Optimizing Sparse Matrix Vector Multiplication on Emerging Multicores

Orhan Kislal, Wei Ding, Mahmut Kandemir
The Pennsylvania State University
Park, Pennsylvania, USA
omk103, wzd109, kandemir@cse.psu.edu

Ilteris Demirkiran
Embry-Riddle Aeronautical University
San Diego, California, USA
demir4a4@erau.edu
Introduction

Importance of Sparse Matrix-Vector Multiplication (SpMV)

- Dominant component for solving eigenvalue problems and large-scale linear systems

- Difference from uniform / regular dense matrix computations

  - Irregular data access patterns
  
- Compact data structure
SpMV is usually in the form of \( b = Ax + b \), where \( A \) is a sparse matrix, and \( x \) and \( b \) are dense vectors. Therefore:

- Only \( x \) and \( b \) can be reused.
- One of the most common data structures for \( A \): Compressed Sparse Row (CSR) format.
**CSR format**

- Each row of A is packed one after the other in a dense array `val`
- integer array `(col)` stores the column indices of each stored element.
- `ptr` keeps track of where each row starts in `val` and `col`.

```
// Basic SpMV implementation,  // b = A*x + b, where A is in CSR and has m rows
// b = A*x + b, where A is in CSR and has m rows
for (i = 0; i < m; ++i) {
    double b = b[i];
    for (k = ptr[i]; k < ptr[i+1]; ++k)
        b+= val[k] * x[col[k]];
    b[i] = b;
}  
```
Motivation

- Computation mapping and scheduling
- Mapping assigns the computation that involves one or more rows of $A$ to a core (computation block)
- Scheduling determines the execution order of those computations
- How to take the on-chip cache hierarchy into account to improve the data locality?
If two computations share data -> better to map them to the cores that share a cache in some layer (more frequent sharing -> higher layer) **Mapping!**

For these two computations, better to let the shared data accessed by two cores in close proximity in time. **Scheduling!**
Data Reordering

- The source vector $x$ is read-only
- Ideally, $x$ can have a customized layout for each row computation $r_x$, i.e., data elements in $x$ that correspond to the nonzero elements in $r$ are placed contiguously in memory (reduce cache footprint)
- However, can we have a smarter scheme?
• Mapping (cache hierarchy-aware)
• Scheduling (cache hierarchy-aware)
• Data Reordering (seek a way to determine the minimal number of layouts for x that keep cache footprint during computation as small as possible)
Only consider the data sharing among the cores

Basic idea: for two computation blocks, higher data sharing means mapping them to higher level of cache.

We quantify the data sharing for two computation blocks as the sum of the number of nonzero elements at the same column (for those computation blocks).
Mapping Con’t

- Constructing the reuse graph
- Vertex: computation block
- Weight on an edge: the amount of data sharing
Mapping Algorithm

- SORT: Edges are sorted by their weights in a decreasing order.

- PARTITION: Vertices are visited based on the order of edges. We then hierarchically partition the reuse graph. The number of partitions is equal to the number of cache levels.

- LOOP: Repeat Step 2 until the partition for the LLC is reached. The assignment of each partition to a set of cores is based on the cache hierarchy.
Mapping-Example

(a)

(b)
Specify an order in which each row block is to be executed

Goal: ensure the data sharing among the computation blocks can be caught in the expected cache level.
SCHEDULING CON’T

- **SORT** (same as the mapping component)

- **INITIAL**: assign the logical time slot for the two nodes (vl and vr) that have the edge in between with the highest weight, and set up the offset \( o(v) \) for each vertex \( v \). \( (o(vl) = +1, o(vr) = -1) \)

- Purpose of employing offset: ensure the nodes mapped to the same core with high data sharing are scheduled to be executed as closely as possible.
SCHEDULE

**CASE 1:** $v_x$ and $v_y$ are mapped to different cores. Then assign $v_x$ and $v_y$ to be executed at the same time slot or $|T(v_x) - T(v_y)|$ is minimized.

**CASE 2:** $v_x$ and $v_y$ are mapped to the same core. If $v_x$ is already assigned, then $v_y$ will be assigned at $T(v_x) + o(v_x)$ and $o(v_y) = o(v_x)$. Otherwise, initialize $v_x$ and $v_y$ at Step 2.

**LOOP:** repeat Step 3 until all vertices are scheduled.
(a) is a portion of the reuse graph and (b) is the illustration of two schedules for v3. The first one places v3 next to v1 and the second one places v3 next to v2. Using the offset, our scheme successfully generates the first schedule instead of the second one.
Find a customized data layout for \( x \) used in each set of rows or row blocks such that the cache footprint generated by the computation of these rows can be minimized.
Case 1: \( r_1 \) and \( r_2 \) have no common nonzero elements, then \( x \) can have the same data layout for \( r_1 x \) and \( r_2 x \) (see (a))

Case 2: otherwise, assuming they have \( p \) common nonzero elements, the memory block size is \( b \), and the number of nonzero elements in \( r_1 \) and \( r_2 \) are \( n_i \) and \( n_j \), respectively. (see (b))
Experiment Setup

Intel Dunnington

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>12 cores (2 sockets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency</td>
<td>2.40GHz</td>
</tr>
<tr>
<td>L1</td>
<td>32KB, 8-way, 64-byte line size, 3 cycle latency</td>
</tr>
<tr>
<td>L2</td>
<td>3MB, 12-way, 64-byte line size, 12 cycle latency</td>
</tr>
<tr>
<td>L3</td>
<td>12MB, 16-way, 64-byte line size, 40 cycle latency</td>
</tr>
<tr>
<td>Off-Chip Latency</td>
<td>about 85 ns</td>
</tr>
<tr>
<td>Address Sizes</td>
<td>40 bits physical, 48 bits virtual</td>
</tr>
</tbody>
</table>

AMD Opteron

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>48 cores (4 sockets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency</td>
<td>2.20GHz</td>
</tr>
<tr>
<td>L1</td>
<td>64KB, full, 64-byte line size</td>
</tr>
<tr>
<td>L2</td>
<td>512KB, 4-way, 64-byte line size</td>
</tr>
<tr>
<td>L3</td>
<td>12MB, 16-way, 64-byte line size</td>
</tr>
<tr>
<td>TLB Size</td>
<td>1024 4K pages</td>
</tr>
<tr>
<td>Address Sizes</td>
<td>48 bits physical, 48 bits virtual</td>
</tr>
</tbody>
</table>

Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Structure</th>
<th>Dimension</th>
<th>Non-zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>caidaRouterLevel</td>
<td>symmetric</td>
<td>192244</td>
<td>1218132</td>
</tr>
<tr>
<td>net4-1</td>
<td>symmetric</td>
<td>88343</td>
<td>2441727</td>
</tr>
<tr>
<td>shallow_water2</td>
<td>square</td>
<td>81920</td>
<td>327680</td>
</tr>
<tr>
<td>ohne2</td>
<td>square</td>
<td>181343</td>
<td>6869939</td>
</tr>
<tr>
<td>lpl1</td>
<td>square</td>
<td>32460</td>
<td>328036</td>
</tr>
<tr>
<td>rmn10</td>
<td>unsymmetric</td>
<td>46835</td>
<td>2329092</td>
</tr>
<tr>
<td>kim1</td>
<td>unsymmetric</td>
<td>38415</td>
<td>933195</td>
</tr>
<tr>
<td>bcsstk17</td>
<td>symmetric</td>
<td>10974</td>
<td>428650</td>
</tr>
<tr>
<td>tsc_opf_300</td>
<td>symmetric</td>
<td>9774</td>
<td>820783</td>
</tr>
<tr>
<td>ins2</td>
<td>symmetric</td>
<td>309412</td>
<td>2751484</td>
</tr>
</tbody>
</table>
Experiment Setup

CON’T

- Different versions in our experiments
  - Default
  - Mapping
  - Mapping + Scheduling
  - Mapping + Scheduling + Layout
EXPERIMENTAL RESULTS

**CON’T**

Performance improvement on Dunnington

![Performance Improvement Chart](chart.png)

- **Mapping**
- **Mapping+Scheduling**
- **Mapping+Scheduling+Layout**

Mapping over Default: 8.1%
Mapping+Scheduling over Mapping: 1.8%
Mapping+Scheduling+Layout over Mapping+Scheduling: 1.7%
EXPERIMENTAL RESULTS

Performance improvement on AMD

Mapping over Default: 9.1%
Mapping+Scheduling over Default: 11%
Mapping+Scheduling+Layout over Default: 14%
THANK YOU!