Code Generation and Autotuning in Computer Algebra

Jeremy Johnson, Werner Krandick, David Richardson
Department of Computer Science
Drexel University, USA

Anatole Ruslanov
Department of Computer and Information Sciences
SUNY Fredonia, USA

ACA 2009, Montreal, Canada
Motivation

- Automatic analysis of empirical performance data can lead to significant performance gains
- Computer architecture today is
  - Highly efficient and complex
  - Often proprietary/trade secret
  - Evolves quickly
  - Difficult to model
- Objective: automatically generate and test many implementations (live or at installation).
Automatic generation & tuning - how?

- High-performance depends on
  - The algorithm → automatically generate and test several/many/all
  - The platform architecture* → iterate on generating and testing with many parameters

- The optimal code/algorith/parameters are determined via runtime experiments

*Pipeline organization, number of registers, integer units, cache and memory hierarchy organization, etc.
Tricky questions (can’t model well)

- How is the pipeline organized?
  - Branch misprediction handling
  - Instruction prefetching, issue, reordering
- How is cache organized?
  - How well does it prefetch? How many ports?
- How many integer units are there?
  - How well can they be engaged in parallel?
- How do compilers use the CPU registers?
- What happens when the code is compiled on one machine but run on another?
Apply to computer algebra

- Automatic code generation and tuning techniques may be applied to symbolic computation and computer algebra systems.

- In this talk, we present an example that demonstrate benefits of these techniques.

- We show that the performance of the Taylor shift operation used in real root isolation can be substantially improved through automatic code generation and tuning.
Classical Taylor shift by 1

Let $A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$

Pascal’s triangle with inputs: $a_n, \ldots, a_0$, and $0, \ldots, 0$

Each element is the sum of its top and left neighbors

$B(x) = A(x+1)$
Traditional computation

Sequence of n addition passes

Input: \( A(x) = a_n x^n + \ldots + a_0 \)
for \( i = 0, \ldots, n-1 \)
for \( k = n-1, \ldots, i \)
\[ a_k \leftarrow a_k + a_{k+1} \]
Output: \( B(x) = a_n x^n + \ldots + a^0 \)

Straightforward methods:
function calls to integer addition
Taylor shift by 1 algorithm redesign

- Performance depends on addition
- Minimize cycles per word addition
  - by reducing memory traffic
  - by removing most carry computations

- Arithmetic ideas:
  - signed digits
  - suspended normalization
  - radix reduction
  - delayed carry propagation
Tiling improves data locality

Sequence of addition passes within each tile.

Force active data structures to have a small memory footprint.
Register tile avoids memory traffic

Key idea: avoid reads by keeping all digits in registers.

- Do additions for the $i$-th order digits only
  - Read coefficient digits
  - Read temporary values
  - Do additions in registers
  - Store back to L1 cache
- No carry propagation
Delayed carry propagation

- Reduce radix to prevent overflow and absorb carries during register tile computation
Schedule register tile to improve instruction-level parallelism (ILP)

- Assist the compiler with scheduling by grouping additions.
- Example pictured is 4x4 register tile.
- The 16 additions consume about 10 cycles on any 2 IEU CPU.
- We did not try scheduling for 3 or more IEU.
Speedup relative to straightforward method

![Graph showing speedup relative to degree](image-url)
Automatic code generation and tuning for Taylor Shift computation

- Each register tile computation is defined / influenced by
  - The tile size
  - A number of parallel additions

- Today’s compilers still need to receive fully unrolled code for best performance
Automatic code generation and tuning for Taylor Shift computation

- We wrote Perl-based code generator that
  - Consists of ~ 1000 lines of code
  - Unrolls the loops
  - Uses performance counters for assessment
  - Selects best tile size automatically

- Then we played with the generator!
Code generator worked hard!

<table>
<thead>
<tr>
<th>Square tile size</th>
<th>Lines of code generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1,124</td>
</tr>
<tr>
<td>6</td>
<td>1,876</td>
</tr>
<tr>
<td>8</td>
<td>3,044</td>
</tr>
<tr>
<td>10</td>
<td>4,724</td>
</tr>
<tr>
<td>12</td>
<td>7,012</td>
</tr>
<tr>
<td>14</td>
<td>10,004</td>
</tr>
<tr>
<td>16</td>
<td>13,796</td>
</tr>
<tr>
<td>Total</td>
<td>41,580</td>
</tr>
</tbody>
</table>
Impact of register tile size on performance
Architecture: AMD Opteron

![Graph of speedup vs. degree for different register tile sizes, showing performance variations across degrees for tiles of 4x4, 6x6, 8x8, 10x10, 12x12, 14x14, and 16x16.]
## Processor architectures

<table>
<thead>
<tr>
<th>processor</th>
<th>word-length</th>
<th>registers</th>
<th>IEUs</th>
<th>cache assoc.</th>
<th>optimal tile-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium4</td>
<td>32</td>
<td>8</td>
<td>2x2</td>
<td>8-way</td>
<td>6x6</td>
</tr>
<tr>
<td>UltraSPARC III</td>
<td>64</td>
<td>32</td>
<td>2</td>
<td>4-way</td>
<td>8x8</td>
</tr>
<tr>
<td>Pentium EE</td>
<td>64</td>
<td>16</td>
<td>2x2</td>
<td>8-way</td>
<td>12x12</td>
</tr>
<tr>
<td>Opteron</td>
<td>64</td>
<td>16</td>
<td>3</td>
<td>4-way</td>
<td>12x12</td>
</tr>
</tbody>
</table>
Summary

- Improved performance through automatic code generation and tuning!
  - Modeling is difficult
  - Invent new implementations
  - Spoon-feed the compilers
  - Automatically experiment/test
  - Choose the best!
Thank you! / Merci!

Questions?