

IN3200/IN4200: Chapter 3

Data access optimization

(Part 2)

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

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

One can *estimate* the theoretically achievable performance of a code, if we know a characteristic ratio that describes the processor (*machine balance*) and a characteristic ratio that describes the code (*code balance*).

Repetition; The concept of “machine balance”

Machine balance, B_m , of a processor is the ratio between the maximum memory bandwidth and the peak FP (*floating-point*) performance:

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

“Word” = one DP (*double-precision*) value (8 bytes)

The machine balance for a modern processor has typically a very small value (meaning the memory is “slow” relative to floating-point operations).

Repetition; The concept of “code balance”

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

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

That is, you should count the number of FP operations (easy), and also count (or estimate) the number of data words transferred over the performance-limiting data path. (Counting the volume of data traffic can be non-trivial.)

Note: $\frac{1}{B_c}$ is called **computational intensity**.

Repetition; The expected maximum performance

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

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

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

Recall: P_{\max} denotes the maximum FP performance, b_{\max} denotes the maximum bandwidth of the performance-limiting data path.

In case $P \ll P_{\max}$: more analysis is needed to find out whether the code balance B_c can be improved by data access optimization (that is, decreasing memory traffic).

Repetition; More realistic “balance analysis”

The theoretical maximum memory bandwidth b_{\max} is difficult to achieve in reality. The realistically achievable memory bandwidth, b_S , can for example be measured by the STREAM benchmark.

Then, the *realistically* achievable maximum FP performance of a code with code balance B_c is estimated as

$$P = \min \left(P_{\max}, \frac{b_S}{B_c} \right)$$

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!)

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 ϕ . Then, memory load traffic needed for computing $\phi_{\text{new}}[k][i]$ 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 $\phi_{\text{new}}[k][i-2]$);
- The $\phi[k][i+1]$ has to be loaded again from memory for computing $\phi_{\text{new}}[k][i]$ (and will be immediately reused for computing $\phi_{\text{new}}[k][i+2]$);
- 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)

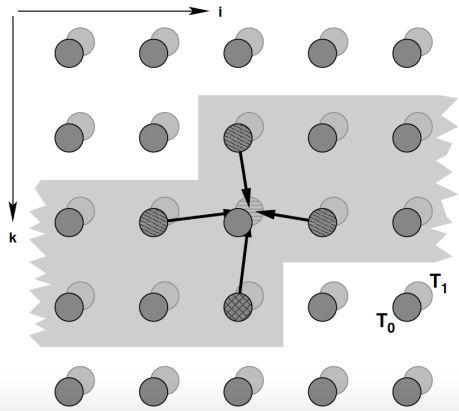
Suppose the cache can store at least two rows of ϕ , but not enough to store the entire array ϕ . Then, memory load traffic needed for computing $\phi_{\text{new}}[k][i]$ is as follows:

- The $\phi[k-1][i]$ value is still in cache (it was first loaded from memory for computing $\phi_{\text{new}}[k-2][i]$);
- 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: $B_c = \frac{5}{2}$
- 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.)

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

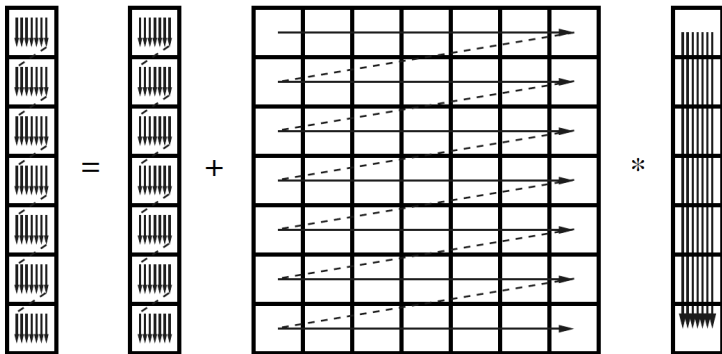


Figure 3.11: Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded N times.

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

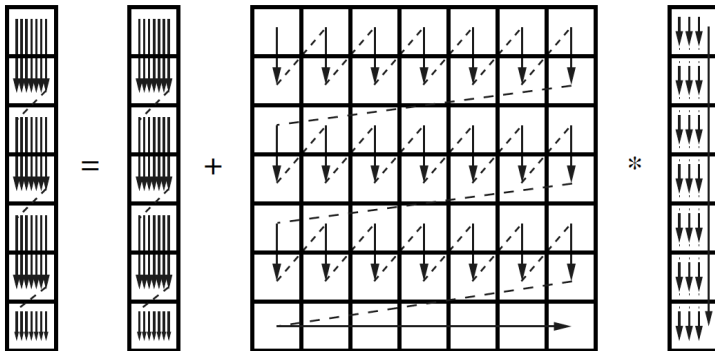


Figure 3.12: Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

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];
  }
```

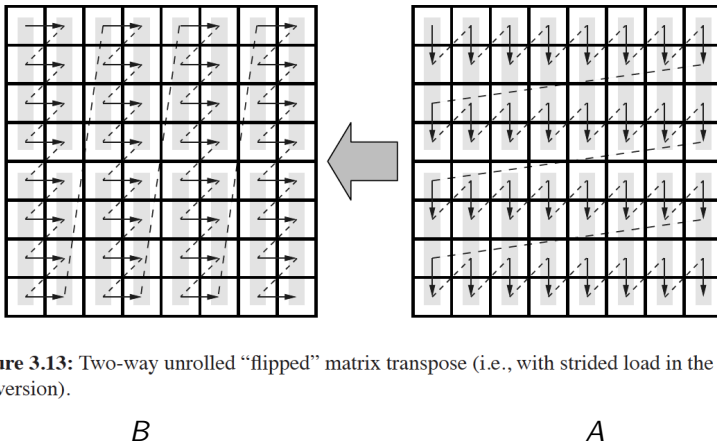


Figure 3.13: Two-way unrolled “flipped” matrix transpose (i.e., with strided load in the original version).

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];  
            }  
    }  
}
```

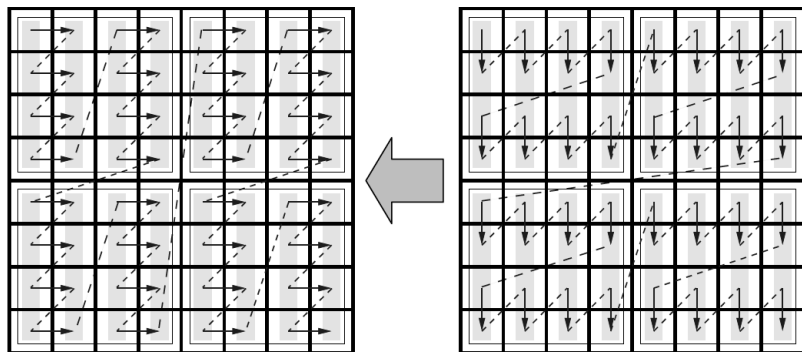



Figure 3.14: 4×4 blocked and two-way unrolled "flipped" matrix transpose.

B

A

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: $N^2/b + N$