Optimizing Sparse Matrix Vector Multiplication on Emerging Multicores

Orhan Kislal, Wei Ding, Mahmut Kandemir

The Pennsylvania State University 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

BACKGROUND

- SpMV is usually in the form of $\boldsymbol{b}=A\boldsymbol{x}+\boldsymbol{b}$, where A is a sparse matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are dense vectors

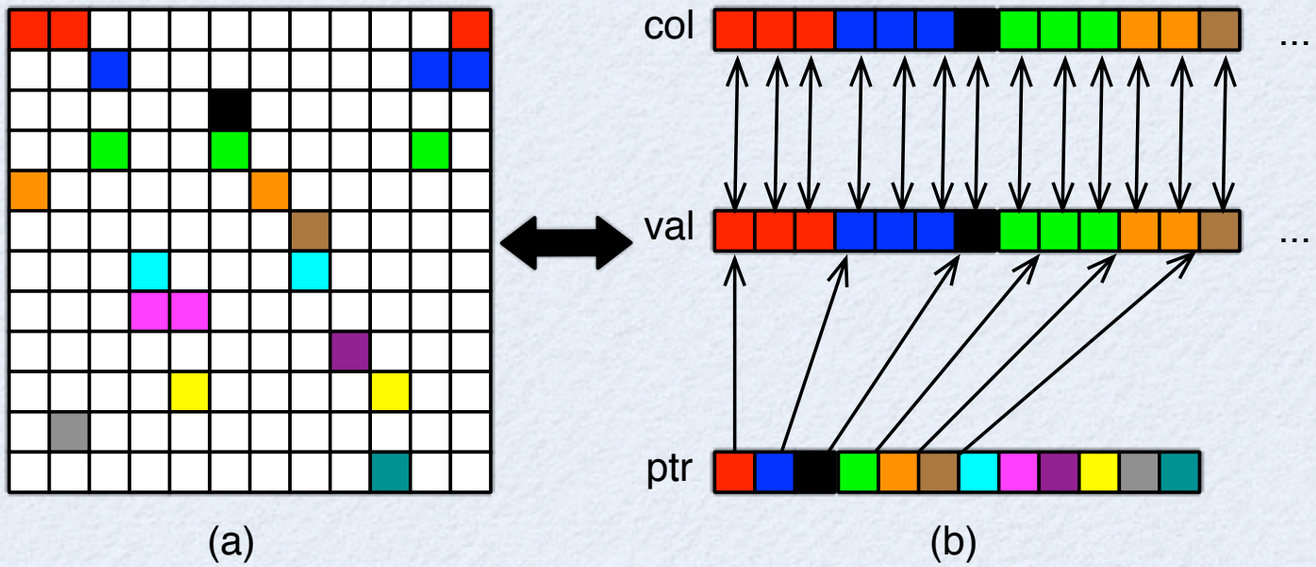
  $\boldsymbol{x}$: source vector

  $\boldsymbol{b}$: destination vector

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

BACKGROUND CON'T

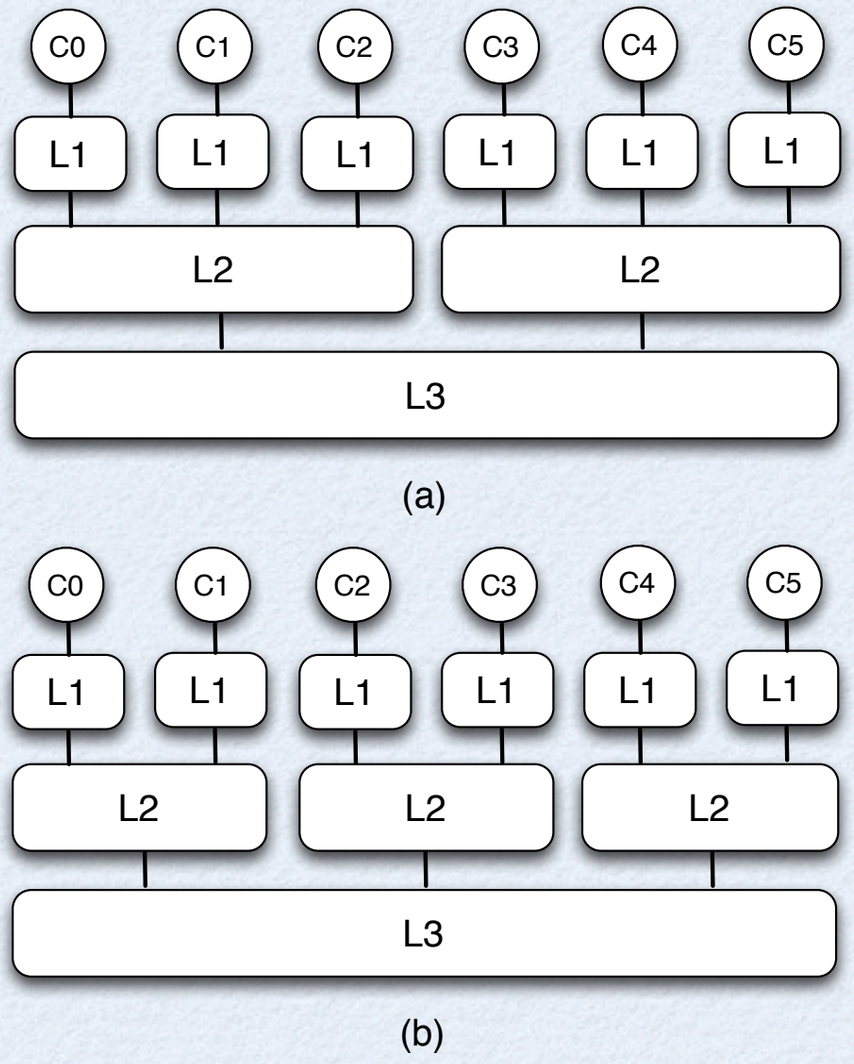
- CSR format




```
// Basic SpMV implementation,
// 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;

}
```

- 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.

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?

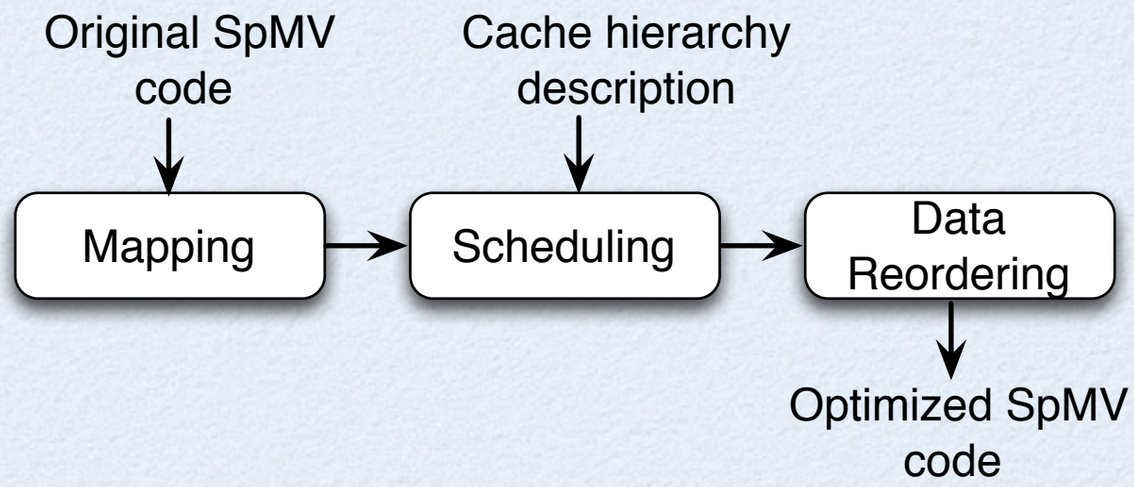
Motivation con't

- 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!

Motivation con't

- Data Reordering
  - The source vector ***x*** is read-only
  - Ideally, x can have a customized layout for each row computation ***rx***, 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?

FRAMEWORK



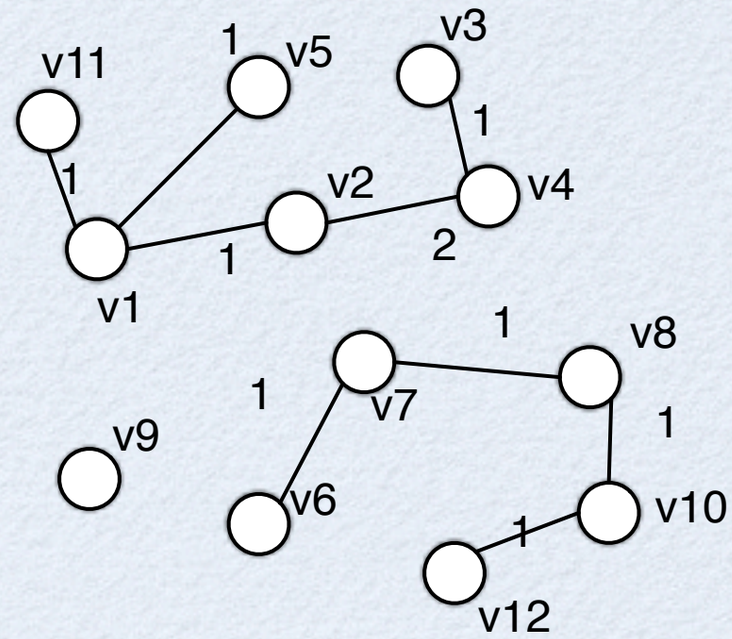
- Mapping (cache hierarchy-aware)
- Scheduling (cache hierarchy-aware)
- Data Reordering (seek a way to determine the minimal number of layouts for x that keep cache footprint during computation as small as possible)

Mapping

- 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

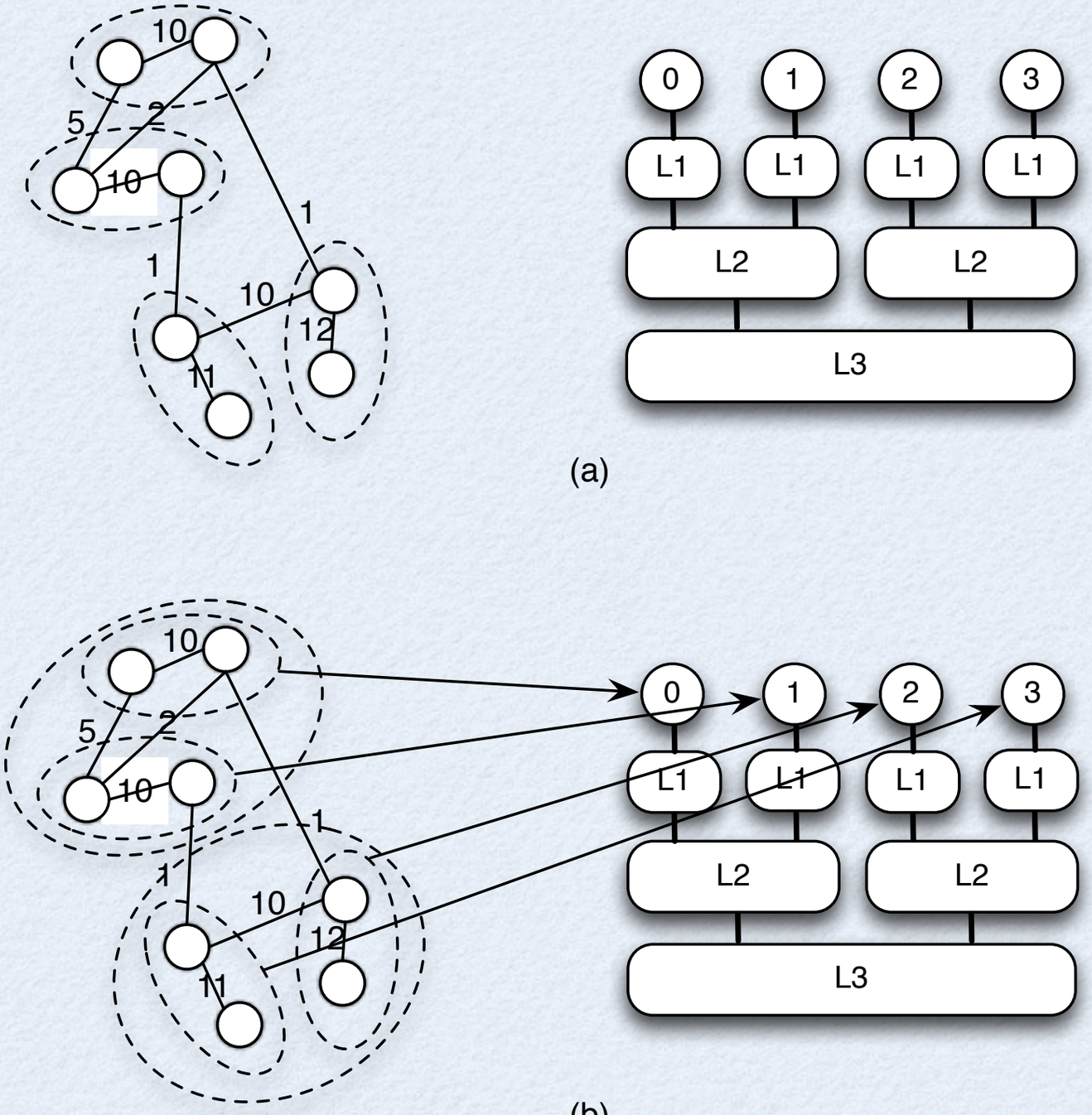
- Constructing the reuse graph
  - Vertex: computation block
  - Weight on an edge: the amount of data sharing



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



SCHEDULING

- 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.

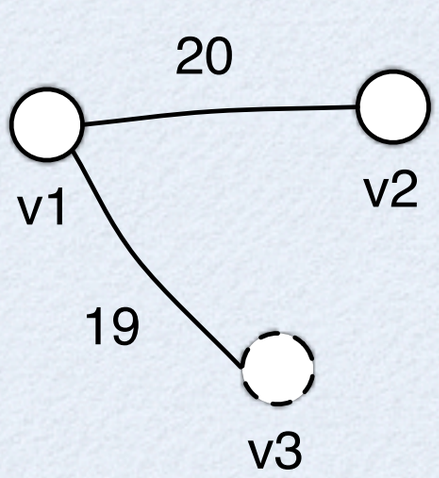
SCHEDULING CON'T

- 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.

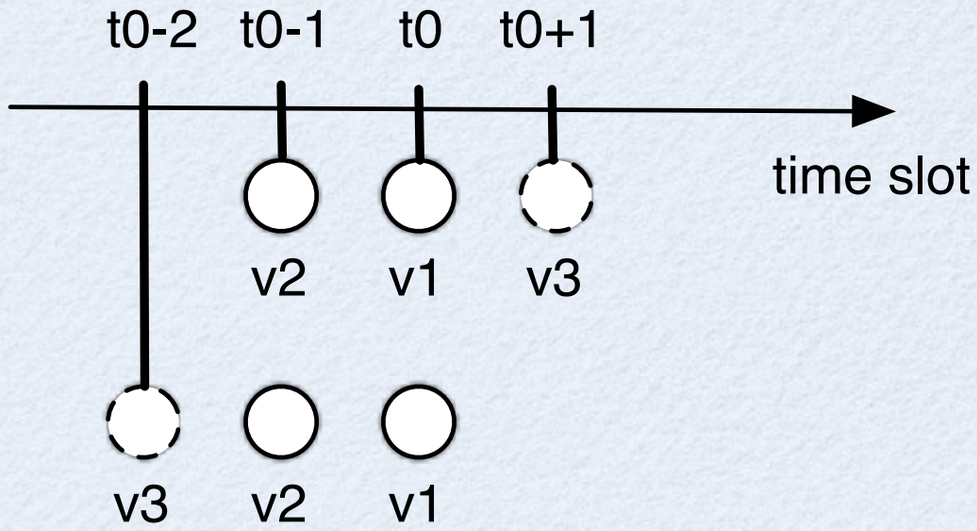
SCHEDULING CON'T

- SCHEDULE
  - CASE 1: vx and vy are mapped to different cores. Then assign vx and vy to be executed at the same time slot or $|T(vx) - T(vy)|$ is minimized
  - CASE 2: vx and vy are mapped to the same core. If vx is already assigned, then vy will be assigned at $T(vx) + o(vx)$ and $o(vy) = o(vx)$. Otherwise, initialize vx and vy at Step 2
- LOOP: repeat Step 3 until all vertices are scheduled.

SCHEDULING-EXAMPLE



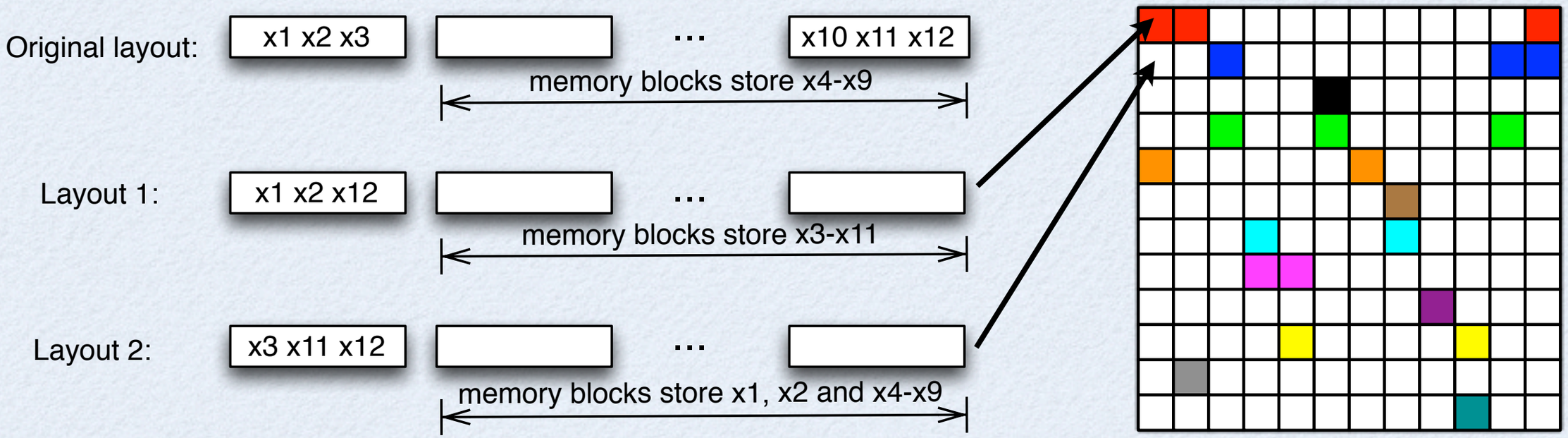
(a)



(b)

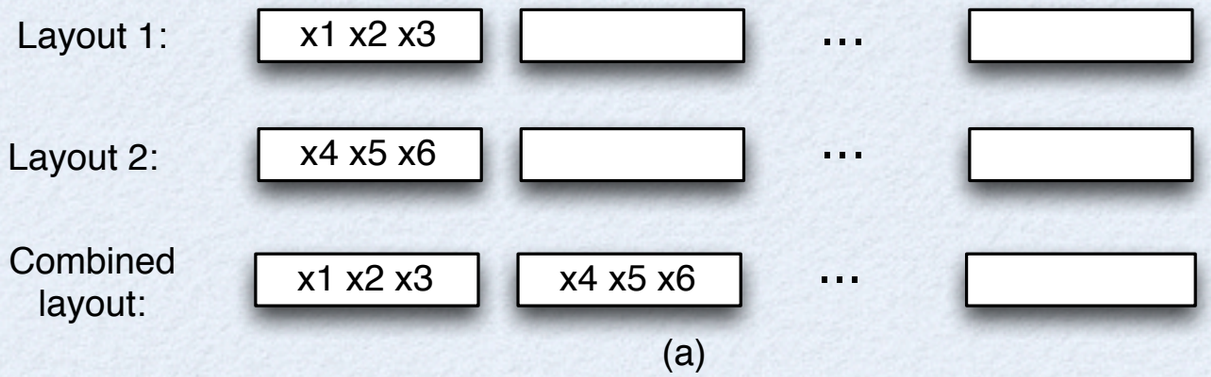
(a) is a portion of the reuse graph and (b) is the illustration of two schedules for v3. The first one places v3 next to v1 and the second one places v3 next to v2. Using the offset, our scheme successfully generates the first schedule instead of the second one.

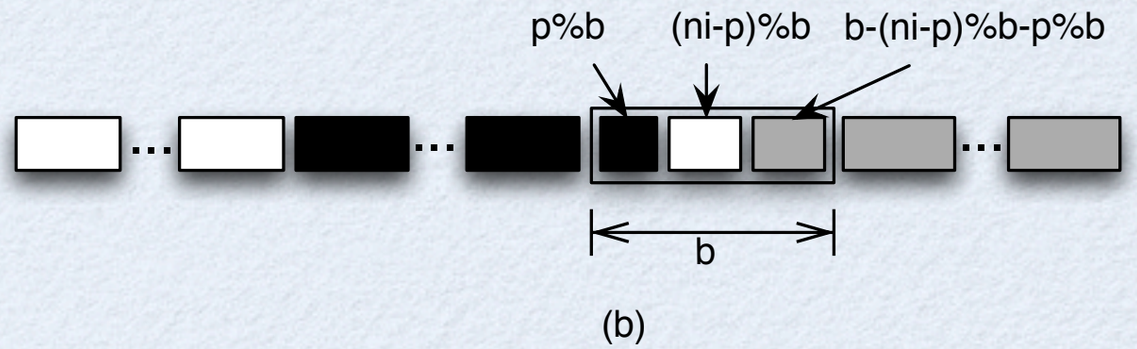
DATA REORDERING



- 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.

DATA REORDERING CON'T





- Case 1: *r1* and *r2* have no common nonzero elements, then *x* can have the same data layout for *r1x* and *r2x* (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 r1 and r2 are ni and nj, respectively. (see (b))

EXPERIMENT SETUP

Intel Dunnington

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

AMD Opteron

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

Benchmarks

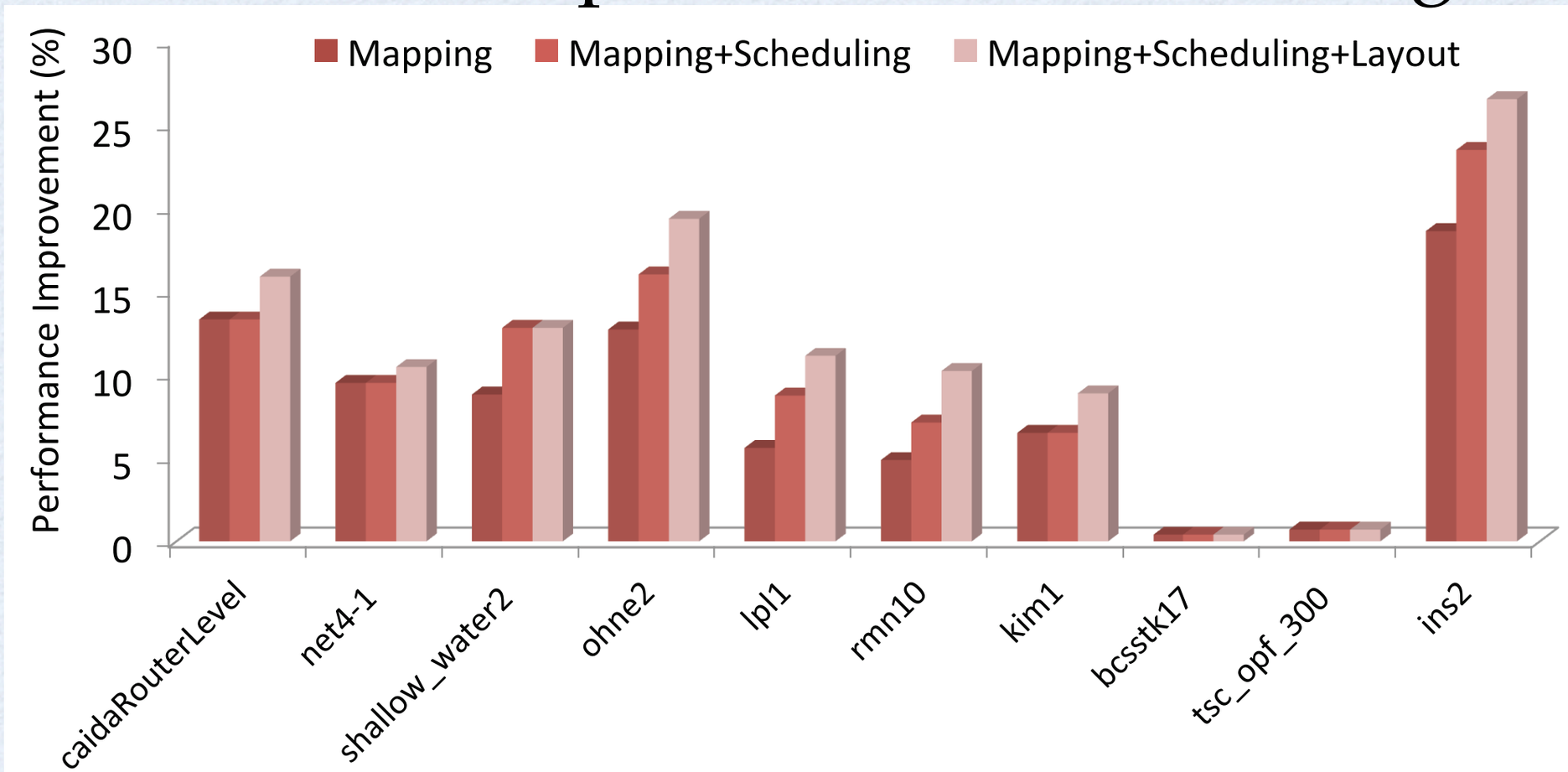
Name	Structure	Dimension	Non-zeros
caidaRouterLevel	symmetric	192244	1218132
net4-1	symmetric	88343	2441727
shallow_water2	square	81920	327680
ohne2	square	181343	6869939
lpl1	square	32460	328036
rmn10	unsymmetric	46835	2329092
kim1	unsymmetric	38415	933195
bcsstk17	symmetric	10974	428650
tsc_opf_300	symmetric	9774	820783
ins2	symmetric	309412	2751484

Experiment Setup Con't

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

EXPERIMENTAL RESULTS CON'T

Performance improvement on Dunnington



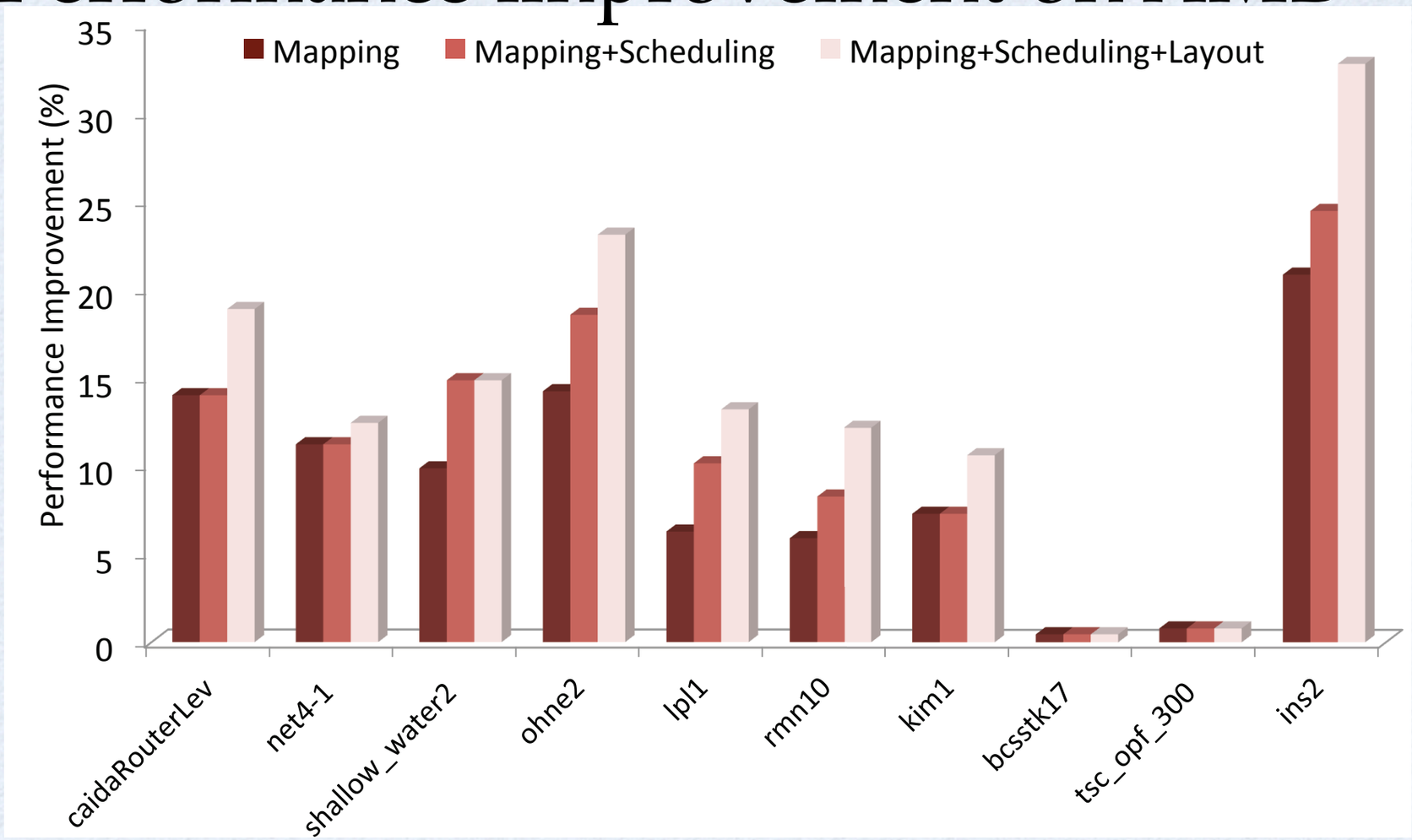
Mapping over Default: 8.1%

Mapping+Scheduling over Mapping: 1.8%

Mapping+Scheduling+Layout over Mapping+Scheduling: 1.7%

EXPERIMENTAL RESULTS CON'T

Performance improvement on AMD



Mapping over Default: 9.1%

Mapping+Scheduling over Default: 11%

Mapping+Scheduling+Layout over Default: 14%

THANK YOU!