Case study:
OpenMP-parallel sparse matrix-vector multiplication

A simple (but sometimes not-so-simple) example for bandwidth-bound code and saturation effects in memory
Sparse matrix-vector multiply (spMVM)

- 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”: $N_{nz} \sim N_r$

(c) RRZE 2014
CRS matrix storage scheme

- **val[]** stores all the nonzeros (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: $N_r$)

```
1 2 3 4 ...
1
2
3
4
...
```

```
val[]
col_idx[]
row_ptr[]
```

(c) RRZE 2014
Case study: Sparse matrix-vector multiply

- **Strongly memory-bound for large data sets**
  - Streaming, with partially indirect access:

```c
!$OMP parallel do
do i = 1,Nr
    do j = row_ptr(i), row_ptr(i+1) - 1
        c(i) = c(i) + val(j) * b(col_idx(j))
    enddo
enddo
!$OMP end parallel do
```

- Usually many spMVMs required to solve a problem

- **Following slides: Performance data on one 24-core AMD Magny Cours node**

(c) RRZE 2014
Bandwidth-bound parallel algorithms: Sparse MVM

- **Data storage format is crucial for performance properties**
  - Most useful general format: Compressed Row Storage (CRS)
  - SpMVM is *easily parallelizable* in shared and distributed memory

- **For large problems, spMVM is inevitably memory-bound**
  - Intra-LD saturation effect on modern multicores

- **MPI-parallel spMVM is often communication-bound**
  - See later part for what we can do about this…
Case 1: Large matrix

Intrasocket bandwidth bottleneck

Good scaling across NUMA domains

Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

(c) RRZE 2014
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- **Case 2: Medium size**

Intrasocket bandwidth bottleneck

Working set fits in aggregate cache

mc2depi, 525825x525825, non-zero: 2100225

(c) RRZE 2014
Case 3: Small size

- No bandwidth bottleneck
- Parallelization overhead dominates
Conclusions from the spMVM benchmarks

- If the problem is “large”, bandwidth saturation on the socket is a reality
  - There are “spare cores”
  - Very common performance pattern

- What to do with spare cores?
  - Let them idle → saves energy with minor loss in time to solution
  - Use them for other tasks, such as MPI communication

- Can we predict the saturated performance?
  - Bandwidth-based performance modeling!
  - What is the significance of the indirect access? Can it be modeled?

- Can we predict the saturation point?
  - … and why is this important?
Example: SpMVM chip performance model

- Sparse MVM in double precision w/ CRS data storage:

```plaintext
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
```

- DP CRS comp. intensity
  - $\alpha$ quantifies traffic for loading RHS
    - $\alpha = 0 \rightarrow$ RHS is in cache
    - $\alpha = 1/N_{nzr} \rightarrow$ RHS loaded once
    - $\alpha = 1 \rightarrow$ no cache
    - $\alpha > 1 \rightarrow$ Houston, we have a problem!
  - “Expected” performance = $b_s \times I_{CRS}$
  - Determine $\alpha$ by measuring performance and actual memory traffic
    - Maximum memory BW may not be achieved with spMVM

$$I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \text{ flops/byte}$$
Determine RHS traffic

\[ I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \text{flops/byte} = \frac{N_{nz} \cdot 2 \text{flops}}{V_{meas}} \]

- \( V_{meas} \) is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for \( \alpha \):
  \[ \alpha = \frac{1}{4} \left( \frac{V_{meas}}{N_{nz} \cdot 2 \text{bytes}} - 6 - \frac{8}{N_{nzr}} \right) \]
- Example: kkt_power matrix from the UoF collection on one Intel SNB socket
  - \( N_{nz} = 14.6 \cdot 10^6, N_{nzr} = 7.1 \)
  - \( V_{meas} \approx 258 \text{ MB} \)
  - \( \rightarrow \alpha = 0.43, \alpha N_{nzr} = 3.1 \)
  - \( \rightarrow \) RHS is loaded 3.1 times from memory
  - and: \( \frac{I_{CRS}^{DP}(1/N_{nzr})}{I_{CRS}^{DP}(\alpha)} = 1.15 \)

15% extra traffic \( \rightarrow \) optimization potential!
Roofline analysis for spMVM

- **Conclusion**
  - The roofline model does not work 100% for spMVM due to the RHS traffic uncertainties
  - We have “turned the model around” and measured the actual memory traffic to determine the RHS overhead
  - Result indicates:
    1. how much actual traffic the RHS generates
    2. how efficient the RHS access is (compare BW with max. BW)
    3. how much optimization potential we have with matrix reordering

- **Consequence:** *If the model does not work, we learn something!*
Input to the roofline model

... on the example of spMVM with kkt_power matrix

Throughput: 1 ADD, 1 MULT + 1 LD + 1ST/cy

Memory-bound!
\[ \alpha = 0.43 \]

Code analysis:
1 ADD, 1 MULT, 
\( (1.5 + 1/N_{nzr}) \) LOADs, 
1/N_{nzr} STOREs + \( \alpha \)

Measured memory traffic \( \approx 258 \) MB
A word on sparse matrix storage formats

CRS
Sliced ELLPACK
SELL-C-σ
Sparse Matrix Format Jungle

Roofline case studies

(c) RRZE 2014
SpMVM in the Heterogeneous Era

- Compute clusters are getting more and more heterogeneous

- A special format per compute architecture
  1. hampers runtime exchange of matrix data
  2. complicates library interfaces

- CRS (CPU standard format) may be problematic (cf. next slide)
  - Vectorization along matrix rows
  - Bad utilization for short rows and wide SIMD units (Intel MIC: 512 bit)

→ We want to have a unified, SIMD-friendly, and high-performance sparse matrix storage format.
Compressed Row Storage (CRS)

- Standard format for CPUs

- Entries and column indices stored row-wise

(c) RRZE 2014
CRS Vectorization

unsigned int i, j;
double tmp;

#pragma omp parallel for schedule(runtime) private (tmp1, tmp2, j)
for (i=0; i<nrows; i++){
    tmp1 = 0.0;
    tmp2 = 0.0;
    for (j=rpt[i]; j<rpt[i+1]; j=j+2){
        tmp1 += val[j] * rhs[col[j]];
        tmp2 += val[j+1] * rhs[col[j+1]];
    }
    lhs[i] += tmp1 + tmp2;
}

- **Potential problem:** Long vector registers on modern CPUs (e.g., 512 bit on Xeon Phi)
  - 512 bit → 8 doubles or 16 integers in a single vector
  - j-loop:16-way unrolling ⇒ problem for short rows

(c) RRZE 2014
Sliced ELLPACK

- Well-known sparse matrix format for GPUs

- Entries and column indices stored column-wise in chunks

- One parameter:
  1. C: Chunk height
Sliced ELLPACK

**Potential problem:**
Depending on the variation in the row length, a more or less significant amount of zeros will be loaded and processed, quantified by $\beta$ ("chunk occupancy"):

$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot cl[i]}$$

$$1/C \leq \beta \leq 1$$

- $\beta = 1/C \Rightarrow$ maximum overhead
- $\beta = 1 \Rightarrow$ no overhead at all

(row length is constant in a chunk)

C............ chunk height
$N_c$........ number of chunks
cl[k]........ max. row length in k-th chunk

(c) RRZE 2014
Minimizing the storage overhead $\Rightarrow$ SELL-C-$\sigma$

- Sort rows within a range $\sigma$ to minimize the overhead
  - $\sigma$ should not be too large in order to not worsen the RHS vector access pattern

- Two parameters:
  1. C: Chunk height
  2. $\sigma$: Sorting scope

(c) RRZE 2014

Roofline case studies
Choosing the Sorting Scope $\sigma$

- The larger the sorting scope, the lower the storage overhead.
- But what happens if the sorting scope gets too large?

![Graph showing Gflop/s vs. $\log(\sigma)$](c) RRZE 2014

Roofline case studies
SELL-C-σ Performance

Using a unified storage format comes with little performance penalty in the worst case and up to a 3x performance gain in the best case for a wide range of test matrices.
Case study: A 3D Jacobi smoother

The basics in two dimensions
Layer conditions
Validating the model
Optimization by spatial blocking
Case study:
A 3D Jacobi smoother

The basics in two dimensions
Layer conditions
Validating the model
Optimization by spatial blocking

Intel® Xeon® Processor E5-2690 v2
10 cores@3 GHz
L3 CacheSize = 25 MB
Memory Bandwidth = 48 GB/s
Stencil schemes

- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically it is a sparse matrix vector multiply (spMVM) embedded in an iterative scheme (outer loop)
- but the regular access structure allows for matrix free coding

```plaintext
do iter = 1, maxit

    Perform sweep over regular grid: y ← x

    Swap y ←→ x

endo
```

- Complexity of implementation and performance depends on
  - stencil operator, e.g. Jacobi-type, Gauss-Seidel-type,…
  - spatial extent, e.g. 7-pt or 25-pt in 3D,…

(c) RRZE 2014
Jacobi-type 5-pt stencil in 2D

\[
\begin{align*}
\text{do } & k=1,k_{\text{max}} \\
\text{do } & j=1,j_{\text{max}} \\
y(j,k) &= \text{const} \times (x(j-1,k) + x(j+1,k) & \\
& + x(j,k-1) + x(j,k+1) )
\end{align*}
\]

enddo
enddo

Appropriate performance metric: "Lattice Updates per second" [LUP/s] (here: Multiply by 4 FLOP/LUP to get FLOP/s rate)
Jacobi 5-pt stencil in 2D: data transfer analysis

\[
\begin{align*}
    y(j,k) &= \text{const} \times (x(j-1,k) + x(j+1,k) + x(j,k-1) + x(j,k+1)) \\
\end{align*}
\]

Available in cache (used 2 iterations before)

LD+ST\ y(j,k) (incl. write allocate)

SWEEP

do k=1,kmax
    do j=1,jmax
        y(j,k) = \text{const} \times (x(j-1,k) + x(j+1,k) \& \\
        + x(j,k-1) + x(j,k+1))
    enddo
enddo

Naive balance (incl. write allocate):

\[
\begin{align*}
    x(:, :) &: 3 \text{ LD} + \\
    y(:, :) &: 1 \text{ ST} + 1\text{ LD}
\end{align*}
\]

\[B_C = 40 \text{ B} / \text{LUP}\] (assuming double precision)
Case study:
A 3D Jacobi smoother

The basics in two dimensions
Layer conditions
Validating the model
Optimization by spatial blocking
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers of grid (assume “Least recently used” replacement strategy)

(c) RRZE 2014
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers of grid (+assume „Least recently used“ replacement strategy)

\[ x(0:j_{\text{max}}+1,0:k_{\text{max}}+1) \]
Analyzing the data flow

Reduce inner $(j)$ loop dimension successively

Best case: 3 “layers” of grid fit into the cache!

$$x(0:j_{\text{max}1}+1,0:k_{\text{max}1}+1)$$

$$x(0:j_{\text{max}2}+1,0:k_{\text{max}2}+1)$$
Analyzing the data flow: Layer condition

\[
\begin{align*}
do & \ k = 1, k_{\text{max}} \\
& \hspace{1cm} \text{do } j = 1, j_{\text{max}} \\
& \hspace{2cm} y(j, k) = \text{const} \times \left( x(j-1, k) + x(j+1, k) \& \\
& \hspace{4cm} + x(j, k-1) + x(j, k+1) \right) \\
& \hspace{1cm} \text{enddo} \\
& \text{enddo}
\end{align*}
\]

3 * j_{\text{max}} * 8B < \text{CacheSize}/2

“Layer condition”

Layer condition:
- No impact of outer loop length \( k_{\text{max}} \)
- No strict guideline (cache associativity – data traffic for \( y \) not included)
- Need to be adapted for other stencils, e.g. 3D 7-pt stencil
Analyzing the data flow: Layer condition

\[
\text{do } k=1, \text{kmax} \\
\quad \text{do } j=1, \text{jmax} \\
\quad \quad y(j,k) = \text{const} \times (x(j-1,k) + x(j+1,k) + x(j,k-1) + x(j,k+1)) \\
\quad \text{enddo} \\
\text{enddo}
\]

3 * jmax * 8B < CacheSize/2

“Layer condition” fulfilled?

\[
\text{do } k=1, \text{kmax} \\
\quad \text{do } j=1, \text{jmax} \\
\quad \quad y(j,k) = \text{const} \times (x(j-1,k) + x(j+1,k) + x(j,k-1) + x(j,k+1)) \\
\quad \text{enddo} \\
\text{enddo}
\]

\(B_C = 24 \text{ B} / \text{LUP}\)

\(B_C = 40 \text{ B} / \text{LUP}\)
From 2D to 3D

2D:

Towards 3D understanding

- Picture can be considered as 2D cut of 3D domain for (new) fixed \( i \)-coordinate:

\[
x(0:j_{\text{max}}+1,0:k_{\text{max}}+1) \Rightarrow x(i, 0:j_{\text{max}}+1,0:k_{\text{max}}+1)
\]
From 2D to 3D

- \(x(0:imax+1, 0:jmax+1, 0:kmax+1)\) – Assume \(i\)-direction contiguous in main memory (Fortran notation)
- Stay at 2D picture and consider one cell of \(j-k\) plane as a contiguous row of elements in \(i\)-direction: \(x(0:imax, j, k)\)
Layer condition: From 2D 5-pt to 3D 7-pt Jacobi-type stencil

\[
3 * \text{jmax} * 8B < \text{CacheSize}/2
\]

do \( k=1, k_{\text{max}} \)
  do \( j=1, \text{jmax} \)
    \[
    y(j,k) = \text{const} * (x(j-1,k) + x(j+1,k) \&
    \quad + x(j,k-1) + x(j,k+1))
    \]
  enddo
enddo

\[
B_c = 24 \text{ B} / \text{LUP}
\]

\[
\text{do } k=1, k_{\text{max}} \\
\text{do } j=1, \text{jmax} \\
\text{do } i=1, \text{imax} \\
\quad y(i,j,k) = \text{const} * (x(i-1,j,k) + x(i+1,j,k) \&
\quad + x(i,j-1,k) + x(i,j+1,k) \&
\quad + x(i,j,k-1) + x(i,j,k+1))
\]
enddo
enddo
enddo

\[
3 * \text{jmax} * \text{imax} * 8B < \text{CacheSize}/2
\]

\[
B_c = 24 \text{ B} / \text{LUP}
\]
3D 7-pt Jacobi stencil (sequential)

```
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) = const. * (x(i-1,j,k) + x(i+1,j,k) &
                           + x(i,j-1,k) + x(i,j+1,k) &
                           + x(i,j,k-1) + x(i,j,k+1) )
    enddo
  enddo
enddo
```

“Layer condition”
3*jmax*imax*8B < CS/2

“Layer condition” OK \(\rightarrow\) 5 accesses to \(x()\) served by cache

**Question:**
Does parallelization/multi-threading change the layer condition?
Jacobi stencil – OpenMP outer loop parallelization (I)

```fortran
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
    do j=1,jmax
        do i=1,imax
            y(i,j,k) = 1/6.*
                  *(x(i-1,j,k) + x(i+1,j,k) &
                   + x(i,j-1,k) + x(i,j+1,k) &
                   + x(i,j,k-1) + x(i,j,k+1))
        enddo
    enddo
enddo
```

Equal chunks in k-direction → Layer condition for each thread

\[
\text{nthreads} \times 3 \times jmax \times imax \times 8B < \text{CS}/2
\]

Layer condition (cubic domain; \( \text{CS} = 25 \text{ MB} \))

1 thread: \( \text{imax}=\text{jmax} < 720 \) → 10 threads: \( \text{imax}=\text{jmax} < 230 \)
Jacobi stencil – OpenMP outer loop parallelization (II)

Layer condition OK: \texttt{ntthreads} \* 3 \* \texttt{jmax} \* \texttt{imax} \* 8B < CS/2

\begin{verbatim}
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) = 1/6. * (x(i-1,j,k) + x(i+1,j,k) &
                         + x(i,j-1,k) + x(i,j+1,k) &
                         + x(i,j,k-1) + x(i,j,k+1) )
    enddo
  enddo
enddo
\end{verbatim}

\textbf{Intel® Xeon® Processor E5-2690 v2}
10 cores@3 GHz
CS = 25 MB (L3)
b\textsubscript{S} = 48 GB/s

\rightarrow P = 2000 MLUP/s
Case study: A 3D Jacobi smoother

The basics in two dimensions
Layer conditions
Validating the model
Optimization by spatial blocking
3D OpenMP Jacobi Stencil – model validation

Validation: Measured data traffic from main memory [Bytes/LUP]

1 thread: Layer condition OK – but can not saturate bandwidth

10 threads: performance drops around $i_{\text{max}} = 230$

Roofline case studies (c) RRZE 2014

Roofline limit (48 GB/s; 24 B/LUP)

Roofline assumption: 24 B/LUP

Validation: Measured data traffic from main memory [Bytes/LUP]
Jacobi Stencil – violated layer condition

Layer condition not OK: \( \text{nthreads} \times 3 \times j_{\text{max}} \times i_{\text{max}} \times 8B > \text{CS}/2 \)

\begin{verbatim}
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,k_{\text{max}}
do j=1,j_{\text{max}}
do i=1,i_{\text{max}}
   \text{y}(i,j,k) = 1/6. * (x(i-1,j,k) + x(i+1,j,k) &
   + x(i,j-1,k) + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1) )
enddo
enddo
enddo
\end{verbatim}

But assume: \( \text{nthreads} \times 3 \times i_{\text{max}} \times 8B < \text{CS}/2 \)

\begin{itemize}
  \item (8+8) B/LUP for \( y() \) (ST+WA)
  \item + 8 B/LUP for \( x(i,j,k+1) \)
  \item + 8 B/LUP for \( x(i,j+1,k) \)
  \item + 8 B/LUP for \( x(i,j,k-1) \)
\end{itemize}

\( B_C = 40 \text{ B/LUP} \)

Roofline: \( P = 1200 \text{ MLUP/s} \)
3D OpenMP Jacobi Stencil – model validation

Roofline case studies

Roofline limit (48 GB/s; 24 B/LUP)

Roofline (40 B/LUP)

Roofline assumption: 24 B/LUP
Case study:  
A 3D Jacobi smoother

The basics in two dimensions
Layer conditions
Validating the model
Optimization by spatial blocking
### Enforcing the layer condition by blocking

Split up domain into subblocks.

*Example: block size = 5*

---

(c) RRZE 2014
Enforcing the layer condition by blocking

<table>
<thead>
<tr>
<th>Block Boundaries</th>
<th>Additional data transfers (overhead) at block boundaries!</th>
</tr>
</thead>
</table>

(c) RRZE 2014
Jacobi Stencil – simple spatial blocking

do jb=1,jmax,jblock ! Assume jmax is multiple of jblock

!$OMP PARALLEL DO SCHEDULE(STATIC)
do  k=1,kmax
    do j=jb,(jb+jblock-1) ! Loop length jblock
        do i=1,imax
            y(i,j,k) = 1/6. *(x(i-1,j,k) +x(i+1,j,k) &
                             + x(i,j-1,k) +x(i,j+1,k) &
                             + x(i,j,k-1) +x(i,j,k+1))
        enddo
    enddo
enddo

Layer condition (j-Blocking)
nthreads*3*jblock*imax*8B < CS/2

Ensure layer condition by choosing jblock appropriately (Cubic Domains):

  jblock < CS/(imax* nthreads* 48B )

Test system: Intel® Xeon® Processor E5-2690 v2 (10 cores / 3 GHz)

  bS = 48 GB/s,    CS = 25 MB (L3)  \rightarrow  P = bS/Bc = 2000 MLUP/s
Jacobi Stencil – simple spatial blocking

Determine:

\[ j_{\text{block}} < \frac{CS}{(2 \times n_{\text{threads}} \times 3 \times \text{imax} \times 8B)} \]

\( i_{\text{max}} = j_{\text{max}} = k_{\text{max}} \)

CS = 10 MB:
~ 90+ % roofline limit

Validation: Measured data traffic from main memory [Bytes/LUP]

(j)block change

Roofline limit (48 GB/s; 24 B/LUP)
Conclusions from the Jacobi example

- We have made sense of the memory-bound performance vs. problem size
  - “Layer conditions” lead to predictions of code balance
  - Achievable memory bandwidth is input parameter
- “What part of the data comes from where” is a crucial question
- The model works only if the bandwidth is “saturated”
  - In-cache modeling is more involved

- Avoiding slow data paths == re-establishing the most favorable layer condition

- Improved code showed the speedup predicted by the model
- Optimal blocking factor can be estimated
  - Be guided by the cache size the layer condition
  - No need for exhaustive scan of “optimization space”