Empirical Optimization

Keshav Pingali

Context: HPC software

- Traditional approach
  - Hand-optimized code: (e.g.) BLAS
  - Problem: tedious to write by hand
- Alternatives:
  - Restructuring compilers
    - General purpose: generate code from high-level specifications
    - Use architectural models to determine optimization parameters
  - Library generators
    - Problem-specific: (e.g.) ATLAS for BLAS, FFTW for FFT
    - Use empirical optimization to determine optimization parameters

How good are these approaches?

Our approach

- Original ATLAS Infrastructure
- Model-Based ATLAS Infrastructure

BLAS

- Let us focus on MMM:

  ```
  for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
  for (k = 0; k < K; k++)
  C[i][j] += A[i][k]*B[k][j]
  ```

- Properties
  - Very good reuse: O(N^2) data, O(N^3) computation
  - Many optimization opportunities
  - Few "real" dependencies
  - Will run poorly on modern machines
    - Poor use of cache and registers
    - Poor use of processor pipelines

Optimizations

- Cache-level blocking (tiling)
  - Atlas blocks only for L1 cache
  - NB: L1 cache time size
  - Register-level blocking
  - Important to hold array values in registers
  - M.U1: register tile size
  - Software pipelining
  - Unroll and schedule operations
  - Latency, xFetch: scheduling parameters
  - Versioning
  - Dynamically decide which way to compute
  - Back-end compiler optimizations
  - Scalar Optimizations
  - Instruction Scheduling

Cache-level blocking (tiling)

- Tiling in ATLAS
  - Only square tiles (NbNxNbNxNb)
  - Working set of tile fits in L1
  - Tiles are usually copied to continuous storage
  - Special "clean-up" code generated for boundaries
- Mini-MMM

```python
for (i = 0; i < M; i++)
for (j = 0; j < N; j++)
for (k = 0; k < K; k++)
C[i][j] = A[i][k]*B[k][j]
```
Register-level blocking

- Micro-MMM
  - A: MM1
  - B: NUx1
  - C: MUxNU
- Mini-MMM with Micro-MMM inside
- Unroll loops by MU, NU, and KU
- MU and NU: optimization parameters
  - MU * NU + MU + NU registers
- Special clean-up code required if NB is not a multiple of MU, NU, KU
- MU, NU, KU: optimization parameters

Scheduling

- FMA Present?
- Schedule Computation
  - Using Latency
- Schedule Memory Operations
  - Using IFetch, NFetch, FFetch

ATLAS Search Strategy

- Multi-dimensional optimization problem:
  - Independent parameters: NB, MU, NU, KU, ...
  - Dependent parameter: MFlops
  - Function from parameters to variables is given implicitly; can be evaluated repeatedly
- One optimization strategy: orthogonal line search
  - Optimize along one dimension at a time, using reference values for parameters not yet optimized
  - Not guaranteed to find optimal point, but might come close

Find Best NB

- Search in following range
  - 16 <= NB <= 80
  - NB^2 <= L1Size
- In this search, use simple estimates for other parameters
  - (eg) KU: Test each candidate for
    - Full unrolling (KU = NB)
    - No unrolling (KU = 1)

Model-based optimization

- Original ATLAS Infrastructure
- Model-Based ATLAS Infrastructure

Modeling for Optimization Parameters

- Optimization parameters
  - NB
  - Hierarchy of Models (later)
  - MU, NU
  - MU * NU + MU + NU + Latency <= NR
  - KU
  - maximize subject to L1 Instruction Cache
  - Latency
  - BL <= 1x2
  - MulAdd
    - hardware parameter
    - MFAdd = set to 2
Largest NB for no capacity/conflict misses

- If tiles are copied into contiguous memory, condition for only cold misses:
  \[ 3^2 \leq \frac{L1\text{Size}}{2^2} \]

Cache model:
- Fully associative
- Line size 1 Word
- Optimal Replacement

Bottom line:
- \( NB^2 + NB + 1 < C \)
- One full matrix
- One row / column
- One element

Summary: Modeling for Tile Size (NB)

- Models of increasing complexity
  - Whole work-set fits in L1
  - \( NB^2 + NB + 1 < C \)
  - Fully Associative
  - Optimal Replacement
  - Line size 1 word
  - Line size > 1 word
  - LRU Replacement

Largest NB for no capacity misses

- MMM:
  - \( \text{Not used in context} \)
- Cache model:
  - Fully associative
  - Line size 1 Word
  - Optimal Replacement

Summary of model

Experiments

- Ten modern architectures
- Model did well on RISC architectures
- UltraSparc: did better
- Model did not do as well on Itanium
- Substantial gap between ATLAS CGw/S and ATLAS Unleashed on some architectures

Some sensitivity graphs for Alpha 21264
Eliminating performance gaps

- Think globally, search locally
- Gap between model-based optimization and empirical optimization can be eliminated by
  - Local search:
    - for small performance gaps
    - in neighborhood of model-predicted values
  - Model refinement:
    - for large performance gaps
    - must be done manually
  - (future) machine learning: learn new models automatically
- Model-based optimization and empirical optimization are not in conflict

Small performance gap: Alpha 21264

- Local search:
  - Around model-predicted NB
  - Hill-climbing not useful
  - Search interval: \([NB-lcm(MU,NU),NB+lcm(MU,NU)]\)
  - Local search for MU,NU
  - Hill-climbing OK

Large performance gap: Itanium

- Memory hierarchy
  - L1 data cache: 16 KB
  - L2 cache: 256 KB
  - L3 cache: 3 MB
- Diagnosis:
  - Model tiles for L1 cache
  - On Itanium, FP values not cached in L1 cache!
  - Performance gap goes away if we model for L2 cache (NB = 105)
  - Obtain even better performance if we model for L3 cache (NB = 360, 4.6 GFlops)
- Problem:
  - Tiling for L2 or L3 may be better than tiling for L1
  - How do we determine which cache level to tile for??
- Our solution: model refinement + a little search
  - Determine tile sizes for all cache levels
  - Choose between them empirically

Large performance gap: Opteron

- Opteron characteristics
  - Small number of logical registers
  - Out-of-order issue
  - Register renaming
- For such processors, it is better to
  - let hardware take care of scheduling dependent instructions,
  - use logical registers to implement a bigger register tile.
  - x86 has 8 logical registers
  - register tiles must be of the form \((x,1)\) or \((1,x)\)
**Refined model**

- Refined model is not complex.
- Refined model by itself eliminates most performance gaps.
- Local search eliminates all performance gaps.

**Bottom line**

- Refined model is not complex.
- Refined model by itself eliminates most performance gaps.
- Local search eliminates all performance gaps.

**Future Directions**

- Repeat study with FFTW/SPIRAL
  - Uses search to choose between algorithms
- Feed insights back into compilers
  - Build a linear algebra compiler for generating high-performance code for dense linear algebra codes
    - Start from high-level algorithmic descriptions
    - Use restructuring compiler technology
- Generalize to other problem domains
- How can we get such systems to learn from experience?