Compiler Autotuning and Supporting Tools

July 9, 2008
CScADS Workshop

Chun Chen, Jacqueline Chame, Muhammad Murtaza, Mary Hall, Jaewook Shin*, Paul Hovland*

*Argonne National Lab
Will self-tuned libraries always outperform compiler generated code?

• On some previous generation processors, our compiler autotuning has shown better performance than hand-tuned libraries in several cases.

• Still a challenge for some processors.
  - Self-tuned library can use hand-tuned kernels
  - Back-end compiler used for autotuning is not as efficient.
Recap of Existing Compiler Limitations

- Data reuse mostly focuses on individual cache level or idealistic cache model
  - Miss the opportunity of efficient data reuse across the entire memory hierarchy.
- Lack of mechanism to compose complex high-level transformations
  - Built-in rigid transformation strategy often generates very different code from manually-optimized, with relatively low performance.
Optimization framework

architecture specifications

code strategy generation

- heuristics for locality
- reuse analysis
- cache models
- register models

transformation framework

- dependence graph
- transformation modules
- code generator

original code

transformation script with bound parameters

search

- guided empirical search

optimized code variant

final optimized code

execution environment

performance monitoring
Transformation Framework

original code

statement +
iteration spaces

dependence graph

transformation
script

permute
tile
unroll-and-jam
...

statements +
transformed iteration
spaces

Omega code generator

transformed
code
Tools available

• Omega Library 2
  - Improved Omega Library from UMD
  - Bug fixes and enhanced functionality
  - Three essential components: Omega test, code generator and command-line calculator

• Robust all-in-one solution for our purpose.

• More sophisticated code generation for loops is left to higher-level tool.
• CHiLL: A Framework for Composing High-Level Loop Transformations
  - Built upon improved Omega Library.
  - Transformation strategy represented as script.
  - Algorithms take care of complex loop bounds and statement order based on dependence graph and iteration spaces even for non-perfectly nested loops.
  - Provide a simple interface to analytical compiler and search engine.

• Can be used to facilitate the process of manual tuning of libraries and applications.
CHiLL: example 1 (simple loop)

DO I = 1, 14, 3
   X(I) = 0

original()
unroll(0, 1, 2)

original()
unroll(0, 1, 10)

original()
unroll(0, 1, 0)

DO T2 = 1, 7, 6
   X(T2) = 0
   X(T2 + 3) = 0
   X(13) = 0

original()
unroll(0, 1, 2)

DO T2 = 1, 7, 6
   X(T2 + 3) = 0
   X(13) = 0
   X(T2 + 9) = 0

original()
unroll(0, 1, 10)

original()
unroll(0, 1, 0)
DO I=0,N
  DO J=I,I+N
    F3(I) += F1(J) * W(I-J)
  END DO
  F3(I) *= DT
END DO

OVER1 = MOD(1+N, 2)
DO T2=0,N-OVER1,2
  F3(T2) += F1(T2) * W(T2-T2)
  DO T4=T2+1,N+T2
    F3(T2) += F1(T4) * W(T2-T4)
    F3(T2+1) += F1(T4) * W(T2+1-T4)
  END DO
  F3(T2+1) += F1(N+T2+1) * W(-N)
  F3(T2) *= DT
  F3(T2+1) *= DT
END DO
IF (1<=OVER1)
  DO T4=N,2*N
    F3(N) += F1(T4) * W(N-T4)
  END DO
ENDIF (1<=OVER1 .AND. 0<=N)
  F3(N) *= DT
ENDIF
CHiLL: example 3 (Matrix Multiply)

\[ TI = 128 \]
\[ TJ = 8 \]
\[ TK = 512 \]
\[ UI = 2 \]
\[ UJ = 2 \]

\textbf{permute([3,1,2])}

\textbf{tile(0,2,TJ)}

\textbf{tile(0,2,TI)}

\textbf{tile(0,5,TK)}

\textbf{datacopy(0,3,2,1)}

\textbf{datacopy(0,4,3)}

\textbf{unroll(0,4,UI)}

\textbf{unroll(0,5,UJ)}

\[ \text{DO T2=1,N,512} \]
\[ \text{DO T4=1,N,128} \]
\[ \text{DO T6=T2,MIN(N,T2+511)} \]
\[ \text{DO T8=T4,MIN(N,T4+127)} \]
\[ \text{P1(T6-T2+1,T8-T4+1)=A(T8,T6)} \]
\[ \text{DO T6=1,N,8} \]
\[ \text{DO T8=T6,MIN(T6+7,N)} \]
\[ \text{DO T10=T2,MIN(N,T2+511)} \]
\[ \text{P2(T10-T2+1,T8-T6+1)=B(T10,T8)} \]
\[ \text{OVER1=MOD(N,2)} \]
\[ \text{DO T8=T4,MIN(T4+126,N-OVER1),2} \]
\[ \text{OVER2=MOD(N,2)} \]
\[ \text{DO T10=T6,MIN(N-OVER2,T6+6),2} \]
\[ \text{DO T12=T2,MIN(T2+511,N)} \]
\[ \text{C(T8:T8+1,T10:T10+1)=P1(T12-T2+1,T8-T4+1:T8-T4+2)*P2(T12-T2+1,T10-T6+1:T10-T6+2)} \]
\[ \text{IF (1<=OVER2 .AND. N<=T6+7)} \]
\[ \text{DO T12=T2,MIN(T2+511,N)} \]
\[ \text{C(T8:T8+1,N)=P1(T12-T2+1,T8-T4+1:T8-T4+2)*P2(T12-T2+1,N-T6+1)} \]
\[ \text{IF (1<=OVER1 .AND. N<=T4+127)} \]
\[ \text{DO T10=T6,MIN(T6+7,N)} \]
\[ \text{DO T12=T2,MIN(N,T2+511)} \]
\[ \text{C(N,T10)=P1(T12-T2+1,N-T4+1)*P2(T12-T2+1,T10-T6+1)} \]
Autotuning Experimental Results

- **AMD Opteron**
  - Pathscale compiler failed to identify temporary arrays are always aligned.
  - Performance still lags behind!

- **Intel Core 2**
  - Intel compiler does not handle scheduling for prefetch intrinsics.
  - Performance still lags behind!
Matrix Multiply on Jacquard (NERSC)

TUNE: TUNE for locality, PATHSCALE for vectorization
ACML: hand-tuned vendor library
PATHSCALE: not vectorized (alignment issues)
Matrix-Matrix Performance Comparison on Intel Core 2 Duo

![Graph showing performance comparison between TUNE, MKL, and Native for problem sizes ranging from 100 to 1400. The graph plots Mflop against Problem Size.](Image)
<table>
<thead>
<tr>
<th>Event</th>
<th>MKL</th>
<th>TUNE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE_PrefNta_Ret</td>
<td>126362</td>
<td>0</td>
</tr>
<tr>
<td>SSE_PrefT1_Ret</td>
<td>32260262</td>
<td>0</td>
</tr>
<tr>
<td>SSE_PrefT2_Ret</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SSE_PrefNta_Miss</td>
<td>46467</td>
<td>0</td>
</tr>
<tr>
<td>SSE_PrefT1_Miss</td>
<td>1038617</td>
<td>0</td>
</tr>
<tr>
<td>SSE_PrefT2_Miss</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DCache_Rep</td>
<td>332297749</td>
<td>18360367</td>
</tr>
<tr>
<td>DCache_Pend_Miss</td>
<td>39019994</td>
<td>140968429</td>
</tr>
<tr>
<td>Data_Mem_Ref</td>
<td>417906963</td>
<td>578392107</td>
</tr>
<tr>
<td>Pref_Rqsts_Up</td>
<td>54180035</td>
<td>30368079</td>
</tr>
<tr>
<td>Pref_Rqsts_Dn</td>
<td>1649884</td>
<td>14441</td>
</tr>
<tr>
<td>UnhltCore_Cycles</td>
<td>545770204</td>
<td>761735797</td>
</tr>
</tbody>
</table>
Lessons Learned

- Memory hierarchy is only part of the story.
  - Efficient locality optimization changes Matrix Multiply into CPU-bounded computation.
  - Need autotuning on instruction scheduling.

- What about other compiler optimizations
  - Many heuristic algorithms used in compiler.
  - Domain-knowledge optimizations provided by user.
**Issue: Instruction Scheduling**

Nek5k 4x4 matrix-multiply (from Jaewook Shin @ ANL):

<table>
<thead>
<tr>
<th>Compiler scheduled: 75% peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>movapd 32+B(%rip), %xmm9</td>
</tr>
<tr>
<td>mulpd %xmm8, %xmm9</td>
</tr>
<tr>
<td>mulpd B(%rip), %xmm8</td>
</tr>
<tr>
<td>movapd 48+A(%rip), %xmm12</td>
</tr>
<tr>
<td>movapd 112+B(%rip), %xmm15</td>
</tr>
<tr>
<td>mulpd %xmm12, %xmm15</td>
</tr>
<tr>
<td>mulpd %xmm12, %xmm14</td>
</tr>
<tr>
<td>movapd %xmm12, %xmm13</td>
</tr>
<tr>
<td>mulpd %xmm12, %xmm12</td>
</tr>
<tr>
<td>addpd %xmm4, %xmm0</td>
</tr>
<tr>
<td>addpd %xmm5, %xmm1</td>
</tr>
<tr>
<td>addpd %xmm6, %xmm2</td>
</tr>
<tr>
<td>addpd %xmm7, %xmm3</td>
</tr>
<tr>
<td>hadddp %xmm1, %xmm0</td>
</tr>
<tr>
<td>movapd %xmm0, C(%rip)</td>
</tr>
<tr>
<td>hadddp %xmm3, %xmm2</td>
</tr>
<tr>
<td>movapd %xmm2, 16+C(%rip)</td>
</tr>
<tr>
<td>addpd %xmm12, %xmm8</td>
</tr>
<tr>
<td>addpd %xmm13, %xmm9</td>
</tr>
<tr>
<td>addpd %xmm14, %xmm10</td>
</tr>
<tr>
<td>addpd %xmm15, %xmm11</td>
</tr>
<tr>
<td>hadddp %xmm9, %xmm8</td>
</tr>
<tr>
<td>movapd %xmm8, 32+C(%rip)</td>
</tr>
<tr>
<td>hadddp %xmm11, %xmm10</td>
</tr>
<tr>
<td>movapd %xmm10, 48+C(%rip)</td>
</tr>
<tr>
<td>movapd 64+A(%rip), %xmm0</td>
</tr>
<tr>
<td>movapd 96+B(%rip), %xmm3</td>
</tr>
<tr>
<td>mulpd %xmm0, %xmm3</td>
</tr>
<tr>
<td>mulpd %xmm2, %xmm3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple scheduler: 81% peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>movapd 32+B(%rip), %xmm9</td>
</tr>
<tr>
<td>mulpd %xmm8, %xmm9</td>
</tr>
<tr>
<td>mulpd B(%rip), %xmm8</td>
</tr>
<tr>
<td>movapd 48+A(%rip), %xmm12</td>
</tr>
<tr>
<td>movapd 112+B(%rip), %xmm15</td>
</tr>
<tr>
<td>mulpd %xmm12, %xmm15</td>
</tr>
<tr>
<td>mulpd %xmm5, %xmm1</td>
</tr>
<tr>
<td>addpd %xmm5, %xmm1</td>
</tr>
<tr>
<td>movapd 80+B(%rip), %xmm14</td>
</tr>
<tr>
<td>mulpd %xmm12, %xmm14</td>
</tr>
<tr>
<td>addpd %xmm6, %xmm2</td>
</tr>
<tr>
<td>movapd 48+B(%rip), %xmm13</td>
</tr>
<tr>
<td>mulpd %xmm12, %xmm13</td>
</tr>
<tr>
<td>addpd %xmm7, %xmm3</td>
</tr>
<tr>
<td>movapd 16+B(%rip), %xmm12</td>
</tr>
<tr>
<td>hadddp %xmm1, %xmm0</td>
</tr>
<tr>
<td>movapd %xmm0, C(%rip)</td>
</tr>
<tr>
<td>hadddp %xmm3, %xmm2</td>
</tr>
<tr>
<td>movapd 96+B(%rip), %xmm3</td>
</tr>
<tr>
<td>mulpd %xmm0, %xmm3</td>
</tr>
<tr>
<td>movapd %xmm2, 16+C(%rip)</td>
</tr>
<tr>
<td>movapd 64+B(%rip), %xmm2</td>
</tr>
<tr>
<td>mulpd %xmm0, %xmm2</td>
</tr>
<tr>
<td>movapd 32+B(%rip), %xmm1</td>
</tr>
<tr>
<td>mulpd %xmm0, %xmm1</td>
</tr>
<tr>
<td>movapd B(%rip), %xmm0</td>
</tr>
<tr>
<td>addpd %xmm12, %xmm8</td>
</tr>
<tr>
<td>movapd 64+A(%rip), %xmm8</td>
</tr>
<tr>
<td>movapd 80+A(%rip), %xmm4</td>
</tr>
<tr>
<td>movapd 112+B(%rip), %xmm7</td>
</tr>
<tr>
<td>mulpd %xmm4, %xmm7</td>
</tr>
<tr>
<td>addpd %xmm13, %xmm9</td>
</tr>
</tbody>
</table>
ADDIFOR haxpy3 function (from Paul Hovland @ ANL):

```fortran
  do j=1,N
    do i=1,N
      Y(i,j) = a0*X0(i,j) + a1*X1(i,j) + a2*X2(i,j) +
               2.0*b00*u0(i)*u0(j) +
               2.0*b11*u1(i)*u1(j) +
               2.0*b22*u2(i)*u2(j) +
               b01*(u0(i)*u1(j) + u1(i)*u0(j)) +
               b02*(u0(i)*u2(j) + u2(i)*u0(j)) +
               b12*(u1(i)*u2(j) + u2(i)*u1(j))
    enddo
  enddo

DCOPY(N*N,X0,1,Y,1)
DSCAL(N*N,a0,Y,1)
DAXPY(N*N,a1,X1,Y,1)
DAXPY(N*N,a2,X2,Y,1)
DSYR(UPLO,N,2*b00,u0,1,Y,N)
DSYR(UPLO,N,2*b11,u1,1,Y,N)
DSYR(UPLO,N,2*b22,u2,1,Y,N)
DSYR2(UPLO,N,b01,u0,1,u1,1,Y,N)
DSYR2(UPLO,N,b02,u0,1,u2,1,Y,N)
DSYR2(UPLO,N,b12,u1,1,u2,1,Y,N)
```

Not optimal for BLAS.
Integrate All-Levels of Autotuning

• High-level autotuning:
  - Polyhedral-based loop transformation, automatically code generation.
  - Compiler can select promising transformation strategies from a vast pool of choices.
  - Constraints on parameter space.

• Low-level autotuning
  - Different instructions preferred for different generations of the same processor family or different application codes.
  - Brute force search on scheduling (e.g. Intel MKL).
Conclusion

- For a few well-studied libraries, performance gap still exists
  - Until every part of compiler catches up
  - Domain knowledge beyond existing compiler technologies

- For applications whose computations are not readily decomposed to well-tuned kernels
  - Can achieve high performance from autotuning
  - Without labor-intensive manual-tuning