Optimizing MMM & ATLAS Library Generator

Recall: MMM miss ratios

L1 Cache Miss Ratio for Intel Pentium III
- MMM with \( N = 1 \ldots 1500 \)
  - 16KB 32B/Block 4-way 8-byte elements

**IJK version (large cache)**

for \( I = 1, N/\)row-major storage
for \( J = 1, N \)
for \( K = 1, N \)
\[
C(I,J) = C(I,J) + A(I,K) \times B(K,J)
\]

**Large cache scenario:**
- Matrices are small enough to fit into cache
- Only cold misses, no capacity misses
- Miss ratio:
  - Data size = \( 3N^2 \)
  - Each miss brings in \( b \) floating-point numbers
  - Miss ratio = \( 3N^2 \times b \times 4N^3 = 0.75bN \) \( \text{[0.019 if } b = 4, N=10] \)

**Small cache scenario:**
- Matrices are large compared to cache
  - Reuse distance is not \( O(1) \) => misses
  - Cold and capacity misses
- Miss ratio:
  - A: \( N^2/b \) misses (good spatial locality)
  - B: \( N^2 \) misses (poor temporal and spatial locality)
  - C: \( N/b \) misses (good temporal locality)
  - Miss ratio \( \rightarrow 0.25(b+1)/b = 0.3125 \) \( \text{[for } b = 4] \)
MMM experiments

L1 Cache Miss Ratio for Intel Pentium III
- MMM with N = 1…1300
  - 16KB 32B/Block 4-way 8-byte elements

Can we predict this?

How large can matrices be and still not suffer capacity misses?

for I = 1, N
for J = 1, N
for K = 1, N
C(I,J) = C(I,J) + A(I,K)*B(K,J)

- How large can these matrices be without suffering capacity misses?
  - Each iteration of outermost loop walks over entire B matrix, so all of B must be in cache
  - We walk over rows of A and successive iterations of middle loop touch same row of A, so one row of A must be in cache
  - We walk over elements of C one line at a time.
  - So inequality is N² + N + b <= Capacity of cache

Check with experiment

- For our machine, capacity of L1 cache is 16KB/8 doubles = 2¹¹ doubles
- If matrices are square, we must solve
  N² + N + 4 = 2¹¹
  which gives us N ~ 45
- This agrees with experiment.

High-level picture of high-performance MMM code

- Block the code for each level of memory hierarchy
  - Registers
  - L1 cache
  - ...
- Choose block sizes at each level using the theory described previously
  - Useful optimization: choose block size at level L+1 to be multiple of the block size at level L
ATLAS

- Library generator for MMM and other BLAS
- Blocks only for registers and L1 cache
- Uses search to determine block sizes, rather than the analytical formulas we used
  - Search takes more time, but we do it once when library is produced
- Let us study structure of ATLAS in little more detail

Our approach

- Original ATLAS Infrastructure
- Model-Based ATLAS Infrastructure

BLAS

- Let us focus on MMM:
  ```
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      for (int 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
  - MU,NU: register tile size
- Filling the processor pipeline
  - 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 (NBxNBxNB)
  - Working set of tile fits in L1
  - Tiles are usually copied to continuous storage
  - Special "clean-up" code generated for boundaries

- Mini-MMM
  ```c
  for (int j = 0; j < NB; j++)
  for (int i = 0; i < NB; i++)
  for (int k = 0; k < NB; k++)
      C[i][j] += A[i][k] * B[k][j]
  ```

- NB: Optimization parameter

Register-level blocking

- Micro-MMM
  ```c
  A: MUx1
  B: 1xNU
  C: MUxNU
  ```
  - MUxNU+MU+NU registers
  - Unroll loops by MU, NU, and KU

- Mini-MMM with Micro-MMM inside
  ```c
  for (int j = 0; j < NB; j += NU)
  for (int i = 0; i < NB; i += MU)
      load C[i..i+MU-1, j..j+NU-1] into registers
      load A[i..i+MU-1,k] into registers
      load B[k,j..j+NU-1] into registers
      multiply A's and B's and add to C's
      store C[i..i+MU-1, j..j+NU-1]
  ```

- Special clean-up code required if NB is not a multiple of MU,NU,KU

Scheduling

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

- Latency, xFetch: optimization parameters

Search Strategy

- Multi-dimensional optimization problem:
  - Independent parameters: NB,MU,NU,KU,...
  - Dependent variable: 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
  - $16 \leq NB \leq 80$
  - $NB^2 \leq L1Size$
- In this search, use simple estimates for other parameters
  - (eg) KU: Test each candidate for
    - Full K unrolling (KU = NB)
    - No K unrolling (KU = 1)

Model-based optimization

- Original ATLAS Infrastructure
- Model-Based ATLAS Infrastructure

Modeling for Optimization Parameters

- Optimization parameters
  - NB
  - MU, NU
    - $MU \cdot NU + MU + NU + \text{Latency} \leq NR$
  - KU
    - maximize subject to L1 Instruction Cache
  - Latency
    - $[L + 1/2]$
  - MulAdd
    - hardware parameter
  - xFetch
    - set to 2

Largest NB for no capacity/conflict misses

- If tiles are copied into contiguous memory, condition for only cold misses:
  - $3 \cdot NB^2 \leq L1Size$
Largest NB for no capacity misses

- MMM:
  - for (int j = 0; i < N; i++)
  - for (int i = 0; j < N; j++)
  - for (int k = 0; k < N; k++)
  - c[i][j] += a[i][k] * b[k][j]

- 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
  - 2NB^2 ≤ C
  - Whole work-set fits in L1
  - NB^2 + NB + 1 ≤ C
  - Fully Associative
  - Optimal Replacement
  - Line Size: 1 word
  - Line Size: > 1 word
  - LRUR 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
  - CISC architectures
- 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
- ATLAS CGw/S:
  - mini-MMM: 1300 MFlops
  - NB = 72
  - (MU,NU) = (4,4)
- ATLAS Model
  - mini-MMM: 1200 MFlops
  - NB = 84
  - (MU,NU) = (4,4)

  - 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
- Performance of mini-MMM
  - ATLAS CGw/S: 4000 MFlops
  - ATLAS Model: 1800 MFlops
- Problem with NB value
  - ATLAS Model: 30
  - ATLAS CGw/S: 80 (search space max)
  - Local search will not solve problem.
Itanium diagnosis and solution

- 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

Opteron diagnosis and solution

- 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)

Large performance gap: Opteron

- MMM Performance
  - Performance of mini-MMM
    - ATLAS CGw/S: 2072 MFlops
    - ATLAS Model: 1282 MFlops
- Key differences in parameter values: MU/NU
  - ATLAS CGw/S: (6,1)
  - ATLAS Model: (2,1)

Refined model
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