High Performance Computing

Sequential code optimization by example

G. Hager, G. Wellein
Regionales Rechenzentrum Erlangen

W. u. E. Heraeus Summerschool
on Computational Many Particle Physics
Sep 18-29, Greifswald, Germany

A warning

“Premature optimization is the root of all evil.”

Donald E. Knuth

Outline

- Warm-up example: Monte Carlo spin simulation
  - „Common sense“ optimizations
    - Strength reduction by tabulation
    - Reducing the memory footprint
- General remarks on algorithms and data access
- Example: Matrix transpose
  - Data access analysis
  - Cache thrashing
  - Optimization by padding and blocking
- Example: Sparse matrix-vector multiplication
  - Sparse matrix formats: CRS and JDS
  - Optimizing data access for sparse MVM
  - Strengths and weaknesses of the two formats

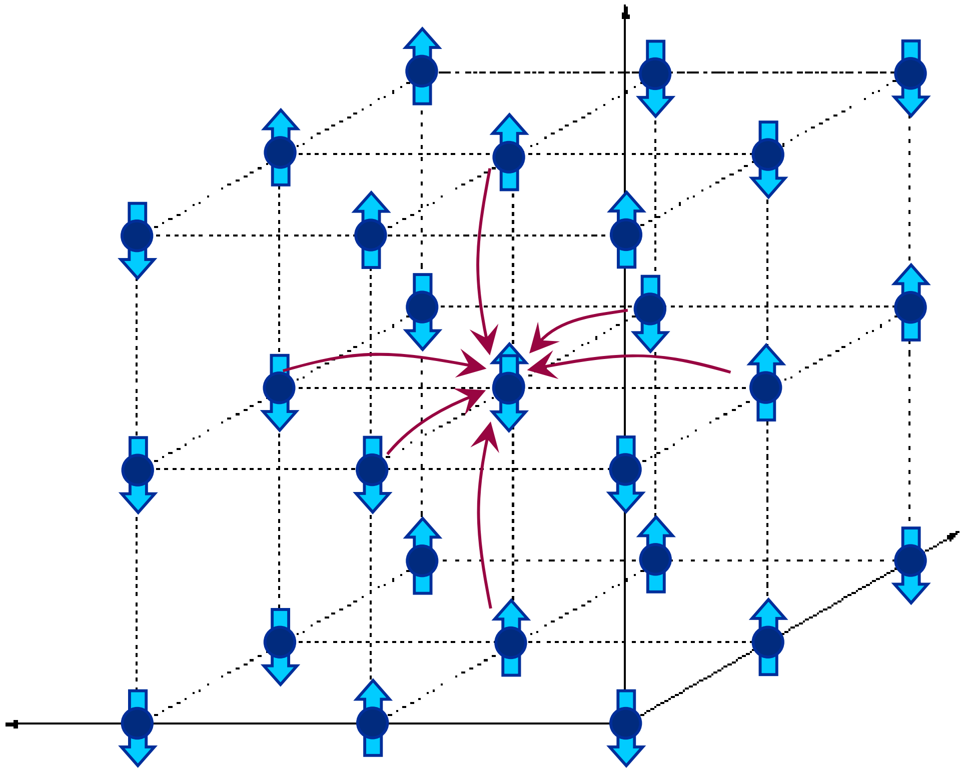
„Common sense“ optimizations:
A Monte Carlo spin code

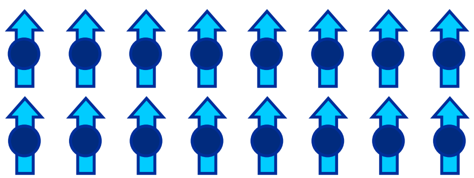
Optimization of a Spin System Simulation: Model

- **3-D cubic lattice**
- **One variable („spin“) per grid point with values**

  +1 or -1

- **Next-neighbour interaction terms**
- **Code chooses spins randomly and flips them as required by MC algorithm**





Optimization of a Spin System Simulation: Model

- **Systems under consideration**
  - **50·50·50=125000 lattice sites**
  - **2^{125000} different configurations**
  - **Computer time: 2^{125000} · 1 ns ≈ 10^{37000} years – without MC ☺**

- **Memory requirement of original program ≈ 1 MByte**

Optimization of a Spin System Simulation: Original Code

- **Program Kernel:**

```
      IA=IZ(KL,KM,KN)
      IL=IZ(KLL,KM,KN)
      IR=IZ(KLR,KM,KN)
      IO=IZ(KL,KMO,KN)
      IU=IZ(KL,KMU,KN)
      IS=IZ(KL,KM,KNS)
      IN=IZ(KL,KM,KNN)

      edelz=iL+iR+iU+iO+iS+iN

C     CRITERION FOR FLIPPING THE SPIN

      BF= 0.5d0*(1.d0+tanh(edelz/tt))
      IF(YHE.LE.BF) then
      iz(kl,km,kn)=1
      else
      iz(kl,km,kn)=-1
      endif
```

Load neighbors of a random spin

calculate magnetic field

decide about spin orientation

Optimization of a Spin System Simulation: Code Analysis

- Profiling shows that
  - 30% of computing time is spent in the `tanh` function
  - Rest is spent in the line calculating `edelz`
- Why?
  - `tanh` is expensive by itself (see previous talk)
  - Compiler fuses spin loads and calculation of `edelz` into a single line
- What can we do?
  - Try to reduce the „strength“ of calculations (here `tanh`)
  - Try to make the CPU move less data
- How do we do it?
  - Observation: argument of `tanh` is always integer in the range -6..6 (`tt` is always 1)
  - Observation: Spin variables only hold values +1 or -1

Optimization of a Spin System Simulation: Making it Faster

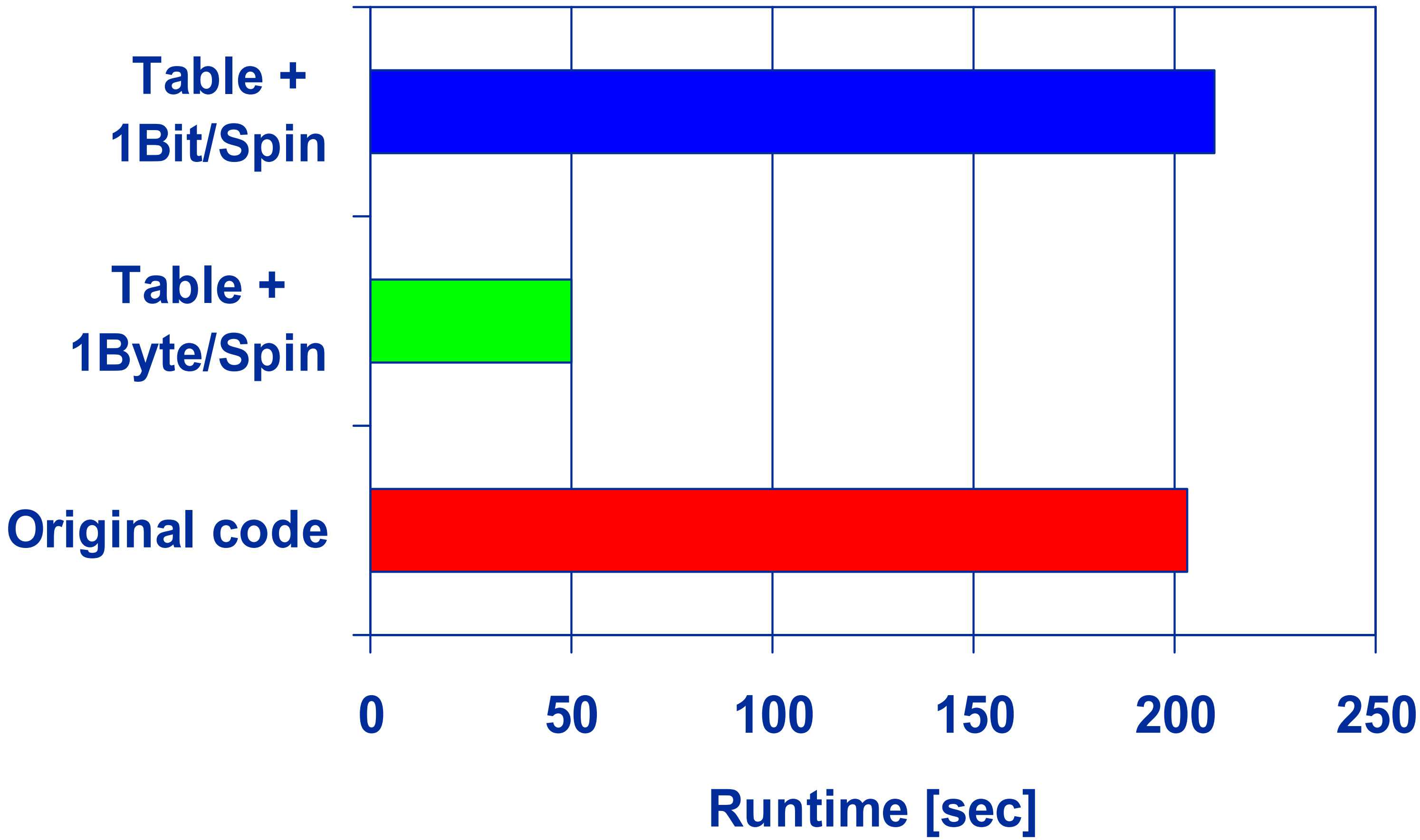
- Strength reduction by tabulation of `tanh` function

```
BF= 0.5d0*(1.d0+tanh_table(edelz))
```

  - Performance increases by 30% as table lookup is done with „lightspeed“ compared to `tanh` calculation
- By declaring spin variables with `INTEGER*1` instead of `INTEGER*4` the memory requirement is reduced to about ¼
  - Better cache reuse
  - Factor 2–4 in performance depending on platform
  - Why don‘t we use just one bit per spin?
    - Bit operations (mask, shift, add) too expensive → no benefit
- Potential for a variety of data access optimizations
  - But: choice of spin must be absolutely random!

Optimization of a Spin System Simulation: Performance Results

- **Pentium 4 (2.4 GHz)**



General remarks on algorithms and data access

Data access

- Data access is the most frequent performance-limiting factor in HPC
- Cache-based microprocessors feature small, fast caches and large, slow memory
  - “Memory Wall”, “DRAM Gap”
  - Latency can be hidden under certain conditions (prefetch, software pipelining)
  - Bandwidth limit cannot be circumvented
    - Instead, modify the code to avoid the slow data paths
- General guideline: examine “traffic-to-work” ratio (balance) of algorithm to get a hint at possible limitations
  - Examination of performance-critical loops is vital
  - Important metric: (“LOADs/STOREs to FLOPs”)
  - Optimization: lower LDST/FLOP ratio
- … and always remember that stride-1 access is best!

“Lightspeed” estimates

- How do you know that your code makes good use of the resources?
- In many cases one can estimate the possible performance limit (lightspeed) of a loop
- Architectural boundary conditions:

Memory bandwidth	GWords/s (1 W = 8 bytes)
Floating point peak performance	GFlops/s

Machine balance

$$B_m = \frac{\text{bandwidth}\,[words/s]}{\text{FP performance}\,[flops/s]}$$

- Typical values (memory): 0.13 W/F (Itanium2 1.5 GHz)
  0.125 W/F (Xeon 3.2 GHz),
  0.5 W/F (NEC SX8)

“Lightspeed” estimates

- **Expected performance on the loop level?**
- **Code balance:**

$$B_c = \frac{\text{data transfer (LD/ST)}\,[\mathit{words}]}{\text{arithmetic operations}\,[\mathit{flops}]}$$

- **Expected fraction of peak performance („lightspeed"):**

$$l = \frac{B_m}{B_c}$$

- **Example: Vector triad A(:)=B(:)+C(:)*D(:) on 3.2 GHz Xeon**

  B_m/B_c = 0.125/2 = 0.0625, i.e. 6.25% of peak performance!

- **Many code optimizations thus aim at lowering B_c**

- **Case 1: O(N)/O(N) Algorithms**
  - O(N) arithmetic operations vs. O(N) data access operations
  - Examples: Scalar product, vector addition, sparse MVM etc.
  - Performance limited by memory bandwidth for large N (“memory bound”)
  - Limited optimization potential for single loops
    - at most constant factor for multi-loop operations
  - Example: successive vector additions

```
do i=1,N
   a(i)=b(i)+c(i)
enddo
```

```
do i=1,N
   z(i)=b(i)+e(i)
enddo
```

$B_c = 3/1$

no optimization potential for either loop

Loop fusion →

$B_c = 5/2$

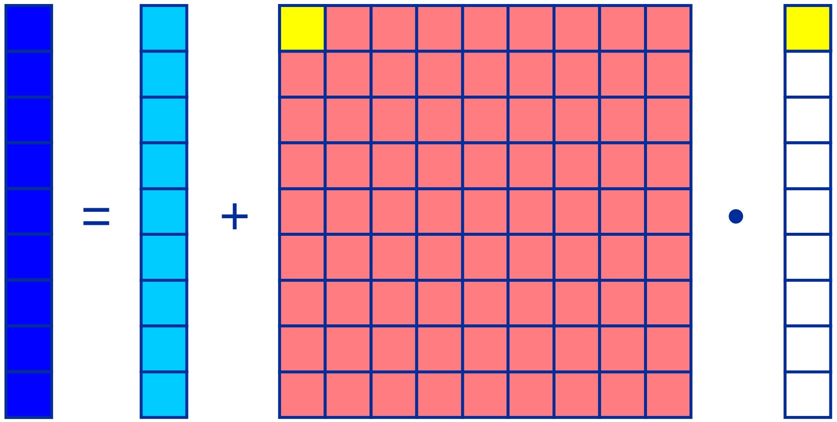
```
do i=1,N
   a(i)=b(i)+c(i)
   z(i)=b(i)+e(i)
enddo
```

fusing different loops allows O(N) data reuse from registers

Data access – general guidelines

- **Case 2: O(N²)/O(N²) algorithms**
  - **Examples: dense matrix-vector multiply, matrix addition, dense matrix transposition etc.**
    - **Nested loops**
  - **Memory bound for large N**
  - **Some optimization potential (at most constant factor)**
    - **Can often enhance LDST/FLOP ratio by outer loop unrolling**
  - **Example: dense matrix-vector multiplication**

```
do i=1,N
 do j=1,N
  c(i)=c(i)+a(i,j)*b(j)
 enddo
enddo
```



Naïve version loads b[] N times!

Data access – general guidelines

- O(N²)/O(N²) algorithms cont'd
  - "Unroll & jam" optimization (or "outer loop unrolling")

```
do i=1,N
 do j=1,N
  c(i)=c(i)+a(i,j)*b(j)
 enddo
enddo
```

unroll

```
do i=1,N,2
 do j=1,N
  c(i)=c(i)+a(i,j)*b(j)
 enddo
 do j=1,N
  c(i+1)=c(i+1)+a(i+1,j)*b(j)
 enddo
enddo
```

jam

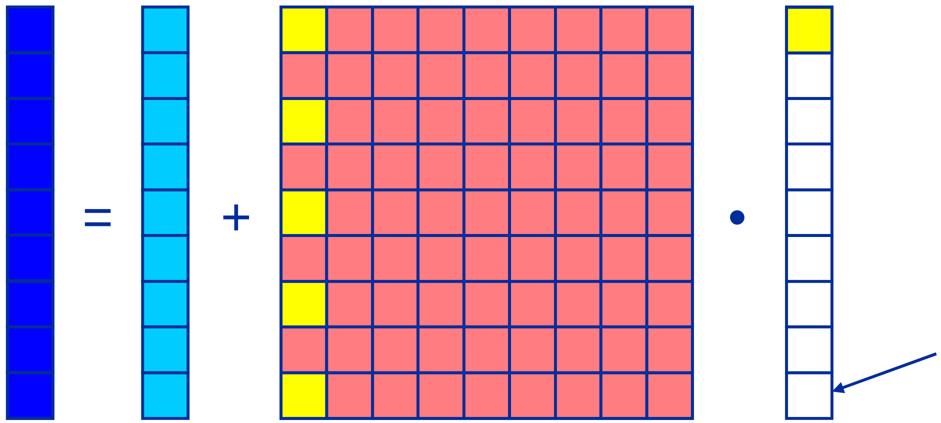
```
do i=1,N,2
 do j=1,N
  c(i)=c(i)+a(i,j)        *b(j)
  c(i+1)=c(i+1)+a(i+1,j)*b(j)
 enddo
enddo
```

b(j) can be reused once from register → save 1 LD operation

Lowered B_c from 1 to 3/4

Data access – general guidelines

- O(N²)/O(N²) algorithms cont'd
  - Data access pattern for 2-way unrolled dense MVM:



Vector b[ ] now only loaded N/2 times!

Remainder loop handled separately

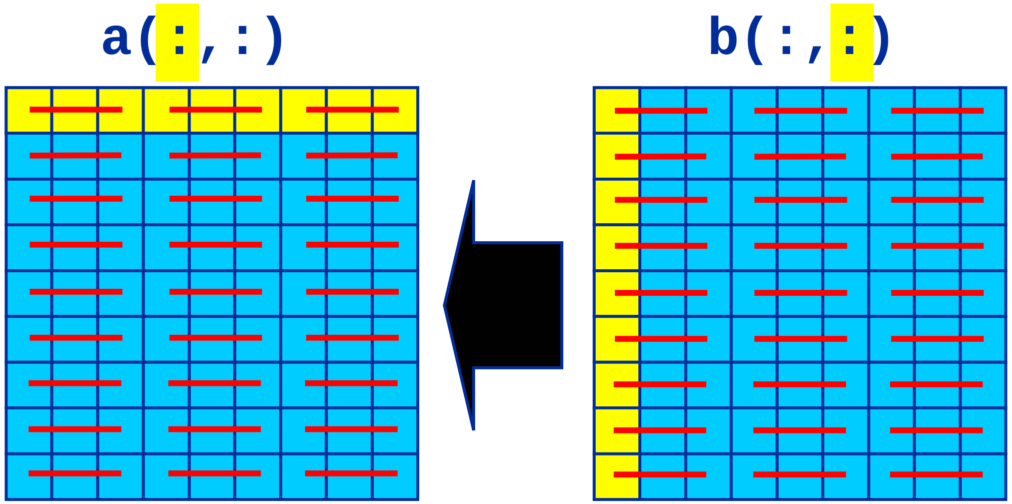
  - Code blance can still be enhanced by more aggressive unrolling (i.e., m-way instead of 2-way)
  - Significant code bloat (try to use compiler directives if possible)
    - Ultimate limit: b[ ] only loaded once from memory ($B_c \approx 1/2$)
    - Beware: CPU registers are a limited resource
    - Excessive unrolling can cause register spills to memory

Optimizing data access for dense matrix transpose

Dense matrix transpose

- Simple example for data access problems in cache-based systems
- Naïve code:

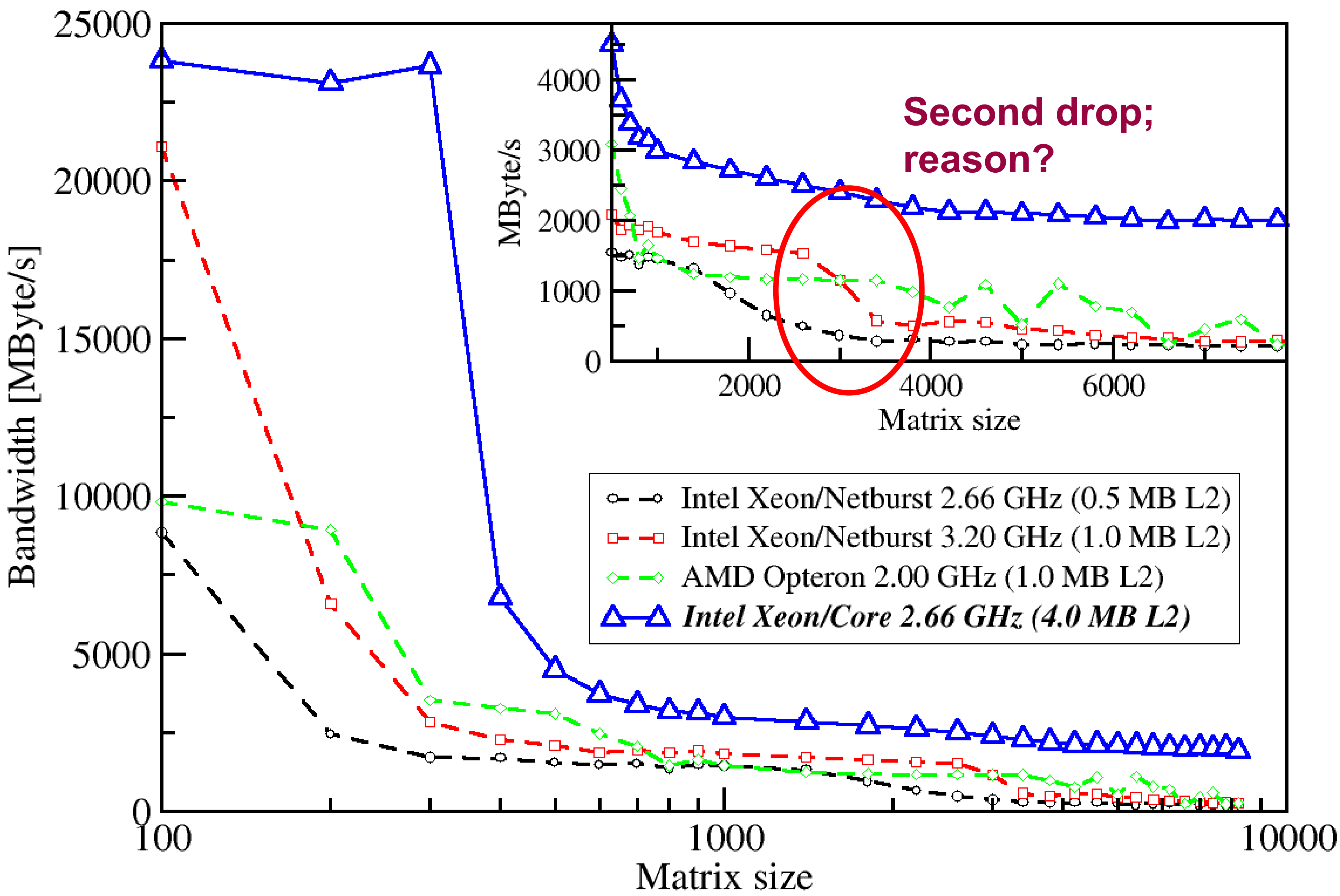
```
do i=1,N
 do j=1,N
  a(j,i) = b(i,j)
 enddo
enddo
```



- Problem: Stride-1 access for a implies stride-N access for b
  - Access to a is perpendicular to cache lines (—)
  - Possibly bad cache efficiency (spatial locality)

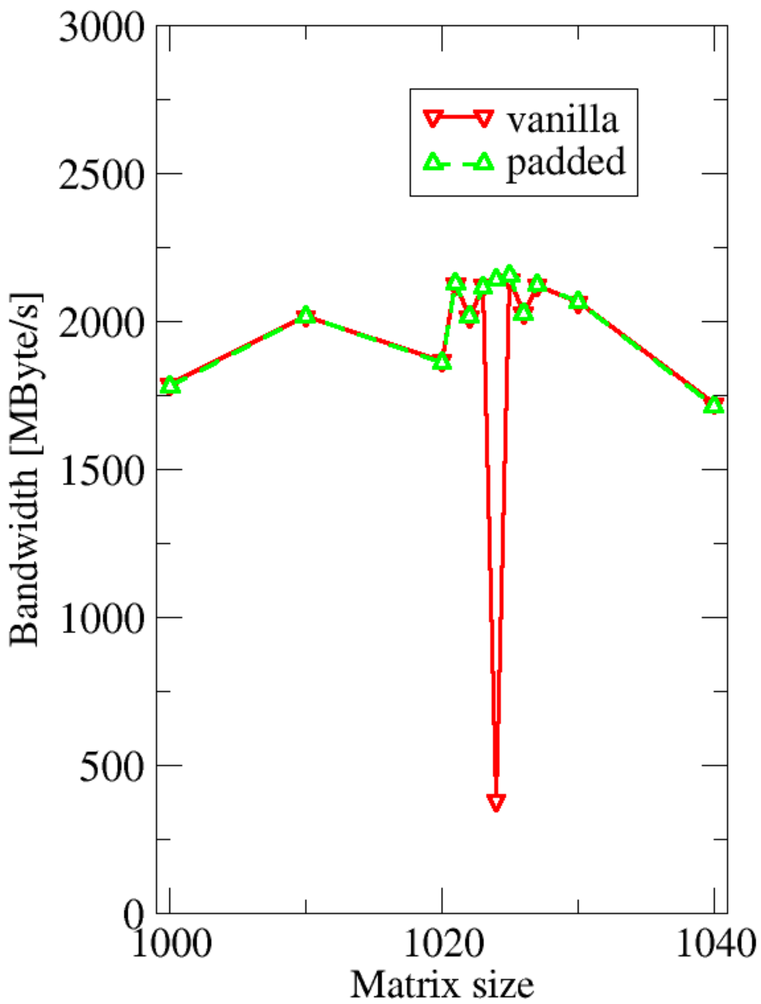
- Remedy: Outer loop unrolling and blocking

Dense matrix transpose: Vanilla version on different architectures



Dense matrix transpose: Cache thrashing

- A closer look (e.g. on Xeon/Netburst) reveals interesting performance characteristics:



- Matrix sizes of powers of 2 seem to be extremely unfortunate
  - Reason: Cache thrashing!
- Remedy: Improve effective cache size by padding the array dimensions!
  - `a(1024,1024)` → `a(1025,1025)`
    `b(1024,1024)` → `b(1025,1025)`
  - Eliminates the thrashing completely
- Rule of thumb: If there is a choice, use dimensions of the form 16•(2k+1)

Dense matrix transpose: Unrolling and blocking

```
do i=1,N
 do j=1,N
  a(j,i) = b(i,j)
 enddo
enddo
```

unroll/jam

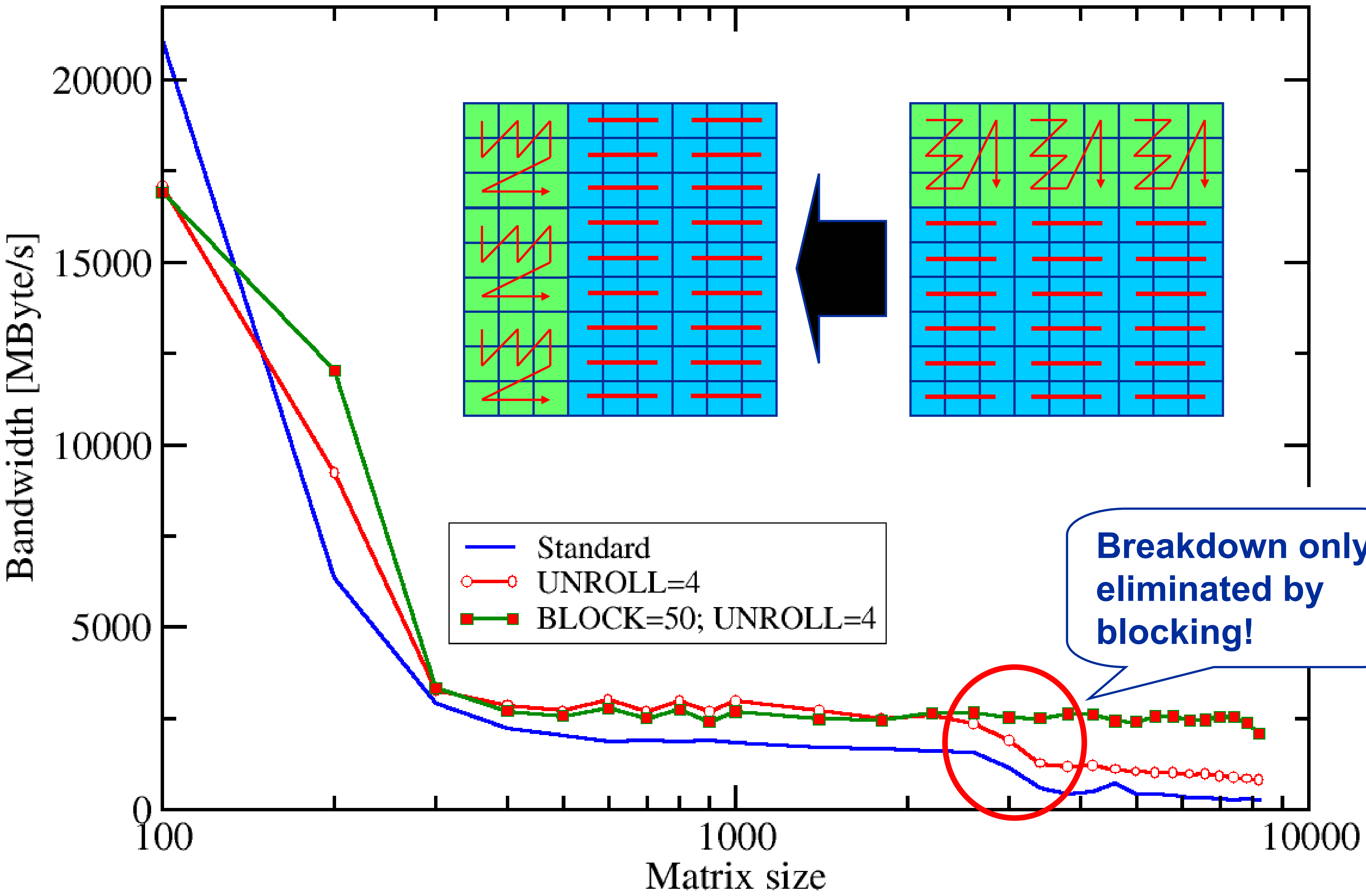
```
do i=1,N,U
 do j=1,N
  a(j,i)       = b(i,j)
  a(j,i+1)     = b(i+1,j)
  ...
  a(j,i+U-1) = b(i+U-1,j)
 enddo
enddo
```

block

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

Blocking and unrolling factors (B,U) can be determined experimentally; be guided by cache sizes and line lengths

Dense matrix transpose: Blocked/unrolled versions on Xeon/Netburst 3.2 GHz



Data access – general guidelines

- Case 3: $O(N^3)/O(N^2)$ algorithms
  - Most favorable case – computation outweighs data traffic by factor of N
  - Examples: Dense matrix diagonalization, dense matrix-matrix multiplication
  - Huge optimization potential: proper optimization can render the problem cache-bound if N is large enough
  - Example: dense matrix-matrix multiplication

```
do i=1,N
 do j=1,N
  do k=1,N
   c(j,i)=c(j,i)+a(k,i)*b(k,j)
  enddo
 enddo
enddo
```

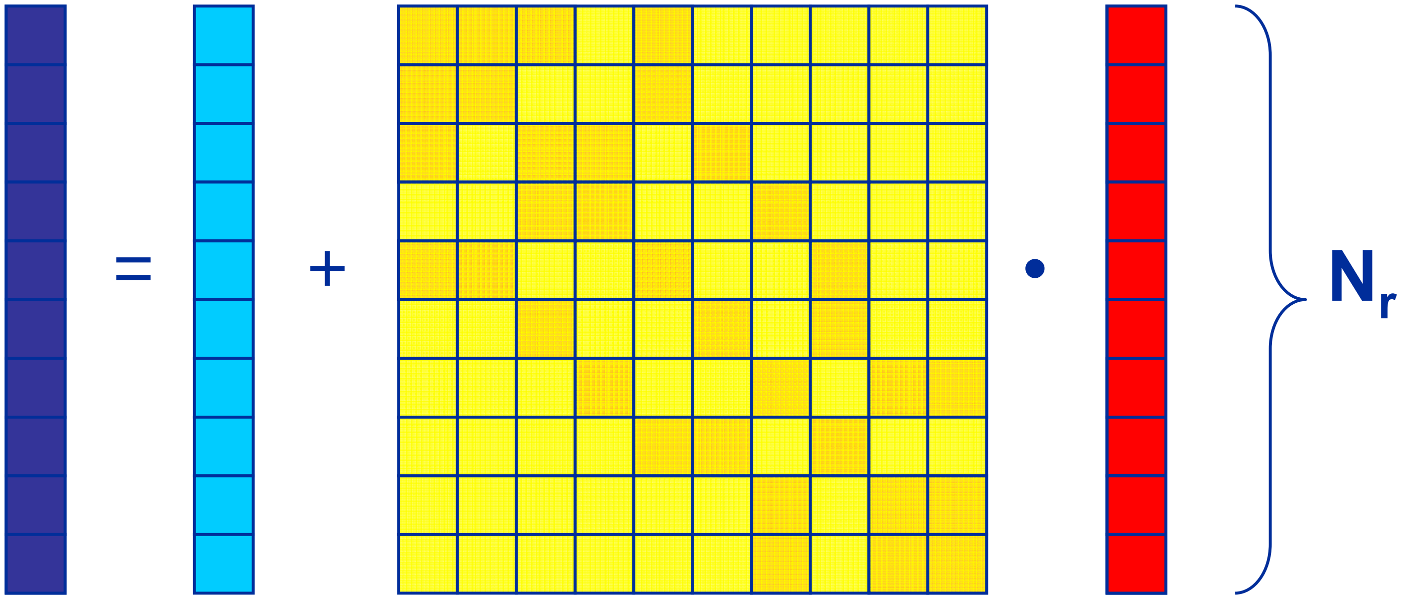
Core task: dense MVM ($O(N^2)/O(N^2)$)
→ memory bound

→ Tutorial exercise: Which fraction of peak can you achieve?

Optimizing sparse matrix-vector multiplication

Sparse matrix-vector multiply (sMVM)

- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson
- **Store only N_{nz} nonzero elements** of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- **"Sparse": $N_{nz} \sim N_r$**
- Type O(N)/O(N) → **memory bound**
  - Nevertheless, there is more than one loop here!

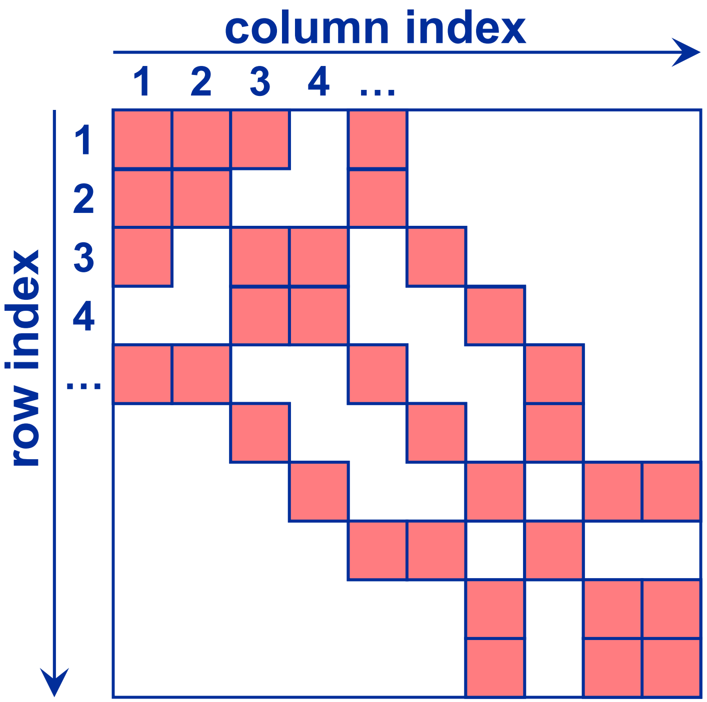


General case: some indirect addressing required!

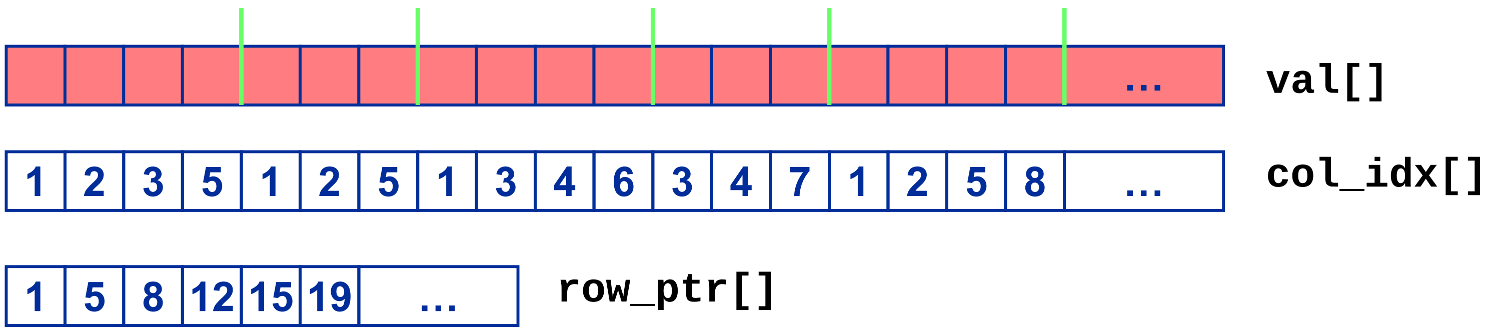
Sparse matrix-vector multiply: Different matrix storage schemes

- Choice of sparse matrix storage scheme is crucial for performance
  - Different schemes yield entirely different performance characteristics
- Most important formats:
  - CRS (Compressed Row Storage)
  - JDS (Jagged Diagonals Storage)
- Other possibilities:
  - CCS (Compressed Column Storage, “Harwell-Boeing”)
  - CDS (Compressed Diagonal Storage)
  - SKS (Skyline Storage)
  - SYDY (Something You Devised Yourself)
- Depending on the storage scheme, the memory access patterns differ vastly between the formats
  - So do the opportunities for optimization
  - Choose the storage scheme that best fits your needs

CRS matrix storage scheme



- `val[]` stores all the nonzeroes (length N_{nz})
- `col_idx[]` stores the column index of each nonzero (length N_{nz})
- `row_ptr[]` stores the starting index of each new row in `val[]` (length: N_r)



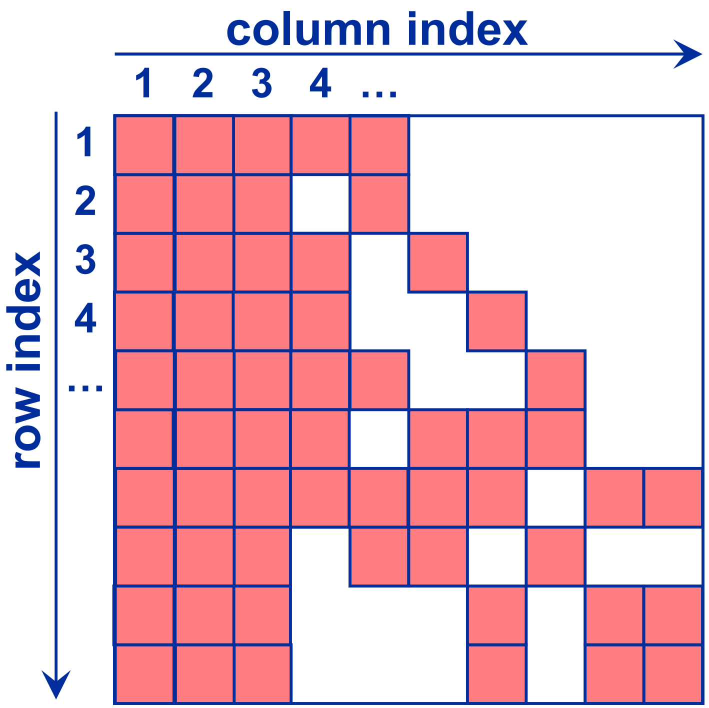
CRS sparse MVM

- Implement `c(:)=m(:,:)*b(:)`
- Only the nonzero elements of the matrix are used
  - Operation count = $2N_{nz}$

```
do i = 1,Nr
 do j = row_ptr(i), row_ptr(i+1) - 1
  c(i) = c(i) + val(j) * b(col_idx(j))
 enddo
enddo
```

- Features
  - Long outer loop (N_r)
  - Probably short inner loop (number of nonzero entries in each respective row)
  - Register-optimized access to result vector `c[]`
  - Stride-1 access to matrix data in `val[]`
  - Indexed (indirect) access to RHS vector `b[]`

JDS matrix storage scheme



- `val[]` stores all the nonzeroes (length N_{nz})
- `col_idx[]` stores the column index of each nonzero (length N_{nz})
- `jd_ptr[]` stores the starting index of each new jagged diagonal in `val[]`
- `perm[]` holds the permutation map (length N_r)

`val[]`

																				…

`col_idx[]`

4	1	1	1	3	1	3	5	7	7	6	2	3	2	6	2	4	6	9	9	…

`jd_ptr[]`

1	11	21	…

`perm[]`

2	6	3	7	4	5	1	8	9	10	…

JDS sparse MVM

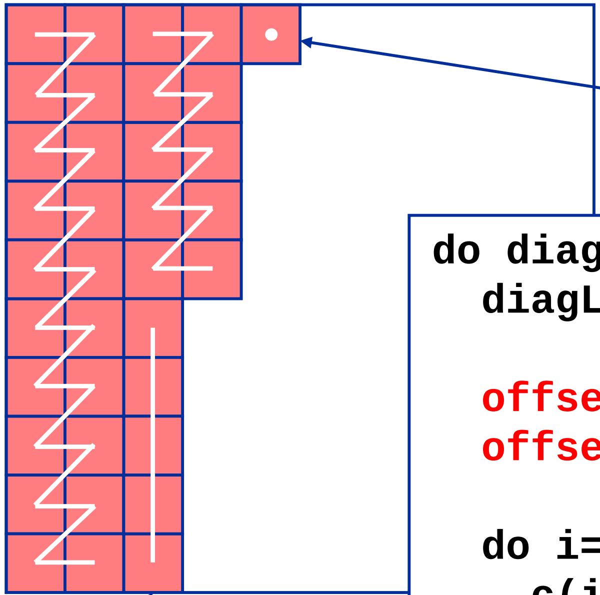
- Implement `c(:)=m(:,:)*b(:)`
- Only the nonzero elements of the matrix are used
  - Operation count = $2N_{nz}$

```
do diag=1, zmax
  diagLen = jd_ptr(diag+1) - jd_ptr(diag)
  offset  = jd_ptr(diag)
  do i=1, diagLen
    c(i) = c(i) + val(offset+i) * b(col_idx(offset+i))
  enddo
enddo
```

- Features
  - Long inner loop (max. N_r)
    - candidate for vectorization/parallelization
  - Short outer loop (number of jagged diagonals)
  - Multiple accesses to each element of result vector `c[]`
    - optimization potential
  - Stride-1 access to matrix data in `val[]`
  - Indexed (indirect) access to RHS vector `b[]`

JDS sparse MVM optimization

- Outer 2-way loop unrolling for JDS (B_c 9/4 → 7/4)



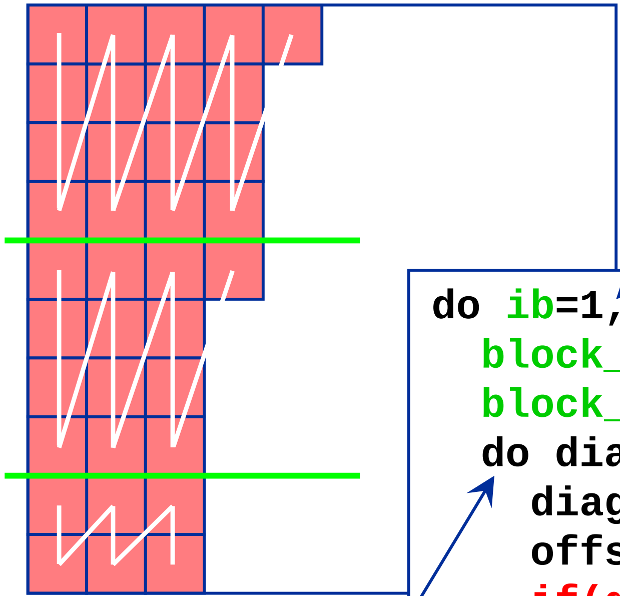
Remainder loop (omitted in code)

Iterations “peeled off”

```
do diag=1,zmax,2
  diagLen = min( (jd_ptr(diag+1)-jd_ptr(diag)) ,\
                 (jd_ptr(diag+2)-jd_ptr(diag+1)) )
  offset1 = jd_ptr(diag)
  offset2 = jd_ptr(diag+1)

  do i=1, diagLen
    c(i) = c(i)+val(offset1+i)*b(col_idx(offset1+i))
    c(i) = c(i)+val(offset2+i)*b(col_idx(offset2+i))
  enddo

  offset1 = jd_ptr(diag)
  do i=(diagLen+1),(jd_ptr(diag+1)-jd_ptr(diag))
    c(i) = c(i)+val(offset1+i)*b(col_idx(offset1+i))
  enddo

enddo
```

JDS sparse MVM optimization

- **Blocking optimization for JDS sparse MVM:**
  - **Does not enhance code balance but cache utilization**



Loop over blocks

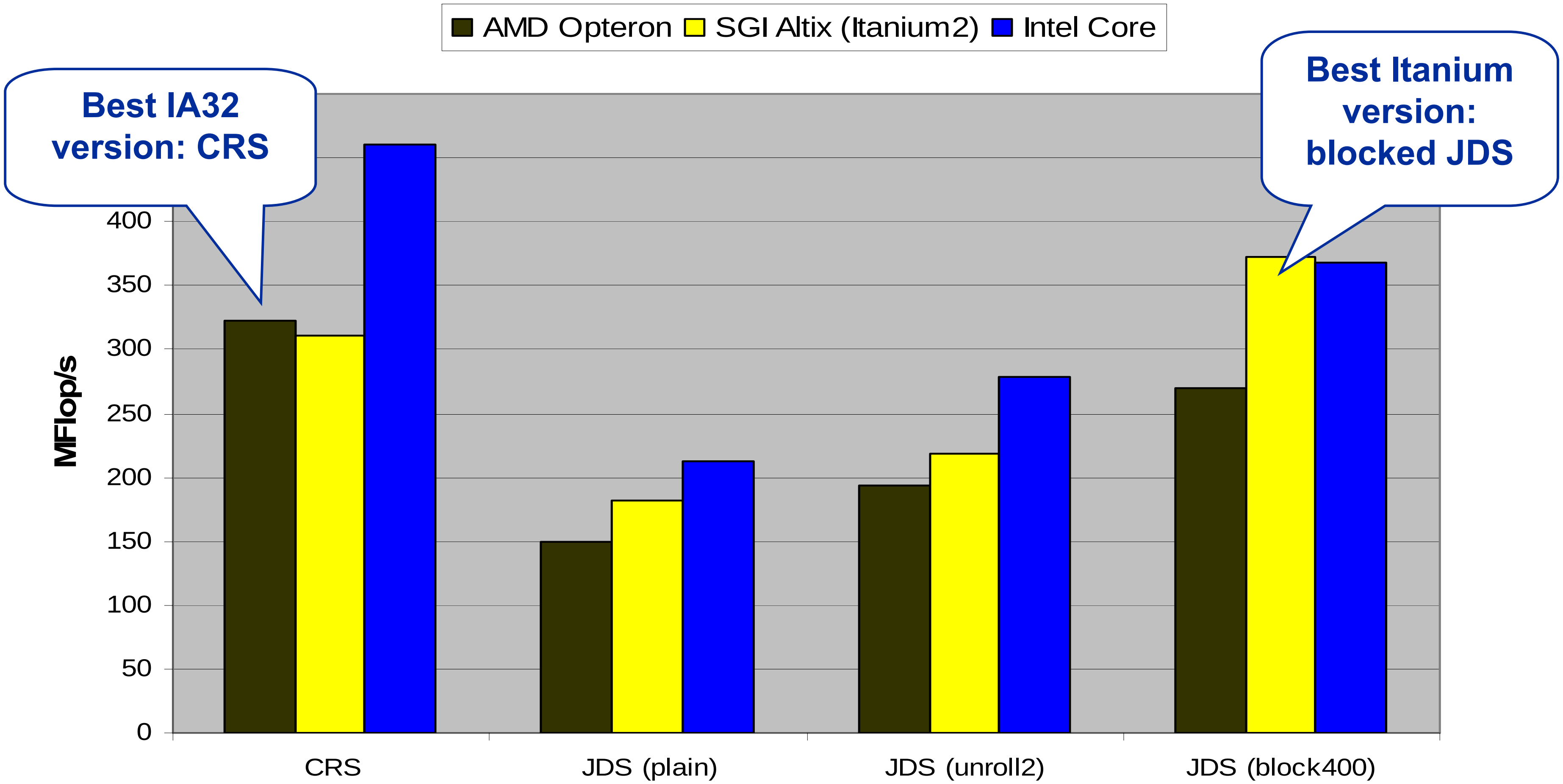
```
do ib=1, maxDiagLen, blocklen
  block_start = ib
  block_end   = min(ib+blocklen-1, maxDiagLen)
  do diag=1, zmax
    diagLen = jd_ptr(diag+1)-jd_ptr(diag)
    offset  = jd_ptr(diag)
    if(diagLen .ge. block_start) then
      do i=block_start, min(block_end,diagLen)
        c(i) = c(i)+val(offset+i)*b(col_idx(offset+i))
      enddo
    endif
  enddo
enddo
```

Loop over diagonal chunks

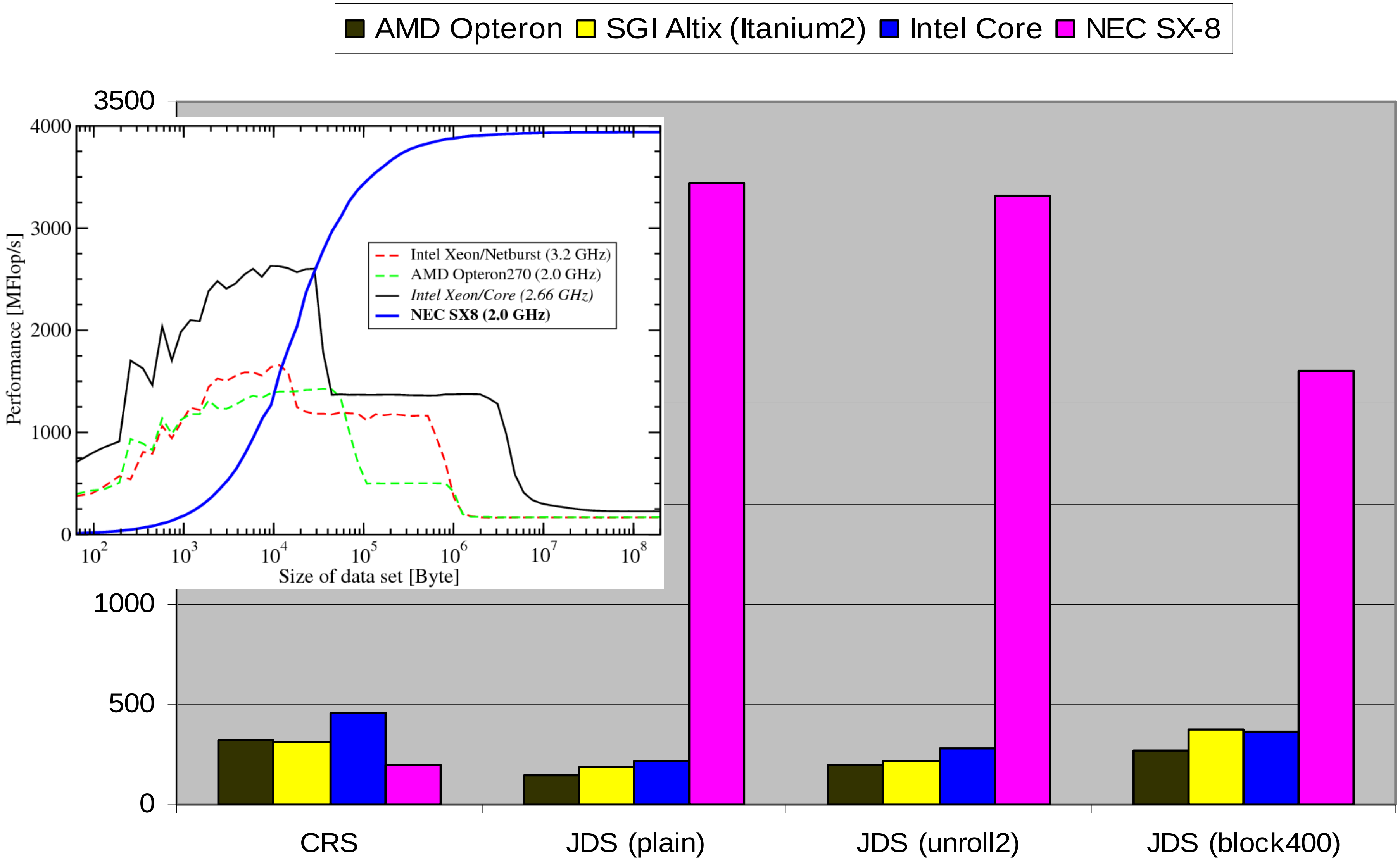
Standard JDS MVM kernel

Performance comparison: CRS vs. JDS

- **CRS: Vanilla version**
- **JDS: Vanilla vs. Unrolled vs. Blocked**
  - **Experimentally determined optimal block length: 400**



Performance comparison: Real programmers use vector computers…



References

- S. Goedecker, A. Hoisie
  Performance Optimization of Numerically Intensive Codes
  Society for Industrial & Applied Mathematics,U.S. (ISBN 0898714842)

- R. Gerber et al.
  The Software Optimization Cookbook, Second Edition
  High-Performance Recipes for IA-32 Platforms
  Intel Press (ISBN 0-9764832-1-1)

- R. Barrett et al.
  Templates for the Solution of Linear Systems:
  Building Blocks for Iterative Methods
  `http://www.netlib.org/templates/Templates.html`