Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors

Qingda Lu\textsuperscript{1}, Christophe Alias\textsuperscript{2}, Uday Bondhugula\textsuperscript{1}, Thomas Henretty\textsuperscript{1}, Sriram Krishnamoorthy\textsuperscript{3}, J. Ramanujam\textsuperscript{4}, Atanas Rountev\textsuperscript{1}, P. Sadayappan\textsuperscript{1}, Yongjian Chen\textsuperscript{5}, Haibo Lin\textsuperscript{6}, Tin-Fook Ngai\textsuperscript{5}

\textsuperscript{1}The Ohio State University, \textsuperscript{2}ENS Lyon – INRIA, \textsuperscript{3}Pacific Northwest National Lab, \textsuperscript{4}Louisiana State University, \textsuperscript{5}Intel Corp., \textsuperscript{6}IBM China Research Lab
The Hardware Trends

• Trend of fabrication technology
  – Gate delay decreases
  – Wire delay increases

• Trend of chip-multiprocessors (CMPs)
  – More processors
  – Larger caches
    • Private L1 cache
    • Shared last-level cache to reduce off-chip accesses
      – We focus on L2 cache
Non-Uniform Cache Architecture

• The problem: Large caches have long latencies

• The Solution
  – Employ a banked L2 cache organization with non-uniform latencies

• NUCA designs
  – Static NUCA: Mapping of data into banks is predetermined based on the block index.
  – Dynamic NUCA: Migrating data among banks. Hard to implement due to overheads and design challenges.
A Tiled NUCA-CMP
Motivating Example: Parallelizing 1D Jacobi on a 4-tile CMP

• 1-D Jacobi Code:
  
  while (condition) {
    for (i = 1; i < N-1; i++)
    for (i = 1; i < N-1; i++)
      A[i] = B[i];
  }

• If we use the standard linear array layout and parallelize the program with the “owner-computes” rule, communication volume is roughly 2N cache lines in every outer iteration.
Motivating Example: Parallelizing 1D Jacobi on a 4-tile CMP

If we divide the iteration space into four contiguous partitions and rearrange the data layout, only 6 remote cache lines are requested in every outer iteration.
The Approach

- A general framework for integrated data layout transformation and loop transformations for enhancing data locality on NUCA CMP’s
- Formalism: polyhedral model
- Two main steps:
  - Localization analysis: search for an affine mapping of iteration spaces and data spaces such that no iteration accesses any data that is beyond a bounded distance in the target space
  - Code generation: generate efficient indexing code using CLooG by differentiating the “bulk” scenario from the “boundary” case via linear constraints.
Polyhedral Model

• An algebraic framework for representing affine programs – statement domains, dependences, array access functions – and affine program transformations

• Regular affine programs
  – Dense arrays
  – Loop bounds – affine functions of outer loop variables, constants and program parameters
  – Array access functions - affine functions of surrounding loop variables, constants and program parameters
for (i=1; i<=7; i++)
    for (j=2; j<=6; j++)
        S1: \( a[i][j] = a[j][i] + a[i][j-1] \)

\[
\mathbf{F}_{S_1,a}(x_{S_1}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}
\]

\[
\mathbf{D}_{S_1}(x_{S_1}) = \begin{pmatrix} 1 & 0 & -1 \\ -1 & 0 & 7 \\ 0 & 1 & -2 \\ 0 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \end{pmatrix}
\]
Computation Allocation and Data Mapping

- Computation allocation:
  - Original iteration space \( \Rightarrow \) Integer vector representing the virtual processor space
  - A one-dimensional affine transform \((\pi)\) for a statement \(S\) is given by
    \[
    \pi_S(x_S) = C_S \cdot \begin{bmatrix} x_S \\ n \\ 1 \end{bmatrix}
    \]

- Data Mapping:
  - Original data space \( \Rightarrow \) Integer vector representing the virtual processor space
  - A one-dimensional affine transform \((\psi)\) for an array \(A\) is given by
    \[
    \psi_A(x_A) = G_A \cdot \begin{bmatrix} x_A \\ n \\ 1 \end{bmatrix}
    \]
Localized Computation Allocation and Data Mapping

• Definition: For a program \( P \), let \( D \) be its index set, computation allocation \( \pi \) and data mapping \( \psi \) for \( P \) are \textit{localized} if and only if for any array \( A \), and any reference \( F_{S,A} \),

\[
\forall \, \vec{i}, \, D_S(\vec{i}) \geq 0 \Rightarrow \left| \pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i})) \right| \leq q ,
\]

where \( q \) is a constant.

• As a special case, \textit{communication-free localization} can be achieved if and only if for any array \( A \) \textit{and any array} reference \( F_{S,A} \) in a statement \( S \), computation allocation \( \pi \) and data mapping \( \psi \) satisfy

\[
\pi_S(\vec{i}) = \psi_A(F_{S,A}(\vec{i}))
\]
Localization Analysis

- **Step 1: Group Interrelated Statements/Arrays**
  - Form a bipartite graph: vertices corresponds to statements / arrays and edges connect each statement vertex to its referenced arrays.
  - Find the connected components in the bipartite graph.

- **Step 2: Find Localized Computation Allocation / Data Mapping** for each connected component
  - Formulate the problem as finding an affine computation allocation $\pi$ and an affine data mapping $\psi$ that satisfy $\left| \pi_S(i) - \psi_{FS,A}(F_{S,A}(i)) \right| \leq q$ for every array reference $F_{S,A}$
  - Translate the problem to a linear programming problem that minimizes $q$. 
Localization Analysis Algorithm

Require: Array access functions are rewritten to access byte arrays

\[ C = \emptyset \]

for each array reference \( F_{S,A} \) do

Obtain new constraints: under \( \tilde{i} \in D_S, \pi_S(\tilde{i}) - \psi_A(F_{S,A}(\tilde{i})) + q \geq 0 \)

and \( \psi_A(F_{S,A}(\tilde{i})) - \pi_S(\tilde{i}) + q \geq 0 \)

Apply Farkas Lemma to new constraints to obtain linear constraints; eliminate all Farkas multipliers

Add linear inequalities from the previous step into \( C \)

Add objective function (\( \min q \))

Solve the resulting linear programming problem with constraints in \( C \)

if \( \psi \) and \( \pi \) are found then return \( \pi, \psi, \) and \( q \)

else return “not localizable”
Data Layout Transformation

• Strip-mining
  – An array dimension $N \rightarrow$ two virtual dimensions $\left(\left\lfloor \frac{N}{d} \right\rfloor, d \right)$
  – Array reference array reference $[\ldots][i][\ldots] \rightarrow [\ldots][i \div d][i \mod d][\ldots]$

• Permutation
  – Array $A(..., N1, ..., N2, ...) \rightarrow A'(..., N2, ..., N1, ...)$
  – Array reference $A[\ldots][i1][\ldots][i2][\ldots] \rightarrow A[\ldots][i1][\ldots][i2][\ldots]$. 
Data Layout Transformation (Cont.)

The 1D jacobi example:
Combination of strip-mining and permutation
Data Layout Transformation (Cont.)

• Padding:
  – **Inter-array padding**: keep the base addresses of arrays aligned to a tile specified by the data mapping function
  – **Intra-array padding**: align elements inside an array with “holes” to make a strip-mined dimension divisible by its sub-dimensions
Data Layout Transformation (Cont.)

With localized computation allocation and data mapping:

• If we find communication-free allocation with all arrays share the same stride along the fastest varying dimension, only padding is applied

• Otherwise we create a blocked view of n-dimensional array $A$ along dimension $k$ with data layout transformations
  
  – If $K=1$, we have data layout transformation $\sigma(i_n, i_{n-1}, \ldots, i_2, i_1) = (i_n, i_{n-1}, \ldots, i_2, (i_1 \mod (N_1/P))/L, i_1/(N_1/P), i_1 \mod L)$, $L$ is cache line size in elements and $P$ is processor number.
  
  – If $K>1$, we have data layout transformation, $\sigma(i_n, i_{n-1}, \ldots, i_2, i_1) = (i_n, \ldots, i_1/L, i_k \mod (N_k/P), \ldots, i_2, i_k/(N_k/P), i_1 \mod L)$. 
Code Generation

• It is challenging to generate efficient code
  – Replacing array reference $A[u(i)]$ with $A'[\sigma(u(i))]$ produces very inefficient code

• We iterate directly in the data space of the transformed array.
  – In the polyhedral model, we specify an iteration domain $D$ and an affine scheduling function for each statement.
  – Efficient code is then generated by CLooG that implements the Quilleré-Rajopadhye-Wilde algorithm.
Code Generation (Cont.)

Key techniques used in generating code accessing layout transformed arrays

• While $\sigma$ is not affine, $\sigma^{-1}$ is. We replace constraints $j_k = \sigma(u_k(i))$ by $\sigma^{-1}(j_k) = i$.

• We separate the domain $D$ into two sub-domains $D_{steady}$ and $D_{boundary}$ and generate code for two domains.
  – $D_{steady}$ covers the majority of $D$ by simplifying constraints such as $(i-u) \ mod \ L$ to $(i \ mod \ L) - u$
  – Boundary cases are automatically handled in $D_{boundary} = D - D_{steady}$
Experimental Setup

• With the Virtutech Simics full-system simulator extended with a timing infrastructure GEMS, we simulated a tiled NUCA-CMP:

<table>
<thead>
<tr>
<th>Processor</th>
<th>16 4-way, 4GHz in-order SPARC cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>private, 64KB I/D cache, 4-way, 64-byte line, 2-cycle access latency</td>
</tr>
<tr>
<td>L2 cache</td>
<td>shared, 8MB unified cache, 8-way, 64-byte line, 8-cycle access latency</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB, 320-cycle (80ns) latency, 8 controllers</td>
</tr>
<tr>
<td>On-chip network</td>
<td>4x4 mesh, 5-cycle per-link latency, 16GB bandwidth per link</td>
</tr>
</tbody>
</table>

• We evaluated our approach with a set of data-parallel benchmarks
  – Focus of the approach is not on finding parallelism
Data Locality Optimization for NUCA-CMPs: The Results

![Graph showing speedup for various tasks: Sum, Mv, Demosaic, Convolve, Life, 1D Jacobi, 2D Jacobi, Skeleton. Speedup ranges from 0 to 3.]
Data Locality Optimization for NUCA-CMPs: The Results

Link Utilization

- Sum
- Mv
- Demosaic
- Convolve
- Life
- 1D Jacobi
- 2D Jacobi
- Skeleton
Data Locality Optimization for NUCA-CMPs: The Results

Remote L2 Access Number

- Sum
- Mv
- Demosaic
- Convolve
- Life
- 1D Jacobi
- 2D Jacobi
- Skeleton

The graph shows the remote L2 access number for different tasks. The tasks are ordered from the highest to the lowest access number.
Conclusion

• We believe that the approach developed in this study provides very general treatment of code generation for data layout transformation
  – analysis of data access affinities
  – automated generation of efficient code for the transformed layout