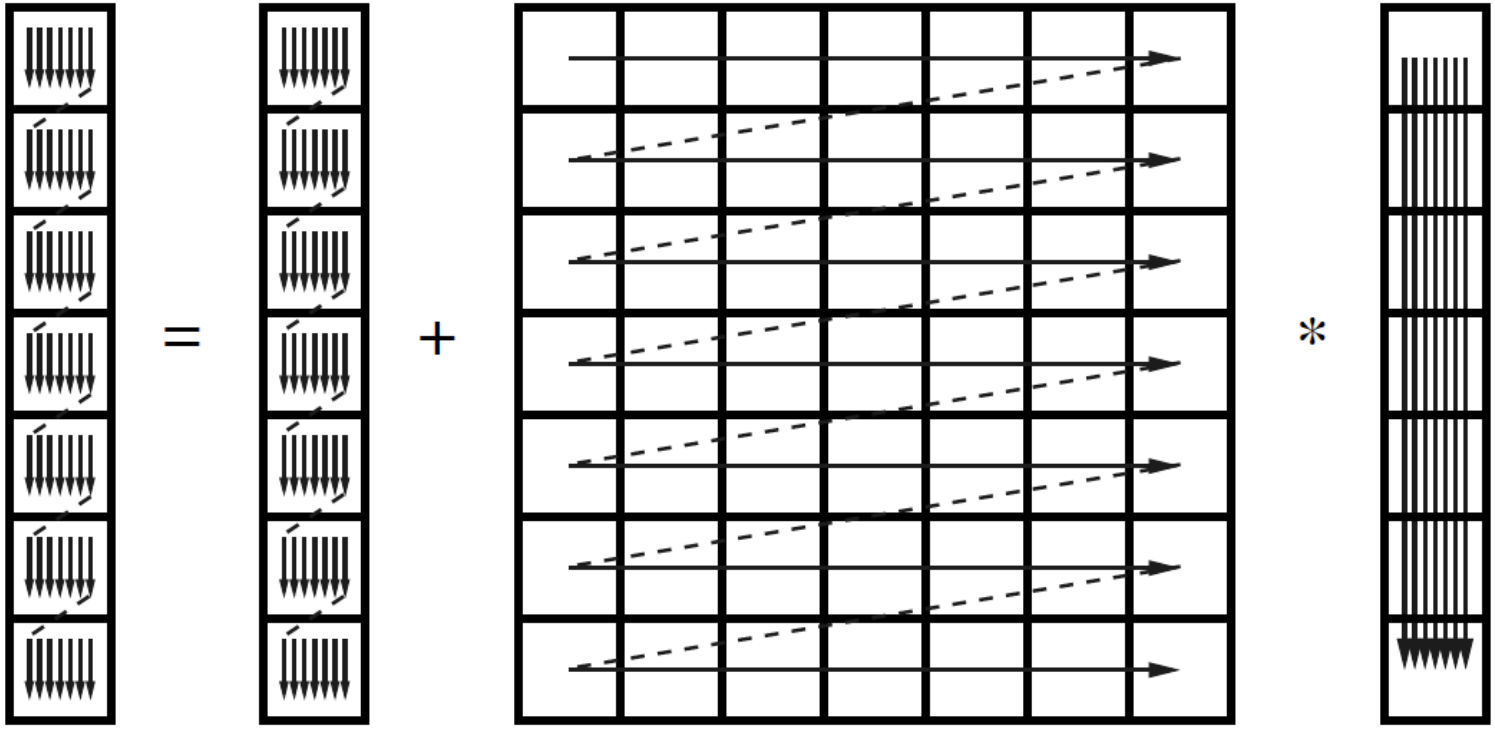


Figure 3.11: Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded N times.

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 and stores: $N^2 + N^2/m + 2N$ (for small cache size)
- Size of *m* shouldn't be too large, to avoid too high *register pressure*

Illustration of the effect of unrolling

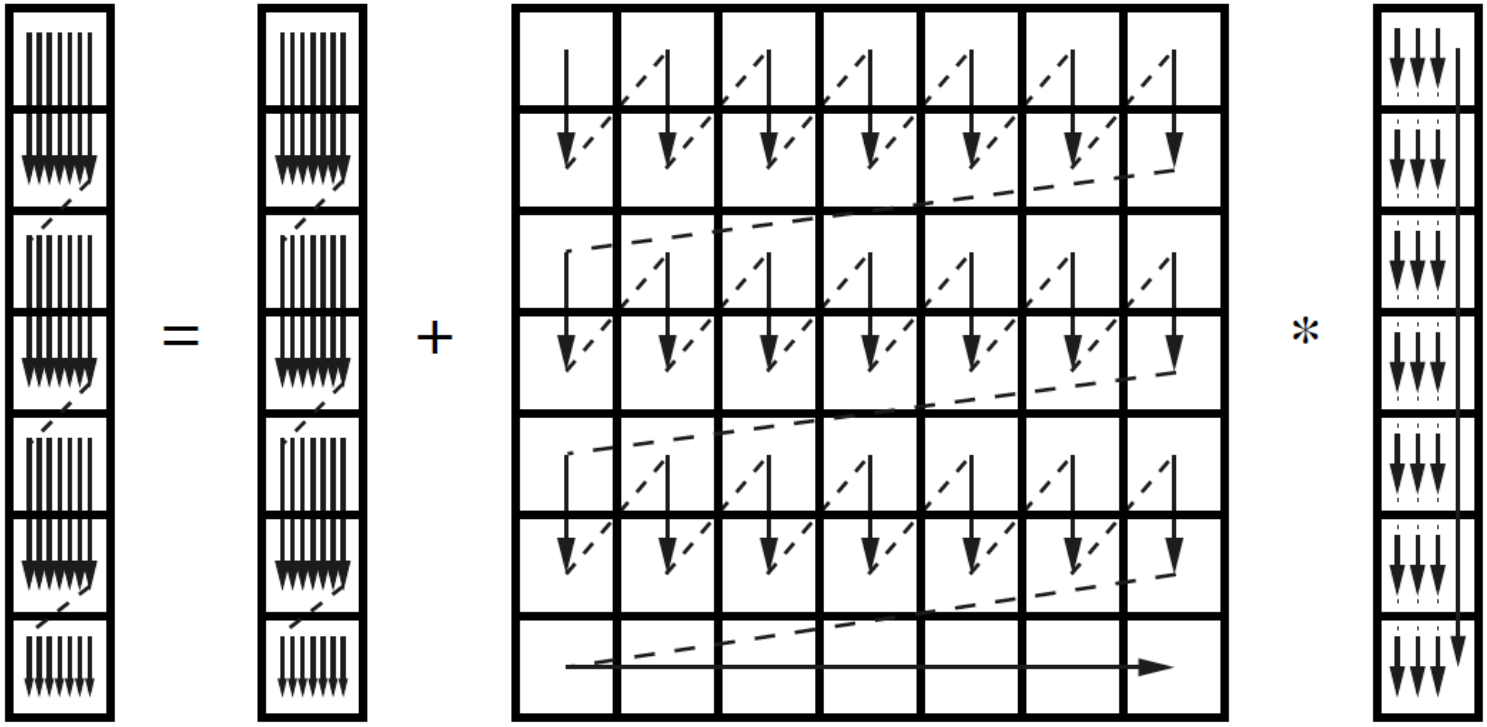


Figure 3.12: Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

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. (**Note: The matrix-transpose example in the textbook is programmed in Fortran, thus column-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

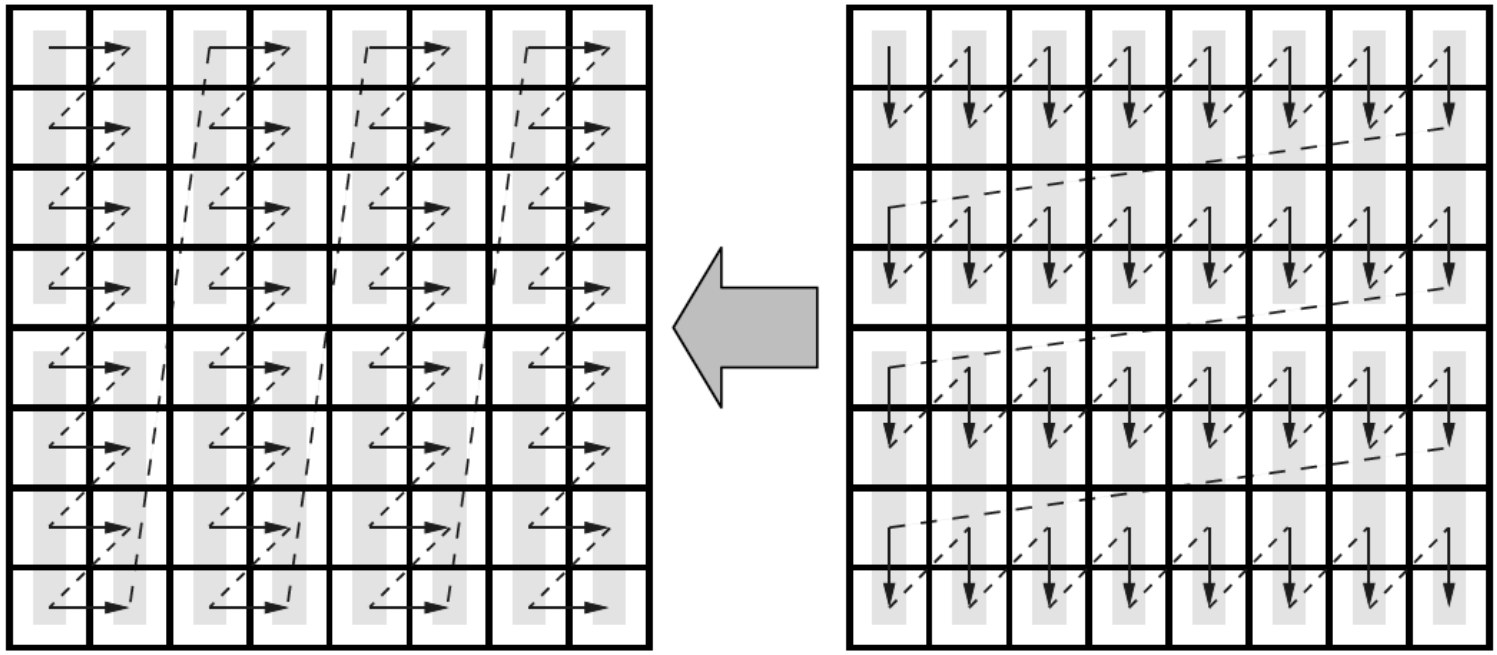


Figure 3.13: Two-way unrolled "flipped" matrix transpose (i.e., with strided load in the original version).

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

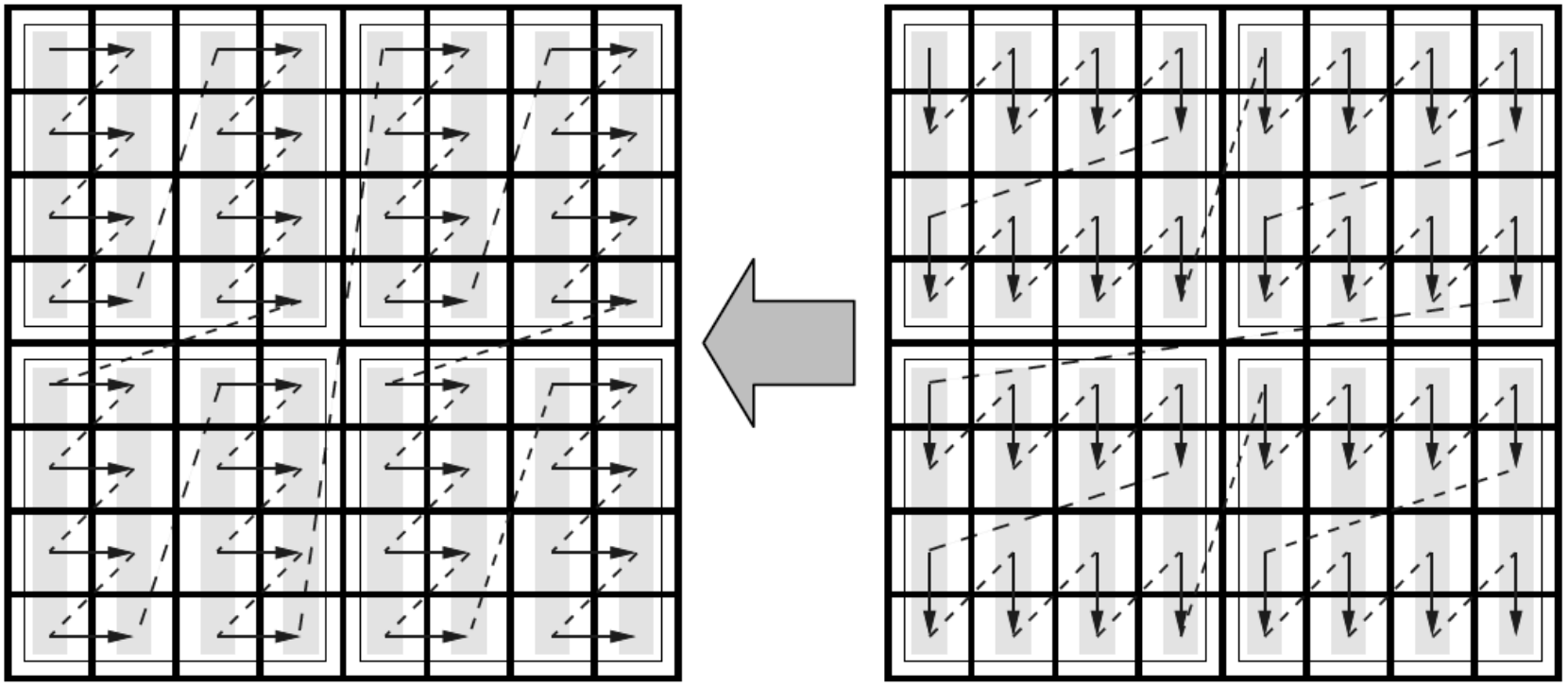


Figure 3.14: 4×4 blocked and two-way unrolled "flipped" matrix transpose.

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 (instead of only once).
- Total number of memory loads: $N^2/b + N$

Straightforward implementation & balance analysis

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

- Total number of floating-point (FP) operations: $2N^2$
- Memory traffic: N^2 loads for 2D array `A`, N loads & N stores for 1D array `C`
- 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

Code balance for the small-cache case:

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

Illustration of array B being loaded N times

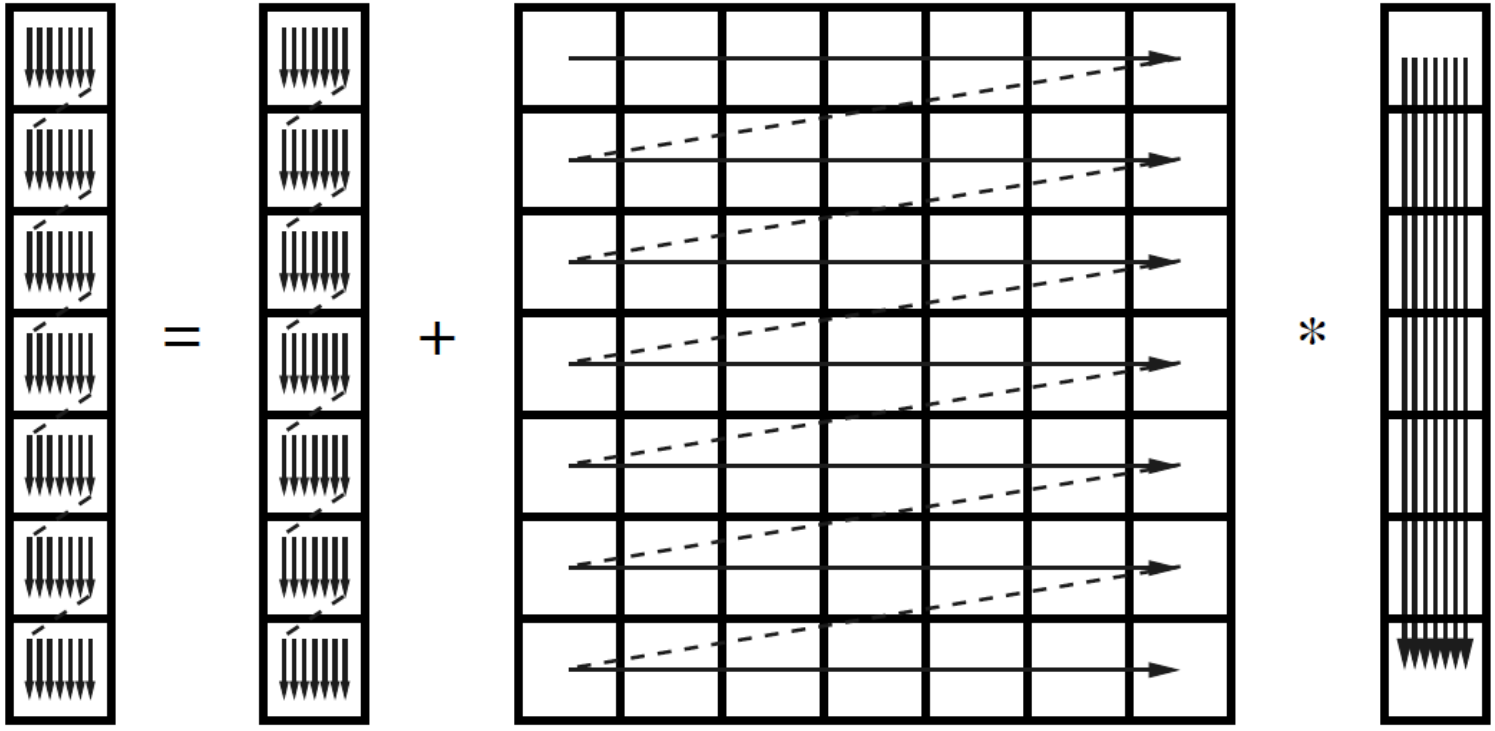


Figure 3.11: Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded N times.

How to reduce memory traffic for small-cache case?

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 for array `B`: N^2/m (for small-cache case)
- Size of *m* shouldn't be too large, to avoid too high *register pressure*

Illustration of the effect of unrolling

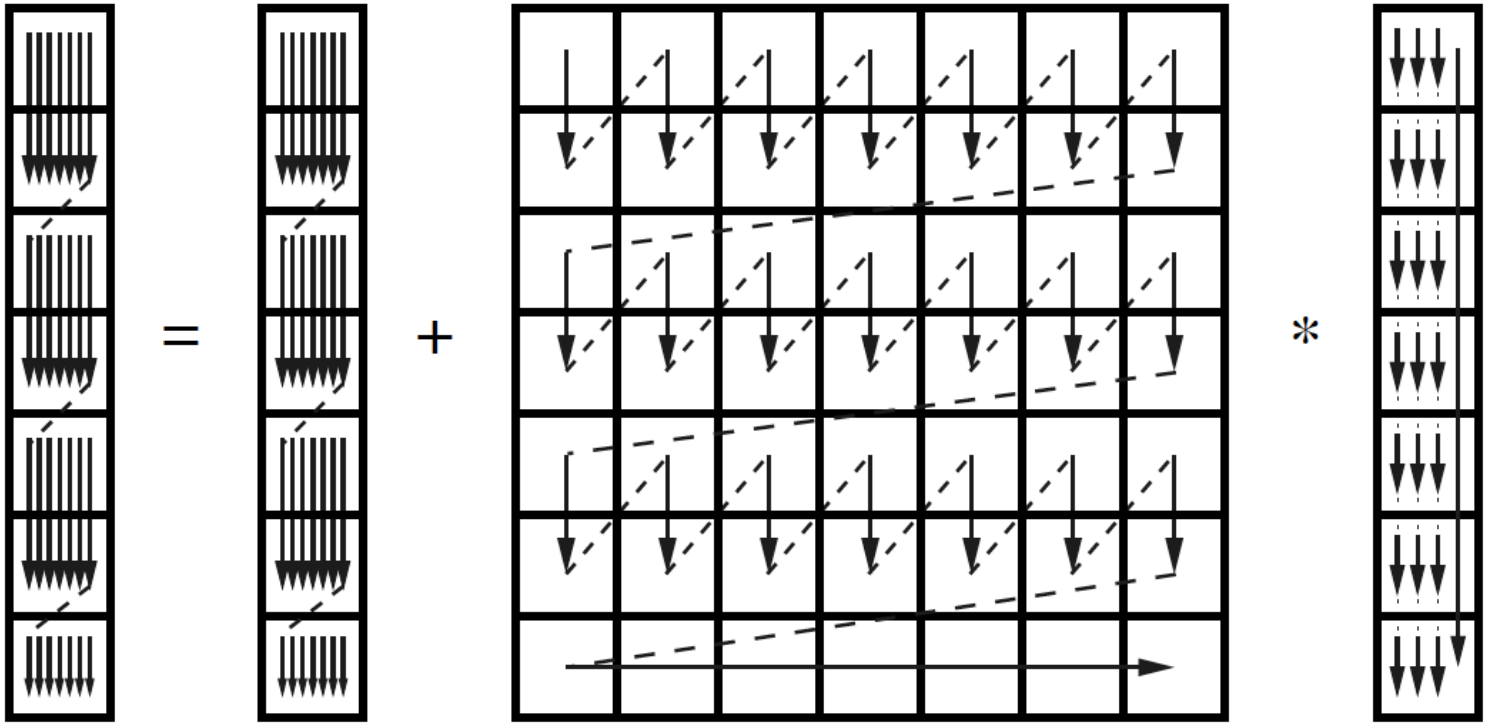


Figure 3.12: Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

Improved code balance

For the small-cache case, unroll and jam will result in the following improved code balance:

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

Illustration of sparse matrix-vector multiply

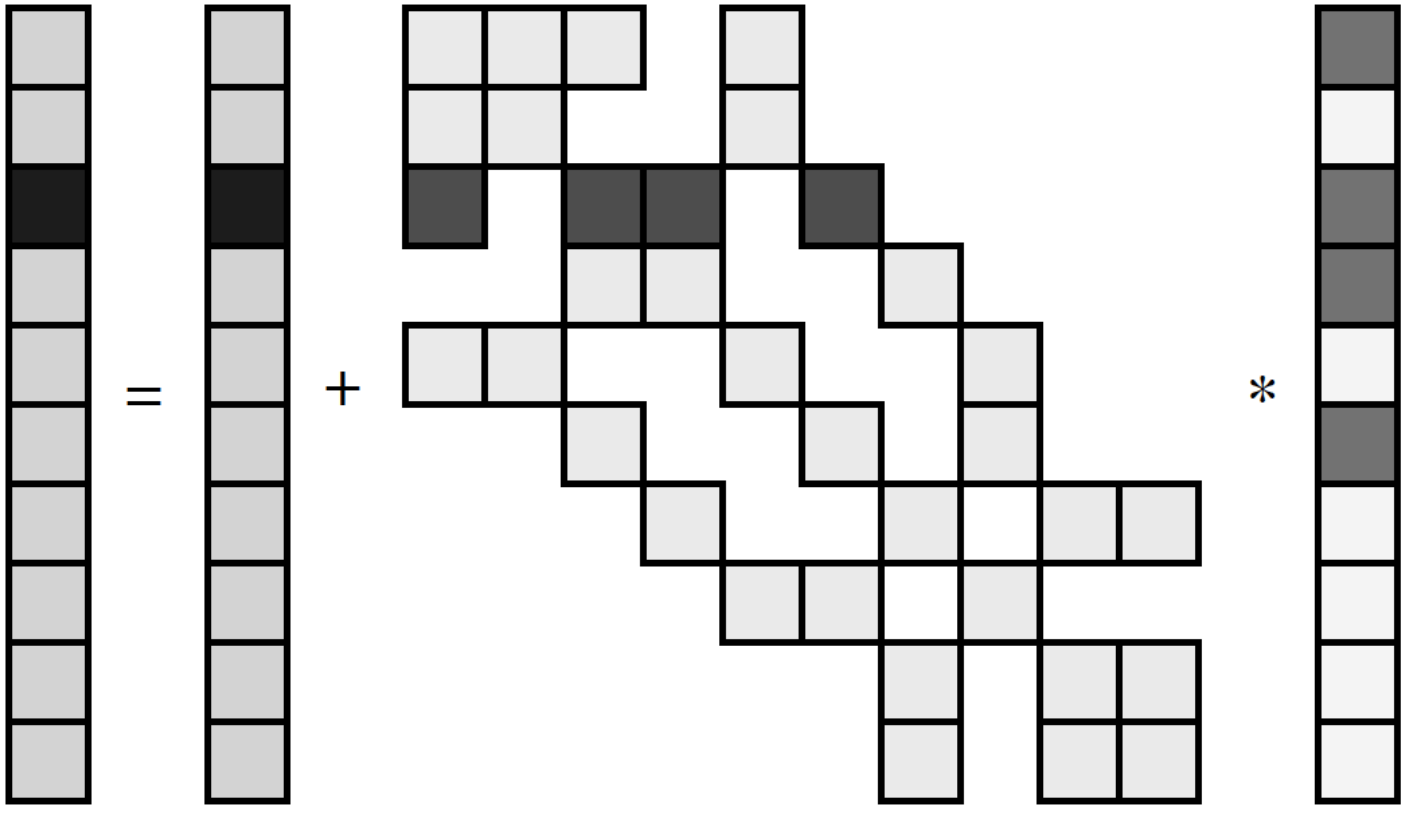


Figure 3.15: Sparse matrix-vector multiply. Dark elements visualize entries involved in updating a single LHS element. Unless the sparse matrix rows have no gaps between the first and last nonzero elements, some indirect addressing of the RHS vector is inevitable.

Compressed row storage (CRS) format

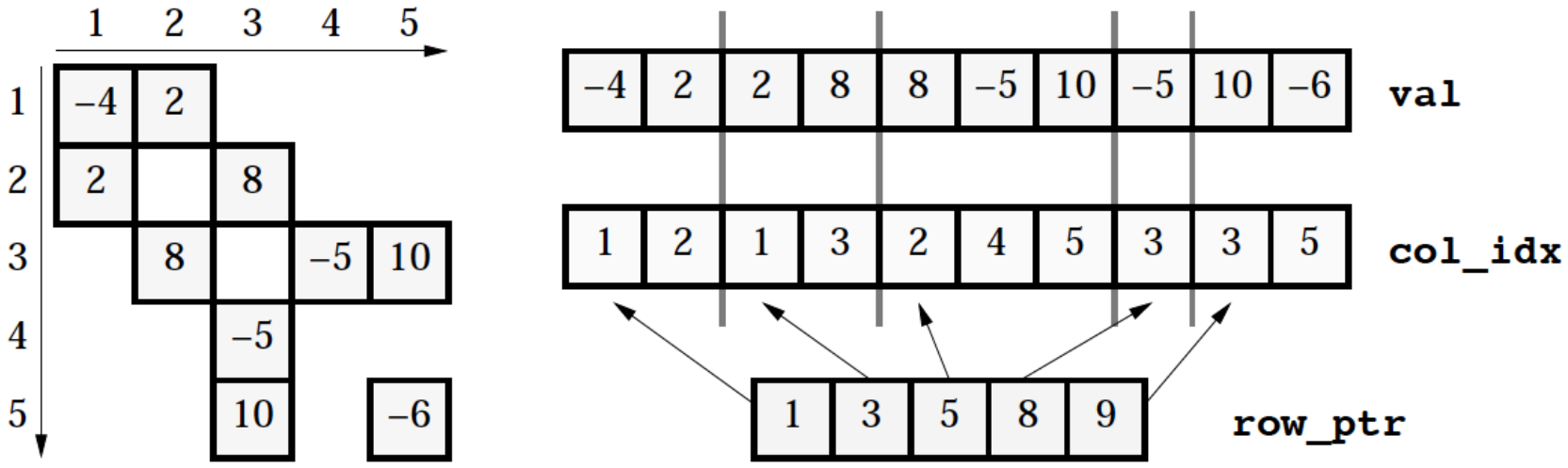


Figure 3.16: CRS sparse matrix storage format.

Three arrarys:

- 1D array `val`, of length N_{nz}, stores all the nonzero values of the sparse matrix
- 1D array `col_idx` , of length N_{nz}, records the original column positions of the all nonzero values
- 1D array `row_ptr`, of length $N + 1$, contains the indices at which new rows start in array `val`

Implementation of matrix-vector multiply using CRS format

```
for (i=0; i<N; i++) {
  tmp = C[i];
  for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
    tmp = tmp + val[j]*B[col_idx[j]];
  C[i] = tmp;
}
```

- There is a long outer loop (of length N)
- The inner loop can be very short
- Access to array `C` will be well optimized by compiler
- Access to array `val` is with stride one
- Access to array `B` is indirect (via `col_idx`) and can be irregular

Code balance analysis of matrix-vector multiply with CRS

Best-case scenario (entire `B` array is cached, needing only N loads), each entry in `row_ptr` and `col_idx` is half a word:

$$\frac{N_{\mathrm{nz}}(1+0.5)+0.5N+N+2N}{2N_{\mathrm{nz}}}$$

Worst-case scenario (`B[col_idx[j]]` needs to be loaded from memory every single time, and only one value is used per cacheline):

$$\frac{N_{\mathrm{nz}}(1+0.5)+0.5N+N_{\mathrm{nz}}\frac{\text{cacheline size}}{\text{word size}}+2N}{2N_{\mathrm{nz}}}$$

Main ideas for improvement

- Continue using CRS format, but with suitable permutations (to reduce the actual memory traffic associated with array `B`)
- Use the JDS format with further optimization (see Sections 3.6.1 & 3.6.2)

Taxonomy of parallel computing paradigms

...for describing the amount of concurrent control and data streams present in a parallel architecture

Dominating concepts:

- **SIMD** (*Single Instruction, Multiple Data*)—A single instruction stream, either on a single processor (core) or on multiple computing elements, provides parallelism by operating on multiple data streams concurrently. (Hardware examples: vector processors, SIMD-capable modern superscalar microprocessors and GPUs.)
- **MIMD** (*Multiple Instruction, Multiple Data*)—Multiple instructions streams on multiple processor (cores) operate on different data items concurrently. (Hardware examples: shared-memory and distributed-memory parallel computers.)

The focus of this chapter is on multiprocessor MIMD parallelism.

Shared-memory computers

A *shared-memory parallel computer* has a number of CPUs (cores) that work on a shared physical address space.

Two varieties:

- *Uniform Memory Access* (UMA) systems hae a "flat" memory model: latency and bandwidth are the same for all processors and all memory locations. (Typically, single multicore processor chips are "UMA machines".)
- *Cache-coherent Nonuniform Memory Access* (ccNUMA) systems have a physically distributed memory that is *logically shared*. The aggregated memory appears as one single address space. Memory access performance depends on the which CPU (core) accesses which parts of memory ("local" vs. "remote" access).

Caches are not (completely) shared

A shared-memory system, no matter UMA or ccNUMA, has multiple CPU cores.

Although there is a single address space (shared memory), there are private caches, or partially shared caches, for the different CPU cores.

Therefore, copies the same cache line **may** reside in several local caches.

Cache coherence

Problematic situations when a cache line resides in several caches:

- If the cache line in one of the caches is modified, the other caches' contents are *outdated* (thus invalid).
- If different parts of the cache line are modified by different processors in their local caches $\rightarrow$ no one has the correct cache line anymore.

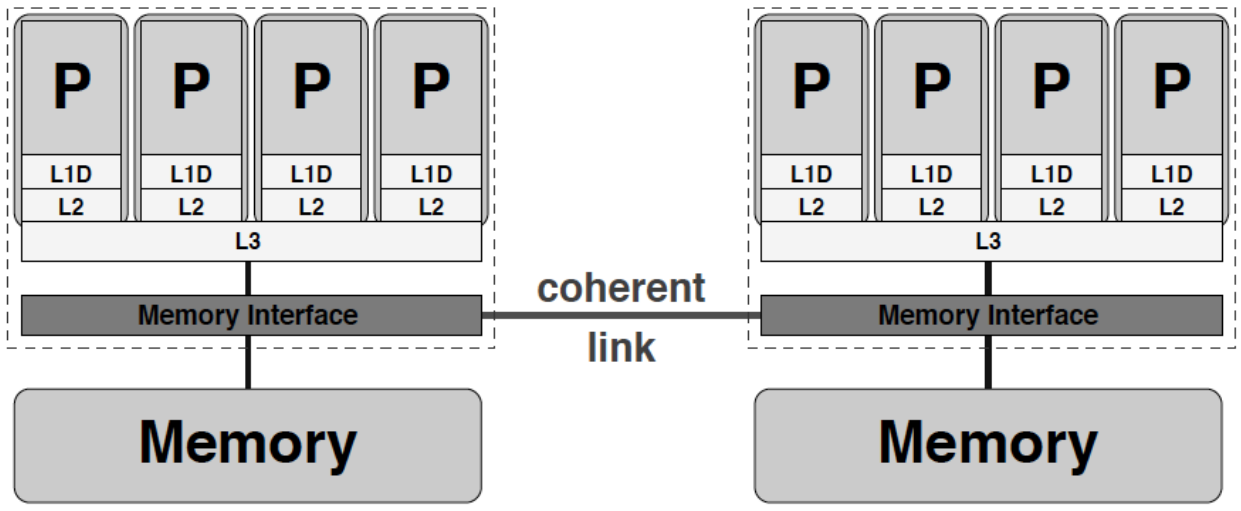
Cache coherence protocols (supported in hardware) guarantee *consistency* between cached data and data in the shared memory at all times.

ccNUMA for scalable memory bandwidth

- A *locality domain* (LD) is a set of processor cores together with locally connected memory. This "local" memory can be accessed by the set of processor cores in the most efficient way, without resorting to a network of any kind.
- Each LD is a UMA building block.
- Multiple LDs are linked via a coherent interconnect, which can mediate direct, cache-coherent memory accesses. (This mechanism is transparent for the programmer.)
- The whole ccNUMA system has a shared address space (memory), runs a single OS instance.

Example of ccNUMA

Figure 4.5: A ccNUMA system with two locality domains (one per socket) and eight cores.



Penalty for non-local transfers

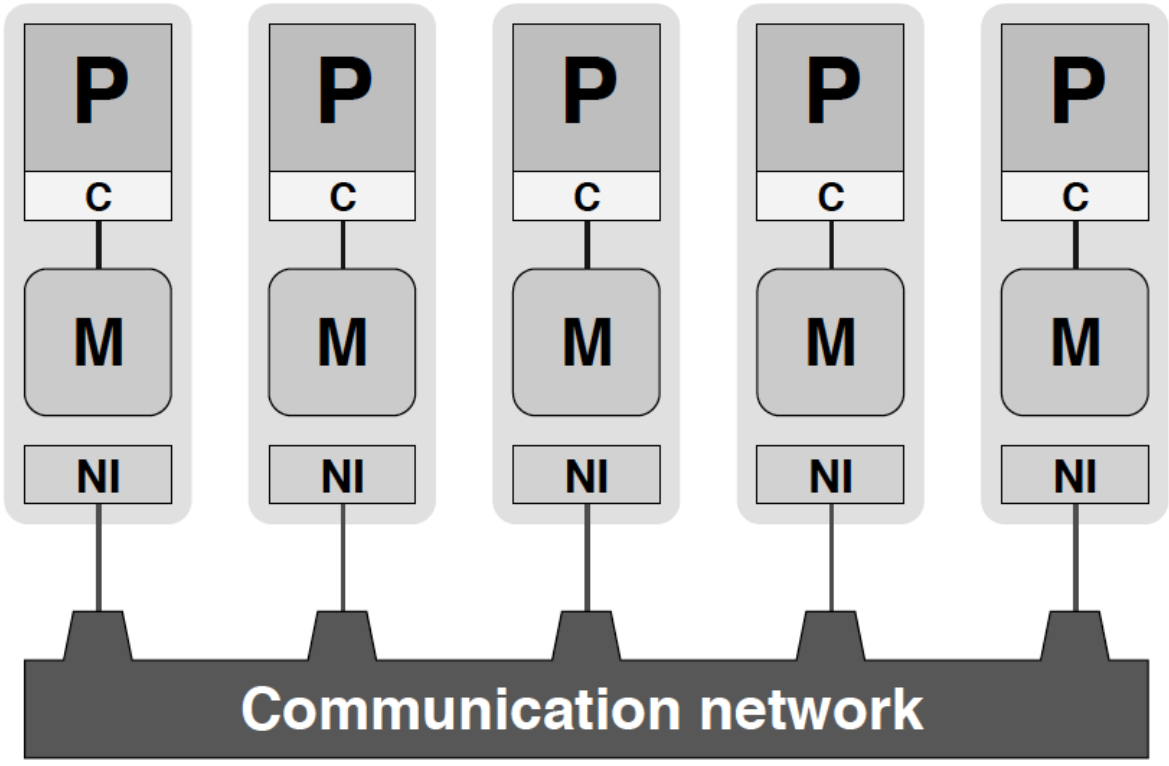
The *locality problem*: Non-local memory transfers (between LDs) are more costly than local transfers (within a LD).

The *contention problem*: If two processors from different LDs access memory in the same LD, fighting for memory bandwidth.

Both problems can be "solved" (alleviated) by carefully observing the data access patterns of an application and restricting data access of each processor (mostly) to its own LD, through proper programming.

A "purely" distributed-memory computer

Figure 4.7: Simplified programmer's view, or "programming model," of a distributed-memory parallel computer: Separate processes run on processors (P), communicating via interfaces (NI) over some network. No process can access another process' memory (M) directly, although processors may reside in shared memory.



"A programmer's view": Each processor is connected to its exclusive local memory (not shared by any other CPUs).

No such "purely" distributed-memory computer today.

Typical modern distributed-memory systems

A cluster of shared-memory "*compute nodes*", interconnected via a *communication network*.

Each node comprises at least one network interface (NI) that mediates the connection to the communication network.

A serial process runs on each CPU (core). Between the nodes, processes can communicate by means of the network.

The layout and speed of the network has a considerable impact on application performance.

Hierarchical hybrid systems

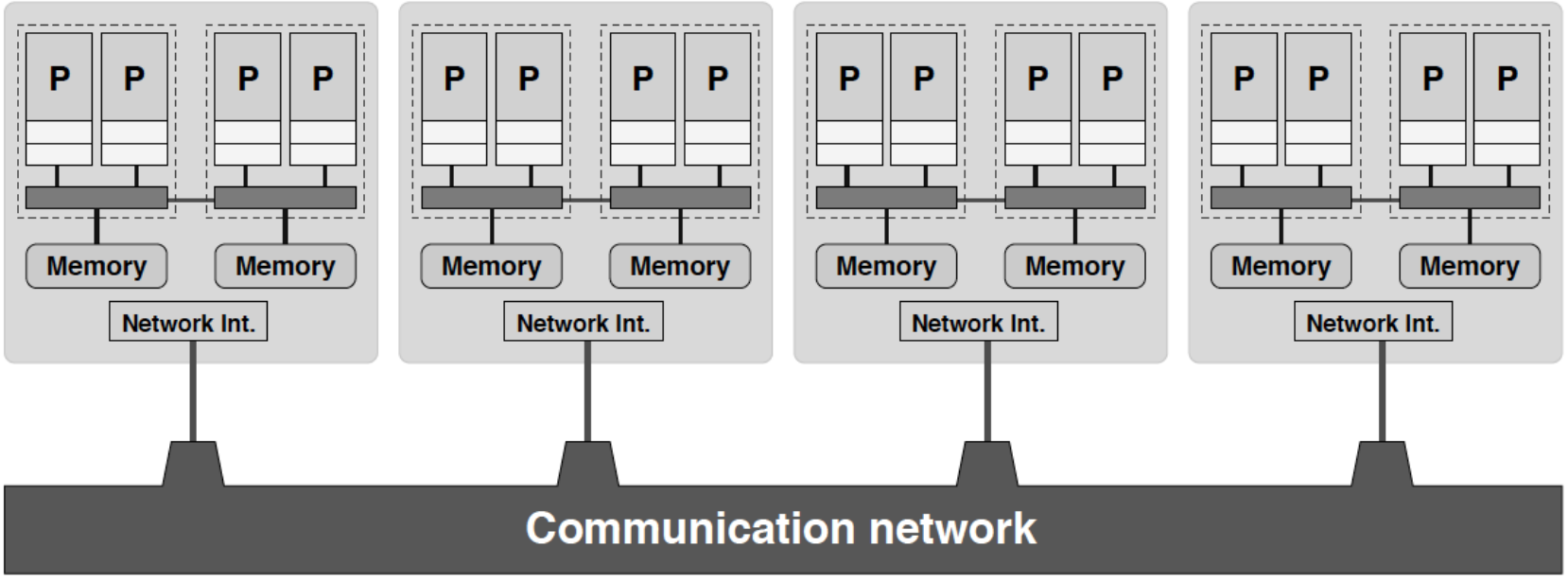


Figure 4.8: Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance "sweet spot" and are thus found in many commodity clusters.