# Chapter 3
Data Access Optimization

Daniel Schicker

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

## Overview / Structure

## Limiting Performance Factor : Data Access

- Imbalance of memory bandwidth and peak arithmetic performance
- „In terms of bandwidth, slowest data paths are three to four orders of magnitude away, and eight in terms of latency“P64
- Capacity miss: cache line was accessed before but is not in cache anymore (limited cache capacity)Alex Breuer,Parallel Computing 1 05_data_locality_dense_linear_algebra_23_12_21.pdf slide 10
- Goals:
  - Reduce use of slow data paths
  - Make data transfer as efficient as posible and reduce Capacity misses

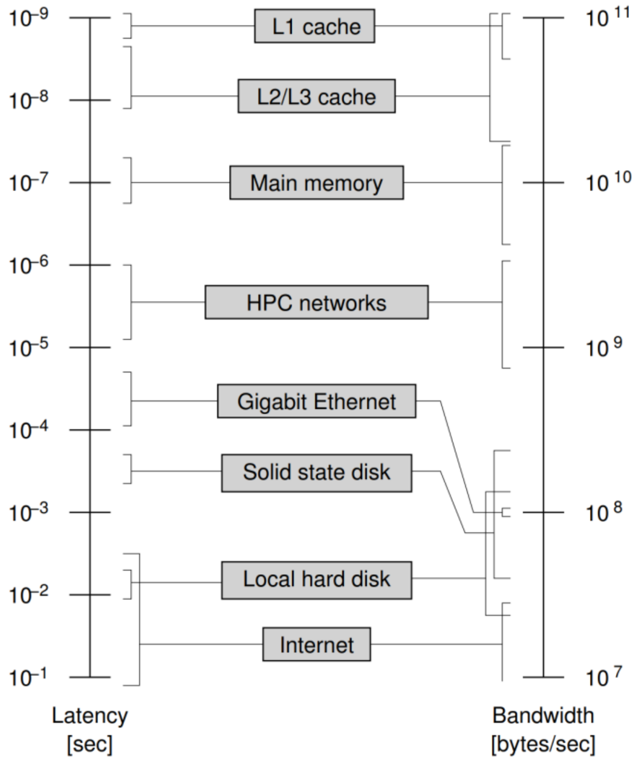


**Figure 3.1:** Typical latency and bandwidth numbers for data transfer to and from different devices in computer systems. Registers have been omitted because their “bandwidth” usually matches the computational capabilities of the compute core, and their latency is part of the pipelined execution.

Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P64.

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

## Limiting Performance Factor : Data Access

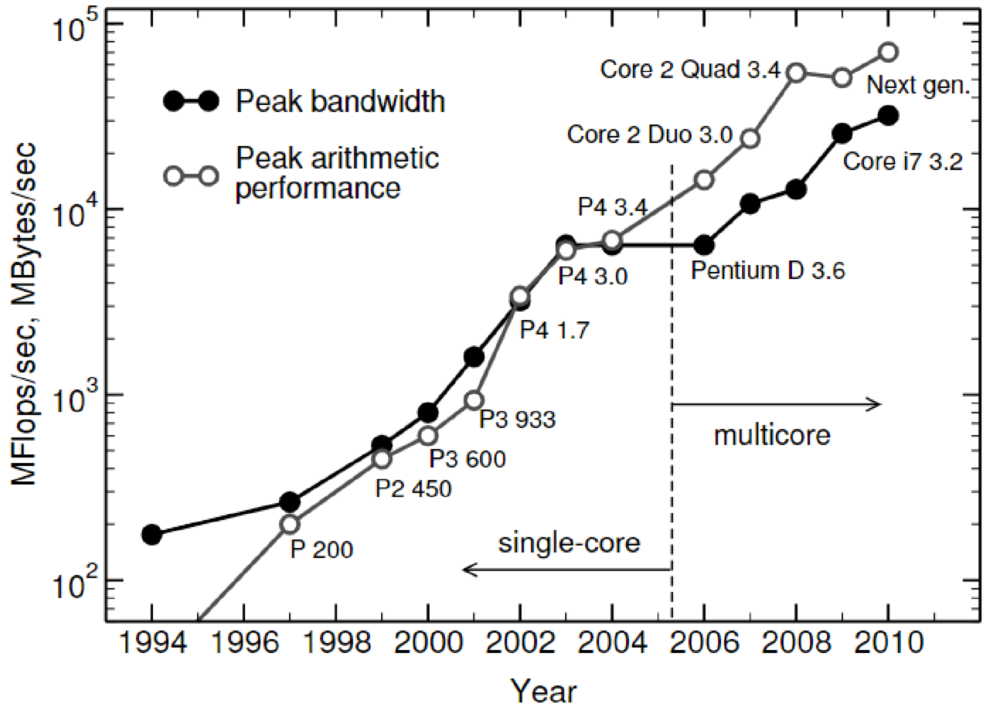


**Figure 3.2:** Progress of maximum arithmetic performance (open circles) and peak theoretical memory bandwidth (filled circles) for Intel processors since 1994. The fastest processor in terms of clock frequency is shown for each year. (Data collected by Jan Treibig.)

Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P64.

## 3.1.1 Bandwidth-Based Perfomance Modeling

Diffrent rules of thumb to:

- Predict performance
- Determine whether program is optimized
- Determine ratio of traffic,FLOP/s,bandwidth…

Machine balance $B_m$ Example:

- Dual core
- clock frequency: 3.0 GHz
- Memory bandwith: 10.6 Gbytes/s
- 4 Flops per core per cycle
- Remember: „Word“ = 8 Byte

$$B_m = \frac{b_{max}}{P_{max}} = \frac{\text{peak memory bandwidth}}{\text{peak arithmetic performance}}$$

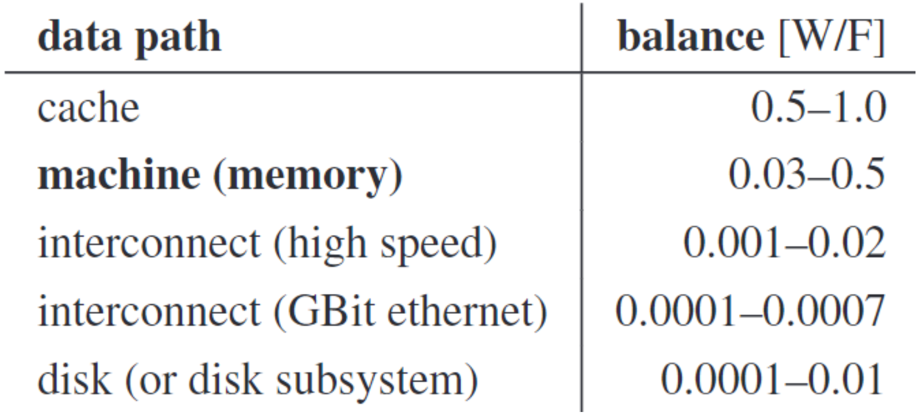
$$b_{max} = \frac{10.6 \text{ Gbytes/s}}{8 \text{ Byte}} = 1.325 \frac{\text{GWords}}{\text{s}}$$

$$P_{max} = 2 * 3.0 \text{ GHz} * 4 = 24 \text{ GHz}$$

$$\frac{b_{max}}{P_{max}} = \frac{1.325}{24} \approx 0.055 = B_m$$

# 3.1.1 Bandwidth-Based Perfomance Modeling

## Typical $B_m$ Values at that time:



| 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.

Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P65.

## 3.1.1 Bandwidth-Based Perfomance Modeling

$B_m$ of the i7-9700K:

- Cores : 8
- Frequency : 3.60 GHz
- Max. bandwidth : 41.6 GByte/s
- Instruction Set : AVX2-256 Bit

- $\frac{41.6 \text{ Gbytes/s}}{8 \text{ Byte}} = 5.2 \text{ GWords/s}$
- $8 \times 3.6 \text{ GHz} \times 16 = 460.8 \text{ GHz}$
- $\frac{5.2 \text{ GWords/s}}{460.8 \text{ GHz}} \approx \mathbf{0.011}$

https://www.intel.de/content/www/de/de/products/sku/186604/intel-core-i79700k-processor-12m-cache-up-to-4-90-ghz/specifications.html

$B_m$ of the i9-14900KS:

- Cores : 24
- Frequency : 4.5 GHz
- Max. bandwidth : 89.6 GByte/s
- Instruction Set : AVX2-256 Bit

- $\frac{89.6 \text{ Gbytes/s}}{8 \text{ Byte}} = 11.2 \text{ GWords/s}$
- $24 \times 4.5 \text{ GHz} \times 16 = 1728 \text{ GHz}$
- $\mathrm{B_m} = \frac{11.2 \text{ GWords/s}}{1728 \text{ GHz}} \approx \mathbf{0.0065}$

https://www.intel.de/content/www/de/de/products/sku/237504/intel-core-i9-processor-14900ks-36m-cache-up-to-6-20-ghz/specifications.html

## 3.1.1 Bandwidth-Based Perfomance Modeling

- $B_c$ : Code balance computes amount of words per Flop

$$B_c = \frac{\text{total data traffic [Words]}}{\text{total floating point ops [FLops]}}$$

- l : expected maximum fraction of peak performance

$$l = \min(1, \frac{B_m}{B_c})$$

- P : maximal performance in GFlops/s

$$P = l * \text{Pmax} = \min(\text{Pmax}, \frac{b_{max}}{B_c})$$

Example:

```
do i = 1 , N
    A(i) = B(i) + C(i) * D(i);
enddo
```

$$B_c = \frac{3+1}{2} = 2$$

$$l = \min\left(1, \frac{0.1}{2}\right) = 0.05 \triangleq 5\ \%$$

No Write Allocates!

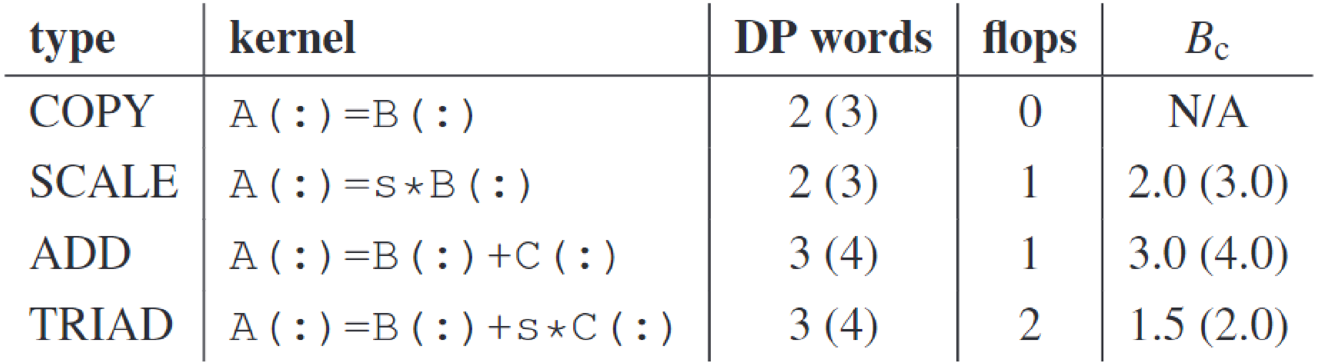
- $B_m = 0.1$

$$P = 0.05 * \text{Pmax}$$

## Assumptions for Balance analysis

- Loop code optimally utilizes all arithmetic units
- Loop code uses floating point arithmetic
- Data transfer and arithmetic overlap perfectly
- Slowest data path determines performance & faster data paths are infinitely fast
- Latency effects are ignored
- Bandwidth is completely saturated

## 3.1.2 The STREAM Benchmarks



| 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) |

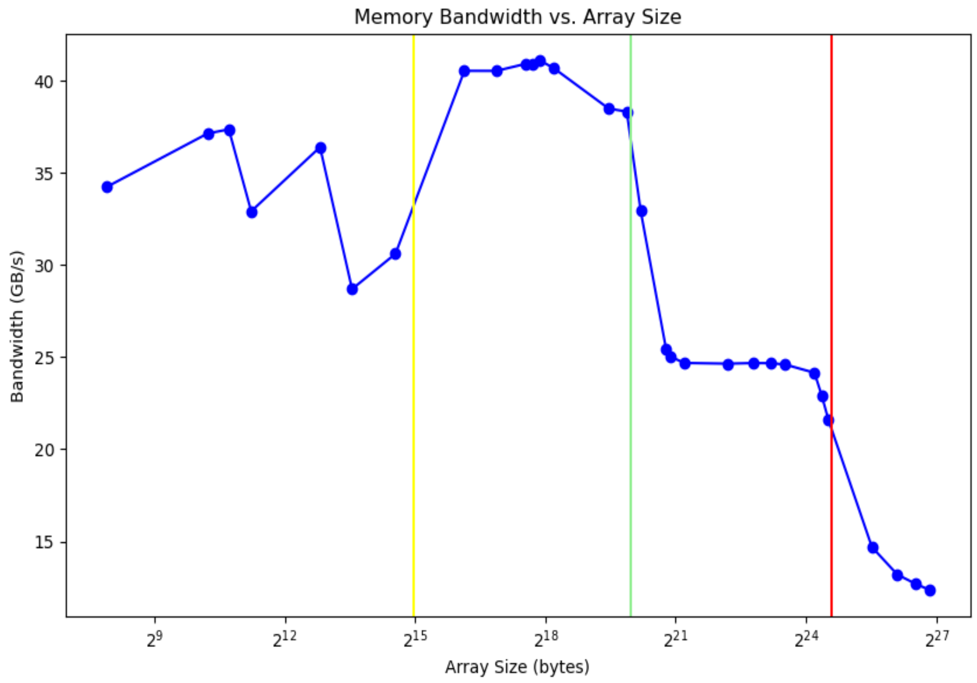
**Table 3.2:** The STREAM benchmark kernels with their respective data transfer volumes (third column) and floating-point operations (fourth column) per iteration. Numbers in brackets take write allocates into account.

Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P67.

## 3.1.2 The STREAM Benchmarks

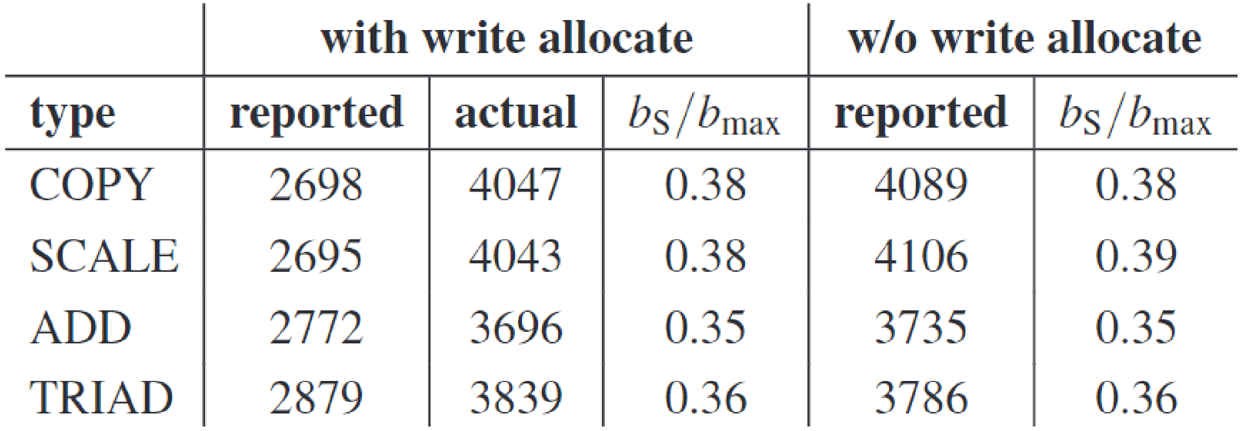
Common factors preventing the STREAM Benchmark from achieving the performance level predicted by balance analysis:

- Peak bandwidth is not accessible in both directions (load/store)
- Large Latencies, Overhead and hardware issues
- Imbalance of read and write lead to underuse of unidirectional data paths



## 3.1.2 The STREAM Benchmarks

- Use $b_S$ instead of $b_{max}$:
  $P = \min\left(P_{max}, \frac{b_S}{B_c}\right)$
- 40% of peak memory bandwidth is achiveable



| | with write allocate | | | w/o write allocate | |
|---|---|---|---|---|---|
| type | reported | actual | $b_S/b_{max}$ | reported | $b_S/b_{max}$ |
| COPY | 2698 | 4047 | 0.38 | 4089 | 0.38 |
| SCALE | 2695 | 4043 | 0.38 | 4106 | 0.39 |
| ADD | 2772 | 3696 | 0.35 | 3735 | 0.35 |
| TRIAD | 2879 | 3839 | 0.36 | 3786 | 0.36 |

**Table 3.3:** Single-thread STREAM bandwidth results in GBytes/sec for an Intel Xeon 5160 processor (see text for details), comparing versions with and without write allocate. Write allocate was avoided by using nontemporal store instructions.

Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P69.

## 3.2 Storage Order

- Both code snippets execude the same task but have diffrent memory access patterns
- Stride-N access reduces spatial locality and lower bandwidth
- The inner loop variable, which serves as an index, should guarantee stride-one access

Fortran

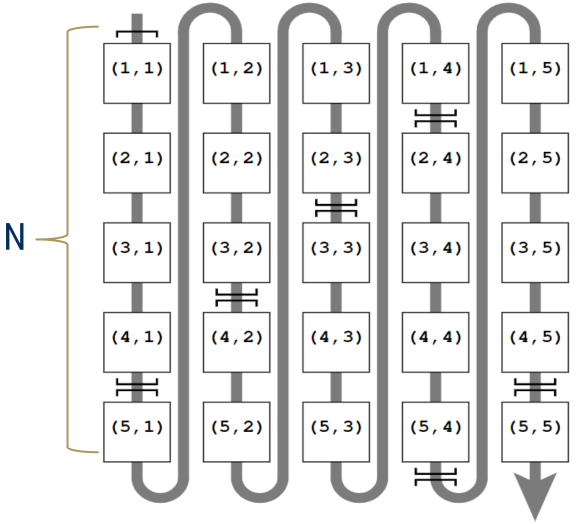
Stride-N access

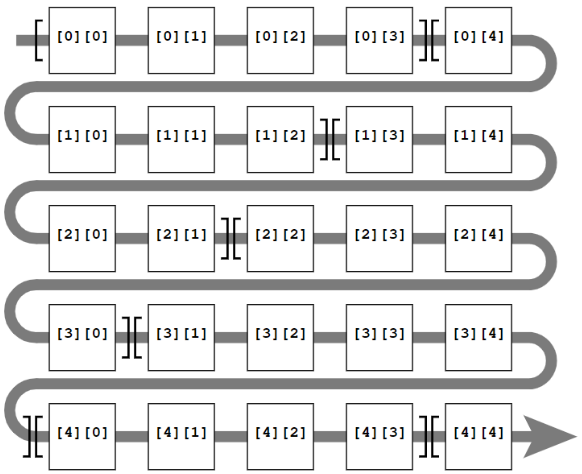
```
do i=1,N
  do j=1,N
    A(i,j) = i*j
  enddo
enddo
```

C

Stride-1 access

```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = i*j;
  }
}
```





Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P70.

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

## 3.3 Case Study: The Jacobi Algorithm

- „A stencil describes the dependency of an output grid point w.r.t. its localneighborhood in the input“

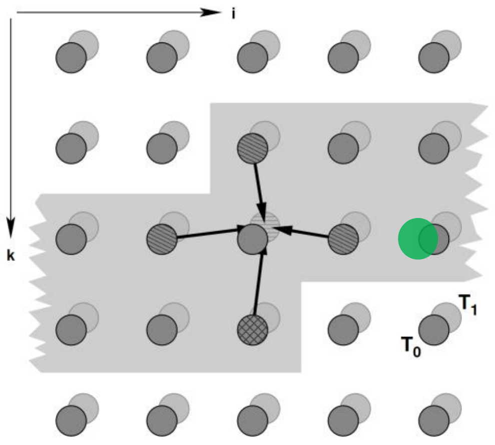
  Source:Alex Breuer.Parallel Computing 1. 04_stencils_23_12_05.pdf slide 12

- Only three loads are needed because phi(i+1, k, t0) is used again two iterations later

- If the cache can hold more than two rows only one store and one load for phi(i, k+1 , t0) are necessary
  - $B_c = \frac{1+1}{4} = 0.5$ W/F

- If the i-dimension is increased, the code balance will rise again

```
double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
integer :: t0,t1
t0 = 0 ; t1 = 1
do it = 1,itmax    ! choose suitable number of sweeps
  do k = 1,kmax
    do i = 1,imax
      ! four flops, one store, four loads
      phi(i,k,t1) = ( phi(i+1,k,t0) + phi(i-1,k,t0)
                    + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
    enddo
  enddo
  ! swap arrays
  i = t0 ; t0=t1 ; t1=i
enddo
```



Source for Figure 3.5 and code snippet: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P72.

**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).

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

## 3.3 Case Study: The Jacobi Algorithm

- MLUPS/sec = lattice side updates per second
- By using the STREAM results of the Xeon 5160 processor and the performance model, one can derive a maximal performance of 675 MFlops/s

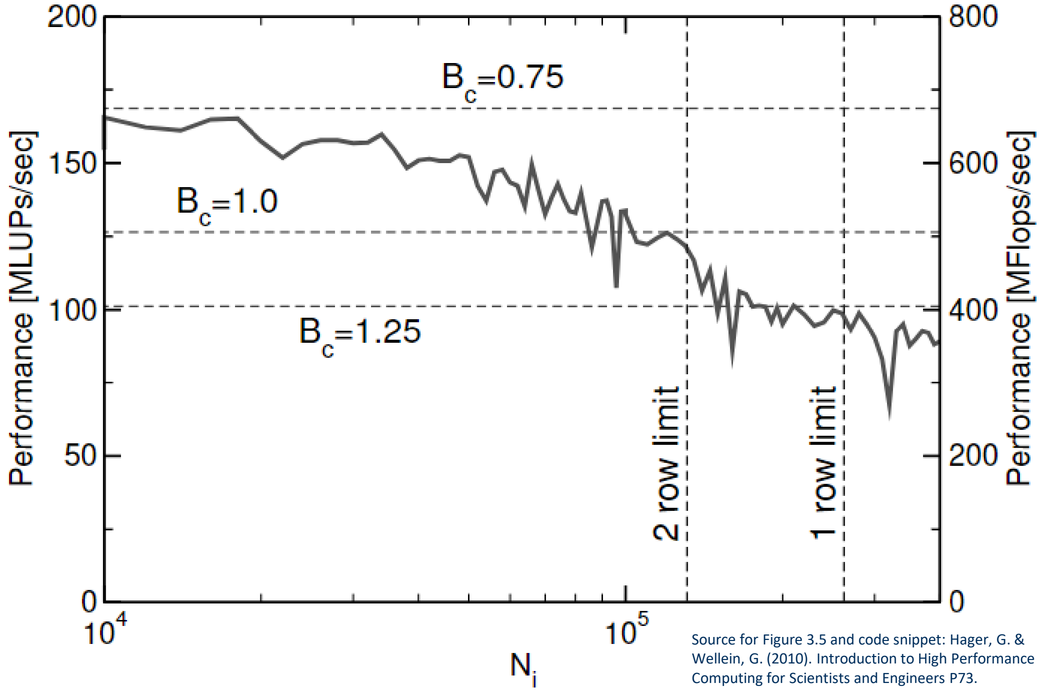


**Figure 3.6:** Performance versus inner loop length (lattice extension in the i direction) for the Jacobi algorithm on one core of a Xeon 5160 processor (see text for details). Horizontal lines indicate predictions based on STREAM bandwidth.

Source for Figure 3.5 and code snippet: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P73.

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

# 3.4 Case study: Dense matrix transpose

- We assume column major order
- C ≙Cache Size;
- $L_c$ ≙ cache line size
- Case one: $2 \times N^2 \leq C$
- Case two: $N * Lc \leq C$
- Case three: $N * Lc \gtrsim C$
  - No spatial locality and a drop in performance is expected

```
do i=1,N
  do j=1,N
    A(i,j) = B(j,i)
  enddo
enddo
```

Source for Figure 3.5 and code snippet: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P74 &75.

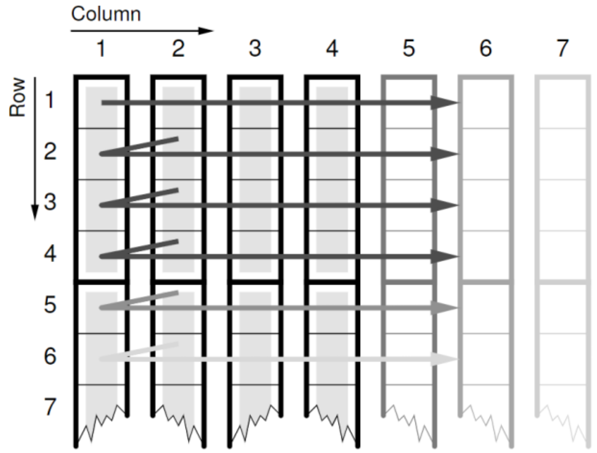


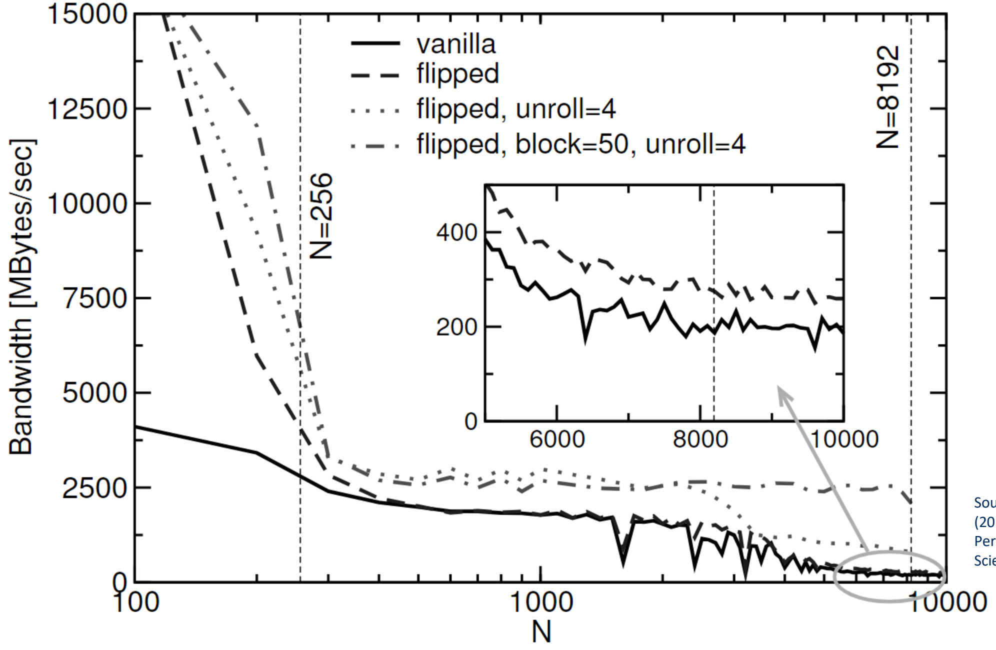
**Figure 3.7:** Cache line traversal for vanilla matrix transpose (strided store stream, column major order). If the leading matrix dimension is a multiple of the cache line size, each column starts on a line boundary.

## 3.4 Case study: Dense matrix transpose

- Additional limiting factors include the fact that the L1 cache operates in a write-through mode, and if N is large enough, every access in the strided stream results in a TLB miss

- Cache thrashing

76 *Introduction to High Performance Computing for Scientists and Engineers*



Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P77.

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

## 3.4 Case study: Dense matrix transpose

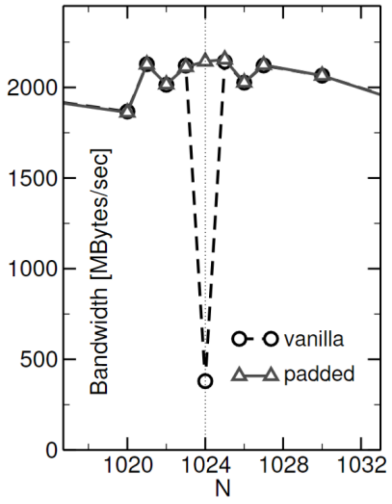


**Figure 3.9:** Cache thrashing for an unfavorable choice of array dimensions (dashed): Matrix transpose performance breaks down dramatically at a dimension of 1024×1024. Padding by enlarging the leading dimension by one removes thrashing completely (solid).

Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P77.

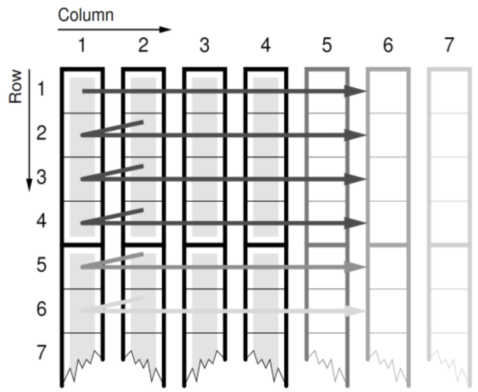


**Figure 3.7:** Cache line traversal for vanilla matrix transpose (strided store stream, column major order). If the leading matrix dimension is a multiple of the cache line size, each column starts on a line boundary.

Source : Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P75.

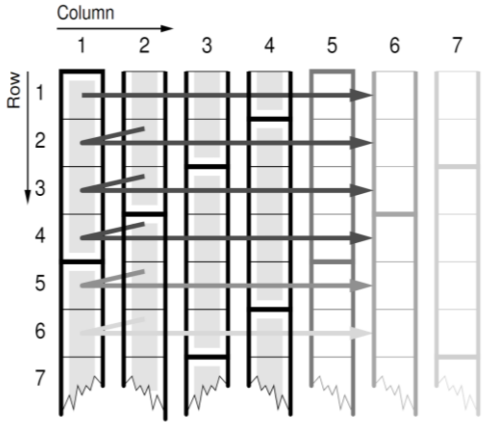
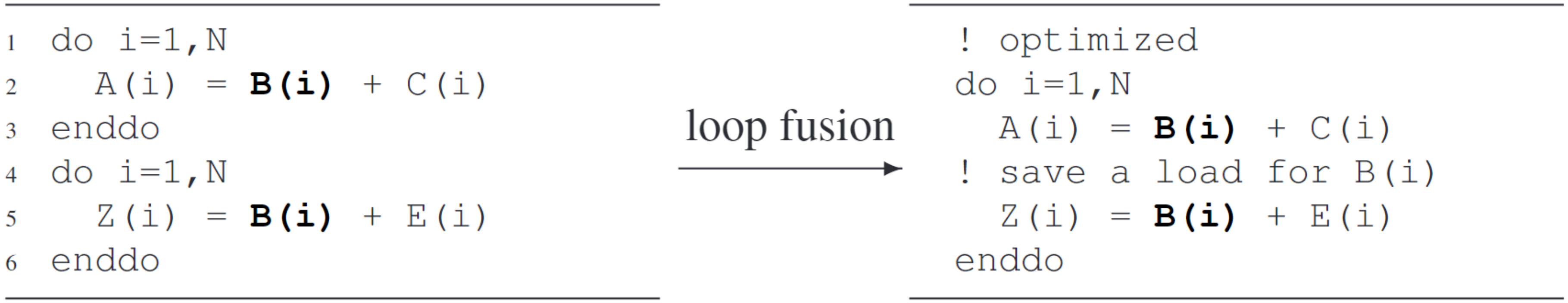


**Figure 3.10:** Cache line traversal for padded matrix transpose. Padding may increase effective cache size by alleviating associativity conflicts.

Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P78.

## 3.5 Algorithm Classification and Access Optimizations O(N)/O(N)



```
do i=1,N
  A(i) = B(i) + C(i)
enddo
do i=1,N
  Z(i) = B(i) + E(i)
enddo
```

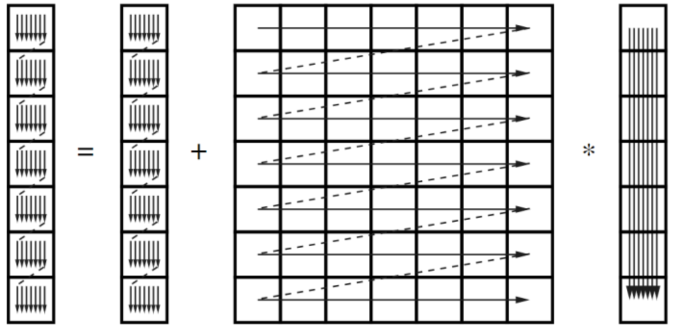
```
! optimized
do i=1,N
  A(i) = B(i) + C(i)
! save a load for B(i)
  Z(i) = B(i) + E(i)
enddo
```

Source : Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P79.

# 3.5 Algorithm Classification and Access Optimizations O(N²)/O(N²)

Total memory traffic: $N^2 + N^2 + N$

```
do i=1,N
  tmp = C(i)
  do j=1,N
                      x  y
    tmp = tmp + A(j,i) * B(j)
  enddo
  C(i) = tmp
enddo
```



loop unrolling $\xrightarrow{\quad}$

Total memory traffic: $N^2 + \frac{N^2}{m} + N$

```
! remainder loop ignored
do i=1,N,m
  do j=1,N
    C(i) = C(i) + A(j,i) * B(j)
    C(i+1) = C(i+1) + A(j,i+1) * B(j)
    ! m times
    ...
    C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
  enddo
enddo
```

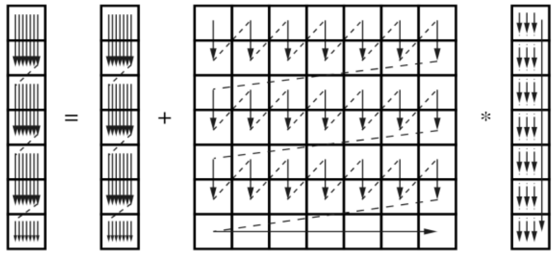


**Figure 3.12:** Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P80,P81,P82.

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

## 3.5 Algorithm Classification and Access Optimizations O(N²)/O(N²)

```
do j=1,N,m
  do i=1,N
    A(i,j)     = B(j,i)
    A(i,j+1)   = B(j+1,i)
    ...
    A(i,j+m-1) = B(j+m-1,i)
  enddo
enddo
```

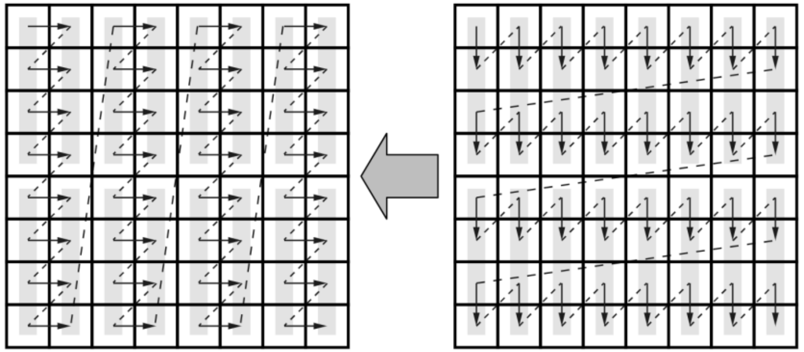
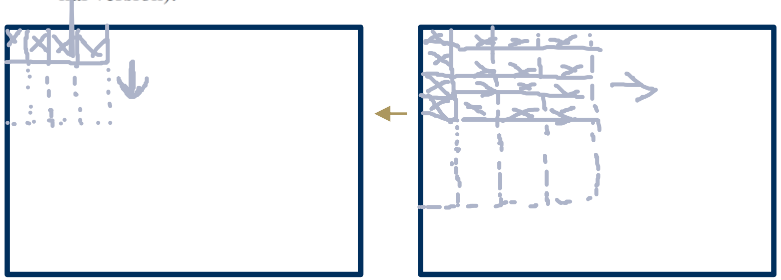


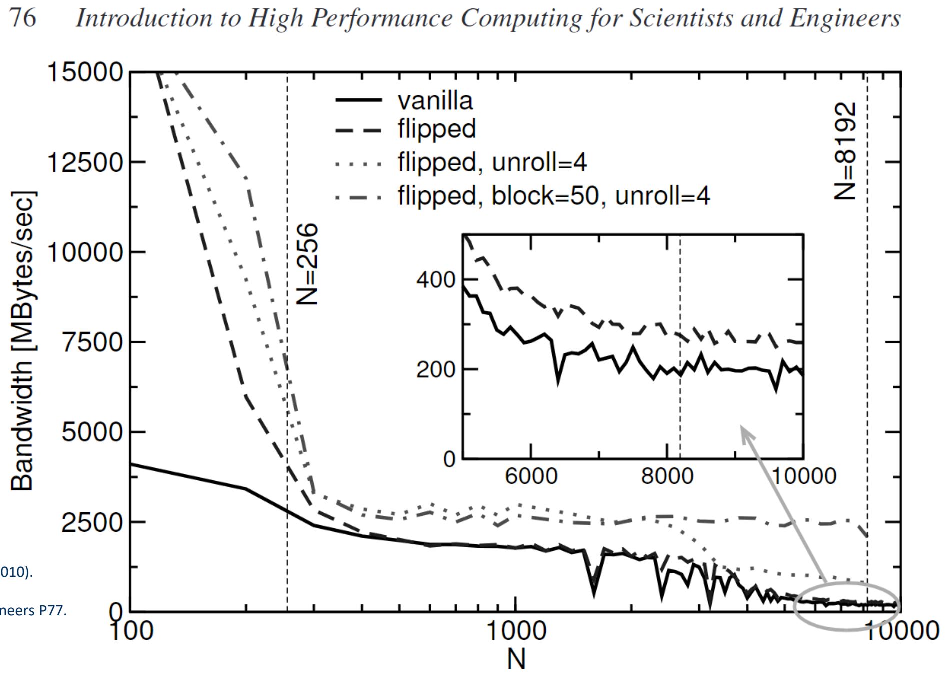
**Figure 3.13:** Two-way unrolled "flipped" matrix transpose (i.e., with strided load in the original version).



Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P80,P81,P82.

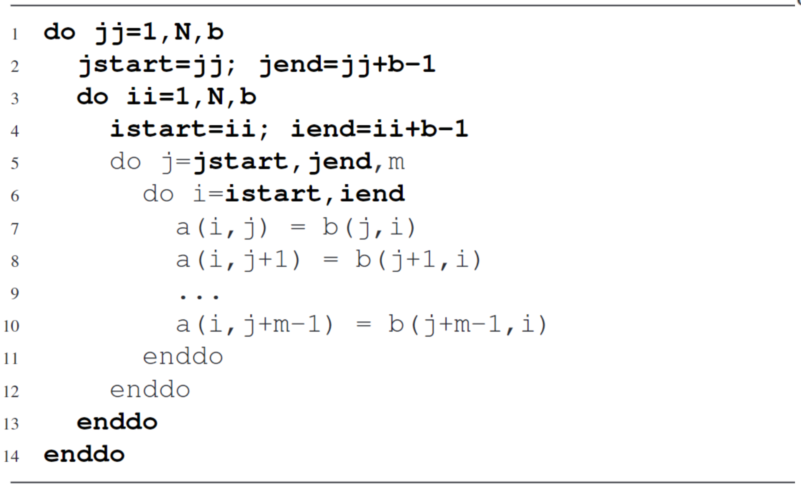
## 3.5 Algorithm Classification and Access Optimizations O(N²)/O(N²)

- Use the remaining Lc – m elements of a cache line as well and hopefully never touch the cache line again



Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P77.

# 3.5 Algorithm Classification and Access Optimizations O(N²)/O(N²)



```
do jj=1,N,b
  jstart=jj; jend=jj+b-1
  do ii=1,N,b
    istart=ii; iend=ii+b-1
    do j=jstart,jend,m
      do i=istart,iend
        a(i,j) = b(j,i)
        a(i,j+1) = b(j+1,i)
        ...
        a(i,j+m-1) = b(j+m-1,i)
      enddo
    enddo
  enddo
enddo
```

Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers 83.

84 *Introduction to High Performance Computing for Scientists and Engineers*

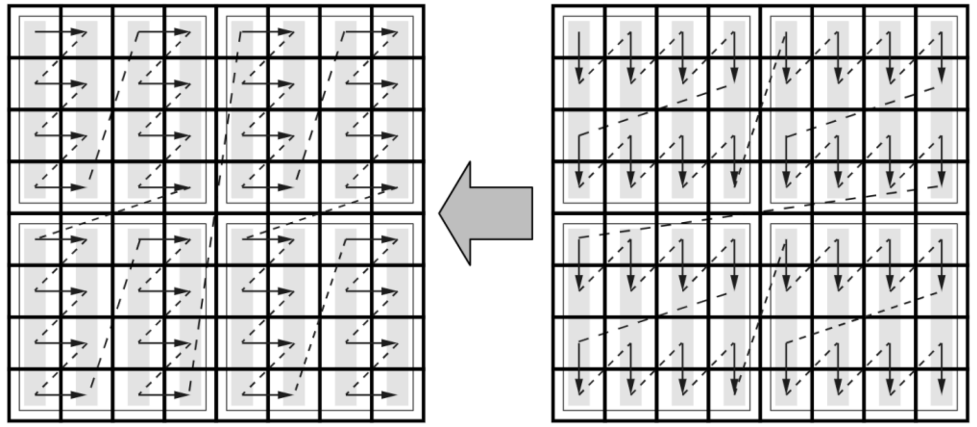
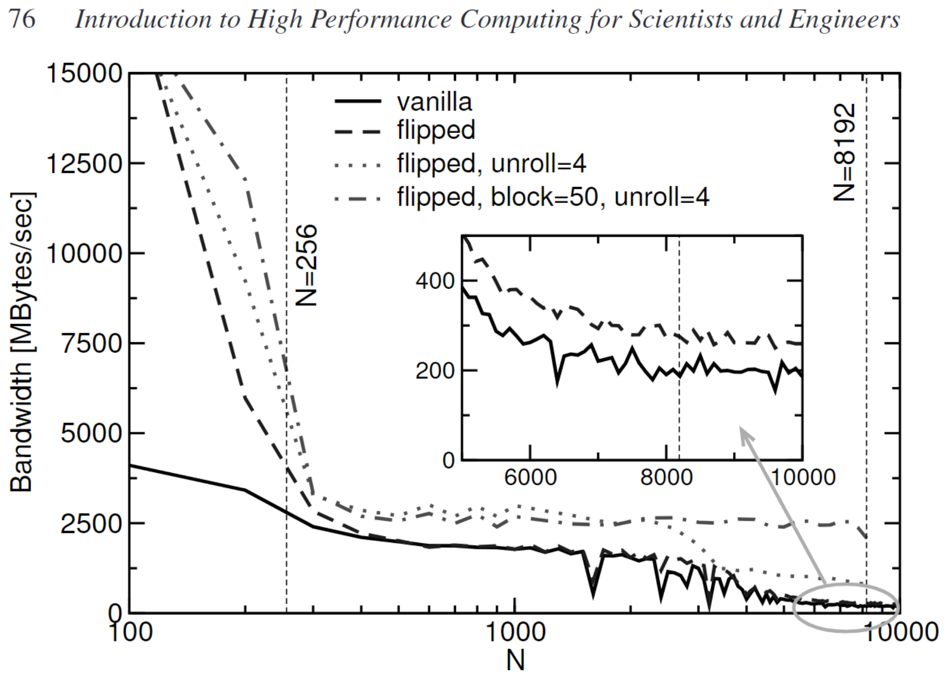


**Figure 3.14:** 4 × 4 blocked and two-way unrolled "flipped" matrix transpose.

Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P84.

# 3.5 Algorithm Classification and Access Optimizations O(N²)/O(N²)



Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P77.

# 3.6 Case Study: Sparse Matrix-Vector Multiply CRS (Compressed Row Storage)

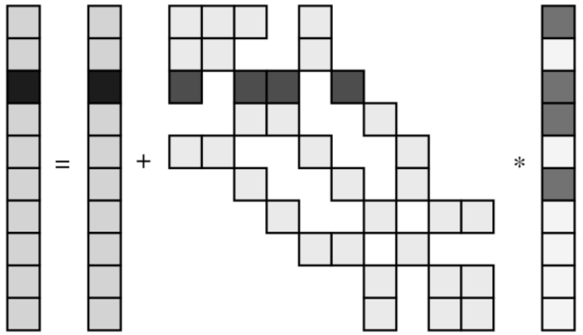


**Figure 3.15:** Sparse matrix-vector multiply. Dark elements visualize entries involved in updating a single LHS element. Unless the sparse matrix rows have no gaps between the first and last nonzero elements, some indirect addressing of the RHS vector is inevitable.

Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P86.

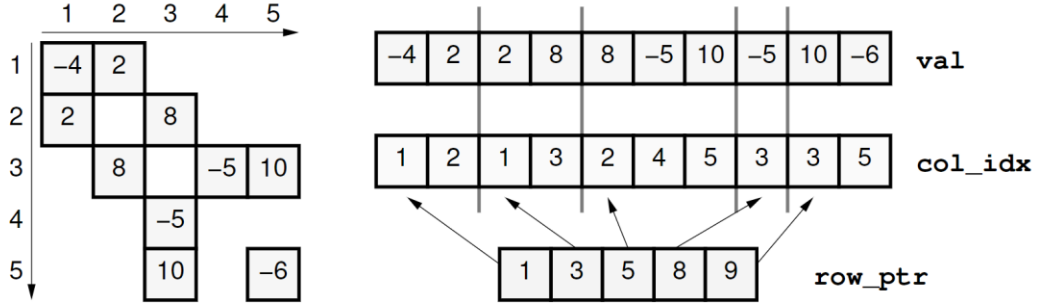
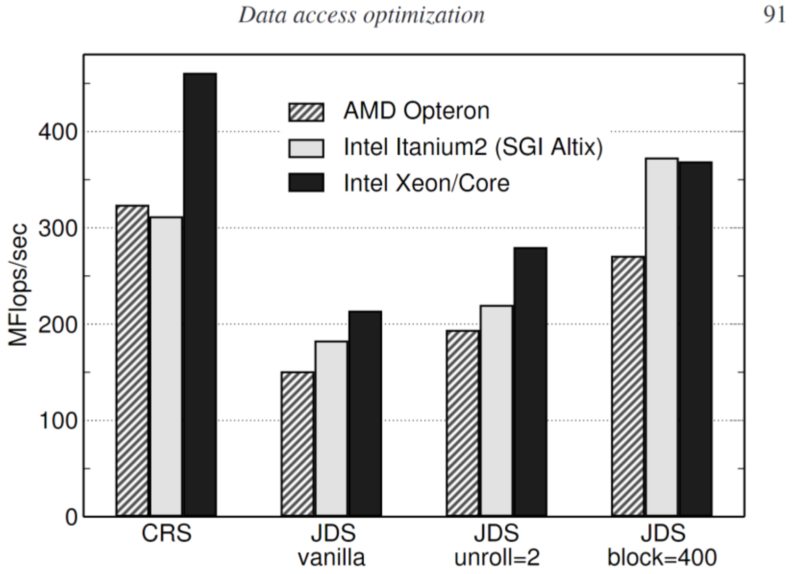


**Figure 3.16:** CRS sparse matrix storage format.

```
do i = 1,$N_r$
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

Source fpr code snippet and Figure 3.16: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P87.

## 3.6 Case Study: Sparse Matrix-Vector Multiply CRS (Compressed Row Storage)



Source: Hager, G. & Wellein, G. (2010). Introduction to High Performance Computing for Scientists and Engineers P91.

## Vielen Dank für Ihre Aufmerksamkeit!

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA