Optimization by Run-time Specialization for Sparse-Matrix Vector Multiplication

Maria J. Garzaran
University of Illinois at Urbana-Champaign

Joint work with Sam Kamin (UIUC) and Baris Aktemur (Ozyegin University, Turkey)
Why autotuning?

• Computers have complex hardware
  – e.g., branch prediction, hardware prefetch, multicores, …
• Generating highly efficient code is difficult
  – Compilers
    • Do not produce expected results
  – Manual
    • Increase cost
    • Low performance if not enough resources
• Must automate tuning
Compilers

• Compilers have limitations
  – Lack semantic information
    • fewer choices
  – Must target all class of applications
  – Must be reasonably fast
Compiler vs. Manual Tuning

Matrix Matrix Multiplication

![Graph showing performance comparison between compiler and manual tuning]

7x

Intel MKL

- MFLOPS
- Matrix Matrix Multiplication


[Forthcoming paper] On the Effectiveness of OpenACC compilers. Joint project with Swapnil Ghike, Ruben Gran, and David Padua.
What is Autotuning?

• Strategy to generate highly efficient codes that adapt to
  – the target platform
  – input set

• Needs multiple versions of a given program
  – different compiler optimizations
  – different algorithms
  – different parameter values for a given algorithm

• Typically uses empirical search (executes each version in the target architecture) and select the fastest
  – can search using analytical models
Outline

- Input-independent
- Input-dependent
  - Pure algorithms
  - Specialization and Runtime code generation
Outline

- Input-independent
- Input-dependent
  - Pure algorithms
  - Specialization and Runtime code generation
Input-dependent performance

• Performance depends on the characteristics of the input:
  – frequent pattern mining
  – sorting
  – sparse matrix-vector(s) multiplication
  – graph problems
Input-dependent Autotuning

Training Phase

Training Data Sets

Empirical Evaluation

Machine Learning

Selection Strategy

Algo Pool

Metrics
Input-dependent Autotuning

Training Phase

- Training Data Sets
- Empirical Evaluation
- Machine Learning
- Selection Strategy

Execution Phase

- Input
- Metrics
- Selection Strategy
- Best Algo
Outline

• Input-independent
• Input-dependent
  – Pure algorithms
    • Sorting
  – Specialization and Runtime code generation
Different Algorithms for Sorting

• Sorting is another interesting problem for autotuning
  – The compiler can do little about sorting
  – Performance depends on machine and input data characteristics.

• No single algorithm is the best for all input and platforms

[CGO’04][CGO’5]
Joint project with Xiaoming Li and David Padua
Different Algorithms for Sorting

Performance (keys per cycle)

Standard Deviation of the input data

- Intel Xeon
- AMD Athlon MP

Different Algorithms:
- CC-Radix
- Quicksort
- Merge Sort

Same input, different performance
Selecting the best algorithm

IBM Power3

[CGO0’04]

Adaptive Sort

Performance (keys per cycle)

Standard Deviation

Quicksort  CC-Radix  Merge Sort  Adaptive Sort
Outline

• Input-independent

• Input-dependent
  – Pure algorithms
  – Specialization and Runtime code generation
    • Sparse Matrix-Vector Multiplication
Input-Dependent Autotuning

Training Phase

- Training Data Sets
- Empirical Evaluation
- Machine Learning
- Selection Strategy

Execution Phase

- Input
- Metrics
- Selection Strategy
- Best Algo

Code for the Best Algorithm can be generated at runtime using specialization and runtime code generation.
Sparse Matrix-Vector Multiplication

• $y = Ax$, $A$ sparse but known

• $y = Ax$ is performed many times
  – Justifies one-time tuning effort

We generate code that is specialized for the matrix $A$
Code is specialized for the location of the nonzeros in $A$
Sparse Matrix-Vector Multiplication

\[ A = \begin{bmatrix}
  b & c & . & . \\
  . & a & . & . \\
  c & . & b & b \\
  . & . & a & . \\
\end{bmatrix} \]

value

\[ \begin{bmatrix}
  b & c & a & c & b & b & a \\
\end{bmatrix} \]

col_idx

\[ \begin{bmatrix}
  0 & 1 & 1 & 0 & 2 & 3 & 3 \\
\end{bmatrix} \]

row

\[ \begin{bmatrix}
  0 & 2 & 3 & 6 & 7 \\
\end{bmatrix} \]
Sparse Matrix Vector Multiplication (CSR)

Matrix A

/* loop over rows */
for (i=0; i<m; i++) {
    double y_i = y[i];
    /* loop over non-zero elements in each row */
    for (jj = row_start[i]; jj < row_start[i+1];
        jj++, col_idx++, value++){
        y_i += value[0]*x[col_idx[0]];
    }
    y[i] = y_i;    // indirect array accessing
}
Specialization 1: Number of Zeros

Matrix A

/* loop over rows */
for (i=0; i<m; i++) {
    double y_i = y[i];
    /* loop over non-zero elements in each row */
    for (jj = row_start[i]; jj<row_start[i+1]; jj++, col_idx++, value++){
        y_i += value[0]*x[col_idx[0]];
    }
    y[i] = y_i;  // indirect array accessing
}
Specialization 1: Number of Zeros

Matrix A

- Still indirect access
- Unrolled loop

```c
4 rows with 2 non-zeros

for (i=0; i<4; i++) {
    row = rows[a++];
    y[row] += value[b]*x[col_idx[b]];
    y[row] += value[b+1]*x[col_idx[b+1]];
    y[row] += y;
    b += 2;  \ completely unrolled inner loop
}

for (i=0; i<2; i++) {
    row = rows[a++];
    y[row] += value[b]*x[cols[b]];
    b += 1;
}
```
Specialization 2: Stencils

- Specialize code based on location of the non-zeros with respect to the diagonal

```c
int stencil_1[1,2];
for (i=0; i<2; i++) {
    row = stencil_1[i];  // no indirect access
    xx=x+row;
    w[row] += value[0]*xx[0];
    w[row] += value[1]*xx[1];
    w[row] += value[2]*xx[2];
    b += 3;
}
for (i=0; i<4; i++) {
    w[row] += value[0]*xx[1];
    b += 1;
}
```

- No indirect access
- Unrolled loop

Matrix A
Specialization 3: Unfolding

- Straight-line code that performs all the floating-point operations.

\[
\begin{align*}
\text{w}[1] & \text{ } + = \text{ value}_{1,2} \times \text{v}[1] \\
\text{w}[2] & \text{ } + = \text{ value}_{2,2} \times \text{v}[2] + \text{value}_{2,3} \times \text{v}[3] + \\
& \hspace{1cm} \text{value}_{2,4} \times \text{v}[4]; \\
\text{w}[3] & \text{ } + = \text{ value}_{3,3} \times \text{v}[3] + \text{value}_{3,4} \times \text{v}[4] + \\
& \hspace{1cm} \text{value}_{4,5} \times \text{v}[5]; \\
\text{w}[4] & \text{ } + = \text{ value}_{4,4} \times \text{v}[4]; \\
\end{align*}
\]

Code size is proportional to the number of non-zeros
Sparse Matrix-Vector Multiplication

• Many methods can be used
  – plain CSR
  – Unroll \{i\}
  – OSKI
  – Diagonal
  – NonZeros
  – Stencil
  – Unfolding
  – GenOSKI
  – Compressed Format

off-line

runtime code generation
Experimental results

• Have run experiments on 3 different platforms:
  – loome2: Intel Core i7
  – loome3: Intel Core i5
  – i2pc3: Intel Xeon E7

• Codes are compiled using clang with –O3.
• Codes run in parallel using OpenMP, 4 threads
• 53 matrices from Matrix Market and University of Florida Sparse Matrix Collection
Overall Speedups

The table shows average speedups with respect to MKL running in parallel with 4 threads.

<table>
<thead>
<tr>
<th></th>
<th>Loome2</th>
<th>Loome3</th>
<th>i2pc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>1.72</td>
<td>1.45</td>
<td>1.53</td>
</tr>
<tr>
<td>NonZeros</td>
<td>23 (1.18)</td>
<td>23 (1.31)</td>
<td>9 (1.45)</td>
</tr>
<tr>
<td>Stencil</td>
<td>18 (1.56)</td>
<td>14 (1.51)</td>
<td>18 (1.23)</td>
</tr>
<tr>
<td>Unfolding</td>
<td>9 (2.33)</td>
<td>10 (1.91)</td>
<td>20 (1.92)</td>
</tr>
<tr>
<td>GenOsKi</td>
<td>2 (1.14)</td>
<td>6 (1.27)</td>
<td>-</td>
</tr>
<tr>
<td>Compressed</td>
<td>1 (1.04)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

53 53 47
Specialization methods have higher speedups

<table>
<thead>
<tr>
<th>Machine Matrix</th>
<th>n</th>
<th>nz</th>
<th>Loome2</th>
<th>Loome3</th>
<th>i2pc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>torso2</td>
<td>115K</td>
<td>1M</td>
<td>Stencil</td>
<td>1.72</td>
<td>Stencil</td>
</tr>
<tr>
<td>fidap011</td>
<td>16K</td>
<td>1M</td>
<td>CSR4</td>
<td>1.04</td>
<td>GenOski</td>
</tr>
<tr>
<td>s3dkq4m2</td>
<td>90K</td>
<td>2,2M</td>
<td>Stencil</td>
<td>1.55</td>
<td>Stencil</td>
</tr>
<tr>
<td>engine</td>
<td>143K</td>
<td>2,4M</td>
<td>Unfolding</td>
<td>3.31</td>
<td>Unfolding</td>
</tr>
</tbody>
</table>

Method Speedup over MKL
Specialization methods have higher speedups

<table>
<thead>
<tr>
<th>Machine Matrix</th>
<th>n</th>
<th>nz</th>
<th>Loome2</th>
<th>Loome3</th>
<th>i2pc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>torso2</td>
<td>115K</td>
<td>1M</td>
<td>Stencil</td>
<td>1.72</td>
<td>1.46</td>
</tr>
<tr>
<td>fidap011</td>
<td>16K</td>
<td>1M</td>
<td>NonZeros</td>
<td>1.04</td>
<td>GenOski</td>
</tr>
<tr>
<td>s3dkq4m2</td>
<td>90K</td>
<td>2,2M</td>
<td>Stencil</td>
<td>1.55</td>
<td>Stencil</td>
</tr>
<tr>
<td>engine</td>
<td>143K</td>
<td>2,4M</td>
<td>Unfolding</td>
<td>3.31</td>
<td>Unfolding</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th># Stencils</th>
<th>#Patterns</th>
<th>#Distinct Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>torso2</td>
<td>3,148</td>
<td>81</td>
<td>806653</td>
</tr>
<tr>
<td>fidap011</td>
<td>7,432</td>
<td>1684</td>
<td>211503</td>
</tr>
<tr>
<td>s3dkq4m2</td>
<td>1,131</td>
<td>380</td>
<td>74282</td>
</tr>
<tr>
<td>engine</td>
<td>84,195</td>
<td>108</td>
<td>1</td>
</tr>
</tbody>
</table>
How to predict the best method?

if (#nz || #distinct values) is small
    Unfolding
else if #stencils is small
    Stencil
    else if #patterns is small
        GenOSKI
    else NonZeros
Status of the project

• Ongoing work
  – Designing more methods for code specialization
    • Compressed formats, hybrids?
  – Need fast runtime code generation
    • Currently using LLVM
    • Parallel code generation
  – Need to select the best specialization method
    • Identify the metrics to select the best method
Summary - I

- Code Specialization can produce speed-ups for most matrices and machines
- Novel way to autotune codes that adapt to the input:
  1. Algorithm selection
  2. Specialization and Runtime Code Generation
- Autotuning can be done in many problem domains
Summary - II

Autotuning produces:

+ highly efficient codes that adapt to the target machine and input
+ codes have longer life span,

but

- requires deep understanding of the algorithms
- codes are more difficult to debug
Questions?

garzaran@illinois.edu