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”: $N_{nz} \sim N_r$

![Diagram showing sparse matrix-vector multiplication]
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: \( N_r \))

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

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

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

- Usually many spMVMs required to solve a problem

- **Following slides: Performance data on one 24-core AMD Magny Cours node**
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…
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- Case 1: Large matrix

Intrasocket bandwidth bottleneck

Good scaling across NUMA domains

(c) RRZE 2014
Case 2: Medium size

Application: Sparse matrix-vector multiply
Strong scaling on one XE6 Magny-Cours node

- Intrasocket bandwidth bottleneck
- Working set fits in aggregate cache

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

Strong scaling on one Magny-Cours node

- 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:**

  ```
  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
  ```

- **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

- **Roofline case studies**

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

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

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

**Memory-bound!**
\( \alpha = 0.43 \)

**Measured memory traffic:** \( \approx 258 \text{ MB} \)
A word on sparse matrix storage formats

CRS
Sliced ELLPACK
SELL-C-σ
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
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 $\rightarrow$ 8 doubles or 16 integers in a single vector
  - j-loop:16-way unrolling $\Rightarrow$ **problem for short rows**
Sliced ELLPACK

- Well-known sparse matrix format for GPUs

- Entries and column indices stored column-wise in chunks

- One parameter:
  1. $C$: Chunk height
**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)
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?
SELL-C-\(\sigma\) 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.

Roofline case studies
(c) RRZE 2014