Overview

- ATLAS: portable BLAS generator
  - Implemented by Dongarra (UTK), Demmel (UCB) et al.
  - We will focus on MMM
- Importance: popularized the idea of auto-tuning
  - Generate-and-test
  - Program generator to generate program variants
  - Test performance of variant by running program
- Program variants in ATLAS
  - Iterative blocked kernels for different levels of memory hierarchy
- Auto-tuning useful for library generators
  - Large upfront cost for generate-and-test

Overview

- ATLAS Library Generator
  - Lecture based on these papers:
    - “A comparison of empirical and model-driven optimization” Yotov et al, PLDI 2003
    - “Is search really necessary to generate high-performance BLAS?” Yotov et al, Proceedings of IEEE, 93(2), 2005
    - “Think globally, search locally” Yotov et al., ICS 2005

BLAS

- Let us focus on MMM:
  ```c
  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(N²) data, O(N³) computation
  - Many optimization opportunities
    - Few “real” dependencies
    - Will run poorly on modern machines
    - Poor use of cache and registers
    - Poor use of processor pipelines
- Key optimization
  - Blocking/tiling to improve temporal locality

Why blocking?

- Assume blocks fit in cache during block computation
- # of cache misses for block data = 3B²/L (L: line size)
- # of block computations = (N/B)³
- Total number of misses = (N/B)³ * (3B²/L) = 3N³BL
- High-level picture:
  - number of cache misses decreases with block size as long as working set of block computation fits in cache
Optimal block size

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

- Easy computation
  - Need space in cache for 3 blocks of B²
    - So choose largest B for which 3B² < C
- Careful accounting: to avoid capacity misses, need space in cache for
  - block of B
  - row of A
  - one element (line) of C
  - loop order determines which matrices
- For our problem:
  - B² + B + L < C (with optimal replacement)
  - B² + 2B < C (with LRU replacement)
- In either case, we get B ~ sqrt(C)

High-level picture of high-performance MMM code

- Block the code for each level of memory hierarchy
  - Registers: requires loop unrolling
  - 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

Importance of multi-level blocking

"An Experimental Comparison of Cache-oblivious and Cache-conscious programs"
Yotov et al, SPAA 2007

MMM experiments

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

Optimal value of B
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

Parameters in ATLAS code

- 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
  - NU,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: nothing to do with ATLAS
  - 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 < MM; j++)
  for (int k = 0; k < MM; k++)
    C[i][j] += A[i][k] * B[k][j];
```

- NB: Optimization parameter

Study in Yotov et al. paper

- Original ATLAS Infrastructure
- Model-Based ATLAS Infrastructure

Parameters in ATLAS code

- 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: nothing to do with ATLAS
  - 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 < MM; j++)
  for (int k = 0; k < MM; k++)
    C[i][j] += A[i][k] * B[k][j];
```

- NB: Optimization parameter
Register-level blocking

- Micro-MM
  - A: MUx1
  - B: 1xNU
  - C: MUxNU
  - MUxNU=MU+NU registers
- Unroll loops by MU, NU, and KU
- Mini-MM with Micro-MM inside
  - 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
  - for (int k = 0; k < NB; k++)
  - 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
- MU, NU, KU: optimization parameters

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: GFlops
  - 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 K unrolling (KU = NB)
    - No K unrolling (KU = 1)
Model-based optimization

- Original ATLAS Infrastructure
  - Detect Hardware Parameters
  - ATLAS Search Engine (MMSearch)
  - ATLAS MM Code Generator (MMCase)
  - Compile, Execute, Measure

- Model-Based ATLAS Infrastructure
  - Detect Hardware Parameters
  - Model
  - ATLAS MM Code Generator (MMCase)
  - Compile, Execute, Measure

Modeling for Optimization Parameters

- Optimization parameters
  - NB
    - Hierarchy of Models (later)
  - MU, NU
    - \( MU \times \text{NU} + \text{Latency} \leq \text{NR} \)
  - KU
    - maximize subject to L1 Instruction Cache
  - Latency
    - \( \text{Latency}_{\text{L1}} = 1/2 \)
  - MulAdd
    - hardware parameter
  - xFetch
    - set to 2

Modeling for Tile Size (NB)

- Models of increasing complexity
  - \( 3 \times \text{NB} \leq C \)
    - Whole work-set fits in L1
  - \( \text{NB}^2 + \text{NB} + 1 \leq C \)
    - Fully Associative
    - Optimal Replacement
  - Line Size: 1 word
    - \( \text{NB} \times \frac{1}{2} \leq C \)
  - Line Size: 1 word
    - \( \text{NB} \times \frac{1}{2} \leq C \)
    - LRU Replacement

Summary of model

- \( L = \frac{\text{NR} - \text{Latency}_{\text{L1}} - 1}{2} \)
- \( \text{Estimating } L \)
- \( \text{Estimating } \text{NB} \) and \( \text{No} \)
- \( \text{Maximize } \text{No} \) subject to L1 Instruction Cache
- \( \text{Latency}_{\text{L1}} = 1/2 \)
- \( \text{Estimating } \text{No} \)
- \( \text{Estimating } P_{\text{F1}}, P_{\text{F2}} \) and \( P_{\text{F3}} \)
- \( P_{\text{F1}}, P_{\text{F2}}, P_{\text{F3}} = 2 \text{NB} - 1 \)
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

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)

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

Large performance gap: Opteron

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)
Refined model

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

Performance of MMM code produced by Intel’s Itanium compiler (-O3)

Goto BLAS obtains close to 99% of peak, so compiler is pretty good!

Things to think about

- What is the space of program variants?
  - Space must include the optimal point or at least points close to it in performance
  - Question: what kinds of MMM implementations are not explored by ATLAS?
- What is the search strategy and is it guaranteed to find the optimal (or at least very good) point?
  - ATLAS uses orthogonal line search
  - One general problem: you spend much more time executing non-optimal program variants than the optimal one
  - Some notion of importance sampling might be useful if search time matters
- What were the key approximations made in the analytical performance model?
  - Need to look at model used for each parameter
  - Are these approximations reasonable?