# IN3200/IN4200: Chapter 3
# Data access optimization
# (Part 2)

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

# Quick repetition of Part 1

**Bandwidth-based performance modeling/prediction**—to get a rough idea about the maximumly achievable performance of a code.

One can *estimate* the maximumly 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**, $B_{\mathrm{m}}$, of a processor is the ratio between the maximum memory bandwidth and the peak FP (*floating-point)* performance:

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

"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 need to count the number of FP operations (easy), and also count (or estimate) the number of data words transfered over the performance-limiting data path. (Counting the actual amount of transfers can be non-trivial.)

# Repetition; To estimate the maximumly achievable performance

When you know the machine balance $B_{\mathrm{m}}$ of a CPU, and you want to run a code with $B_{\mathrm{c}}$ as its code balance.

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

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

Recall: $P_{\mathsf{max}}$ denotes the theoretical peak FP performance, $b_{\mathsf{max}}$ denotes the theoretical maximum bandwidth of the performance-limiting data path. (To be more realistic, $b_{\mathsf{max}}$ can be replaced by $b_{\mathrm{S}}$, which denotes realistically achievable memory bandwidth.)

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

# Revisiting the first 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 $\Rightarrow B_c = \frac{3+1}{2} = 2$
- Indeed, data traffic to/from memory is always in cachelines, but this doesn't change the code balance for this example:
  - Suppose each cacheline has space for 8 words
  - One cacheline containing 8 values of array `A` is stored to memory every 8th iteration
  - One cacheline containing 8 values of array `B` is loaded from memory every 8th iteration
  - One cacheline containing 8 values of array `C` is loaded from memory every 8th iteration
  - One cacheline containing 8 values of array `D` is loaded from memory every 8th iteration
  - For every 8 iterations, 4 cachelines are stored/loaded to/from memory, that is, 32 words of data traffic
  - During 8 iterations, 16 floating-point operations executed
  - Code balance is still $2 = \frac{32}{16}$

# 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]+phi[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.*)

# An important observation

During every `it` iteration, each `phi[k][i]` value (except those on the boundary) will participate in computing **4** different values of `phi_new`:

1. as the "below" neighbor when computing `phi_new[k-1][i]`
2. as the "right" neighbor when computing `phi_new[k][i-1]`
3. as the "left" neighbor when computing `phi_new[k][i+1]`
4. as the "above" neighbor when computing `phi_new[k+1][i]`

# If there is no cache....

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 three times already);
- The `phi[k][i-1]` value has to be loaded from memory again (although it was loaded from memory twice already);
- The `phi[k][i+1]` value has to be loaded from memory again (although it was loaded from memory once already);
- The `phi[k+1][i]` value has to be loaded from memory (for the first time)

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

# 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 (a little over) 2 rows fit in cache

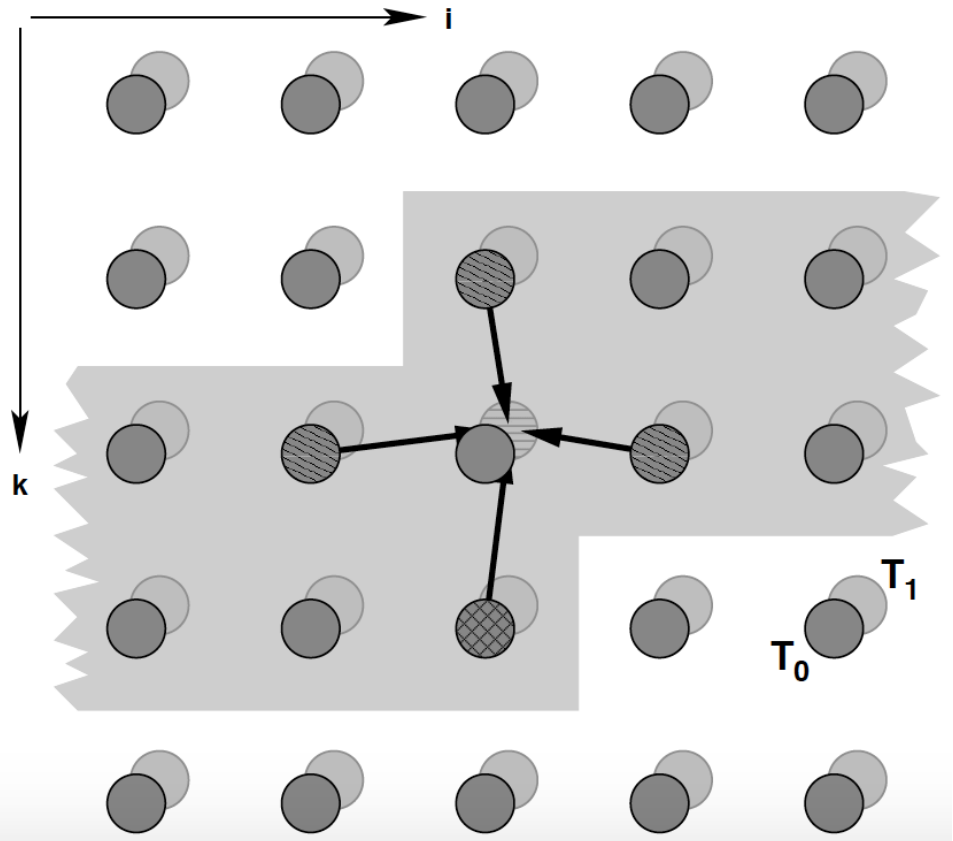


**Figure 3.5:** Stencil update for the plain 2D Jacobi algorithm. If at least two successive rows can be kept in the cache (shaded area), only one $T_0$ site per update has to be fetched from memory (cross-hatched site).

# 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.

# An example of two 1D loops

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];
}
```

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

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

# Loop *fusion*

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;
}
```

- Total number of FP: $2N^2$
- Total number of loads & stores: $N^2$ loads for 2D array `A`, $2N$ for 1D array `C` ($N$ loads, $N$ stores)
- **No memory traffic for** `tmp` (the compiler keeps it only in register)
- 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

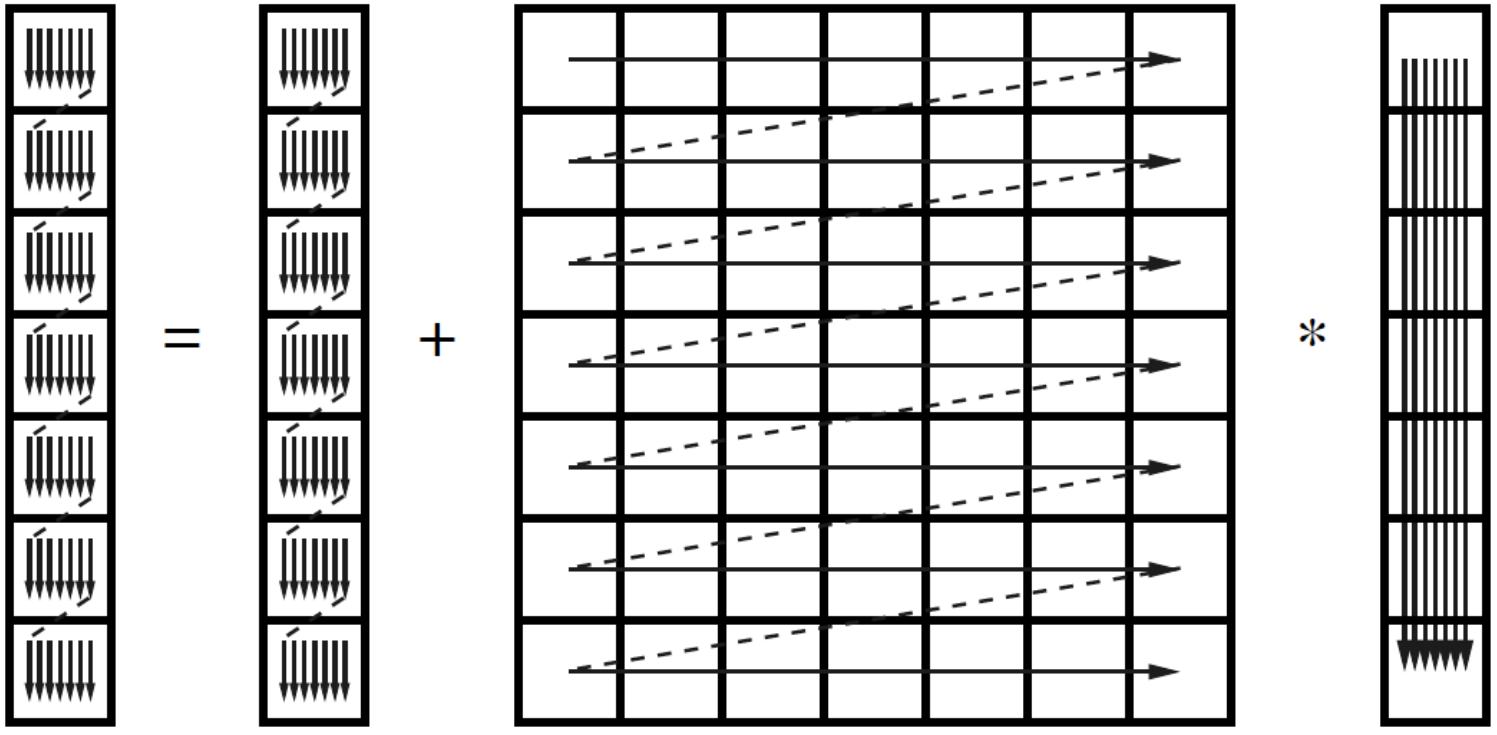

**Figure 3.11:** Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded $N$ times.

# Loop *unroll* and *jam*

*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

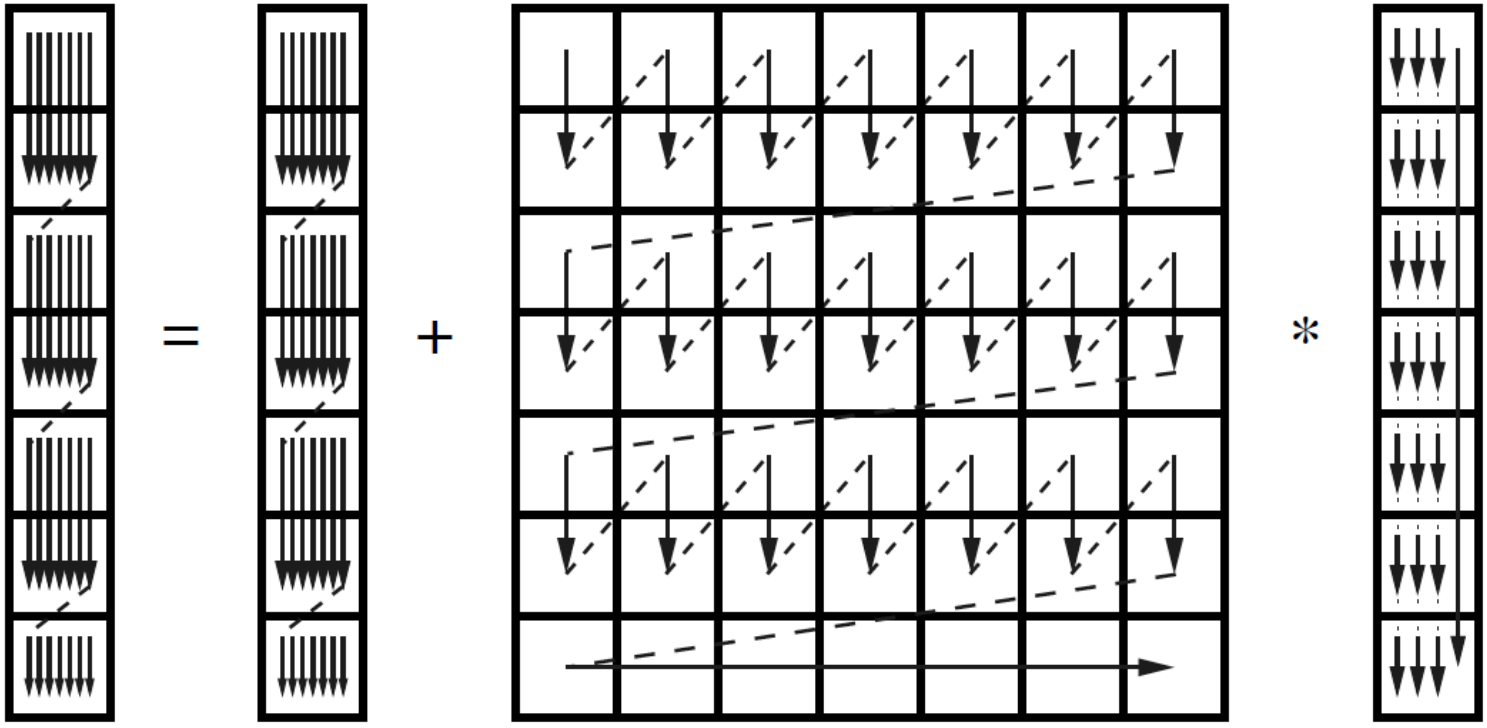

**Figure 3.12:** Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.