Optimizing MMM & ATLAS Library Generator
Recall: MMM miss ratios

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

DO I = 1, N // row-major storage
  DO J = 1, N
    DO K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)

- Large cache scenario:
  - Matrices are small enough to fit into cache
  - Only cold misses, no capacity misses
  - Miss ratio:
    - Data size = 3 N^2
    - Each miss brings in b floating-point numbers
    - Miss ratio = 3 N^2 / b*4N^3 = 0.75/bN = 0.019 (b = 4, N=10)
IJK version (small cache)

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

- Small cache scenario:
  - Matrices are large compared to cache
    - reuse distance is not O(1) => miss
  - Cold and capacity misses
  - Miss ratio:
    - C: $N^2/b$ misses (good temporal locality)
    - A: $N^3/b$ misses (good spatial locality)
    - B: $N^3$ misses (poor temporal and spatial locality)
    - Miss ratio $\rightarrow 0.25 \frac{(b+1)}{b} = 0.3125$ (for $b = 4$)
MMM experiments

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

Can we predict this?
How large can matrices be and still not suffer capacity misses?

DO I = 1, M
    DO J = 1, N
        DO K = 1, P
            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 at a time.
  - So inequality is NP + P + 1 <= C
Check with experiment

- For our machine, capacity of L1 cache is 16KB/8 doubles = $2^{11}$ doubles
- If matrices are square, we must solve $N^2 + N + 1 = 2^{11}$
  which gives us $N = 45$
- This agrees well 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**
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: $O(N^2)$ 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-MMMM**
  - A: MUx1
  - B: 1xNU
  - C: MUxNU
  - MUxNU+MU+NU registers

- **Unroll loops by MU, NU, and KU**

- **Mini-MMMM with Micro-MMMM inside**

```java
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: 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
  - $16 \leq NB \leq 80$
  - $NB^2 \leq L1\text{Size}$
- 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
  - L1Size, NR, MulAdd, L*
  - ATLAS Search Engine (MMSearch)
  - NB, Mu, Nu, Ku, xFetch, MulAdd, Latency
  - ATLAS MM Code Generator (MMCase)

  → MFLOPS

  → MiniMMM Source

- **Model-Based ATLAS Infrastructure**

  - Detect Hardware Parameters
  - L1Size, L1I$Size, NR, MulAdd, L*
  - Model
  - NB, Mu, Nu, Ku, xFetch, MulAdd, Latency
  - ATLAS MM Code Generator (MMCase)

  → MiniMMM Source
Modeling for Optimization Parameters

- Optimization parameters
  - NB
    - Hierarchy of Models (later)
  - MU, NU
    - $MU \times NU + MU + NU + \text{Latency} \leq NR$
  - 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:
  - $3 \times \text{NB}^2 \leq \text{L1Size}$
Largest NB for no capacity misses

- **MMM:**
  ```java
  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
  - $3 \times NB^2 \leq C$
    - Whole work-set fits in L1
  - $NB^2 + NB + 1 \leq C$
    - Fully Associative
    - Optimal Replacement
    - Line Size: 1 word
  - $\left\lfloor \frac{NB^2}{B} \right\rfloor + \left\lfloor \frac{NB}{B} \right\rfloor + 1 \leq \frac{C}{B}$ or $\left\lfloor \frac{NB^2}{B} \right\rfloor + NB + 1 \leq \frac{C}{B}$
    - Line Size > 1 word
  - $\left\lfloor \frac{NB^2}{B} \right\rfloor + 2 \left\lfloor \frac{NB}{B} \right\rfloor + \left( \left\lfloor \frac{NB}{B} \right\rfloor + 1 \right) \leq \frac{C}{B}$ or $\left\lfloor \frac{NB^2}{B} \right\rfloor + 3NB + 1 \leq \frac{C}{B}$
    - LRU Replacement
Summary of model

- **Estimating FMA:**
  Use the machine parameter FMA

- **Estimating \( L_s \):**

\[
L_s = \left\lfloor \frac{L_s \times |ALU_{FP}| + 1}{2} \right\rfloor
\]

- **Estimating \( M_U \) and \( N_U \):**

\[
M_U \times N_U + N_U + M_U + L_s \leq N_R
\]

1) \( M_U, N_U \leftarrow u \).
2) Solve constraint for \( u \).
3) \( M_U \leftarrow \max(u, 1) \).
4) Solve constraint for \( N_U \).
5) \( N_U \leftarrow \max(N_U, 1) \).
6) If \( M_U < N_U \) then swap \( M_U \) and \( N_U \).

- **Estimating \( N_B \):**

\[
\left\lfloor \frac{N_B^2}{B_1} \right\rfloor + 3 \left\lfloor \frac{N_B \times N_U}{B_1} \right\rfloor + \left\lfloor \frac{M_U}{B_1} \right\rfloor \times N_U \leq \frac{C_1}{B_1}
\]

Trim \( N_B \), to make it a multiple of \( M_U, N_U, \) and 2.

- **Estimating \( K_U \):**

Choose \( K_U \) as the maximum value for which mini-MMM fits in the L1 instruction cache. Trim \( K_U \) to make it divide \( N_B \) evenly.

- **Estimating \( F_F, I_F, \) and \( N_F \):**

\[
F_F = 0, I_F = 2, N_F = 2
\]
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

MMM Performance

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
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)
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

- **Estimating $FMA$:**
  Use the machine parameter $FMA$

- **Estimating $L_s$:**
  
  \[
  L_s = \left\lfloor \frac{L_\ast \times |ALU_{FP}| + 1}{2} \right\rfloor
  \]

- **Estimating $M_U$ and $N_U$:**
  
  \[
  M_U \times N_U + N_U + M_U + L_s \leq N_R
  \]
  
  1) $M_U, N_U \leftarrow u$.
  2) Solve constraint for $u$.
  3) $M_U \leftarrow \max(u, 1)$.
  4) Solve constraint for $N_U$.
  5) $N_U \leftarrow \max(N_U, 1)$.
  6) If $M_U < N_U$ then swap $M_U$ and $N_U$.
  7) **Refined Model:** If $N_U = 1$ then
     - $M_U \leftarrow N_R - 2$
     - $N_U \leftarrow 1$
     - $FMA \leftarrow 1$

- **Estimating $N_B$:**
  
  \[
  \left\lfloor \frac{N_B^2}{B_1} \right\rfloor + 3 \left\lfloor \frac{N_B \times N_U}{B_1} \right\rfloor + \left\lfloor \frac{M_U}{B_1} \right\rfloor \times N_U \leq \frac{C_1}{B_1}
  \]
  
  Trim $N_B$, to make it a multiple of $M_U$, $N_U$, and 2.

- **Estimating $K_U$:**
  Choose $K_U$ as the maximum value for which mini-MMM fits in the L1 instruction cache. Trim $K_U$ to make it divide $N_B$ evenly.

- **Estimating $F_F$, $I_F$, and $N_F$:**
  
  $F_F = 0, I_F = 2, N_F = 2$
• 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
    - Part IBM PERCS Project
  - Generalize to other problem domains