CoE 163

Computing Architectures and Algorithms

Linear Algebra Software Libraries

Recap

- Previous discussion showed how careful implementation of an algorithm can improve memory/cache behavior of MMM
- Some techniques that were explored:
  - Choosing the better loop order
  - Blocking
- General ideas
  - If matrix is large, cache cannot hold all the matrix operands -> cache misses are costly
  - Shorter strides can be advantageous in traversing elements of a matrix

Recap

- Parameters to consider for optimization
  - Column-major vs row-major
  - Size of cache
  - Size of matrices
  - (MMM with blocking) Size of blocks
    - $N = \boldsymbol{n}\sqrt{\frac{\boldsymbol{3}}{\boldsymbol{M}}}$
    - Selection of N that optimizes our algorithm depends on size of matrix n and size of cache M

Basic Linear Algebra Subroutines (BLAS)

Standardizing common operations can be cost-effective

- Operations like MMM are so common
- Manufacturers have standardized these common operations as the Basic Linear Algebra Subroutines (BLAS)
- Can achieve portability and efficiency for wide range of kernel scientific computations

The BLAS (http://www.netlib.org/blas/)

- High quality "building block" routines for basic vector and matrix operations
  - Level 1: scalar, vector, & vector-vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- Provides specification of the semantics and syntax for the operations
- Computer vendors or software vendors provide implementations of BLAS that are optimized for specific machine architectures

The BLAS (http://www.netlib.org/blas/)

- Platform independent and free library alternatives are available:
  - ATLAS automatically generates an optimized BLAS library for a given architecture
  - OpenBLAS (a fork of GotoBLAS) is a free open-source alternative to the vendor BLAS implementations
    - Packaged on many end-user Linux distributions such as Ubuntu
    - Readily available for users who perform calculations on their personal computers
    - Decent speed and fairly competitive with Vendor BLAS

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **saxpy** operation ("sum of $\boldsymbol{\alpha x}$ plus $\boldsymbol{y}$"):

$$\boldsymbol{y} \coloneqq \boldsymbol{\alpha x} + \boldsymbol{y}$$

where $\boldsymbol{\alpha} \in \mathbb{R}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{\boldsymbol{n}}$

Example values when $n = 2$:

$$\alpha = 3, \qquad y = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \qquad x = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation ("sum of αx plus y"):

$$\boldsymbol{y} := \alpha \boldsymbol{x} + \boldsymbol{y}$$

Let's compute the number of memory operations (read or write to/from memory)

**Assume an optimized/efficient algorithm is being used

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the `saxpy` operation ("sum of αx plus y"):

$$y := \alpha x + y$$

Load α
into
register
once

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation (“sum of αx plus y”):

$$y := \alpha x + y$$

Load α into register once

Read n elements of x into cache

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation ("sum of αx plus y"):

$$y := \alpha x + y$$

Load α into register once

Read n elements of x into cache

Read n elements of y into cache

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation ("sum of $\boldsymbol{\alpha x}$ plus $\boldsymbol{y}$"):

$$\boldsymbol{y} := \boldsymbol{\alpha x} + \boldsymbol{y}$$

Write/update n elements of y back to memory

Load α into register once

Read n elements of x into cache

Read n elements of y into cache

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation ("sum of $\boldsymbol{\alpha x}$ plus $\boldsymbol{y}$"):

$$\boldsymbol{y} \coloneqq \boldsymbol{\alpha x} + \boldsymbol{y}$$

Write/update n elements of y back to memory

Load α into register once

Read n elements of x into cache

Read n elements of y into cache

$$n \quad + \quad 1 \quad + \quad n \quad + \quad n$$

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation ("sum of $\boldsymbol{\alpha x}$ plus $\boldsymbol{y}$"):

$$\boldsymbol{y} := \boldsymbol{\alpha x} + \boldsymbol{y}$$

Write/update n elements of y back to memory

Load α into register once

Read n elements of x into cache

Read n elements of y into cache

$3n + 1$ memory operations

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation ("sum of $\boldsymbol{\alpha x}$ plus $\boldsymbol{y}$"):

$$\boldsymbol{y} \coloneqq \boldsymbol{\alpha x} + \boldsymbol{y}$$

$2n$ floating point operations

$3n + 1$ memory operations

Performance of BLAS

- Level 1: scalar, vector, & vector-vector operations

Consider the **`saxpy`** operation ("sum of $\boldsymbol{\alpha x}$ plus $\boldsymbol{y}$"):

$$\boldsymbol{y} \coloneqq \boldsymbol{\alpha x} + \boldsymbol{y}$$

Approximately 3 memory operations for
every 2 floating point operation

$$q = \frac{f}{m} = \frac{2n}{3n+1}$$

$$\boldsymbol{q} \approx \frac{\mathbf{2}}{\mathbf{3}}$$

Performance of BLAS

- Level 2: matrix-vector operations

$$\boldsymbol{y} \coloneqq \boldsymbol{A}\boldsymbol{x} + \boldsymbol{y}$$

where $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{\boldsymbol{n}}$

- $\boldsymbol{n}\times\boldsymbol{n} = \boldsymbol{n}^2$ data reads for the matrix
- $\boldsymbol{3n}$ for reading x, y from memory and writing y to memory
- $\boldsymbol{m} = \boldsymbol{n}^2 + \boldsymbol{3n} \approx \boldsymbol{n}^2$ memory operations

Performance of BLAS

- Level 2: matrix-vector operations

$$\boldsymbol{y} := \boldsymbol{A}\boldsymbol{x} + \boldsymbol{y}$$

where $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{\boldsymbol{n}}$

- $\boldsymbol{n}\times\boldsymbol{n} = \boldsymbol{n}^2$ data reads for the matrix
- $\boldsymbol{3n}$ for reading x, y from memory and writing y to memory
- $\boldsymbol{m} = \boldsymbol{n}^2 + \boldsymbol{3n} \approx \boldsymbol{n}^2$ memory operations
- $\boldsymbol{f} = 2(\boldsymbol{n}\times\boldsymbol{n}) = \boldsymbol{2n}^2$ floating point operations

Performance of BLAS

- Level 2: matrix-vector operations

$$\boldsymbol{y} := \boldsymbol{A}\boldsymbol{x} + \boldsymbol{y}$$

where $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{\boldsymbol{n}}$

- $\boldsymbol{n}\times\boldsymbol{n} = \boldsymbol{n}^2$ data reads for the matrix
- $\boldsymbol{3n}$ for reading x, y from memory and writing y to memory
- $\boldsymbol{m} = \boldsymbol{n}^2 + \boldsymbol{3n} \approx \boldsymbol{n}^2$ memory operations
- $\boldsymbol{f} = 2(\boldsymbol{n}\times\boldsymbol{n}) = 2\boldsymbol{n}^2$ floating point operations
- $\boldsymbol{q} \approx 2\boldsymbol{n}^2/\boldsymbol{n}^2 \approx 2$

Performance of BLAS

- Level 2: matrix-vector operations

$$\boldsymbol{y} := \boldsymbol{A}\boldsymbol{x} + \boldsymbol{y}$$

where $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{\boldsymbol{n}}$

- $\boldsymbol{n}\times\boldsymbol{n} = \boldsymbol{n}^2$ data reads for the matrix
- $\boldsymbol{3n}$ for reading x, y from memory and writing y to memory
- $\boldsymbol{m} = \boldsymbol{n}^2 + \boldsymbol{3n} \approx \boldsymbol{n}^2$ memory operations
- $\boldsymbol{f} = 2(\boldsymbol{n}\times\boldsymbol{n}) = \boldsymbol{2n}^2$ floating point operations
- $\boldsymbol{q} \approx \boldsymbol{2n}^2/\boldsymbol{n}^2 \approx \boldsymbol{2}$
- Level 2 operations have slightly better q value
  - slightly more efficient than Level 1

Performance of BLAS

- Level 3: matrix-matrix operations

$$\boldsymbol{C} := \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C}$$

where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{\boldsymbol{n} \times \boldsymbol{n}}$

- $\boldsymbol{n}^2$ reads for A
- $\boldsymbol{n}^2$ reads for B
- $\boldsymbol{2n}^2$ memory operations for C (read and write)
- $\boldsymbol{m} = \boldsymbol{4n}^2$ memory operations

Performance of BLAS

- Level 3: matrix-matrix operations

$$\boldsymbol{C} := \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C}$$
where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{\boldsymbol{n} \times \boldsymbol{n}}$

- $\boldsymbol{n}^2$ reads for A
- $\boldsymbol{n}^2$ reads for B
- $\mathbf{2}\boldsymbol{n}^2$ memory operations for C (read and write)
- $\boldsymbol{m} = \mathbf{4}\boldsymbol{n}^2$ memory operations
- $\boldsymbol{f} = \mathbf{2}\boldsymbol{n}^3$ floating point operations (recall previous slides)

Performance of BLAS

- Level 3: matrix-matrix operations

$$C := A \cdot B + C$$

where $A, B, C \in \mathbb{R}^{n \times n}$

- n^2 reads for A
- n^2 reads for B
- $2n^2$ memory operations for C (read and write)
- $m = 4n^2$ memory operations
- $f = 2n^3$ floating point operations (recall previous slides)
- $q = \frac{2n^3}{4n^2} = \frac{n}{2}$
- Level 3 is most efficient

Performance of BLAS

- Level 3: matrix-matrix operations

$$\boldsymbol{C} \coloneqq \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C}$$

where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{\boldsymbol{n} \times \boldsymbol{n}}$

- $\boldsymbol{n}^2$ reads for A
- $\boldsymbol{n}^2$ reads for B
- $\boldsymbol{2n}^2$ memory operations for C (read and write)
- $\boldsymbol{m} = \boldsymbol{4n}^2$ memory operations
- $\boldsymbol{f} = \boldsymbol{2n}^3$ floating point operations (recall previous slides)
- $\boldsymbol{q} = \frac{\boldsymbol{2n}^3}{\boldsymbol{4n}^2} = \frac{\boldsymbol{n}}{\boldsymbol{2}}$
- (we can further optimize MMM, as discussed previously)

Performance of BLAS

Operation	Definition	f	m	$q = f/m$
saxpy (BLAS1)	$y = \alpha \cdot x + y$ or $y_i = \alpha x_i + y_i$, $i = 1, \ldots, n$	$2n$	$3n + 1$	$2/3$
Matrix-vector mult (BLAS2)	$y = A \cdot x + y$ or $y_i = \sum_{j=1}^{n} a_{ij} x_j + y_i$, $i = 1, \ldots, n$	$2n^2$	$n^2 + 3n$	2
Matrix-matrix mult (BLAS3)	$C = A \cdot B + C$ or $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{jk} + c_{ij}$, $i, j = 1, \ldots, n$	$2n^3$	$4n^2$	$n/2$

Table taken from James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

- BLAS level 3 is most efficient

Performance of BLAS

Operation	Definition	f	m	$q = f/m$
saxpy (BLAS1)	$y = \alpha \cdot x + y$ or $y_i = \alpha x_i + y_i$ $i = 1, \dots, n$	$2n$	$3n+1$	$2/3$
Matrix-vector mult (BLAS2)	$y = A \cdot x + y$ or $y_i = \sum_{j=1}^{n} a_{ij} x_j + y_i$ $i = 1, \dots, n$	$2n^2$	$n^2 + 3n$	2
Matrix-matrix mult (BLAS3)	$C = A \cdot B + C$ or $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{jk} + c_{ij}$ $i, j = 1, \dots, n$	$2n^3$	$4n^2$	$n/2$

Table taken from James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

- BLAS level 3 is most efficient
  - If we have an optimized MMM subroutine, we can improve the performance of our computations by reordering our algorithm in terms of MMM versus `saxpy` or matrix-vector mult

Optimized subroutines vary from machine to machine

- Architecture affects what is algorithm will achieve better memory behavior
- Parameters to consider: blocking factors, loop unrolling depths, software pipelining strategies, loop ordering, register allocations, instruction scheduling
- Example:
  - Cache size and how many levels of cache impact the ideal matrix block sizes and shapes to use
  - Instructions are also cached -- we cannot unroll all the loops if cache size is too limited

Automatically Tuned Linear Algebra Software

Automatic generation of highly efficient Level 3 BLAS

- Code generator to automatically create optimized on-chip, cache contained, (i.e., in Level 1 (L1) cache) matrix multiply
  - Timings determine the correct blocking and loop unrolling factors for on-chip matrix multiply
- Isolate the machine-specific features of the operation to several routines that deal with on-chip matrix multiply
- The rest of the code is fixed across architectures
  - Handles looping, blocking, etc. to build complete matrix-matrix multiply from the on-chip multiply

Automatically generated on-chip matrix multiply

- $\boldsymbol{C} \leftarrow \boldsymbol{A}^{\boldsymbol{T}}\boldsymbol{B} + \boldsymbol{C}$
  - Chosen as opposed to $C \leftarrow AB + C$
  - generates largest $(flops)/(cache\ misses)$ ratio when the loops are written with no unrolling
- Matrix A brought into cache, loops over columns of B (arbitrary choice of which matrix to bring in and loop over the other)

Automatically generated on-chip matrix multiply

- Factors considered for maximal cache reuse:
  - All of A must fit into cache, and at least two columns of B and 1 cache line of C

Automatically generated on-chip matrix multiply

- Factors considered for maximal cache reuse:
  - Instruction cache overflow – Not all of the loops can be unrolled; on-chip multiply instructions must fit L1 cache

Automatically generated on-chip matrix multiply

- Factors considered for maximal cache reuse:
  - Floating point instruction ordering
    - Most modern computers have pipelined floating point units
    - Results of an operation may not be available until X cycles later, where X is number of stages in floating point pipe
    - “Latency hiding” – separate multiply and add; issue unrelated instructions between them

Automatically generated on-chip matrix multiply

- Factors considered for maximal cache reuse:
  - Loop overhead
    - Remove loop overhead by loop unrolling
    - If order of instructions must not change, unroll the loop over the dimension common to A and B (i.e. unroll the “k” loop)
    - Unrolling over other dimensions changes order of instructions and memory access patterns

Automatically generated on-chip matrix multiply

- Factors considered for maximal cache reuse:
  - Exposure of possible parallelism
    - Many modern architectures have multiple floating point units
    - For perfect parallel speedup: memory fetch should also be able to operate in parallel (hardware limitation)
    - Compiler must recognize opportunities for parallelization
      - Unroll “i" and/or “j” loops; choose correct register allocations to avoid false dependencies

Automatically generated on-chip matrix multiply

- Factors considered for maximal cache reuse:
  - The number of outstanding cache misses the hardware can handle before execution is blocked
    - maximal number of cache misses should be issued each cycle, until all memory is in cache or used
    - Use "i" and "j" loop unrolling to control cache-hit ratio

How does ATLAS automatically generate the code?

- Code generator coupled with a timer routine to take initial information
- Tries different strategies for loop unrolling and latency hiding
- Chooses the case which demonstrated the best performance
- User may enter size of L1 cache, or program tries to calculate it

Performance of ATLAS

- ”Has been able to match or exceed the performance of the vendor supplied version of matrix multiply in almost every case”
- ATLAS is used by:
  - MATLAB (v6.0 or higher)
  - Octave

Which BLAS are used by NumPy Python module?

Check out the output of
numpy.show_config()

Summary

- BLAS have been defined for commonly used linear algebra operations
- Vendors implement optimized BLAS specific to their machine architecture
- ATLAS automatically tunes the on-chip matrix multiply
- Reordering of your program to use the BLAS, especially BLAS Level 3 (MMM), optimizes the performance of your code
- Try it yourself!
  - Compare the performance of an algorithm that uses self-written MMM function versus one that uses what numpy offers