Motivation

Modern (high-end) computer architectures are too complex
- Some final machine parameters may not be statically (well-)known
  - Caches (multiple levels, capacity, associativity, replacement policy)
  - Memory latency
- ILP and pipelining:
  - Dynamic dispatch, out-of-order execution, speculation, branching
- Parallelism and contention for shared resources
- OS scheduler
- Paging
- Perf. not well predictable e.g. for manual or compiler optimization
- Some program parameters (problem sizes, data locality etc.) may not be statically known
- Different algorithms / implementation variants may exist for a computation
- Hardcoded manual optimizations lead to non-performance-portable code
- Compiler optimizations are limited and may have unexpected side effects / interferences

Motivation (cont.)

- Thousands of knobs that we could turn to tune performance!
  - Which ones and how?
  - Avoid hardcoding of performance tuning

Idea: Autotuning – Automatic optimization for unknown target system using Machine Learning

- Given: Training data and initial program version
  - Observed performance on target
  - Machine learning algorithm
  - Optimization strategy (choice of some parameter(s))
  - Automatic code generation / adaptation for target platform and possibly repeat this process
- for libraries: autotuning library generators,
  for compilers: iterative compilation
  for dynamic composition: context-aware composition
- Typical examples:
  - Find the best blocking factor(s) for loops or loop nests to automatically adapt to target cache behavior
  - Find the right sequence and settings of compiler optimizations
  - Select among different algorithms for same operation
  - How many cores/threads / which processors/accelerators to use?

Performance Portability for User-level code?

Avoid hard-coded adaptations / optimizations such as:

- if (avail_num_threads() > 1)
  - in_parallel {
  - sort(a, n/2); // on first half of resources
  - sort( &a[n/2], n-n/2); // on the other half
  - else ... (do it in serial)

- if (available(GPU))
  - gpusort(a,n);
  - else
  - qsort(a,n);

- if (n < CACHESIZE/4)
  - mergesort(a,n);
  - else
  - quicksort(a,n);

Recall: Tiled Matrix-Matrix Multiplication (1)

- Matrix-Matrix multiplication
  - $C = A \times B$
  - here for square $(n \times n)$ matrices $C, A, B$, with $n$ large ($\sim 10^3$):
  - $C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$ for all $i, j = 1...n$
- Standard algorithm for Matrix-Matrix multiplication (here without the initialization of C-entries to 0):

  for $(i=0; i<n; i++)$
  - for $(j=0; j<n; j++)$
  - for $(k=0; k<n; k++)$
  - $C[i][j] += A[i][k] \times B[k][j]$;

Recall: Good spatial locality on $A, C$

Recall: Bad spatial locality on $B$

Recall: many capacity misses
Recall: Tiled Matrix-Matrix Multiplication (2)

- Block each loop by block size S (choose S so that a block of A, B, C fit in cache together), then interchange loops
- Code after tiling:

```latex
\text{for } (ii=0; ii<n; ii+=S) \\
\quad \text{for } (jj=0; jj<n; jj+=S) \\
\quad\quad \text{for } (kk=0; kk<n; kk+=S) \\
\quad\quad\quad \text{for } (i=ii; i < ii+S; i++) \\
\quad\quad\quad\quad \text{for } (j=jj; j < jj+S; j++) \\
\quad\quad\quad\quad\quad \text{for } (k=kk; k < kk+S; k++) \\
\quad\quad\quad\quad\quad\quad C[i][j][k] += A[i][k][j] \times B[k][j][k];
```

Good spatial locality for A, B and C

What is the best choice for the blocking factor S?

Recall: Loop Unroll-And-Jam

- Unroll the outer loop and fuse the resulting inner loops:

```latex
\text{for } i \text{ from } 1 \text{ to } N \text{ do} \\
\quad \text{for } j \text{ from } 1 \text{ to } N \text{ do} \\
\quad\quad a[i][j] \leftarrow a[i][j] + b[i][j]; \\
\quad \text{unroll} & \text{ jam:} \\
\quad\quad a[i][j] \leftarrow a[i][j] + b[i][j];
```

The same conditions as for loop interchange (for the two innermost loops after the unrolling step) must hold (for a formal treatment see [Allen/Kennedy/02, Ch. 4.11])

- + increases reuse in inner loop
- + less overhead

Auto-tuning linear algebra library ATLAS (1)

- BLAS = Basic Linear Algebra Subroutines
  - standard numerical library for Fortran, C
  - frequently used in high-performance applications
- Level-1 BLAS: Vector-vector operations e.g. dot product
- Level-2 BLAS: Matrix-vector operations
- Level-3 BLAS: Matrix-matrix operations, esp., generic versions of dense LU decomposition and Matrix mult.
  - SGEMM: $C := \alpha A \times B + \beta C$
  - for matrices A, B, C, scalars $\alpha, \beta$
  - is ordinary Matrix-Matrix multiplication for $\alpha=1, \beta=0$

Auto-tuning linear algebra library ATLAS (2)

- ATLAS is a generator for optimized BLAS libraries
  - Tiling to address L1 cache
  - Unroll-and-jam / scalar replacement to exploit registers
  - Use multiply-accumulate and SIMD instructions where available
  - Schedule computation and memory accesses
  - Outperforms vendor-specific BLAS implementations

Remark

- Off-line sampling and tuning by greedy heuristic search
  - Happens once for each new system at library deployment (generation) time
  - Can be expensive
- Not practical for less static scenarios or costly sampling
  - Fast predictors needed – full execution or even simulation is not feasible
  - Usually constructed by machine learning
  - Shortens the feedback loop
  - Could be adapted dynamically (on-line sampling/tuning)

Further auto-tuning library generators

- Linear Algebra
  - ATLAS
  - PhiPAC
  - OSKI
- FFT and other signal processing
  - FFTW [Frigo’99]
  - SPIRAL [Püschel et al. 2005]
- Sorting, searching etc.
  - STAPL [Rauchwerger et al.]
  - [Li, Padua, Garzaran CGO’94]
  - [Brewer’95]
  - [Olszewski, Voss PDPTA-2004]
Generalize this in a compiler!

- Iterative compilation / autotuning compilers
  - Optimization of compiler transformation sequences
  - GCC MILEPOST project 2007-2008
  - CAPStuner, www.caps-entreprise.com
  - ActiveHarmony search engine + CHILL source-to-source loop transformation framework

Auto-tuning component composition

- www.caps-entreprise.com
- Autotuning SkePU (Dastgeer, Enmyren, K. 2011)
- Generalizing the PEPPHER component based approach to performance tuning interface
- Coordinating energy-affecting knobs in application code,
- ActiveHarmony search engine + CHiLL source
- Performance-aware:
  - Possible loop transformations, code specializations
  - Resource allocation and scheduling for independent tasks
- At run time, automatically select
  - expected best implementation variant for each call,
  - expected best resource allocation and schedule for indep. subtasks, given run-time information on actual parameters and available resources
  - Look up dispatch tables prepared off-line (by machine learning)
- Examples
  - K./Löwe 2007/2012: Performance-aware components
  - Autotuning SkePU (Dastgeer, Enmyren, K. 2011)
  - EU FP7 project PEPPHER (IEEE Micro Sep/Oct. 2011)

Performance-aware components: Interfaces, implementations, descriptors

One PEPPER component of the application

- Component source files
- Interface descriptor
- Implementation variants, e.g. different algorithms, exec. units,
- Composition: Variant selection (static or dynamic or both)
- PEPPER main application

Implementation parameters, e.g. L1 size, Avg. latency,
Loop unroll factors, switches,
Feedback on performance
Measure time on target
Compiled code generator
Compiled code

One step further: Auto-tunable components and run-time composition

- Component programmer exposes the knobs for optimization in a performance tuning interface
  - Tunable function parameters e.g., problem sizes
  - Equivalent implementation variants (different algorithms, ...) at calls
  - Possible loop transformations, code specializations
  - Resource allocation and scheduling for independent tasks
- At run time, automatically select
  - expected best implementation variant for each call,
  - expected best resource allocation and schedule for indep. subtasks, given run-time information on actual parameters and available resources
  - Look up dispatch tables prepared off-line (by machine learning)
- Examples
  - K./Löwe 2007/2012: Performance-aware components
  - Autotuning SkePU (Dastgeer, Enmyren, K. 2011)
  - EU FP7 project PEPPHER (IEEE Micro Sep/Oct. 2011)

Annotated Components in PEPPHER

- User-level components
  - Sequential or parallel implementations, possibly platform-specific
  - Internal parallelism encapsulated, multiple programming models
- Performance-aware:
  - Annotate with performance prediction metadata, mark up performance-relevant variation points
  - Algorithmic variants
  - Operand data representation and storage
  - Tunable parameters: Buffer sizes, blocking factors, ...
  - Schedule and resource allocation at independent subtasks
  - Portable composition and coordination of components
  - Support by run-time system (StarPU)
- Goal: Performance Portability and Programmability
- Main target: heterogeneous multi-/manycore (esp. GPU) systems
- EU FP7 research project, 2010-2012
  - www.peppher.eu

EXCESS

- EU FP7 project, 2013-2016
  - Chalmers, Linköping U, Movidius Ltd, HLRS Stuttgart, U. Tromsø
- Generalizing the PEPPER component based approach to application-level energy optimization
  - Holistic energy modeling and optimization
  - Across the whole system software stack
  - Coordinating energy-affecting knobs in application code, libraries (skeletons, concurrent shared data structures), run-time system, OS
- Consider multicore systems, GPU-based systems and a low-power target platform (Movidius Myriad)
- www.excess-project.eu

Summary: Auto-tuning

- Code optimization is difficult and very platform specific. Avoid hardcoding.
  Instead, expose what is tunable and let the system learn suitable configurations from training data.
- Auto-tuning library generators
  - Fixed domain, implicit or explicit human guidance of search space
- Auto-tuning compilers
  - General-purpose programs (HPC)
  - Program structure (loop nests) defines optimization search space
  - Limited influence by programmer (e.g., some #pragmas)
- Auto-tuning component composition
  - Programmer-exposed performance tuning interfaces, install-time learning, run-time composition
  - Can incorporate library and compiler based autotuning
Master thesis projects available!

- Infrastructure e.g. extending composition tool, containers, ...
- Case studies: PEPPHERing applications, new platforms
- Improved static prediction support
- ...

References

- On ATLAS:
- On FFTW:
- On SPRAL:
- On iterative compilation:
  On ActiveHarmony + CHiLL:
- On General Component Autotuning: