Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Intel Xeon Phi

Athena Elafrou Georgios Goumas Nectarios Koziris

National Technical University of Athens

iWAPT

International Workshop on Automatic Performance Tuning

Orlando, Florida

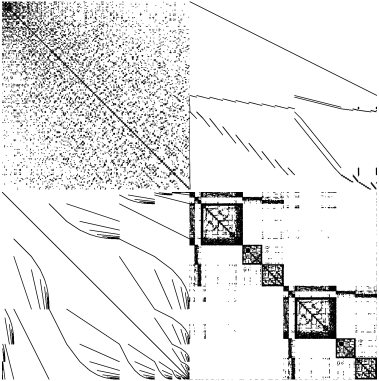
2 June 2017

Outline

Motivation & Background

Sparse Matrices

- Dominated by zeroes ($> 90\%$)
- Application domains:
  - Scientific, discretization of PDEs
  - Graph analytics
  - Machine learning
  - Linear programming
- Compact representation
  - Non-zero values
  - Topological information



Motivation & Background

Storing a sparse matrix: the Compressed Sparse Row (CSR) format

$$A = \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & 0 & 0 & 0 \\ 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & 0 & 0 & 2.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{pmatrix}$$

`rowptr:` (0 4 7 10 12 14 16)

`colind:` (0 1 2 3 0 1 2 0 1 2 0 3 4 5 4 5)

`val:` (7.5 2.9 2.8 2.7 6.8 5.7 3.8 2.4 6.2 3.2 9.7 2.3 5.8 5.0 6.6 8.1)

- Stores the columns of the non-zero elements and "pointers" to the first non-zero element of each row
- Most commonly used format
  - Relatively compact representation ($\approx 12 \cdot NNZ$ bytes)
  - Straightforward implementation

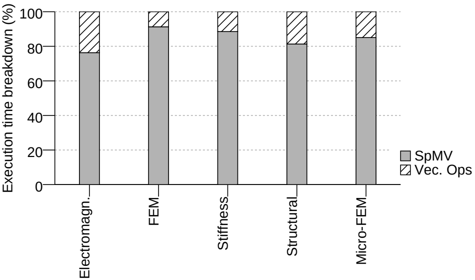
Motivation & Background

The Sparse Matrix-Vector Multiplication (SpMV) kernel

$$y = A \cdot x$$

A is a sparse matrix
x, y are dense vectors

- Ubiquitous in scientific and engineering applications
- Dominates the execution time of many iterative methods for the solution of large sparse linear systems (e.g., CG, GMRES)



Motivation & Background

SpMV using CSR

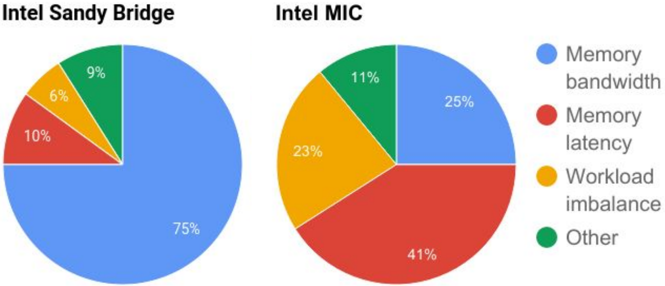
SpMV kernel

```
for (i = 0; i < N; i++)
    for (j = rowptr[i]; j < rowptr[i+1]; j++)
        y[i] += val[j] * x[colind[j]];
```

- SpMV is characterized by:
  - Extremely low *flop:byte* ratio (< 0.25)
    - inherently memory bound (according to the Roofline Model)
  - Irregular memory accesses to vector x
    - may cause excessive cache misses
  - Loop overheads in case of shorts rows
  - Workload imbalance in case of highly uneven row lenghts

Performance analysis on Intel Xeon Phi

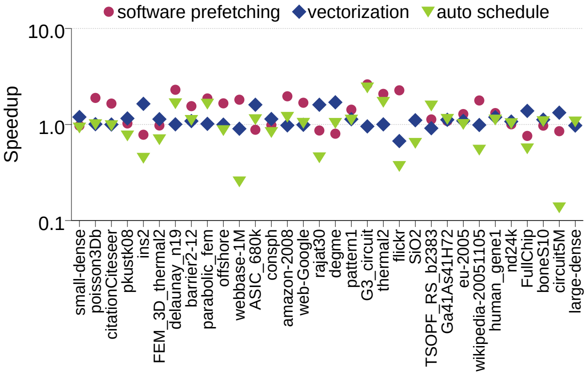
Which performance issues are more prominent?



- On traditional multicore systems
  - More compact formats (BCSR, SSS, CSX, CSB, ELL etc.)
  - Reordering techniques
  - Load balancing techniques
- On modern manycore systems?

Performance analysis on Intel Xeon Phi

Effect of different optimizations on CSR SpMV



- Blindly applying/combining optimizations may hinder performance!
- We need to intelligently select the optimization(s) for every matrix
  - Based on its performance bottleneck(s)

Performance analysis on Intel Xeon Phi

How do we define and determine bottlenecks?

- We define four bottlenecks
  - *MB*: **M**emory **B**andwidth
  - *ML*: **M**emory **L**atency
  - *IMB*: Workload **IMB**alance
  - *CMP*: **C**o**MP**utation
- We define a performance bound for every bottleneck
  - Perform a "bound and bottleneck" analysis
  - Estimate the performance that may be gained by eliminating each bottleneck
- We design heuristics that determine the bottleneck(s) of a matrix based on the estimated performance bounds

Performance analysis on Intel Xeon Phi

Per-bottleneck performance bounds (1/2)

MB

$$P_{MB} = \frac{2 \cdot NNZ}{\frac{M_{A_{format},min} + M_{xy,min}}{B_{max}}}$$

where B_{max} is the maximum sustainable DRAM bandwidth of the system, $M_{A_{format},min}$ and $M_{xy,min}$ is the minimum memory traffic that can be generated by the matrix stored in *format* and the vectors respectively

Performance analysis on Intel Xeon Phi

Per-bottleneck performance bounds (1/2)

MB

$$P_{MB} = \frac{2 \cdot NNZ}{\frac{M_{A_{format},min} + M_{xy,min}}{B_{max}}}$$

where B_{max} is the maximum sustainable DRAM bandwidth of the system, $M_{A_{format},min}$ and $M_{xy,min}$ is the minimum memory traffic that can be generated by the matrix stored in *format* and the vectors respectively

ML

We run a modified SpMV kernel, where irregular accesses to the right-hand side vector are completely eliminated by setting the column indices of all nonzero elements to zero

Performance analysis on Intel Xeon Phi

Per-bottleneck performance bounds (2/2)

IMB

$$P_{IMB} = \frac{2 \cdot NNZ}{t_{median}}$$

where t_{median} is the median execution time of all threads

Performance analysis on Intel Xeon Phi

Per-bottleneck performance bounds (2/2)

IMB

$$P_{IMB} = \frac{2 \cdot NNZ}{t_{median}}$$

where t_{median} is the median execution time of all threads

CMP

We run a modified SpMV kernel, where we completely eliminate indirect memory references, resulting in unit-stride accesses only.

Performance analysis on Intel Xeon Phi

Per-bottleneck performance bounds (2/2)

IMB

$$P_{IMB} = \frac{2 \cdot NNZ}{t_{median}}$$

where t_{median} is the median execution time of all threads

CMP

We run a modified SpMV kernel, where we completely eliminate indirect memory references, resulting in unit-stride accesses only.

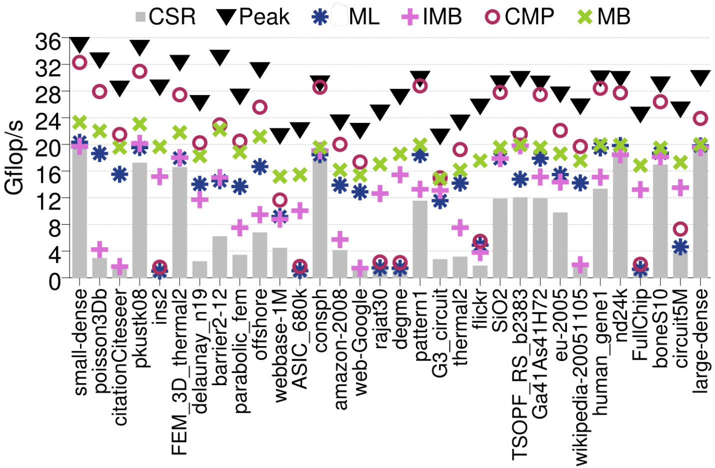
Peak

$$P_{peak} = \frac{2 \cdot NNZ}{\frac{M_{A,min}+M_{xy,min}}{B_{max}}}$$

where $M_{A,min}$ assumes we can only compress the indexing information of a sparse matrix (not the values).

Performance analysis on Intel Xeon Phi

“Bound and bottleneck” analysis



Performance analysis on Intel Xeon Phi

Heuristics for bottleneck detection

procedure CLASSIFY(P_{CSR}, P_{MB}, P_{ML}, P_{IMB}, P_{CMP}, P_{peak})
  $class \leftarrow \{\}$
  if $(P_{CSR} \approx P_{MB} \approx P_{ML} \approx P_{IMB})$ **then**
    $class \leftarrow class \cup \{MB\}$
  end if
  if $(\frac{P_{IMB}}{P_{CSR}} > T_{IMB})$ **then**
    $class \leftarrow class \cup \{IMB\}$
  end if
  if $(\frac{P_{ML}}{P_{CSR}} > T_{ML})$ **then**
    $class \leftarrow class \cup \{ML\}$
  end if
  if $(P_{CMP} > P_{peak}$ **and** $\frac{P_{CMP}}{P_{peak}} > T_{CMP1})$ **or**
   $(P_{CMP} < P_{MB}$ **and** $\frac{P_{CMP}}{P_{CSR}} > T_{CMP2})$ **then**
    $class \leftarrow class \cup \{CMP\}$
  end if
  return $class$
end procedure

– We tune the hyperparameters $T_{CMP1/2}$,T_{ML} and T_{IMB} using grid search

Optimization auto-tuning

Formulation as a classification problem

- Classes represent performance bottlenecks
- A matrix is classified (multilabel classification)
  - We propose two classifiers
    - *profiling-based* (hand-tuned, more expensive)
    - *feature-based* (trained with machine learning, very cheap)
- Optimizations that target the detected performance bottlenecks are jointly applied
  - Runtime code generation
  - Focus on cheap CSR-based optimizations

Class	**Optimization**
MB	column index compression through delta coding
ML	software prefetching on vector x
IMB	matrix split or *auto* scheduling (OpenMP)
CMP	inner loop unrolling + vectorization

Optimization auto-tuning

Classifier A: profiling-based

- Uses the classification algorithm presented earlier
- Relies on micro-benchmarks to be run on-the-fly to estimate some of the per-class upper bounds
  - hence “profiling-based”
  - hence more expensive

Optimization auto-tuning

Classifier A: profiling-based

- Uses the classification algorithm presented earlier
- Relies on micro-benchmarks to be run on-the-fly to estimate some of the per-class upper bounds
  - hence "profiling-based"
  - hence more expensive

Can we do any better?

Optimization auto-tuning

Classifier B: feature-based (1/2)

- Uses real-valued structural features of the matrix
- Trained with supervised machine learning techniques (Decision Tree)
  - Has to be trained on the target hardware platform, offline
- Training data set
  - 215 matrices from the UF Sparse Matrix Collection
  - Not balanced in terms of class representation
- Labeling
  - profiling-based classifier
  - labels may not be accurate
- Machine-learning toolkit
  - *scikit-learn*
- Only performs feature extraction on-the-fly
  - hence “feature-based”
  - hence cheap

Optimization auto-tuning

Classifier B: feature-based (2/2)

Feature	Definition	Complexity
$size$	0:exceeds or 1:fits in LLC	$\Theta(1)$
$density$	$\frac{NNZ}{N^2}$	$\Theta(1)$
nnz_{min}	$\min\{nnz_1, \ldots, nnz_N\}$	$\Theta(N)$
nnz_{max}	$\max\{nnz_1, \ldots, nnz_N\}$	$\Theta(N)$
nnz_{avg}	$\frac{1}{N}\sum_{i=1}^{N} nnz_i$	$\Theta(N)$
nnz_{sd}	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(nnz_i - nnz_{\text{avg}})^2}$	$\Theta(2N)$
bw_{min}	$\min\{bw_1, \ldots, bw_N\}$	$\Theta(N)$
bw_{max}	$\max\{bw_1, \ldots, bw_N\}$	$\Theta(N)$
bw_{avg}	$\frac{1}{N}\sum_{i=1}^{N} bw_i$	$\Theta(N)$
bw_{sd}	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(bw_i - bw_{avg})^2}$	$\Theta(2N)$
$scatter_{\text{avg}}$	$\frac{1}{N}\sum_{i=1}^{N} scatter_i$	$\Theta(N)$
$scatter_{\text{sd}}$	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(scatter_i - scatter_{avg})^2}$	$\Theta(2N)$
$clustering_{avg}$	$\frac{1}{N}\sum_{i=1}^{N} clustering_i$	$\Theta(NNZ)$
$misses_{avg}$	$\frac{1}{N}\sum_{i=1}^{N} misses_i$	$\Theta(NNZ)$

Performance evaluation

Experimental setup

- Hardware platform
  - Intel Xeon Phi 3120P coprocessor
    - 56 cores, 224 threads
- 64-bit Linux OS
- ICC 15.0.0
- OpenMP parallel programming API
- double-precision floating-point
- 215 matrices from the UF Sparse Matrix Collection

Performance evaluation

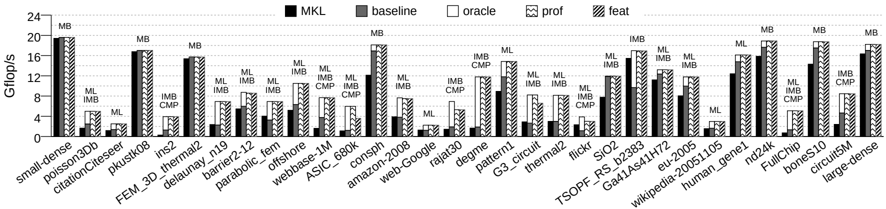
Feature-based classifier accuracy

- Leave-One-Out cross validation
  - assuming labels generated from profiling-based classifier
- *Exact Match Ratio*
  - the percentage of samples for which the predicted set of classes is fully correct
- *Partial Match Ratio*
  - the percentage of samples for which at least one prediction is correct

Features	**Complexity**	**Accuracy Exact (%)**	**Accuracy Partial (%)**
$nnz_{\{min,max,sd\}}$ bw_{avg} $dispersion_{\{avg,sd\}}$	O(N)	80	95
$size, bw_{\{avg,sd\}}$ $nnz_{\{min,max,avg,sd\}}$ $misses_{avg}$ $dispersion_{sd}$	O(NNZ)	84	100

Performance evaluation

Raw performance



- We compare to Intel MKL's `mkl_dcsrmv()`
- Significant speedups for matrices that belong to the ML/IMB classes
- Performance stability
  - No slowdowns!

Performance evaluation

Runtime overhead

Minimum number of solver iterations required to amortize cost

$$N_{iters} \gg \frac{t_{pre}}{t_{spmv} - t'_{spmv}}$$

t_{pre}: online preprocessing time

t_{spmv}: the execution time of SpMV before optimization

t'_{spmv}: the execution time of SpMV after optimization

Optimizer	$N_{iters,best}$	$N_{iters,avg}$	$N_{iters,worst}$
trivial-single	470	928	8460
trivial-combined	2062	3802	38400
profiling-based	149	297	3200
feature-based	26	62	601

Summary – Future directions

Bottleneck-oriented optimization tuning for SpMV offers

- Performance stability
  - Successfully captures diversity in sparsity patterns and hardware platforms
- Low online overhead
  - Using the feature-based classifier

Future directions

- Experiment on more platforms, e.g., GPUs
- Improve accuracy of feature-based classifier
- Expand the optimization pool

Thank you!

Questions – Discussion