Notes

• Midterm exam in on Thursday, Nov. 10
  • on readings through previous week
• Group Project interim report due Nov. 7
Autotuning for HPC Applications

• Overall goal is *performance portability*
  • Across diverse HPC architectures
  • Which has not been achievable through languages and compilers

• Involves “automatic generation of a search space of possible implementations of a computation that are evaluated through models and/or empirical measurement to identify the most desirable implementation”

• Search space is a set of code variants functionally equivalent
  • Paper says to an original implementation, but could be to a specification (e.g., an API)
Autotuning

• **Empirical autotuners**
  • Execute each code variant
  • Measure runtime (or another objective function)
  • Evaluate performance of each variant
  • Run the best performing variant
  • Need intelligent search methods and models to prune a potentially very large search space
  • Can also use runtime prediction models, esp. for long-running kernels

• **Code variants**
  • Different code organization, data structures, algorithms, low-level implementation details
  • Parallelization strategies
  • Memory hierarchy optimization (data placement, blocking/tiling, tile size)

• **Can be applied offline, or online while the application is running, or even incrementally**
Tools

• Libraries
  • Isolate performance critical functions behind a standard API
  • Examples include Atlas (linear algebra), SPIRAL (digital signal processing), Sparsity (sparse matrix computations), FFTW (fast Fourier transforms)

• Compilers and code generators
  • Generate a collection of architecture-specific codes from same high-level input
  • Examples include CHiLL (USC, Utah, UMD), Orio (Oregon, Ohio State), POET (Georgia Tech, LLNL)
  • Can include parallelization – SIMD pragmas, OpenMP directives, CUDA, etc.
  • And various loop optimizations – tiling, unrolling, permutation, fusion, distribution, prefetching, software pipelining, ...
    • And what order to apply them
Application-level tools

• Tools allow expressing tunable parameters and code variants representing alternate implementations
  • Can select code variant based on problem size, to target different levels of memory hierarchy or parallelism
  • Must be done at runtime if depends on input dataset
    • Active Harmony (UMD) and Adapt (Purdue) can create, link, test new variants in parallel with execution during iterative computations

• Disadvantage is that each application developer has to specify autotuning

• New frameworks like RAJA (LLNL) and Kokkos (SNL) can specialize high level code using C++ template abstractions around loops and data structures

• Domain specific languages (DSLs) for some application areas – e.g., Halide for image processing, others for stencil computations (PDEs)
Search

• Evaluate points in the search space (parameter values, code variants) to find optimal solution

• Complete enumeration
  • Doesn’t scale since there can be too many points in the search space

• Two ways to limit search space to a subset
  • Model-free – global or local search
    • Global includes simulated annealing, genetic algorithms, particle swarm optimization – guaranteed to find global optimum if given long enough search time, but in practice stop earlier
    • Local includes Nelder-Mead simplex, orthogonal search, variable neighborhood search – move from current to nearby point in search space, so can terminate in a local optimum

• Model-based
  • Use performance prediction metrics (analytical or empirical models)
  • Limited by accuracy of models
Software Engineering Challenges

• Offline autotuning makes compilation slow
  • Many variants need to be compiled and executed

• Empirical autotuning makes developer manage the tuning process

• Build process for autotuning can be complex
  • Can be different while autotuning vs. running autotuned code (library, application, etc.)

• Package management systems (e.g., Spack) help
  • Can wrap compilers to generate autotuning variants

• Debugging autotuned code can be difficult
  • You may be running automatically generated code!
  • But the generated code is more likely than yours to be correct ...
ATLAS

• Automatically Tuned Linear Algebra Software
  • Library produced by autotuning – they call it automated empirical optimization of software (AEOS)

• Start from well-known, widely used API for linear algebra core operations
  • BLAS – basic linear algebra subroutines
  • For linear algebra, need to characterize parameters that vary across machines – biggest one is blocking factor for blocked LA algorithms, which affects cache utilization
  • Can also try different source code implementations
    • Multiple implementations or code generation

• To produce highly tuned code, not enough to understand the hardware
  • Because of complex interactions between hardware features, compiler, OS, ...
  • So we’re back to an empirical process – try code variants, parameter values, etc. to find the best implementation on a specific machine
ATLAS

• **Goal is portable, efficient implementation of BLAS**
  • BLAS are building blocks for performing vector and matrix operations
    • Level 1 is vector-vector
    • Level 2 is matrix-vector
    • Level 3 is matrix-matrix
  • Level 1 has no possible memory reuse, so not addressed
  • Level 2 memory blocking allows for reuse of vector operands, but not matrix
    • Reduces movement of vector operands from $O(N^2)$ to $O(N)$
    • Allows for modest speedups – 10-300%
  • Level 3 blocking allows for reuse of both operands
    • Blocking reduces $O(N^3)$ fetch costs to $O(N^2)$
    • Also better optimizes $O(N^3)$ computation costs than many compilers (run on non-blocked code)
    • Can give orders of magnitude performance improvements
ATLAS

- Level 3 BLAS mainly targets generalized matrix multiplication (GEMM)
  - $C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C$, $\text{op}(X) = X$ or $X^T$
  - $C$ is an $M \times N$ matrix, $\text{op}(A)$ and $\text{op}(B)$ are $M \times K$ and $K \times N$

- Uses both parameterized adaptation and code generation to adapt to a new machine
  - To generate L1 cache-contained matrix multiply kernel

- Most of the paper goes into the details of how to generate the MM kernel that fits into L1 cache
  - All sorts of decisions need to be made about copying matrices, which matrix is in the outermost loop, writing output to $C$ or to an output temporary matrix, choosing loop structure to help with L2 cache reuse
  - ATLAS determines size of L1 data cache, but not L2 (instead computes a value that represents the amount usable for its blocking)
ATLAS

• Other optimizations
  • Instruction cache reuse – fit code into L1 instruction cache
  • Floating point instruction ordering – to hide pipeline latencies (if no fused multiply-add) – modern processors do out-of-order execution in hardware, so this is not needed
  • Reduce loop overhead by loop unrolling
  • Expose instruction-level parallelism – for floating point computations and for memory fetches

• Search heuristic uses a code generator coupled with a timer routine
  • Start with some initial good guesses, then try different loop unrolling and latency hiding strategies to find the best performing variant and parameter values

• Performance results show that ATLAS produces code that is as good as vendor BLAS implementations and much better than what a compiler can do
  • For 500x500 matrices

• Paper also discusses Level 2 BLAS optimization process
  • More complex in some ways than Level 3!