Information Sciences Institute

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

USC

USC Viterbi
School of Engineering

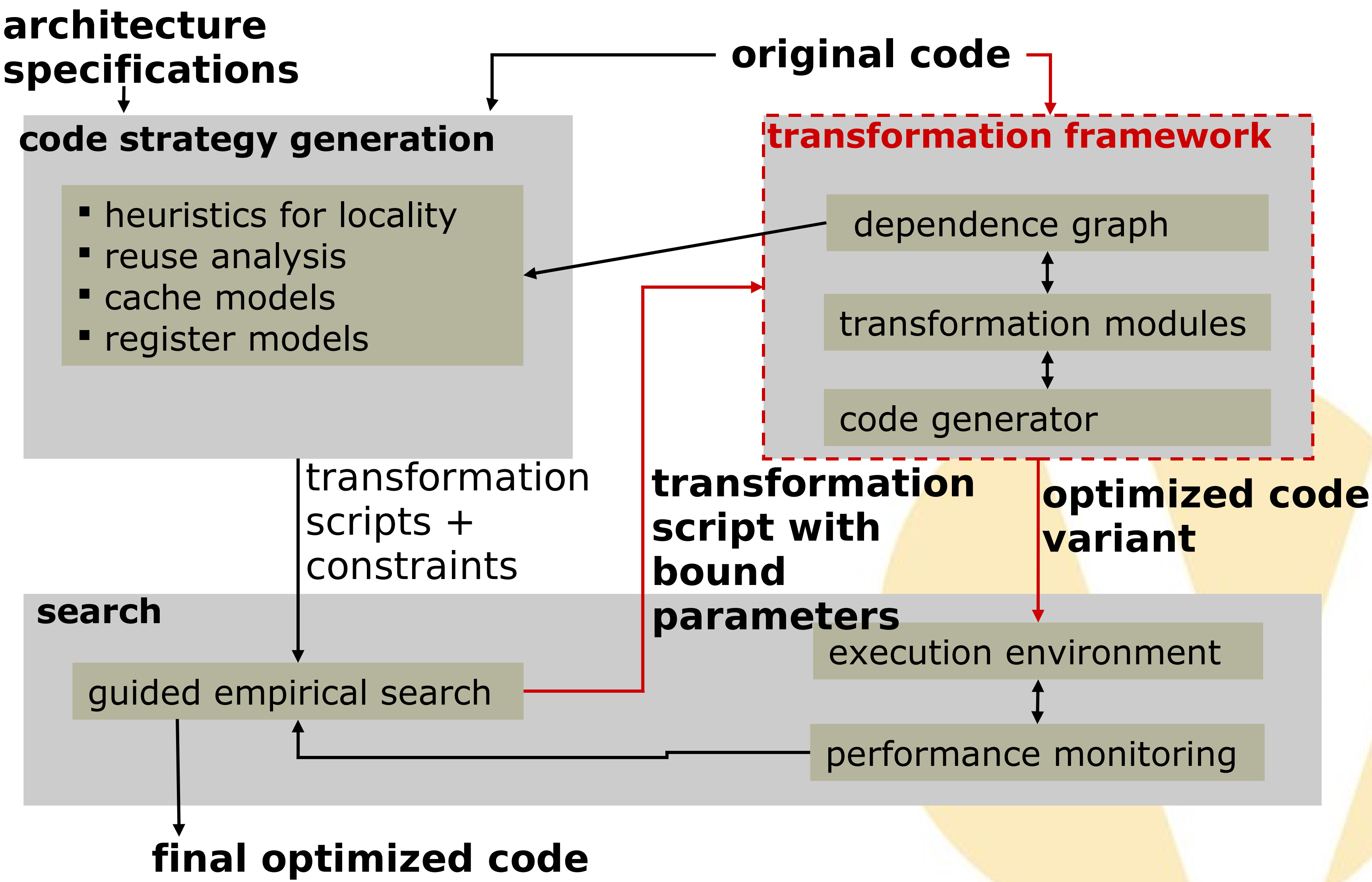
Will self-tuned libraries always outperform compiler generated code?

- 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.

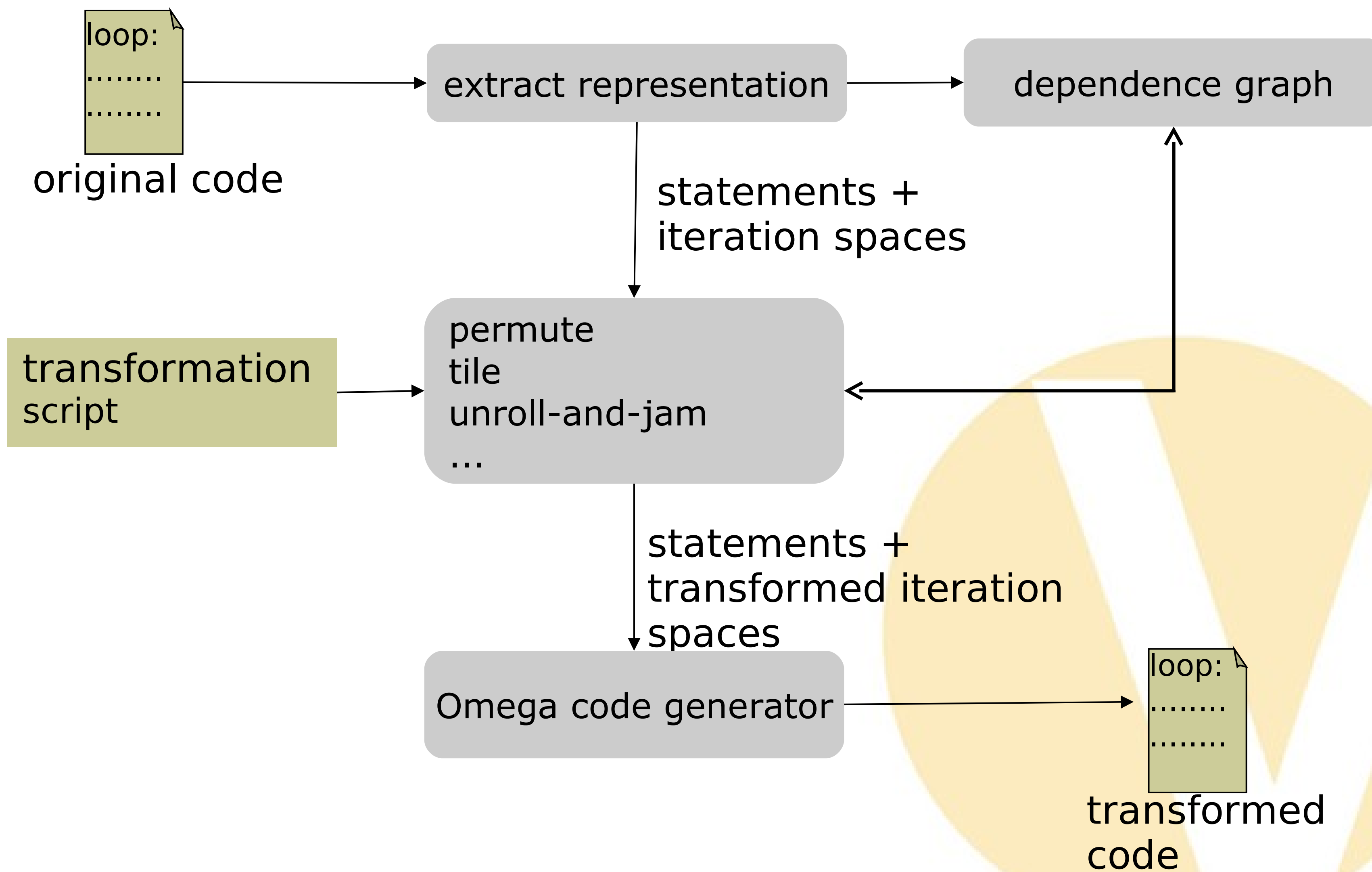
Recap of Existing Compiler Limitations

- 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



Transformation Framework



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.

USC Viterbi School of Engineering

Tools available (Cont.)

Information Sciences Institute

- 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
```

Statement#

Loop level

Unroll amount (adjustable)

original()
unroll(0,1,2)

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

original()
unroll(0,1,10)

```
X(1)=0
X(1+3)=0
X(1+6)=0
X(1+9)=0
X(13)=0
```

original()
unroll(0,1,0)

```
X(1)=0
X(1+3)=0
X(1+6)=0
X(1+9)=0
X(13)=0
```

CHiLL: example 2 (imperfect loop)

```
DO  I=0,N
   DO  J=I,I+N
      F3(I)+=F1(J)*W(I-J)
   F3(I)*=DT
```

original()
unroll(0,1,2)

→

```
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)
   F3(T2+1)+=F1(N+T2+1)*W(-N)
   F3(T2)*=DT
   F3(T2+1)*=DT
IF  (1<=OVER1)
   DO  T4=N,2*N
      F3(N)+=F1(T4)*W(N-T4)
IF  (1<=OVER1  .AND.  0<=N)
   F3(N)*=DT
```

CHiLL: example 3 (Matrix Multiply)

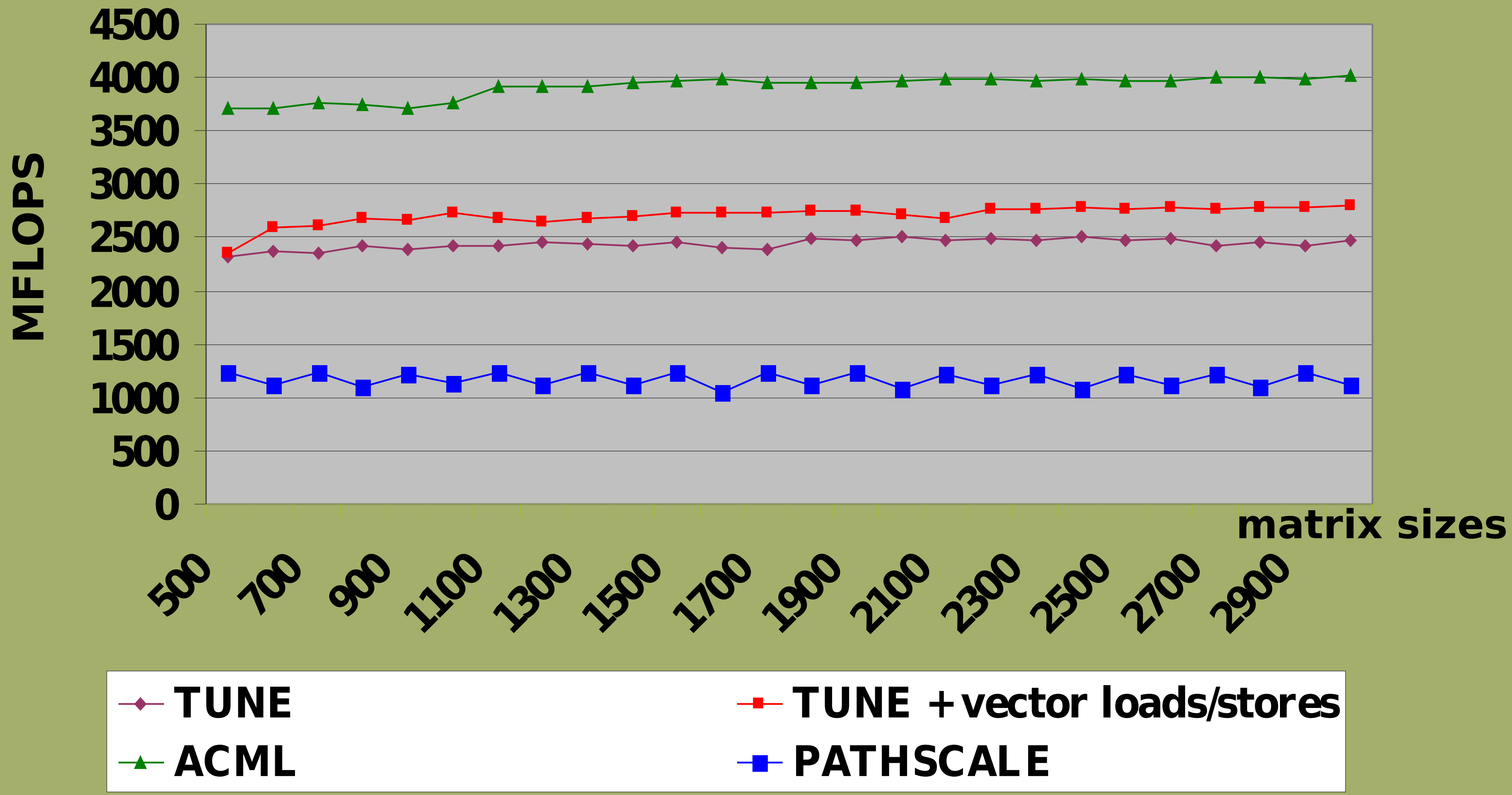
TI=128
TJ=8
TK=512
UI=2
UJ=2
permute([3,1,2])
tile(0,2,TJ)
tile(0,2,TI)
tile(0,5,TK)
datacopy(0,3,2,1)
datacopy(0,4,3)
unroll(0,4,UI)
unroll(0,5,UJ)

```
DO T2=1,N,512
  DO T4=1,N,128
    DO T6=T2,MIN(N,T2+511)
      DO T8=T4,MIN(N,T4+127)
        P1(T6-T2+1,T8-T4+1)=A(T8,T6)
    DO T6=1,N,8
      DO T8=T6,MIN(T6+7,N)
        DO T10=T2,MIN(N,T2+511)
          P2(T10-T2+1,T8-T6+1)=B(T10,T8)
      OVER1=MOD(N,2)
      DO T8=T4,MIN(T4+126,N-OVER1),2
        OVER2=MOD(N,2)
        DO T10=T6,MIN(N-OVER2,T6+6),2
          DO T12=T2,MIN(T2+511,N)
            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)
        IF (1<=OVER2 .AND. N<=T6+7)
          DO T12=T2,MIN(T2+511,N)
            C(T8:T8+1,N)+=P1(T12-T2+1,T8-T4+1:T8-T4+2)*
                          P2(T12-T2+1,N-T6+1)
      IF (1<=OVER1 .AND. N<=T4+127)
        DO T10=T6,MIN(T6+7,N)
          DO T12=T2,MIN(N,T2+511)
            C(N,T10)+=P1(T12-T2+1,N-T4+1)*P2(T12-T2+1,T10-T6+1)
```

Autotuning Experimental Results

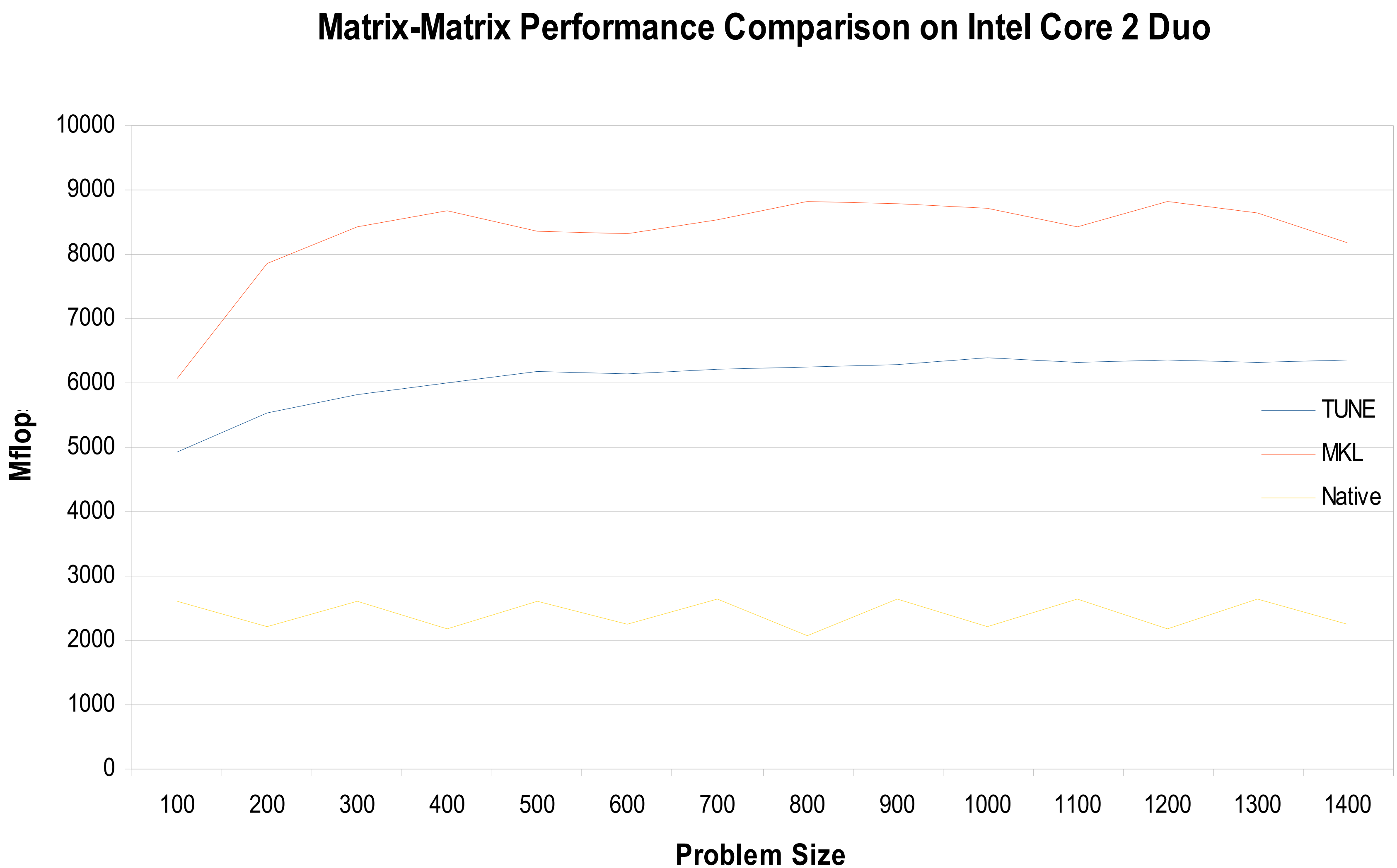
- AMD Opteron
  - Pathscale compiler failed to identify temporary arrays are always aligned.
  - Performance still lags behind!
- Intel Core 2
  - Intel compiler does not handle scheduling for prefetch intrinsics.
  - Performance still lags behind!

Matrix Multiply on Jacquard (NERSC)



TUNE: TUNE for locality, PATHSCALE for vectorization
ACML: hand-tuned vendor library
PATHSCALE: not vectorized (alignment issues)

Intel Core 2


Matrix-Matrix Performance Comparison on Intel Core 2 Duo
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Mflop
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
Problem Size
TUNE
MKL
Native

Intel Core 2

	MKL	TUNE
SSE_PrefNta_Ret	126362	0
SSE_PrefT1_Ret	32260262	0
SSE_PrefT2_Ret	0	0
SSE_PrefNta_Miss	46467	0
SSE_PrefT1_Miss	1038617	0
SSE_PrefT2_Miss	0	0
DCache_Rep	332297749	18360367
DCache_Pend_Miss	39019994	140968429
Data_Mem_Ref	417906963	578392107
Pref_Rqsts_Up	54180035	30368079
Pref_Rqsts_Dn	1649884	14441
UnhltCore_Cycles	545770204	761735797

USC Viterbi School of Engineering

Lessons Learned

ISI Information Sciences Institute

- 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):

Compiler scheduled: 75% peak

```
movapd    32+B(%rip), %xmm9
     mulpd      %xmm8, %xmm9
     mulpd      B(%rip), %xmm8
     movapd     48+A(%rip), %xmm12
movapd    112+B(%rip), %xmm15
     mulpd      %xmm12, %xmm15
     movapd     80+B(%rip), %xmm14
     mulpd      %xmm12, %xmm14
movapd    48+B(%rip), %xmm13
     mulpd      %xmm12, %xmm13
     mulpd      16+B(%rip), %xmm12
addpd     %xmm4, %xmm0
addpd     %xmm5, %xmm1
addpd     %xmm6, %xmm2
addpd     %xmm7, %xmm3
haddpd    %xmm1, %xmm0
movapd    %xmm0, C(%rip)
haddpd    %xmm3, %xmm2
movapd    %xmm2, 16+C(%rip)
addpd     %xmm12, %xmm8
addpd     %xmm13, %xmm9
addpd     %xmm14, %xmm10
addpd     %xmm15, %xmm11
haddpd    %xmm9, %xmm8
movapd    %xmm8, 32+C(%rip)
haddpd    %xmm11, %xmm10
movapd    %xmm10, 48+C(%rip)
     movapd     64+A(%rip), %xmm0
movapd    96+B(%rip), %xmm3
     mulpd      %xmm0, %xmm3
     movapd     64+B(%rip), %xmm2
     mulpd      %xmm0, %xmm2
```

Simple scheduler: 81% peak

```
movapd    32+B(%rip), %xmm9
     mulpd      %xmm8, %xmm9
     mulpd      B(%rip), %xmm8
     movapd     48+A(%rip), %xmm12
addpd     %xmm4, %xmm0
movapd    112+B(%rip), %xmm15
     mulpd      %xmm12, %xmm15
addpd     %xmm5, %xmm1
     movapd     80+B(%rip), %xmm14
     mulpd      %xmm12, %xmm14
addpd     %xmm6, %xmm2
movapd    48+B(%rip), %xmm13
     mulpd      %xmm12, %xmm13
addpd     %xmm7, %xmm3
     mulpd      16+B(%rip), %xmm12
haddpd    %xmm1, %xmm0
movapd    %xmm0, C(%rip)
     movapd     64+A(%rip), %xmm0
haddpd    %xmm3, %xmm2
movapd    96+B(%rip), %xmm3
     mulpd      %xmm0, %xmm3
movapd    %xmm2, 16+C(%rip)
     movapd     64+B(%rip), %xmm2
     mulpd      %xmm0, %xmm2
movapd    32+B(%rip), %xmm1
     mulpd      %xmm0, %xmm1
     mulpd      B(%rip), %xmm0
addpd     %xmm12, %xmm8
     movapd     80+A(%rip), %xmm4
movapd    112+B(%rip), %xmm7
     mulpd      %xmm4, %xmm7
addpd     %xmm13, %xmm9
```

Issue: Compose Computation

ADDIFOR haxpy3 function (from Paul Hovland @ ANL):

```
 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

USC Viterbi School of Engineering

Information Sciences Institute

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