Current and coming OSKI features

Mark Hoemmen
mhoemmen@cs.berkeley.edu

University of California Berkeley

November 4, 2009

Our goals

1. Introduce OSKI sparse matrix library
2. Show both current and proposed features
3. Solicit advice from users:
   - Help us prioritize our work
   - Help us choose interfaces that balance simplicity and cost
   - Teach us about new kernels and optimization possibilities
4. Present current sparse kernels and algorithms research

Who am I?

- I work on communication-avoiding algorithms for sparse and dense linear algebra
- I understand OSKI algorithms and optimizations, but
- Not (yet) an OSKI developer
  - Will likely be more involved in the future
  - I'm here representing OSKI
- Work funded by
  - DOE, NSF, ACM/IEEE, Intel, Microsoft

What is OSKI?

Figure: Oski the Bear (UC Berkeley mascot)

What is OSKI?

- Optimized Sparse Kernel Interface
- "BLAS" for sparse matrix, dense vector ops
- Autotuning C library
  - Automatically picks fast implementation
  - Based on build-time and runtime search
  - Accepts, but does not require, user hints
- Targets cache-based superscalar platforms
  - Shared-memory parallel coming soon!
  - Ongoing research into other platforms

OSKI collaborators

- Project leaders
  - James Demmel, Kathy Yelick
- Current developers
  - Ben Carpenter, Erin Carson, Armando Fox, Rich Vuduc
- Contributed OSKI code
  - Jen Hsu, Shoaib Kamil, Ben Lee, Rajesh Nishtala
- Optimizations and algorithms research
  - Various members of UC Berkeley Benchmarking and Optimization (BeBOP) group
  - For details: `bebop.cs.berkeley.edu`

Kernels currently supported

- What's a kernel?
  - NOT: integral equations, operating systems
  - Computational building block that...
  - ...exposes potential optimizations
- "Classic" kernels
  - Sparse matrix-vector multiply (SpMV)
  - Sparse triangular solve (SpTS)
- "Exotic" kernels that exploit locality
  - Matrix <u>and</u> its transpose: $(x, y) \mapsto (Ax, A^T y)$
  - Matrix <u>times</u> its transpose: $x \mapsto (Ax, A^T Ax)$
  - Power of a matrix: $x \mapsto A^k x$, $k \in \{2, 3, \dots\}$

How much faster?

- Sequential:
  - SpMV: $4\times$
  - SpTS: $1.8\times$
  - $x \mapsto A^T A x$: $4.2\times$
- Parallel:
  - SpMV: $11.3\times$ on 8 cores
- How? Autotuning
  1. Humans develop algorithms and optimizations
  2. Humans write code generation scripts (in any scripting language)
  3. Scripts generate code variants in target language (C)
  4. Offline + runtime search (over code variants and parameters)

How does OSKI work?

- Offline phase (library build time)
  1. Human-written scripts generate code variants
  2. Benchmarks profile hardware
- Online phase (application run time)
  1. Accept sparse matrix in standard format
  2. User can give tuning hints
  3. Library profiles kernels calls to gauge workload
  4. Tune only by explicit user request
  5. User can save tuning strategy, reuse later

Why explicit tuning?

- Tuning <u>expensive</u>
  - Involves copying matrix into new data structure
  - The data structure <u>is</u> the tuning
  - 5–40 SpMVs
- OSKI will NOT tune unless it thinks it pays
  - Users can give <u>workload hints</u>
  - "Will call SpMV 500 $\times$"
  - OSKI counts # kernel calls to guess workload

Three proposed features

Three proposed features

- Many features coming; these three first
- Two proposed by Mike Heroux:
  - Adding nonzeros
  - Graph-only tuning
- One benchmarked and ready to integrate:
  - Shared-memory parallel SpMV
- Give us interface feedback!

Feature 1: Adding nonzeros to an existing matrix

- Many applications
  - Unstructured mesh changes
  - Dynamic graph algorithms
- Not efficiently supported by many sparse data structures
  - May require full copy for one nonzero
- OSKI does not currently support adding nonzeros
- We're exploring ways to change the interface that
  - Minimize costs (memory and time)
  - Are more natural to users

Adding nonzeros = adding sparse matrices

- Adding nonzeros same as adding sparse matrices
  - Old nonzeros: matrix A_1
  - New nonzero(s): matrix A_2
- Merge operation:
  - $(A_1, A_2) \mapsto A$ where $A = A_1 + A_2$
  - Result A: standard sparse matrix data structure
  - Tuning of A_1, A_2 lost

Two possible interfaces for adding nonzeros

- Merge only model
  - Adding a nonzero requires merge
  - Merge may lose tuning info
- Sum of matrices model
  - Treat matrix as (implicit) sum of k matrices
  - SpMV: $Ax = A_1x + A_2x + \cdots + A_kx$
  - Each A_j retains its tuning
  - Option to merge
    - Only per user request for now
    - Automatic tuning decision later

Advantages and disadvantages of Models 1 and 2

- Merge only model
  - Less work for us
  - Narrower API (OSKI is in C)
  - Slower if frequent, small structure changes
- Sum of matrices model
  - Implementation vehicle for many optimizations
  - Naturally supports element assembly (FEM)
  - More work (may be overkill) & wider API
  - Hinders $A^T Ax$ and $A^k x$ (2^k cross terms)

Why split into sum and tune separately?

- Example: UBCSR (Rich Vuduc 2003 PhD thesis)
  - Sum of register-blocked matrices
  - Different register block dimensions for each term
  - Speedup: $2.1\times$ (Itanium 2)
  - Almost always saves memory ($> 50\%$)
- Linear programming and other optimization problems
  - Dense rows / columns common
  - Typical preprocessing step:
    1. Extract dense structures
    2. Express matrix as sparse matrix plus outer product

Feature 2: Graph-only tuning

- Tuning a matrix using only its graph structure
  - No need to store nonzeros sometimes (e.g., Laplacian graph)
  - Multiple matrices with same structure but different nonzeros
  - Avoid copying nonzeros (not needed for tuning)
- Partly supported by OSKI already
  - Can save and restore tuning transformations
  - Tune on a matrix with "dummy" nonzero values
  - Recycle tuning for matrices with same structure
- Mainly software engineering
  - OSKI only tunes using matrix structure anyway
  - But we haven't explored no-explicit-nonzeros case

Feature 3: shared-memory parallel backend

- SpMV only, benchmark-quality prototype
- Ankit Jain, UC Berkeley CS Master's Thesis, 2008
- Excellent speedups over <u>optimized</u> serial:
  - 9x on AMD Santa Rosa (2 socket $\times$ 2 core)
  - 11.3x on AMD Barcelona (2 socket $\times$ 4 core)
  - 7.2x on Intel Clovertown (2 socket $\times$ 4 core)
- More speedup than # cores, due to
  - Search over 2-D block layouts
  - NUMA optimizations

Shared-memory parallel interface

- Ankit made new interface for parallel version
  - Looks sequential to users
  - Pool of fixed # of Pthreads underneath
- Question: is that the interface you want?
  - Sequential front-end, parallel back-end? or
  - Single Program Multiple Data (SMPD) – MPI-style?
  - Pthreads, OpenMP, TBB, ...?

Problem: nested parallel library calls

- What if user library makes parallel calls to OSKI?
  - Special case of nested parallelism ("parallel calls parallel")
- Nested parallelism example: sparse QR
  - Cilk or Intel TBB for parallelism in elimination tree
  - Each thread may then call (multithreaded) BLAS
- At best: libraries fight for cores
  - Each library expects to own all cores
  - Some libraries demand exclusive ownership
- At worst: horrible bugs

Nested parallelism is ongoing research

- UC Berkeley ParLab project: Lithe
- Leave user code alone, change system libraries
  - Any sequential-looking interface can be parallel inside
  - Use OpenMP, TBB, Pthreads, . . . as before. . .
  - . . . but each of these needs new Lithe-based scheduler.
- Proof of concept: sparse QR calling BLAS
- Invasive to system libraries, work in progress

Questions on shared-memory parallel version

- Users want parallel now, before Lithe
- We will likely support some non-Lithe parallel version
- Questions:
  - Will you call OSKI in a parallel context?
  - Do your systems support Pthreads, OpenMP, ...?
  - Will you want to restrict # cores used by OSKI?
  - Our NUMA optimizations target Linux – other platforms?

Higher-level languages (HLLs) in OSKI

Higher-level languages (HLLs) in OSKI

- Why we want HLLs inside OSKI
- Why users might want HLL interfaces
- Audience feedback

Why OSKI developers want HLLs inside OSKI

- Already there!
  - Embedded domain-specific language
  - Algebra for matrix data structure transformations
  - Not meant for users (yet)
- Tuning decisions vs. tuned kernels
  - Tuning decision code not performance-critical. . .
  - . . . yet often source of most bugs and development time.
  - It's why prototyped kernels take so long to deploy in OSKI!
  - HLL dramatically increases (our) productivity
- HLLs as development accelerators
  - Implement new features first in HLL (calling into C)
  - If performance demands it, push new features into C

Why users might want HLL interface to OSKI

- Interfaces in lower-level languages mix productivity and efficiency code
  - "Productivity code": computation users want to do
  - "Efficiency code": tuning and implementation choices for performance
  - Mixing constrains tuning and kills user productivity
- HLLs natural fit for interface of tuned libraries
  - Separate tuning policy from computation
  - OSKI free to experiment with complex optimizations...
  - ..."in parallel" while users experiment with computation.

PySKI: Python Sparse Kernel Interface

- Use Python because of SciPy
  - Popular Matlab-like Python environment
  - `scipy.sparse`: Sparse matrix wrapper
- Modify `scipy.sparse` to call OSKI methods
- Tuned OSKI data structures live as before in C world
  - Python code only deals with pointers
  - Minimizes memory and copy overhead
  - Preserves tuning
- Experimental vehicle for HLL interfaces

Audience questions on HLLs

- Does HLL inside OSKI scare you?
  - Even if users never see it?
- Will OSKI users (= Trilinos developers?) want HLL interface?
- How portable must the HLL be? (OS, compiler, hardware)
  - Some HLLs only need a C compiler, but fewer features
  - Python heavier-weight, but has libraries we want

Proposed feature: Matrix powers kernel

Proposed feature: Matrix powers kernel

- $(A, x) \mapsto (Ax, A^2x, \dots, A^s x)$ (or similar)
- Can compute for same communication cost as one SpMV
- See Demmel et al.\ 2007, 2008, 2009 (SC09)
- Includes multicore optimizations (SC09)
- Applications
  - Chebyshev iteration
  - Lookahead for nonsymmetric Lanczos / BiCG
  - s-step iterative methods

s-step iterative methods

- Reorganization of existing Krylov subspace methods
- Compute s Krylov subspace basis vectors
  - All at once, using matrix powers kernel
- Use BLAS 3 to orthogonalize them
  - Tall Skinny QR (TSQR): stable and optimal communication
- CG, GMRES, (symmetric) Lanczos, Arnoldi
- Details in SC09, and my thesis (almost done!)

Kernel co-tuning

- Our SC09 GMRES has three kernels
  - Matrix powers
  - Tall Skinny QR
  - Block Gram-Schmidt
- Tuning for one affects others
  - Data layout essential to performance
  - Copy in/out btw formats too slow
- Workload fraction per kernel depends on runtime params
  - Restart length
  - Sparse matrix structure
- Must tune entire app / composition of kernels

Challenges

Challenges (1 of 2)

- Composing multiple optimizations
  - Some optimizations change sparsity structure
    - Register blocking adds nonzeros
    - Changes optimizations that partition the matrix
    - Cache blocking, matrix powers, reordering for locality, ...
  - If noncommutative, which order? not all orders make sense
- Co-tuning (Composing multiple kernels)
  - Multiple kernels share data layout, but...
  - ...data layout part of tuning!
  - What interface should kernels export for co-tuning?

Challenges (2 of 2)

- Correctness
  - Performance depends on many autogenerated code variants
  - Some matrix data structures have tricky corner cases
  - Current correctness proofs effort at UC Berkeley
- Search: Combinatorial explosion
  - Heterogeneous and rapidly evolving hardware
    - Multiple levels of memory hierarchy
    - NUMA: Nonuniform memory latencies and bandwidths
    - Compute accelerators like GPUs
  - More and more optimizations and parameters
  - Runtime benchmarking expensive
  - Need smarter search
    - Performance bounds as stopping criterion
    - More information out of fewer samples

Conclusions

- OSKI: optimized "sparse matrix BLAS"
- New features and optimizations in progress
- Interesting research and software development challenges
- We want user feedback!

Extra slides

Why no distributed-memory OSKI?

- Dist-mem search too expensive
  - Single-node already takes hours
  - Build-time search must discover hardware
  - Network topology runtime-dependent
    - Number of procs
    - Job scheduling
  - Must also discover matrix structure at runtime
- Memory bandwidth matters
  - Clearly dominates single-node performance
  - vs.\ message latency – not always
  - Multicore / GPU: more procs, less bw
- Intended use: inside dist-mem library
  - Already wrapped inside PETSc