Compiler-Based Autotuning Technology

Lecture 1: Autotuning and Its Origins

Mary Hall

July, 2011

* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.
Instructor: My Research Timeline


1998-2004: DEFACTO design environment for FPGAs (C to VHDL)


2001-2006: Compilation for multimedia extensions (DIVA, AltiVec and SSE)

2005-present: Auto-tuning compiler technology (memory hierarchy, multimedia extensions, multi-cores and GPUs)

2007-present: Reports on compiler, exascale software and archiving research directions

ACACES 2011, L1: Autotuning and its Origins
(DOE SciDAC) PERI Autotuning Tools

HPC Toolkit (Rice)
ROSE (LLNL)

CHiLL (USC/ISI and Utah)
ROSE (LLNL)
Orio (Argonne)

OSKI (LBNL)

Active Harmony (UMD)
GCO (UTK)

PerfTrack (LBNL, SDSC, RENCI)
Motivation: A Looming Software Crisis

- Architectures are getting increasingly complex
  - Multiple cores, deep memory hierarchies, software-controlled storage, shared resources, SIMD compute engines, heterogeneity, ...

- Performance optimization is getting more important
  - Today’s sequential and parallel applications may not be faster on tomorrow’s architectures.
  - Especially if you want to add new capability!
  - Managing data locality even more important than parallelism.
  - Managing power of growing importance, too.

Complexity!
Motivation: What is Autotuning?

• Definition:
  - Automatically generate a “search space” of possible implementations of a computation
  • A code variant represents a unique implementation of a computation, among many
  • A parameter represents a discrete set of values that govern code generation or execution of a variant
  - Measure execution time and compare
  - Select the best-performing implementation

• Key Issues:
  - Identifying the search space
  - Pruning the search space to manage costs
  - Off-line vs. on-line search
Motivation: My Philosophy

• Identify search space through a high-level description that captures a large space of possible implementations
• Prune space through compiler domain knowledge and architecture features
• Provide access to programmers! (controversial)
• Uses source-to-source transformation for portability, and to leverage vendor code generation
• Requires restructuring of the compiler
Motivation: Collaborative Autotuning “Compiler”

Traditional view:
- code
- input data

(Semi-)Autotuning Compiler:
- code
- input data (characteristics)

Experiments Engine

ACACES 2011, L1: Autotuning and its Origins
Outline of Course

L1: Autotuning and its Origins (today!)

L2: Tuning code with CHiLL

L3: A Closer Look at Polyhedral Compiler Frameworks

L4: Autotuning for GPU Code Generation

L5: Autotuning High-End Applications
Today’s Lecture: Autotuning and its Origins

1. Traditional Compiler Organization
2. Origins in hardware optimization
3. Related Compiler Organization
   • Use of learning algorithms in compiler
4. Autotuning systems
   • Library-specific autotuning
   • Application-specific autotuning
   • Compiler-based autotuning
5. Detailed look at ATLAS, OSKI, SPIRAL, Active Harmony, PetaBricks and Sequoia
1. Historical Organization of Compilers

Perform Analysis

Arch. Spec.

Search and Apply Transformations
- Safety/Profitability
- Parameters
- Composition

Application Code

Optimized Code

Performance Monitoring Support

Execution Environment

Don’t like performance? Rewrite code!

Input Data Set
• What’s not working
  - Transformations and optimizations often applied in isolation, but significant interactions
  - Static compilers must anticipate all possible execution environments
  - Potential to slow code down; many users say “never use O3”
  - Users write low-level code to get around compiler which makes things even worse
1. Example of 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
2. Related Approach in Hardware Design

- Autotuning is related to hardware (and hardware-software) design space exploration
  - The process of analyzing various functionally equivalent implementations to identify the one that best meets objectives.
- Early example:
2. Automatic Design Space Exploration in DEFACTO

- Overall, less than 2 hours
- 5 minutes for optimized design selection

ACACES 2011, L1: Autotuning and its Origins
3. Related Compiler Organization: Iterative Compilation with Learning

- A preceding body of work on using learning techniques (and sometimes profiling) to make optimization decisions
  - Cooper et al., Eigenmann et al., Stephenson et al, Cavazos et al., ...
- Examples from
  - Instruction scheduling, optimization flag selection, optimization sequence, unroll factor selection, ...

ACACES 2011, L1: Autotuning and its Origins
4. Three Types of Autotuning Systems

a. Autotuning libraries
   - Library that encapsulates knowledge of library’s performance under different execution environments
   - Dense linear algebra: ATLAS, PhiPAC
   - Sparse linear algebra: OSKI
   - Signal processing: SPIRAL, FFTW

b. Application-specific autotuning
   - Active Harmony provides parallel rank order search for tunable parameters and variants
   - Sequoia and PetaBricks provide language mechanism for expressing tunable parameters and variants

c. Compiler-based autotuning
   - Focus of this course
4a. Motivation for Autotuning Libraries

- Many codes spend the bulk of their computation time performing very common operations
  - Particularly linear algebra and signal processing
- Enhance performance without requiring low-level programming of the application
- Much research has been devoted to achieving high performance
  - Search space reasonably well understood
  - Performance can still be improved using autotuning
3a. ATLAS (BLAS)

- Self-tuning linear algebra library
- Early description in SIAM 2000
- ATLAS first popularized notion of self-tuning libraries
- Clint Whaley quote: “No such thing as enough compute speed for many scientific codes”
- Precursor: PhiPAC, 1997
3a. ATLAS (BLAS)

ATLAS Method of Software Adaptation

① Parameterization:
- Parameters provide different implementations (e.g., tile size)
- Easy to implement but limited

② Multiple Implementations:
- Linear search of routine list (variants)
- Simple to implement, simple for external contribution
- Low adaptability, ISA independent, kernel dependent

③ Source Generator:
- Heavily parameterized program generates varying implementations
- Very complicated to program, search and contribute
- High adaptability, ISA independent, kernel dependent
3a. Structure of ATLAS Source Generator

GEMM as building block for other Level 3 BLAS functions
3a. OSKI (Sparse BLAS)

- Sparse matrix-vector multiply < 10% peak, decreasing
  - Indirect, irregular memory access
  - Low computational intensity vs. dense linear algebra
  - Depends on matrix (run-time) and machine

- Tuning is becoming more important
  - 2× speedup from tuning, will increase

- Unique challenge of sparse linear algebra
  - Matrix structure dramatically affects performance
  - To the extent possible, exploiting structure leads to better performance
3a. Example of Matrix Structure in OSKI

- Exploit $8 \times 8$ blocks
  - Store blocks & unroll
  - Compresses data
  - Regularizes accesses
- As $r \times c \uparrow$, speed $\uparrow$

Slide source: Rich Vuduc
3a. Example of Matrix Structure in OSKI: Speedups on Itanium 2 for different block sizes

Slide source: Rich Vuduc
3a. Structure of OSKI

1. Build for Target Arch.
   - Generated code variants

2. Benchmark
   - Benchmark data

Library Install-Time (offline) → Application Run-Time

1. Evaluate Models
   - Matrix
   - Workload from program monitoring

2. Select Data Struct. & Code
   - Heuristic models
   - To user: Matrix handle for kernel calls

To user: Matrix handle for kernel calls

Slide source: Rich Vuduc
**3a. SPIRAL (Signal Processing)**

*Complete automation* of the implementation and optimization task.

**Basic ideas:**
- *Declarative representation* of algorithms
- *Rewriting systems* to generate and optimize algorithms at a high level of abstraction
- *Similar concepts in FFTW*

---

ACACES 2011, L1: Autotuning and its Origins

Slide source: Franz Franchetti
**Linear Transforms**

\[
\begin{align*}
DFT_n & \rightarrow (\text{DFT}_k \otimes I_m) T_m^k (I_k \otimes \text{DFT}_m) L_n^p, \quad n = km \\
DFT_n & \rightarrow P_n (\text{DFT}_k \otimes \text{DFT}_m) D_n, \quad n = km, \gcd(k, m) = 1 \\
DFT_p & \rightarrow R_p^T (I_1 \otimes \text{DFT}_{p-1}) D_p (I_1 \otimes \text{DFT}_{p-1}) R_p, \quad p \text{ prime} \\
DCT-3_n & \rightarrow (I_m \otimes J_m) L_n^p (\text{DCT-3}_m (1/4) \otimes \text{DCT-3}_m (3/4)) \\
& \quad \cdot (F_2 \otimes I_m) \left[ \begin{array}{c}
I_m \\
\frac{0 - J_m - 1}{\sqrt{2}} (I_1 \otimes 2 I_m)
\end{array} \right], \quad n = 2m \\
DCT-4_n & \rightarrow S_n \text{DCT-2}_n \text{ diag}_{0 \leq k < n} \left( \frac{1}{2 \cos((2k + 1)\pi/4n)} \right) \\
\text{IMDCT}_{2m} & \rightarrow (J_m \otimes I_m \otimes I_m \otimes J_m) \left( \begin{bmatrix} 1 & 0 \\ 0 & -1 \\ -1 & 0 \\ 0 & 1 \end{bmatrix} \otimes I_m \otimes I_m \otimes I_m \right) J_{2m} \text{DCT-4}_{2m} \\
\text{WHT}_{2^k} & \rightarrow \prod_{i=1}^k (I_2 \otimes \cdots \otimes I_2) \otimes \text{WHT}_{2^{k-1}} \otimes I_2^{1+2^k+\cdots+k_l}, \quad k = k_1 + \cdots + k_l \\
\text{DFT}_2 & \rightarrow F_2 \\
\text{DCT-2} & \rightarrow \text{ diag}(1, 1/\sqrt{2}) F_2 \\
\text{DCT-4} & \rightarrow J_2 F_{13\pi/8}
\end{align*}
\]

**Matrix-Matrix Multiplication**

\[
\begin{array}{c|c|c|c}
\text{M} & \text{M} & \text{M} & \text{M} \\
\hline
1, 1, 1 & \otimes & 1 & \otimes
\end{array}
\]

\[
\begin{align*}
\text{MMM}_{1, 1, 1} & \rightarrow (\cdot)_1 \\
\text{MMM}_{m, n, k} & \rightarrow (\otimes)_{m/m_k \times 1} \otimes \text{MMM}_{m, n, k} \\
\text{MMM}_{m, n, k} & \rightarrow \text{MMM}_{m, n, k} \otimes (\otimes)_{1 \times n/n_k} \\
\text{MMM}_{m, n, k} & \rightarrow (\Sigma_{k/b_k} \circ (\cdot)_{k/b_k}) \otimes \text{MMM}_{m, n, k} \\
\text{MMM}_{m, n, k} & \rightarrow (L_{k/b_k} \otimes I_{k}}) \circ (L_{k/b_k} \otimes I_{k}}) \circ (L_{m/b_k} \otimes I_{m}})
\end{align*}
\]

**Viterbi Decoding**

\[
\begin{align*}
\text{Vit} & \rightarrow \left( \prod (L \otimes I) \circ (I \otimes C) \right) \circ \text{Id} \\
\text{vec}(v) & \rightarrow \left( \prod (L \otimes I) \circ (I \otimes C) \right) \circ \text{Id} \\
\mathbf{z} & \rightarrow \left( \prod (L \otimes I) \circ (I \otimes C \otimes I) \right) \circ (\tilde{L} \otimes I) \circ \text{Id} \\
& \rightarrow \prod (L \otimes I) \circ (I \otimes (B \otimes I) \circ (\tilde{L} \otimes I)
\end{align*}
\]

**Synthetic Aperture Radar (SAR)**

Slide source: Franz Franchetti
3b. Motivation for Application-level tuning

- Parameters and variants arise naturally in portable application code
- Programmer expresses tunable parameters, input data set properties and algorithm variants
- Tools automatically generate code and evaluate tradeoff space of application-level parameters

**Example: Molecular Dynamics Visualization**

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

ncell = boxLength/cellSize

for i = 1, ncell
  /* perform computation */

**Const** cellSize = 48

ncell = boxLength/48

for i = 1, 48
  /* perform computation */
3b. Application-level tuning using Active Harmony

- Search-based collaborative approach
  - Simultaneously explore different tunable parameters to search a large space defined by the user
    - e.g., Loop blocking and unrolling factors, number of OpenMP threads, data distribution algorithms, granularity controls, ...
  - Supports both online and offline tuning
  - Central controller monitors performance, adjusts parameters using search algorithms, repeats until converges
  - Can also generate code on-demand for tunable parameters that need new code (e.g. unroll factors) using code transformation frameworks (e.g. CHiLL)
• All, but the best point of simplex moves

• Computations can be done in parallel

• N parallel evaluations for N +1 point simplex
4b. Language support for application-level tuning using PetaBricks

- Algorithmic choice in the language is the key aspect of PetaBricks
- Programmer can define multiple rules to compute the same data
- Compiler re-uses rules to create hybrid algorithms
- Can express choices at many different granularities

Example: Sort in PetaBricks

```plaintext
1 transform Sort
2 from A[n]
3 to B[n]
4 {
5     from(A a) to(B b) {
6         tunable WAYS;
7         /* Mergesort */
8     } or {
9         /* Insertionsort */
10     } or {
11         /* Radixsort */
12     } or {
13         /* Quicksort */
14     }
15 }
```
4b. Language support for application-level tuning using PetaBricks

1. PetaBricks source code is compiled
2. An autotuning binary is created
3. Autotuning occurs creating a choice configuration file
4. Choices are fed back into the compiler to create a final binary
4b. Application-level tuning is similar using Sequoia

- Example shows variants representing hierarchical implementation of matrix multiply
- These two tasks represent different variants for different levels of the memory system
- Tunable parameters P, Q and R adjust data decomposition

Example from Mike Houston, CScaDS 2007
4c. Motivation for Compiler-Based Autotuning Framework

- Parameters and variants arise from compiler optimizations
  - Parameters such as tile size, unroll factor, prefetch distance
  - Variants such as different data organization or data placement, different loop order or other representation of computation
- Beyond libraries
  - Can specialize to application context (libraries used in unusual ways)
  - Can apply to more general code
- Complementary and easily composed with application-level support
4c. CHiLL Compiler-Based Autotuning Framework
4c. Combining Models, Heuristics and Empirical Search

Compiler Models (static)
- How much data reuse?
- Data footprint in memory hierarchy levels
- Profitability estimates of optimizations

Heuristics
- "Place" data in specific memory hierarchy level based on reuse
- Copy data tiles mapped to caches or buffers

Empirical Search
- Generate parameterized code variants
- Measure performance to evaluate and choose next point to search
- Heuristics limit variants
- Constraints from models limit parameter values

ACACES 2011, L1: Autotuning and its Origins
Sampling of autotuning systems
- Autotuning libraries
- Application-level autotuning
- Compiler-based autotuning

“Search space” of implementations arises from
- Parameters
- Variants

Lecture mostly focused on structure of systems and expressing/generating search space
References


ACACES 2011, L2: Tuning code with CHiLL