A Crash Course in Compilers for Parallel Computing

Mary Hall
Fall, 2008
Overview of “Crash Course”

• L1: Data Dependence Analysis and Parallelization (Oct. 30)
• L2 & L3: Loop Reordering Transformations, Reuse Analysis and Locality Optimization (Nov. 6)
• L4: Autotuning Compiler Technology (Nov. 13)
Outline of Lecture

I. Summary of Previous Weeks
   - What will we use today?

II. Motivation

III. Autotuning for Locality in ATLAS

IV. Generalized Autotuning Compiler
   - On application code
   - Discussion of SSE and Multi-core
   - Empirical search

V. Potential Future Research Directions
Equivalence to Integer Programming

• Need to determine if \( F(i) = G(i') \), where \( i \) and \( i' \) are iteration vectors, with constraints \( i, i' \geq L, U \geq i, i' \)

• Example:

\[
\text{for (i=2; i<=100; i++)} \\
A[i] = A[i-1];
\]

• Inequalities:

\[
0 \leq i_1 \leq 100, \quad i_2 = i_1 - 1, \quad i_2 \leq 100
\]

integer vector \( I \), \( AI \leq b \)

\[
\begin{bmatrix}
-1 & 0 \\
1 & 0 \\
-1 & 1 \\
1 & -1 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
i_1 \\
i_2
\end{bmatrix}
\leq
\begin{bmatrix}
0 \\
100 \\
-1 \\
1 \\
100
\end{bmatrix}
\]

Solution exist? Yes \( \iff \) dependence
How do we get locality (in caches)?

- Data locality:
  - data is reused and is present in cache
  - same data or same cache line
- Data reuse:
  - data used multiple times
  - intrinsic in computation
- If a computation has reuse, what can we do to get locality?
  - code reordering transformations (today)
  - data layout
How to select optimal tile size? (topic for next week)

Square Tile with Lowest L1 Misses

Slide source: Jacqueline Chame
I. Motivation

Tiling for Real Caches

- Tiling reduces capacity misses
- In real life:
  - Caches are direct-mapped/small associativity
  - Tiling may introduce conflict misses:
    - Data within a tile is not contiguous in memory
  - Conflict misses may offset benefits of tiling
  - Tiling may conflict with other performance optimizations (e.g., prefetching)
Overheads and Other Complexity

- Complex interactions lead to unpredictability
  - SIMD execution in SSE-3
  - Impact on hardware prefetching
  - Register spill
  - Interaction with instruction-level parallelism
  - Overhead of additional control for small problem sizes
  - ...

II. The self-tuning Library ATLAS

- High-performance Basic Linear Algebra Subprograms (BLAS) library code
- Level-3 BLAS
  \[ C = \alpha A \ast B + \beta C \]
  - \( A, B, C \): matrices
  - \( \alpha, \beta \): scalars
- Matrix multiplication: \( \alpha=1, \beta=0 \)
- Level-2 BLAS
  - matrix-vector operations
- Level-1 BLAS
  - vector-vector operations

Increased data reuse.
Increased performance gap between hand-tuned and compiled.
ATLAS Approach to Generating Optimized BLAS (GEMM)

- Sequence of experiments to apply code transformations on the high-level code
  - L1 cache-level tiling
  - register-level “tiling” (unroll-and-jam and scalar replacement)
  - scheduling computation and memory accesses in the inner loop body of the transformed loop nest (e.g., MAC instruction? SSE?)

Aside: Unroll & Jam  
(Scalar Replacement on Next Slide)

UR&J equivalent to tiling followed by unroll inner tile (safe if tiling safe)

```
DO K = 1, N by T_K
  DO I = 1, N by 4
    DO J = 1, N
      DO KK = K, min(KK+ T_K, N)
        C(I, J) = C(I, J) + A(I, KK) * B(KK, J)
        C(I+1, J) = C(I+1, J) + A(I+1, KK) * B(KK, J)
        C(I+2, J) = C(I+2, J) + A(I+2, KK) * B(KK, J)
        C(I+3, J) = C(I+3, J) + A(I+3, KK) * B(KK, J)
  END DO
END DO
END DO
```

Now fine-grain parallel computations are exposed
Aside: Scalar Replacement

Scalar Replacement: Replace accesses to $C$ with scalars

```
DO K = 1, N by TK
  DO I = 1, N by 4
    DO J = 1, N
      C1 = C(I,J); C2 = C(I+1,J); C3 = C(I+2,J); C4 = C(I+3,J)
      DO KK = K, min(KK+ TK, N)
        C1 = C1 + A(I, KK) * B(KK, J)
        C2 = C2 + A(I+1, KK) * B(KK, J)
        C3 = C3 + A(I+2, KK) * B(KK, J)
        C4 = C4 + A(I+3, KK) * B(KK, J)
      END DO KK
      C(I,J) = C1; C(I+1,J) = C2; C(I+2,J) = C3; C(I+3,J) = C4
    END DO J
  END DO I
END DO K
```

Now $C$ accesses can be mapped to “named registers”
ATLAS empirical-driven optimizer

**Diagram:**
- **ATLAS empirical-driven optimizer**
- **Detect hardware parameters**
  - L1 size
  - num regs
  - latency
- **ATLAS search engine**
  - unroll factors
  - tile sizes
  - fetch
- **ATLAS code generator**
- **MM source**
- **Effective capacity**
- **MFLOPS**
- **Execute & measure**
ATLAS Empirical Optimization (now called Autotuning)

- **ATLAS search engine**
  - performs empirical search to determine optimization parameter values

- **ATLAS code generator**
  - generates code given parameter values determined by search engine
Cache-level tiling

- Matrix multiplication is converted to a sequence of smaller matrix multiplications with data sets that fit in cache
  - Mini-MMMs
    - MBxKB sub-matrix of A
    - KBxNB sub-matrix of B
    - MBxNB sub-matrix of C
  - Micro-MMMs
    - MUx1 sub-matrix of A
    - 1xNU sub-matrix of B
    - MUxNU sub-matrix of C
ATLAS mini-MMMs

- Tiling for L1 cache only
- Square tiles only: $NB = MB = KB$
- $NB$ is an optimization parameter

```c
/* mini-MMM of size NBxNB */
for (j = 0; j < NB; j++)
    for (i = 0; i < NB; i++)
        for (k = 0; k < NB; k++)
            C[i][j] += A[i][k] * B[k][j];
```
III. How to Generalize in a Compiler?

- Imperfect loop nests and a large suite of code transformations
- Automate code generation and empirical search
  - Can also automate derivation of optimization strategy
- Provide interface to sophisticated application developer
  - Provides high-level description of experiments to be run
II. Autotuning

TUNE Compiler: Model-Guided Empirical Optimization

- architecture specifications
- original code
- code variant generation
  - heuristics for locality
  - reuse analysis
  - cache models
  - register models
- transformation framework
  - dependence graph
  - transformation modules
  - code generator
- transformation scripts with unbound parameters
- transformation script with bound parameters
- search
  - guided empirical search
- optimized code variant
- execution environment
- performance monitoring
- final optimized code
**Polyhedral Transformation Framework**

Original code:

\[
\begin{align*}
I(s_1): & \{[k,i,j] \mid 1 \leq k \leq N-1 \land k+1 \leq i \leq N \land j=k+1\} \\
I(s_2): & \{[k,i,j] \mid 1 \leq k \leq N-1 \land k+1 \leq i,j \leq N\}
\end{align*}
\]

```
DO K=1,N-1
  DO I=K+1,N
    s1 A(I,K)=A(I,K)/A(K,K)
  DO I=K+1,N
    DO J=K+1,N
    s2 A(I,J)=A(I,J)-A(I,K)*A(K,J)
```

Extract representation

- Dependence analysis
- Dependence spaces
- Statements + iteration spaces
- Transformation script
- Tile loops

Transformed code

```
DO K=1,N-1
  DO I=K+1,N
    s1 A(I,K)=A(I,K)/A(K,K)
  DO I=K+1,N
    DO J=K+1,N
    s2 A(I,J)=A(I,J)-A(I,K)*A(K,J)
```
LU Example:
Polyhedral Loop Transformation Framework

Existing iteration space:
is1: \{[k, i, j] | 1 \leq k \leq N-1 \land k+1 \leq i \leq N \land j = k+1}\nis2: \{[k, i, j] | 1 \leq k \leq N-1 \land k+1 \leq i, j \leq N \}\n
Mapping relations:
t1: \{[k, i, j] \rightarrow [0, k, 0, i, 0, j, 0]\}t2: \{[k, i, j] \rightarrow [0, k, 0, i, 1, j, 0]\}\n
Transformed iteration space:
is1: \{[0, k, 0, i, 0, j, 0] | 1 \leq k \leq N-1 \land k+1 \leq i \leq N \land j = k+1\nis2: \{[0, k, 0, i, 1, j, 0] | 1 \leq k \leq N-1 \land k+1 \leq i, j \leq N \}\n
Omega code generation

\begin{align*}
\text{DO } T2 &= 1, N-1 \\
\text{DO } T4 &= T2+1, N \\
A(T4, T2) &= A(T4, T2)/A(T2, T2) \\
\text{DO } T6 &= T2+1, N \\
A(T4, T6) &= A(T4, T6) - A(T4, T2) \ast A(T2, T6)
\end{align*}
Transformation Script for LU Factorization

DO K=1,N-1
   DO I=K+1,N
      s1   A(I,K)=A(I,K)/A(K,K)
      DO I=K+1,N
         s2   A(I,J)=A(I,J)-A(I,K)*A(K,J)
   END

permute([0,1,2])
tile(s1,5,64,1)
split(s1,3,\[d3 \leq d1-2\])
permute(s2,[1,3,7,5])
permute(s1,[1,5,7,3])
split(s1,3,\[d3 \geq d1-1\])
tile(s3,3,32,3)
split(s3,5,\[d9 \leq d3-1\])
tile(s3,9,32,5)
datacopy(s3,7,2,1)
datacopy(s3,7,3)
unroll(s3,9,4)
tile(s1,7,32,3)
tile(s1,5,32,5)
datacopy(s1,7,2,1)
datacopy(s1,7,3)
unroll(s1,9,4)
Automatically Generated Code for LU

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)
          DO T10=T8, MIN(T4-1, T8+31)
            P1(T8-T6+1, T10-T4+1)=A(T10, T8)
          END
          DO T10=T8, MIN(N, T8+31, T4-1)
            P2(T10-T6+1, T8-T2+1)=A(T10, T8)
          END
          DO T8=T4, MIN(T2-2, T4+31)
            OVER1=MOD(-1+N, 4)
            DO T10=T8, MIN(N-OVER1, T8+60), 4
              DO T12=T6, MIN(T8+61, 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)
              END
            END
            DO T10=MAX(N-OVER1+1, T8), MIN(T8+63, N)
              DO T12=T6, MIN(T8+61, T4-1, T6+31)
                A(T8, T10)=A(T8, T10)-A(T8, T10)*A(T10, T8)
              END
            END
          END
        END
      END
    END
  END
END

TRSM

TRSM

unroll cleanup
Automatically-Generated Code for LU (Cont.)

```plaintext
IF (66<=T2)
DO T4=1,T2-33,32
  DO T6=T2-1,N,32
    DO T8=T4,T4+31
      DO T10=T6,M1N(N,T6+31)
        P3(T8-T4+1,T10-T6+1)=A(T10,T8)
      DO T8=T2,M1N(T2+63,N)
      DO T10=T4,T4+31
        P4(T10-T4+1,T8-T2+1)=A(T10,T8)
      DO T8=T6,M1N(T6+31,N)
      OVER2=M0D(-1+N,4)
      DO T10=T2,M1N(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,M1N(N-1,T2+62)
        DO T8=T4+1,N
          A(T8,T4)=A(T8,T4)/A(T4,T4)
        DO T6=T4+1,M1N(T2+63,N)
        DO T8=T4+1,N
          A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)
```

23 November 13, 2008

Compilers for Parallel Computing,
L3: Autotuning Compilers
Matrix Multiply: Comparison with ATLAS, vendor BLAS and native compiler

matrix multiply on SGI R10K
Preliminary Core2Duo Results

Matrix Multiply on Core2Duo (dgemm)
### Differences: PAPI Measurements

<table>
<thead>
<tr>
<th></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>

- **Need Software Prefetching**
- **More L1 Misses**
- **Lower Penalty**
- **Fewer References**
- **Hardware Prefetching**
- **28% gap**
Matrix Multiply on Jacquard (Opterons at NERSC)

TUNE: TUNE for locality, PATHSCALE for vectorization
ACML: hand-tuned vendor library
PATHSCALE: not vectorized (alignment issues)
Original Code Variant Generation Algorithm

- **Key Insights:**
  - Target data structures to specific levels of the memory hierarchy based on reuse analysis
  - Compose code transformations and determine constraints

For each memory hierarchy level in (Register, L1, L2, ...), use models to:

1. Select the data structure $D$ which has maximum reuse from reuse analysis (if possible, one that has not been considered)
2. Permute the relevant loops and apply tiling (unroll-and-jam for registers) according to newly selected reuse dimension
3. Generate copy variant if copying is beneficial
4. Determine constraints based on $D$ and current memory hierarchy level characteristics, using register/cache/TLB footprint analysis
5. Mark $D$ as considered

[CGO05] 28 November 13, 2008
Compilers for Parallel Computing, L3: Autotuning Compilers
Mapping Reuse to Memory Levels

\[ \text{do } j=1,n \]
\[ \text{do } k=1,n \]
\[ \text{do } i=1,n \]
\[ c(i,j) += a(i,k) \times b(k,j) \]

\( c(i,j) \) has temporal reuse in \( k \)

\[ \text{Loop order: I,J,K} \]
\[ \text{Unroll&Jam I,J} \]

\( b(k,j) \) has temporal reuse in \( i \)

\[ \text{Loop order: JJ,KK,I,J,K} \]
\[ \text{Tiling J,K} \]

\( a(i,k) \) has temporal reuse in \( j \)

\[ \text{Loop order: KK,II,JJ,I,J,K} \]
\[ \text{Tiling I} \]

---

\[ \text{constraint on UI and UJ based on register size} \]

\[ \text{constraint on TJ and TK based on L1 cache size} \]

\[ \text{constraint on TK and TI based on L2 cache size} \]

---

\[ \text{constraint on } T_J \text{ and } T_K \text{ based on L1 cache size} \]

\[ \text{constraint on } T_K \text{ and } T_I \text{ based on L2 cache size} \]
Example: Transformation Scripts for Matrix Multiply

code variant I

permute(2,0,1)
tile(s0,3,Tj)
tile(s0,3,Ti)
tile(s0,9,Tk)
datacopy(s0,5,2,1)
datacopy(s0,7,3,0)
unroll (s0,7,Ui)
unroll (s0,9,Uj)

code variant II

permute(s0,2,1)
tile(s0,3,Ti)
tile(s0,3,Tj)
tile(s0,9,Tk)
datacopy(s0,5,3)
datacopy(s0,7,2, transposed)
unroll (s0,7,Uj)
unroll (s0,9,Ui)

Ti, Tj, Tk, Ui, Uj are unbound parameters
Integration with Parameter Search Framework (Parallel Simplex)

Active Harmony (UMD) & CHiLL (USC-ISI) Integration
Matrix Multiplication

- native
- search_only_atlas
- chill
- atlas (full)
- mkl

N

MFLOPS

November 13, 2008
Compilers for Parallel Computing, L3: Autotuning Compilers
Tiling and Parallelization

- Parallel machines
  - Conflicting goals if
    - parallel loop also carries reuse
  - Works well if
    - coarse-grain parallelism (outer loops)
    - inner loops carry reuse
- Data partitioning may cause false sharing
  - choose block sizes multiple of cache line size
Impact of SSE and other Multimedia Extensions

- Data must be contiguous in memory for SIMD operations (spatial reuse)
  - Transpose during data copy accomplishes this for GEMM
- Data alignment
- Small number of SSE registers changes register locality strategy
- Native compilers are limited in generating high-quality code
- Tradeoffs for ILP, loop-level parallelism and SIMD parallelism
Future Research Directions

• Much work to be done on managing large optimization search spaces
  - A combination of models, pruning heuristics, search algorithms
  - Optimization search spaces grow as complexity increases: composing decisions, increased architectural complexity, more global optimizations

• New architectures
  - Multi-core with complex memory hierarchies (e.g., NUCA)
  - Specialized processors such as GPUs (CS6963!)
  - Heterogeneous platforms (partitioning, code generation, composition)
Future Research Challenges, cont.

- Effective interactions with application programmers
  - Understanding what is useful by working on real applications
  - New transformations (e.g., data reorganization)
  - Mechanisms for code generation and auto-tuning integrated into code

- Validation
  - Does a transformation script represent a valid reordering of the original sequential computation?