Compiler-Assisted Performance Tuning

Mary Hall

July 10, 2007

* This work has been sponsored by the NSF NGS and CSR programs, Intel Corporation, and the DOE SciDAC as part of the Performance Engineering Research Institute (PERI).
Collaborators

• Compiler group:
  - Jacqueline Chame (research scientist)
  - Chun Chen (postdoctoral researcher)
  - Spundun Bhatt (programmer)
  - Yoonju Lee Nelson, Muhammad Murtaza, Melina Demertzi (Phd students)

• ISI collaborators:
  - Ewa Deelman, Yolanda Gil, Kristina Lerman, Robert Lucas

• Alumnus collaborator:
  - Jaewook Shin (Argonne)

• Other USC collaborators:
  - Rajiv Kalia, Aiichiro Nakano, Priya Vashishta
Goal for this Workshop

- Discuss role of compiler technology in various tuning efforts
  - Application programmer assistant
  - Library developer assistant (e.g., intelligent code generation and search)
  - Fully automatic tuning (next talk)

- Our interests
  - New applications
  - New architectures
  - Other collaborations
Key Research Themes

• Compiler-based performance tuning tools
  - Use vast resources of petascale systems
  - Enumerate options, generate code, try, measure, record (conceptually)

• Optimizing compilers built from modular, understandable chunks
  - Easier to bring up on new platforms
  - Facilitates collaboration, moving the community forward

A Systematic, Principled Approach!
Outline

1. Motivation
2. Approach & potential of compiler-assisted tuning
   - New flexible and systematic compiler technology
   - Scenarios from application tuning
   - Automatic performance tuning
3. Overview of results (more in next talk)
• Long-term goal is to automate the process of tuning software to maximize its performance.

• Reduces performance portability challenge for computational scientists.

• Addresses the problem that performance experts are in short supply.

• Builds on forty years of human experience and recent success with linear algebra libraries.

Slide source: Bob Lucas and David Bailey

PERI automatic tuning framework
A New Kind of "Compiler"

Traditional view:

- Code
- Input data

Batch Compiler
Performance Tuning “Compiler”

transformation script(s)

Experiments Engine

code

input data (characteristics)

search script(s)

Code Translation
1. Programmer expresses application-level parameters and input data set properties. (ref. Active Harmony and Rose compiler)
Scenario 1: Application-Level Parameters

- Programmer expresses parameters to be searched, input data set (e.g., Visualization of MD Simulation)

- Tools automatically generate code and evaluate tradeoff space of application-level parameters

Parameter: cellSize, range = 48:144, step 16

```plaintext
ncell = boxLength/cellSize
for i = 1, ncell
    /* perform computation */
```

Const: cellSize = 48

```plaintext
ncell = boxLength/48
for i = 1, 48
    /* perform computation */
```
2. Application programmer interacts with compiler to guide optimization.
Scenario 2: Programmer-guided Transformations

- Application programmer has written code variants for every possible unroll factor of two innermost loops
- Straightforward for compiler to generate this code and test for best version


CScADS, July 2007
High-Level Concept: Exploit what compilers do well

- Complex translation and transformation (rewriting rules)
- Domain knowledge of optimizations and optimization impact
- Analyze code to derive “features”
- Source-to-source
  - Rely on investment in backend native compilers to achieve ILP
Model-guided empirical optimization (our autotuning)

- **Model-guided optimization**
  - Static models of architecture, profitability

- **Empirical optimization**
  - Empirical data guide optimization decisions
  - ATLAS, PhiPAC, FFTW, SPIRAL etc.

- **Exploit complementary strengths of both approaches**
  - Compiler models prune unprofitable solutions
  - Empirical data provide accurate measure of optimization impact

**Goal:** Hand-tuned levels of performance from compiler-generated code for loop-based computation that is portable to new architectures.
Automatic Performance Tuning
(Model-Guided Empirical Optimization)

Phase 1:
- Application code
- Code variant generation
- Set of parameterized code variants + constraints on unbound parameters
- Analysis/models
- Transformation modules

Phase 2:
- Search engine
- Performance monitoring support
- Optimized code + representative input data set
- Optimized code
- Execution environment
- Architecture specification
Transformation Framework

- Uniform representation of transformations
- Direct mapping from transformation representation to generated code
- Mostly independent of compiler infrastructure

⇒ Straightforward to name alternative code variants and generate code, useful for search
foreach memory hierarchy level $M$
select unmarked data structure $D$ and loop $L$
s.t. $D$ has maximum reuse, carried by $L$
if (level == register)
make $L$ innermost and unroll $L$
else {
permute & tile $L$
generate copy variant if profitable
}
determine constraints based on $D$ and $M$
mark $D$

do $k=1,n-1$
do $i=k+1,n$
da($i,k$) = $a(i,k)/a(k,k)$
do $i=k+1,n$
do $j=k+1,n$
da($i,j$) = $a(i,j) - a(i,k)*a(k,i)$

transformations

- original iteration space
$s1 = \{ [k,i,j]: 1 <= k <= n-1 \land k+1 <= i <= n \land j = k+1 \}$
$s2 = \{ [k,i,j]: 1 <= k <= n-1 \land k+1 <= i <= n \land k+1 <= j <= n \}$

- permute loops $k$ and $j$
t1 $: = \{ [k,i,j] \rightarrow [0, j, 0, i, 0, k, 0] \}$
t2 $: = \{ [k,i,j] \rightarrow [0, j, 0, i, 1, k, 0] \}$

- tile loops
t1 $: = \{ [k,i,j] \rightarrow [0, jj, 0, kk, 0, j, 0, i, 0, k, 0] :$
$jj = 2 + 16r \land \& \& kk = 1 + 128\alpha \land \& \& i-15, 2 <= ii <= i$
$\& \& kk-127, 1 <= kk <= k \}$
t2 $: = \{ [k,i,j] \rightarrow [0, jj, 0, kk, 0, j, 0, i, 1, k, 0] :$
$jj = 2 + 16r \land \& \& kk = 1 + 128\alpha \land \& \& i-15, 2 <= ii <= i$
$\& \& kk-127, 1 <= kk <= k \}$

dependence analysis

reuse analysis

register model

cache model

analysis and models
Transformed Code for LU
(Automatically Generated)

```
REAL*8 P1(32,32),P2(32,64),P3(32,32),P4(32,64)
OVER1=0
OVER2=0
DO T2=2,N,64
  IF (66<=T2)
    DO T4=2,T2-32,32
      DO T6=1,T4-1,32
        DO T8=T6,MIN(T4-1,T6+31)
          P1(T8-T6+1,T10-T4+1)=A(T10,T8)
        DO T8=T2,MIN(T2+63,N)
          DO T10=T6,MIN(T6+31,T4-1)
            P2(T10-T6+1,T8-T2+1)=A(T10,T8)
        DO T8=T4,MIN(T2-2,T4+31)
          OVER1=MOD(-1+N,4)
          DO T10=T2,MIN(N-OVER1,T2+60),4
            DO T12=T6,MIN(T6+31,T4-1)
              A(T8,T10)=A(T8,T10)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10-T2+1)
              A(T8,T10+1)=A(T8,T10+1)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+1-T2+1)
              A(T8,T10+2)=A(T8,T10+2)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+2-T2+1)
              A(T8,T10+3)=A(T8,T10+3)-P1(T12-T6+1,T8-T4+1)*P2(T12-T6+1,T10+3-T2+1)
          DO T10=MAX(N-OVER1+1,T2),MIN(T2+63,N)
          DO T12=T6,MIN(T4-1,T6+31)
            A(T8,T10)=A(T8,T10)-A(T6,T10)*A(T10,T8)
        DO T6=T4+1,MIN(T4+31,T2-2)
      DO T8=T2,MIN(N,T2+63)
    DO T10=T4,MIN(T6-1,T10+T2)
    A(T6,T8)=A(T6,T8)-A(T6,T10)*A(T10,T8)
```

- **Data Copy**: Tracked at the beginning of the code, where all variables are copied to ensure they are available for usage.
- **Unroll by 4**: The loop unroll mechanism is applied by a factor of 4, as indicated by the note "unroll by 4.
- **Unroll Cleanup**: This is done to handle the remaining elements of the loop after the unroll by 4, ensuring no dangling references.

The diagram illustrates these transformations with annotations indicating the points where data copy, unroll by 4, and unroll cleanup are applied.
Transformed Code for LU

(Cont.)

IF (66<=T2)
DO T4=1,T2-33,32
DO T6=T2-1,N,32
DO T8=T4,T4+31
DO T10=T6,MIN(N,T6+31)
P3(T8-T4+1,T10-T6+1)=A(T10,T8)
DO T8=T2,MIN(T2+63,N)
DO T10=T4,T4+31
P4(T10-T4+1,T8-T2+1)=A(T10,T8)
DO T8=T6,MIN(T6+31,N)
OVER2=MOD(-1+N,4)
DO T10=T2,MIN(N-OVER2,T2+60),4
DO T12=T4,T4+31
A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
A(T8,T10+1)=A(T8,T10+1)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+1-T2+1)
A(T8,T10+2)=A(T8,T10+2)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+2-T2+1)
A(T8,T10+3)=A(T8,T10+3)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10+3-T2+1)
DO T10=MAX(T2,N-OVER2+1),MIN(T2+63,N)
DO T12=T4,T4+31
A(T8,T10)=A(T8,T10)-P3(T12-T4+1,T8-T6+1)*P4(T12-T4+1,T10-T2+1)
DO T4=T2-1,MIN(N-1,T2+62)
DO T8=T4+1,N
A(T8,T4)=A(T8,T4)/A(T4,T4)
DO T6=T4+1,MIN(T2+63,N)
DO T8=T4+1,N
A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)
Search Space

- **Set of variants**
  - Different loop orders, copy yes or no, different loop splitting strategies, different prefetch strategies
  - Select variant with the best performance

- **Integer parameter values**
  - Unroll factors, tile sizes, prefetch distances
  - Each parameter has unique search properties

- **Constraints**
  - Limit unrolling amount by register capacity
  - Limit tiling parameters by cache/TLB capacity and set associativity
## Comparison of Search Cost

<table>
<thead>
<tr>
<th>Code</th>
<th>SGI R10K</th>
<th>Sun Ultrasparc IIe</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM (ATLAS)</td>
<td>35 min</td>
<td>14 min</td>
</tr>
<tr>
<td>MM (ECO)</td>
<td>8 min (60 pts)</td>
<td>6 min (44 pts)</td>
</tr>
</tbody>
</table>
Matrix Multiply: Comparison with ATLAS, vendor BLAS and native compiler

Combining with SIMD Optimizations

• **Motivation**
  - Multimedia extension architectures (SSE3, AltiVec, ...)
  - Node processors in high-end systems (*e.g.*, Intel and Opteron clusters)

• **Developed SLP compiler**
  - Initial approach by Larsen and Amarasinghe (PLDI '00)
  - Locality optimizations for superword registers, control flow support and other extensions, Shin, Chame and Hall, PACT '02, MSP '02, JILP '03, MSP '04, CGO '05

• **Impact**
  - Code variants generated anticipating SLP optimizations
  - Requires close integration with backend (in our case) or more search
**code variant generation**

**Phase 1**

- select loop order
- cache and TLB optimizations
- unroll loop nests for SLP compiler

**Phase 2**

- on unrolled code:
  - pack isomorphic operations
  - align operands
  - register optimizations: superword replacement, register packing
  - low-level optimizations

**Transformation Modules**

- analysis/models

**Architecture Specification**

- C code with SSE-3 "assembly"

**Empirical Search Engine**

- performance monitoring

**Execution Environment**

- optimized code + representative input data

CScADS, July 2007
Pentium M:
Combined Locality + SIMD Compiler

\[
do \ i \\
do \ j \\
do \ k \\
\quad c(i,j) = c(i,j) + a(i,k)*b(k,j) \\
\]

<table>
<thead>
<tr>
<th>MM Version (3200x3200)</th>
<th>Automatically-Generated</th>
<th>Intel MKL 8.0.2</th>
<th>ATLAS 3.7.14</th>
<th>Intel ifort compiler v9.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (Single precision)</td>
<td>2.957 Gflops</td>
<td>2.895 Gflops</td>
<td>3.076 Gflops</td>
<td>0.692 Gflops</td>
</tr>
</tbody>
</table>

## Full Set of Experiments to Date

<table>
<thead>
<tr>
<th></th>
<th>SGI R10000</th>
<th>Pentium M</th>
<th>UltraSparc IIE</th>
<th>Pentium D</th>
<th>PowerPC AltiVec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Matrix-Matrix</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Matrix-Vector</strong></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Matrix-Vector (Transpose)</strong></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Triangular Solve</strong></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>LU Factorization</strong></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Jacobi</strong></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>MADNESS</strong></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## Performance Summary

### Subset of results

<table>
<thead>
<tr>
<th>Architecture</th>
<th>kernel</th>
<th>opt</th>
<th>ATLAS</th>
<th>vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pentium M</strong></td>
<td>mv</td>
<td>1.47x</td>
<td>1.33x</td>
<td>1.00x</td>
</tr>
<tr>
<td></td>
<td>mvT</td>
<td>1.47x</td>
<td>1.34x</td>
<td>0.99x</td>
</tr>
<tr>
<td></td>
<td>mm</td>
<td>3.35x</td>
<td>3.39x</td>
<td>3.04x</td>
</tr>
<tr>
<td></td>
<td>lu</td>
<td>11.44x</td>
<td>-</td>
<td>12.88x</td>
</tr>
<tr>
<td><strong>SGI R10000</strong></td>
<td>mv</td>
<td>1.22x</td>
<td>1.02x</td>
<td>1.20x</td>
</tr>
<tr>
<td></td>
<td>mvT</td>
<td>1.03x</td>
<td>0.87x</td>
<td>0.99x</td>
</tr>
<tr>
<td></td>
<td>mm</td>
<td>1.72x</td>
<td>1.58x</td>
<td>1.73x</td>
</tr>
<tr>
<td></td>
<td>lu</td>
<td>2.75x</td>
<td>-</td>
<td>3.53x</td>
</tr>
</tbody>
</table>

**Baseline performance: best native compiler optimizations**

What next?

• Where compilers can beat libraries
  - PERI: Auto-tuning of application code
  - Libraries used in unusual ways (e.g., MM on long, skinny matrices)
  - Composing library calls

• Other ways compilers can make programmers more productive in tuning their code
  - Search for best values of application-level parameters
  - Apply user-directed code transformations
  - Tune for particular problem sizes
Concluding Remarks

• Three core technical ideas
  – **Compiler technology**: Modular compilers, systematic approach to optimization, empirical search, *hand-tuned performance*
  – **User Tools**: Access to transformation system, express parameters for automatic search, express expected problem size
  – **Systematic**: Express/derive parameters for search

• Lessons for other SciDAC projects
  – **PERI Outreach**: Working with applications informs tool development
## Non-trivial Performance Tradeoffs

<table>
<thead>
<tr>
<th>Version</th>
<th>TI</th>
<th>TJ</th>
<th>TK</th>
<th>Pref?</th>
<th>Loads</th>
<th>L1 misses</th>
<th>L2 misses</th>
<th>TLB misses</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm1</td>
<td>32</td>
<td>64</td>
<td>N</td>
<td></td>
<td>4.20B</td>
<td>142M</td>
<td>21.6M</td>
<td>231K</td>
<td>10.2B</td>
</tr>
<tr>
<td>mm2</td>
<td>16</td>
<td>128</td>
<td>N</td>
<td></td>
<td>4.10B</td>
<td>210M</td>
<td>35.3M</td>
<td>105M</td>
<td>12.5B</td>
</tr>
<tr>
<td>mm3</td>
<td>8</td>
<td>256</td>
<td>256</td>
<td>N</td>
<td>4.08B</td>
<td>319M</td>
<td>7.19M</td>
<td>4.42M</td>
<td>9.70B</td>
</tr>
<tr>
<td>mm4</td>
<td>16</td>
<td>512</td>
<td>128</td>
<td>N</td>
<td>4.11B</td>
<td>182M</td>
<td>8.01M</td>
<td>2.78M</td>
<td>9.47B</td>
</tr>
<tr>
<td>mm5</td>
<td>16</td>
<td>512</td>
<td>128</td>
<td>Y</td>
<td>5.12B</td>
<td>188M</td>
<td>8.04M</td>
<td>2.78M</td>
<td>9.18B</td>
</tr>
<tr>
<td>j1</td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td>25.5M</td>
<td>8.78M</td>
<td>1.65M</td>
<td>3.52K</td>
<td>181M</td>
</tr>
<tr>
<td>j2</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td>34.0M</td>
<td>8.82M</td>
<td>1.64M</td>
<td>3.49K</td>
<td>137M</td>
</tr>
<tr>
<td>j3</td>
<td>16</td>
<td>8</td>
<td>N</td>
<td></td>
<td>28.0M</td>
<td>6.10M</td>
<td>1.32M</td>
<td>18.3K</td>
<td>155M</td>
</tr>
<tr>
<td>j4</td>
<td>16</td>
<td>8</td>
<td>Y</td>
<td></td>
<td>40.8M</td>
<td>7.62M</td>
<td>1.32M</td>
<td>18.6K</td>
<td>125M</td>
</tr>
<tr>
<td>j5</td>
<td>300</td>
<td>16</td>
<td>N</td>
<td></td>
<td>25.5M</td>
<td>8.79M</td>
<td>1.18M</td>
<td>9.99K</td>
<td>159M</td>
</tr>
<tr>
<td>j6</td>
<td>300</td>
<td>16</td>
<td>Y</td>
<td></td>
<td>34.0M</td>
<td>8.84M</td>
<td>1.19M</td>
<td>9.87K</td>
<td>122M</td>
</tr>
</tbody>
</table>

Observation: The best performance comes from balancing all optimization goals.