High Performance Linear Algebra

Lecture 4: Starting with BLAS, BLAS Level 1: AXPY

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

November 19, 2025 — 14.00:16.00

Dipartimento di Matematica
Università di Pisa

Table of Contents

Motivation: Cholesky factorization example

Symmetric Matrix

A matrix $A \in \mathbb{R}^{n \times n}$ is called symmetric if $A = A^\top$, meaning that it is equal to its transpose.

Eigenvalue and Eigenvector

Given a square matrix $A \in \mathbb{R}^{n \times n}$, a non-zero vector $\mathbf{v} \in \mathbb{R}^n$ is called an eigenvector of A if there exists a scalar $\lambda \in \mathbb{R}$ such that:

$$A\mathbf{v} = \lambda \mathbf{v}$$

The scalar λ is referred to as the eigenvalue corresponding to the eigenvector $\mathbf{v}$. All eigenvalues of a symmetric matrix are real.

Motivation: Cholesky factorization example

Positive Definite Matrix

A symmetric matrix $A \in \mathbb{R}^{n\times n}$ is called positive definite if for all non-zero vectors $\mathbf{x} \in \mathbb{R}^n$:

$$\mathbf{x}^\top A\mathbf{x} > 0$$

This implies that all eigenvalues of A are positive.

Examples of symmetric positive definite matrices

- Covariance/correlation matrices in statistics and machine learning.
- Normal equations: $A^\top A$ from least squares; SPD if A has full column rank.
- Gram/kernel matrices: $K_{ij} = k(x_i, x_j)$ with strictly PD kernels (e.g., Gaussian/RBF).
- Precision (inverse covariance) matrices in Gaussian Markov random fields.

Motivation: Cholesky factorization example

- The Cholesky factorization is a method for decomposing a positive definite matrix A into the product of an upper triangular matrix U and its transpose:

$$A = U^\top U$$

- It is useful for solving systems of linear equations, and inverting matrices.
- It is computationally efficient, requiring approximately $\frac{1}{3}n^3$ operations for an $n \times n$ matrix.

Theorem (Existence and uniqueness)

Every symmetric positive definite matrix A has a unique Cholesky factorization $A = U^\top U$, where U is an upper triangular matrix with positive diagonal entries.

Motivation: Cholesky factorization example

Consider the Cholesky factorization $A = U^\top U$:

Algorithm

for $j = 1$ to n **do**
  for $i = 1$ to $j - 1$ **do**
    $u_{ij} \leftarrow \frac{1}{u_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj} \right)$
  end for
  $u_{jj} \leftarrow \sqrt{a_{jj} - \sum_{k=1}^{j-1} u_{kj}^2}$
end for

- Easy to translate to any language
- But…“reinventing the wheel”
- Similar patterns appear repeatedly
- Lots of code duplication

The key observation

Similar code patterns resurface over and over again
in linear algebra algorithms

Natural strategy

"Define a set of operators such that any algorithm can be expressed as their application to the data at hand."

- Some languages provide native operators (MATLAB, Fortran, Julia)
- Algorithms = sequences of primitive operator calls

Benefits of standardized building blocks

1. **Code reuse**
   - Write once, use many times
   - Amortize cost of high-quality implementation
2. **Standardized interfaces**
   - Explore alternative implementations
   - Preserve overall code behavior
3. **Architecture-aware optimizations**
   - Exploit cache hierarchies
   - Use block/submatrix operations (not just vectors)
4. **Portability across systems**
   - Same interface, optimized per platform

Scope of application

- Cholesky is just one example
- Same reasoning applies to:
  - Dense linear algebra (LU, QR, eigensolvers, ...)
  - Sparse linear algebra (SpMV, iterative solvers, ...)
  - Many other numerical algorithms
- Encapsulation enables:
  - Performance tuning without changing user code
  - Leveraging hardware accelerators (GPUs, vector units)
  - Evolution of implementations over time

This is the foundation of BLAS and LAPACK

Table of Contents

The Basic Linear Algebra Subprograms (BLAS)

- Set of low-level routines for common linear algebra operations
- Designed to be efficient and portable
- Building block for higher-level libraries (LAPACK, ScaLAPACK, PSBLAS, PETSc)
- Available in many programming languages (C, Fortran, Python)

Focus of this section

Dense BLAS: routines for dense matrices and vectors

BLAS organization: three levels

Level 1: Vector operations

- Examples: dot product, vector addition, scaling
- Complexity: $\mathcal{O}(n)$
- Memory-bound

Level 2: Matrix-vector operations

- Examples: matrix-vector multiplication, rank-1 updates
- Complexity: $\mathcal{O}(n^2)$
- Memory-bound

Level 3: Matrix-matrix operations

- Examples: matrix-matrix multiplication (GEMM)
- Complexity: $\mathcal{O}(n^3)$
- Compute-bound (high data reuse)

Popular BLAS implementations

OpenBLAS: Open-source implementation of BLAS and LAPACK

ATLAS: Automatically Tuned Linear Algebra Software; open-source, self-optimizing

Intel MKL: High-performance library optimized for Intel processors

cuBLAS: GPU-accelerated BLAS for NVIDIA GPUs

BLIS: Portable, high-performance, modern BLAS framework

Key takeaway

Same interface, different implementations $\Rightarrow$ performance portability

Finding BLAS with CMake

- CMake provides a built-in module to find BLAS libraries
- Use `find_package(BLAS REQUIRED)` to locate BLAS
- Link against the found BLAS library using `target_link_libraries(<target> PRIVATE ${BLAS_LIBRARIES})`
- Information are available on the webpage: FindBLAS module documentation.

Example CMake snippet

```
find_package(BLAS REQUIRED)
target_link_libraries(<target> PRIVATE ${BLAS_LIBRARIES})
```

Table of Contents

Level 1 BLAS: Overview

3 Level 1 BLAS: Vector operations

types	name	(size arguments)	description	equation	flops	data
s, d, c, z	axpy	(n, alpha, x, incx, y, incy)	update vector	$y = y + \alpha x$	$2n$	$2n$
s, d, c, z, cs, zd	scal	(n, alpha, x, incx)	scale vector	$y = \alpha y$	n	n
s, d, c, z	copy	(n, x, incx, y, incy)	copy vector	$y = x$	0	$2n$
s, d, c, z	swap	(n, x, incx, y, incy)	swap vectors	$x \leftrightarrow y$	0	$2n$
s, d	dot	(n, x, incx, y, incy)	dot product	$= x^T y$	$2n$	$2n$
c, z	dotu	(n, x, incx, y, incy)	(complex)	$= x^T y$	$2n$	$2n$
c, z	dotc	(n, x, incx, y, incy)	(complex conj)	$= x^H y$	$2n$	$2n$
sds, ds	dot	(n, x, incx, y, incy)	(internally double precision)	$= x^T y$	$2n$	$2n$
s, d, sc, dz	nrm2	(n, x, incx)	2-norm	$= \|x\|_2$	$2n$	n
s, d, sc, dz	asum	(n, x, incx)	1-norm	$= \|\mathrm{Re}(x)\|_1 + \|\mathrm{Im}(x)\|_1$	n	n
s, d, c, z	i_amax	(n, x, incx)	∞-norm	$= \mathrm{argmax}_i(\lvert\mathrm{Re}(x_i)\rvert + \lvert\mathrm{Im}(x_i)\rvert)$	n	n
s, d, c, z	rotg	(a, b, c, s)	generate plane (Given's) rotation (c real, s complex)		$O(1)$	$O(1)$
s, d, c, z †	rot	(n, x, incx, y, incy, c, s)	apply plane rotation (c real, s complex)		$6n$	$2n$
cs, zd	rot	(n, x, incx, y, incy, c, s)	apply plane rotation (c & s real)		$6n$	$2n$
s, d	rotmg	(d1, d2, a, b, param)	generate modified plane rotation		$O(1)$	$O(1)$
s, d	rotm	(n, x, incx, y, incy, param)	apply modified plane rotation		$6n$	$2n$

Level 1 BLAS: Vector operations

- Basic operations on vectors
- Examples:
  - Dot product: `DOT`
  - Vector addition: `AXPY`
  - Scaling: `SCAL`
  - Copy: `COPY`
  - Norms: `NRM2`
- **Memory-bound** operations

Data types:

- s: single real
- d: double real
- c: single complex
- z: double complex

Naming convention: `<data type><operation>`

AXPY: Definition

AXPY (Add X times Y):

$$y \leftarrow \alpha x + y$$

- α scalar, x, y vectors
- Level 1 BLAS (memory-bound)
- The output vector y is overwritten
- Widely used in numerical algorithms (e.g., iterative methods)

Routine name: `daxpy` (double precision)

```
call daxpy(n, alpha, x, incx, y, incy)
```

- n: vector length
- alpha: scalar
- x, y: vectors
- incx, incy: strides (usually 1)

Fortran example (double precision)


```
program axpy_example
    use iso_fortran_env, only: int64, real64, output_unit
    implicit none
    integer(kind=int64), parameter :: n = 5
    real(kind=real64) :: x(n), y(n), alpha
    integer(kind=int64) :: i
    ! Initialize the vectors and scalar
    x = [1.0, 2.0, 3.0, 4.0, 5.0]
    y = [10.0, 20.0, 30.0, 40.0, 50.0]
    alpha = 2.0
    ! Call the AXPY routine
    call daxpy(n, alpha, x, 1, y, 1)
    ! Print the result
    write(output_unit, '("Resulting vector y:")')
    do i = 1, n
        write(output_unit, '(F6.2)', advance='no') y(i)
    end do
    write(output_unit, '("")')
    return
end program axpy_example
```

Compiling (OpenBLAS)

Install (Ubuntu): `apt-get install libopenblas-dev`

```
gfortran -o axpy_example axpy_example.f90 -lopenblas
./axpy_example
```

Sample output

```
Resulting vector y:
 12.00 24.00 36.00 48.00 60.00
```

Compiling (OpenBLAS)

Install (Ubuntu): `apt-get install libopenblas-dev`

```
gfortran -o axpy_example axpy_example.f90 -lopenblas
./axpy_example
```

Sample output

```
Resulting vector y:
 12.00 24.00 36.00 48.00 60.00
```

There are quite a few inconvenient things!

An object oriented packaging

Let us make an **object oriented packaging** of this BLAS operations using modern Fortran.

- Create a Git repository for our package

```
mkdir objblas
cd objblas
git init
git branch -m main
```

An object oriented packaging

Let us make an **object oriented packaging** of this BLAS operations using modern Fortran.

- Create a Git repository for our package

```
mkdir objblas
cd objblas
git init
git branch -m main
```

- Create a `CMakeLists.txt` file for our project with content

```
cmake_minimum_required(3.28)
project(objblas LANGUAGES Fortran)
find_package(BLAS REQUIRED)
```

An object oriented packaging

Let us make an **object oriented packaging** of this BLAS operations using modern Fortran.

- Create a Git repository for our package

```
mkdir objblas
cd objblas
git init
git branch -m main
```

- Create a `CMakeLists.txt` file for our project with content

```
cmake_minimum_required(3.28)
project(objblas LANGUAGES Fortran)
find_package(BLAS REQUIRED)
```

- Create a directory which will contain the code:

```
mkdir src
touch src/blas.f90
```

An object oriented packaging

We will now create a Fortran module, which will package the BLAS library we are going to use, hence we write into the `src/blas.f90` file the following:

```
module blas
    use iso_fortran_env, only: real64, real32
    implicit none

     < interfaces >

contains

     < implementations >

end module blas
```

Let us start with the implementations


```fortran
subroutine daxpy_blas(alpha, x, y, incx, incy)
    use iso_fortran_env, only: real64
    implicit none
    real(real64), intent(in) :: alpha
    real(real64), intent(in) :: x(:)
    real(real64), intent(inout) :: y(:)
    integer, intent(in), optional :: incx, incy
    ! Local variables
    integer :: incx_, incy_
    incx_ = 1
    incy_ = 1
    if (present(incx)) incx_ = incx
    if (present(incy)) incy_ = incy
    call daxpy(size(x),alpha,x,incx_,y,incy_)
end subroutine daxpy_blas
```

- `intent()` tells the subroutine if the argumenti is an input, an output, or both,
- `optional` tells if the argument can be omitted, and `present` checks if it has been passed or not.
- We use `size(x)` to get the length of the vector.
- We can write similar subroutines for the other data types (single, complex, double complex).

Now the interfaces


```fortran
private

interface axpy
    module procedure daxpy_blas
    ! Other data types procedures
end interface axpy

public :: axpy
```

- `private` makes all the module contents private by default,
- `public :: axpy` makes the `axpy` interface public.

- The `interface` block allows to define multiple procedures with the same name but different argument types.
- Here we define the interface for `daxpy`, which maps to the implementation `daxpy_blas`.
- We can add other procedures for different data types (single, complex, double complex).

CMake configuration

We need to tell CMake how to build our package, so we add the following lines to the `CMakeLists.txt` file:

```
add_library(objblas src/blas.f90)
target_link_libraries(objblas PUBLIC BLAS::BLAS)
```

- `add_library()` creates a library target named `objblas` from the source file.
- `target_link_libraries()` links the BLAS libraries to our package.

CMake configuration

We need to tell CMake how to build our package, so we add the following lines to the `CMakeLists.txt` file:

```
add_library(objblas src/blas.f90)
target_link_libraries(objblas PUBLIC BLAS::BLAS)
```

- `add_library()` creates a library target named `objblas` from the source file.
- `target_link_libraries()` links the BLAS libraries to our package.

Now we need to write a tester: we create a `test` directory and inside it a `CMakeLists.txt` file and a `axpy_test.f90` file and add the following lines to the `test/CMakeLists.txt` file:

```
add_executable(test_axpy test/test_axpy.f90)
target_link_libraries(test_axpy PRIVATE objblas)
```

Test program

```fortran
program test_axpy
    use iso_fortran_env, only: real64,
    ↪  output_unit
    use blas
    implicit none
    integer, parameter :: n = 10
    real(real64) :: x64(n), y64(n),
    ↪  alpha64
    x64 = [1,2,3,4,5,6,7,8,9,10]
    y64 = 0.0_real64
    alpha64 = 2.0_real64
    call axpy(alpha64, x64, y64)
    write(output_unit,*) "Double Precision
    ↪  AXPY Result:"
    write(output_unit,*) y64
end program test_axpy
```

- We test the double precision AXPY operations through the `axpy` interface.
- We initialize vectors and scalars, call the `axpy` method from our package, and print the results.
- ✓ This modular approach makes it easy to extend and maintain the BLAS wrapper.

Test program


```fortran
program test_axpy
    use iso_fortran_env, only: real64,
    ↪  output_unit
    use blas
    implicit none
    integer, parameter :: n = 10
    real(real64) :: x64(n), y64(n),
    ↪  alpha64
    x64 = [1,2,3,4,5,6,7,8,9,10]
    y64 = 0.0_real64
    alpha64 = 2.0_real64
    call axpy(alpha64, x64, y64)
    write(output_unit,*) "Double Precision
    ↪  AXPY Result:"
    write(output_unit,*) y64
end program test_axpy
```

- We test the double precision AXPY operations through the `axpy` interface.
- We initialize vectors and scalars, call the `axpy` method from our package, and print the results.
- This modular approach makes it easy to extend and maintain the BLAS wrapper.

Exercise

Implement the single precision, complex, and double complex versions of AXPY in the `blas` module and test them in the test program.

AXPY: A Starting Point for Parallelism

- AXPY is a simple routine, ideal for exploring parallelism
- The operation can be parallelized by splitting vectors into chunks
- Each chunk can be computed independently in parallel

OpenMP Parallelization

- OpenMP: API for shared memory parallel programming
- Uses **compiler directives** (special comments)
- Supports C, C++, and Fortran
- Portable and scalable for multi-core processors

OpenMP AXPY Example

3 Level 1 BLAS: Vector operations

```fortran
program axpy_opm_example
    use iso_fortran_env, only: int64, real64, output_unit
    use omp_lib
    implicit none
    integer(kind=int64), parameter :: n = 5
    real(kind=real64) :: x(n), y(n), alpha
    integer(kind=int64) :: i
    ! Initialize the vectors and scalar
    x = [1.0, 2.0, 3.0, 4.0, 5.0]
    y = [10.0, 20.0, 30.0, 40.0, 50.0]
    alpha = 2.0
    ! Write the OpenMP directive to parallelize the for loop
    !$omp parallel do
    do i = 1, n
        y(i) = y(i) + alpha * x(i)
```

OpenMP AXPY Example

```
    end do
    !$omp end parallel do
    ! Print the result
    write(output_unit, '("Resulting vector y:")')
    do i = 1, n
        write(output_unit, '(F6.2)', advance='no') y(i)
    end do
    write(output_unit, '("")')
    ! Return
    return
end program axpy_opm_example
```

- Include OpenMP: `use omp_lib`
- Directive: `!$omp parallel do`
- Compiler spawns threads to distribute loop iterations

Compiling with OpenMP

Compile the OpenMP program:

```
gfortran -o axpy_omp_example axpy_omp_example.f90 -fopenmp
```

Run the program:

```
./axpy_omp_example
```

Controlling Thread Count

Set number of threads via environment variable:

```
export OMP_NUM_THREADS=4
./axpy_omp_example
```

Or in code: `call omp_set_num_threads(4)`

Querying Thread Information

Get the number of threads being used:

```
integer(kind=int64) :: nthreads
!$omp parallel
!$omp single
    nthreads = omp_get_num_threads()
!$omp end single
!$omp end parallel
write(output_unit, '("Number of threads: ", I2)') nthreads
```

- Use `omp_get_num_threads()` to query
- `!$omp single` ensures only one thread updates
- Must be called within a parallel region

Two Key Questions

1. **How are threads scheduled?**
   - How are loop iterations distributed among threads?
2. **Who owns what data?**
   - Which variables are shared vs. private?

OpenMP Scheduling Policies

Specified using `schedule` clause:

- `static`: Equal-sized chunks (default)
- `static, chunk_size`: Fixed chunk size
- `dynamic`: Iterations assigned as threads become available
- `guided`: Dynamic with decreasing chunk sizes
- `runtime`: Determined by `OMP_SCHEDULE` environment variable
- `auto`: Compiler decides

```fortran
!$omp parallel do schedule(static, chunk_size)
do i = 1, n
    y(i) = alpha * x(i) + y(i)
end do
!$omp end parallel do
```

Data Sharing Clauses

Control variable visibility between threads:

`shared`: Single instance visible to all threads

`private`: Each thread has its own uninitialized copy

`firstprivate`: Like private, but initialized from original

`lastprivate`: Private, with final value copied back

Example for AXPY:

```
!$omp parallel do shared(x, y, alpha) private(i) schedule(dynamic)
do i = 1, n
    y(i) = alpha * x(i) + y(i)
end do
!$omp end parallel do
```

Performance Measurement Strategy

- Use `omp_get_wtime()` for accurate timing
- Run multiple iterations for reliable measurements
- Use sufficiently large problem sizes
- Read problem size from command line
- Use allocatable arrays for dynamic sizing

Compilation with optimization

```
gfortran -O3 -march=native -mtune=alderlake -o axpy_omp axpy_omp_time.f90 -fopenmp
```

Timing Code Example


```
do i = 1, 1000
    elapsed_time = 0.0
    t1 = omp_get_wtime() ! Start timer
    !$omp parallel do shared(x, y, alpha) private(j) schedule(static)
    do j = 1, n
        y(j) = alpha * x(j) + y(j)
    end do
    !$omp end parallel do
    t2 = omp_get_wtime() ! Stop timer
    elapsed_time = elapsed_time + (t2 - t1)
end do
```

- Average over many iterations
- Use `omp_get_wtime()` instead of `cpu_time`
- Measure wall-clock time

Performance Analysis: roofline model

Let us analyze the performance of our OpenMP AXPY implementation using the roofline model.

First the characteristics of the AXPY operation:

- AXPY operation: $y \leftarrow \alpha x + y$
- Floating-point operations (FLOPs): $2n$ (1 multiplication + 1 addition per element)
- Data movement: $3n$ (read x, read y, write y)
- Operational intensity: $\frac{2n}{3\times 8n} = \frac{2}{24} = \frac{1}{12}$ FLOPs/byte

To plot the roofline model, we need to measure/know:

- Peak computational performance (FLOPs/s)
- Memory bandwidth (bytes/s)

On my CPU (Intel® Core™ i9-14900HX) I have:

- Peak performance: 844.8 GFLOPs (double precision)
- Memory bandwidth: 89.6 GB/s (measured with *stream*)

The **operational intensity** of AXPY is $1/12$ FLOPs/byte, which is independent of n, this is a clear indication of a **memory-bound** operation.

On my CPU (Intel® Core™ i9-14900HX) I have:

- Peak performance: 844.8 GFLOPs (double precision)
- Memory bandwidth: 89.6 GB/s (measured with *stream*)

The **operational intensity** of AXPY is $^1/_{12}$ FLOPs/byte, which is independent of n, this is a clear indication of a **memory-bound** operation.

The memory-bound performance ceiling is given by:

$$\begin{aligned}\text{Performance}_{\text{max, memory-bound}} &= \text{Operational Intensity} \times \text{Memory Bandwidth}\\ &= \frac{1}{12} \times 89.6\ \text{GFLOPs} \approx 7.47\ \text{GFLOPs}\end{aligned}$$

Measuring Performance

After running the different AXPY implementation with a large vector size (e.g., $n = 10^8$) and measuring the execution time, we can compute the achieved performance:

Measuring Performance

After running the different AXPY implementation with a large vector size (e.g., $n = 10^8$) and measuring the execution time, we can compute the achieved performance:

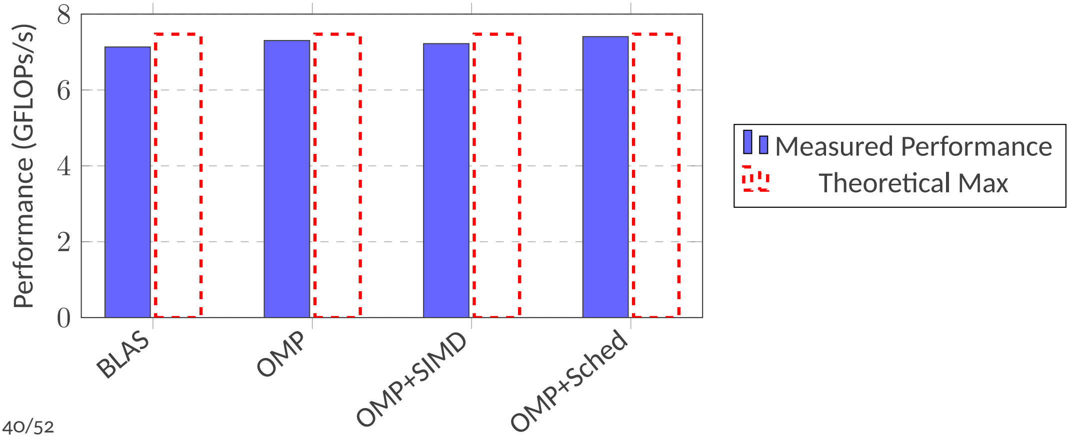
Assuming we measured an execution time of t seconds, the achieved performance is:

$$\text{Achieved Performance} = \frac{2n}{t}\ \text{FLOPs/s}$$

```fortran
do rep = 1, reps
      y = 0.0_dp ! Reset y for each repetition
      t0 = omp_get_wtime()
      call daxpy(n, alpha, x, 1, y, 1)
      t1 = omp_get_wtime() ! BLAS implementation
      time_blas = time_blas + (t1 - t0)
end do
```

Visualizing results

I tested it my machine with $n = 100000$ over 50 repetitions and obtained the following:



Some caveats

To obtain reasonable numbers from our implementations we **need to enable compiler optimizations**:

```
gfortran -O3 -march=native -mtune=native
```

- `-O3` enables high-level optimizations
- `-march=native` enables instructions for the host CPU
- `-mtune=native` optimizes for the host CPU microarchitecture

If we want to enable them in our **CMake project** we need to add the following lines to the `CMakeLists.txt` file:

```
set(CMAKE_Fortran_FLAGS_RELEASE "-O3 -march=native -mtune=native")
set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
```

Full example code

The **full example code** for the OpenMP AXPY implementation with performance measurement is available at:

github.com/High-Performance-Linear-Algebra/objblas/tree/main

As usual, it can be obtained by doing

```
git clone git@github.com:High-Performance-Linear-Algebra/objblas.git
cd objblas
```

and the example run by doing

```
mkdir build
cd build
cmake ..
make # or ninja
./axpy_perf
```

Varying the number of threads

We can analyze the performance of our OpenMP AXPY implementation by varying the number of threads used.

- Set the number of threads using the `OMP_NUM_THREADS` environment variable
- Measure execution time and compute achieved performance for each thread count
- Plot performance vs. number of threads to visualize scaling behavior

Example command to run with different thread counts

```
export OMP_NUM_THREADS=4
./axpy_perf
```

Implementation of the AXPY scaling driver

Since we want to vary the number of threads, and we want to visualize the performance, we can implement a simple bash script which will do the job for us.

```bash
#!/bin/bash
BUILD_DIR=../../build
THREADS=(1 2 4 8 16 32)
SIZES=(1000000 5000000 10000000 50000000 100000000)
repetitions=50

for size in "${SIZES[@]}"; do
    for threads in "${THREADS[@]}"; do
        export OMP_NUM_THREADS=$threads
        echo "Running axpy_scaling with size=$size and threads=$threads"
        $BUILD_DIR/axpy_scaling $size $repetitions
    done
done
```

Implementation of the AXPY scaling driver

The `axpy_scaling` program needs to read the size and number of repetitions from the command line, so we can implement it as follows:

```
integer :: reps, n
character(len=20) :: n_str, reps_str
if (command_argument_count() < 2) then
    write(error_unit, *) 'Usage: axpy_scaling <problem_size> <repetitions>'
    stop
end if
call get_command_argument(1, n_str)
call get_command_argument(2, reps_str)
read(n_str, *) n
read(reps_str, *) reps
```

Implementation of the AXPY scaling driver

We allocate the arrays dynamically:

```fortran
real(dp), allocatable :: x(:), y(:)
real(dp) :: alpha
integer :: stat
! Allocate and initialize data
allocate(x(n), y(n),stat=stat)
if (stat /= 0) then
    write(error_unit, *) 'Error allocating arrays of size ', n
    stop
end if

x = [(real(i, dp), i = 1, n)]
y = 0.0_dp
alpha = 2.0_dp
```

Implementation of the AXPY scaling driver

Finally we can implement the timing loop as follows:

```fortran
! Benchmark BLAS AXPY
time_blas = 0.0_dp
do rep = 1, reps
    y = 0.0_dp
    t0 = omp_get_wtime()
    call daxpy(n, alpha, x, 1, y, 1)
    t1 = omp_get_wtime()
    time_blas = time_blas + (t1 - t0)
end do
```

```fortran
! Benchmark OpenMP AXPY
time_omp = 0.0_dp
do rep = 1, reps
    y = 0.0_dp
    t0 = omp_get_wtime()
    call axpy_omp(n, alpha, x, 1, y, 1)
    t1 = omp_get_wtime()
    time_omp = time_omp + (t1 - t0)
end do
```

We then compute the averages, and print the results to screen:

```fortran
write(output_unit, *) 'Average time BLAS AXPY: ', time_blas/reps, ' seconds'
write(output_unit, *) 'Average time OpenMP AXPY: ', time_omp/reps, ' seconds'
```

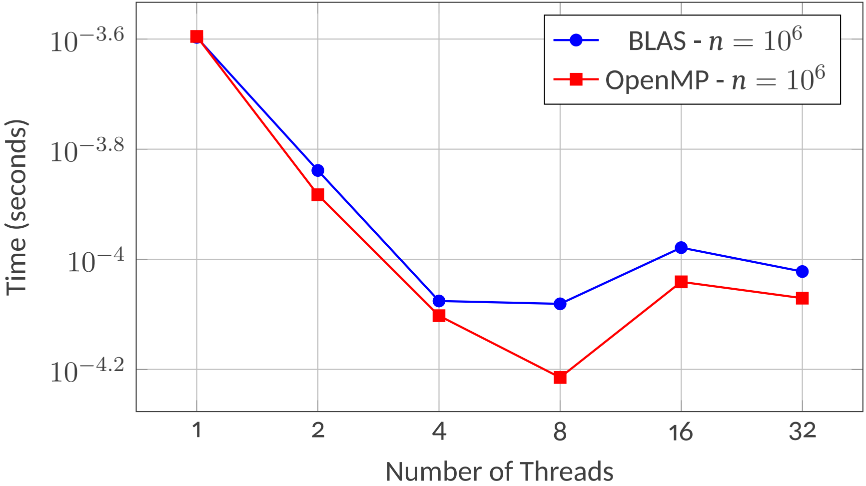
Writing to file for plotting

It is convenient to write the results to a file for later plotting. We can do it as follows:

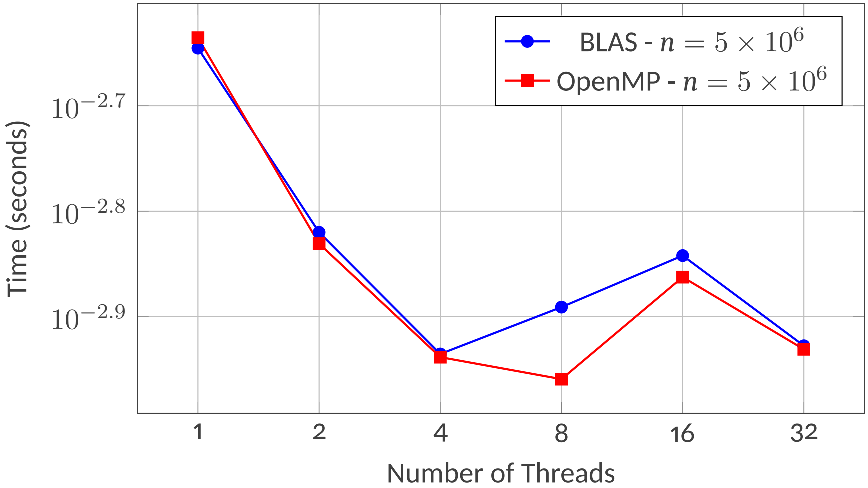
```
open(unit=10, file='axpy_scaling_results.csv', status='unknown',
↪  action='write', position='append')
write(10, '(I10,I10,1X,F15.6,F15.6)') n, nthreads, time_blas, time_omp
close(10)
```

- `open` opens a file for writing
- `status='unknown'` creates the file if it doesn't exist, other possibilities for this argument are `'old'`, `'new'`, and `'replace'`
- `position='append'` adds data to the end of the file, other possibilities are `'rewind'` and `'replace'`, which start writing from the beginning of the file, and overwrite existing content.
- `write` formats and writes the data: problem size, thread count, and timings
- `close` closes the file

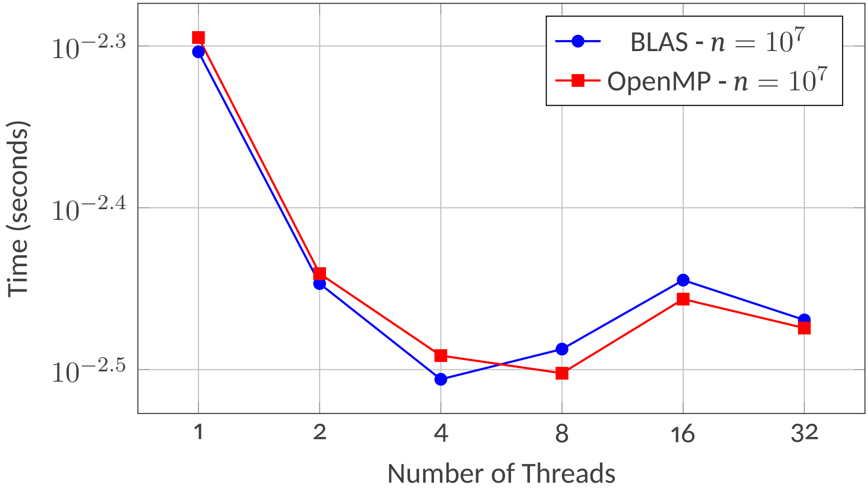
Strong Scaling Results



Strong Scaling Results

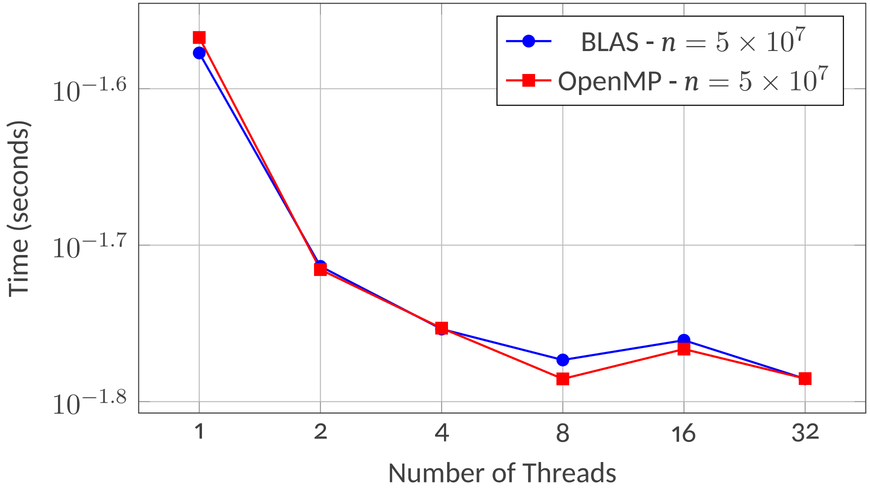


Strong Scaling Results

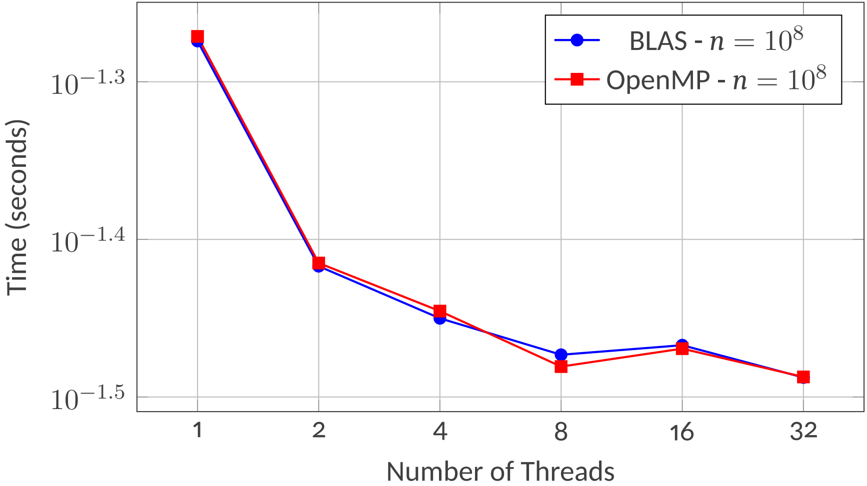


Strong Scaling Results

3 Level 1 BLAS: Vector operations



Strong Scaling Results



Strong Scaling Results: Analysis

- For small problem sizes ($n = 10^6$), execution time increases with more threads
  - Thread creation overhead dominates computation
  - Memory bandwidth not fully utilized

Strong Scaling Results: Analysis

- For small problem sizes ($n = 10^6$), execution time increases with more threads
  - Thread creation overhead dominates computation
  - Memory bandwidth not fully utilized
- For large problem sizes ($n \geq 5 \times 10^7$), execution time decreases with more threads
  - Computation becomes significant enough to benefit from parallelism
  - Better amortization of parallel overhead
  - Improved memory bandwidth utilization across cores

Strong Scaling Results: Analysis

- For small problem sizes ($n = 10^6$), execution time increases with more threads
  - Thread creation overhead dominates computation
  - Memory bandwidth not fully utilized
- For large problem sizes ($n \geq 5 \times 10^7$), execution time decreases with more threads
  - Computation becomes significant enough to benefit from parallelism
  - Better amortization of parallel overhead
  - Improved memory bandwidth utilization across cores
- **Memory-bound nature persists:**
  - Scaling plateaus beyond 8-16 threads
  - Limited by memory bandwidth, not computation
  - Multiple threads saturate available bandwidth

Strong Scaling Results: Analysis

- For small problem sizes ($n = 10^6$), execution time increases with more threads
- For large problem sizes ($n \geq 5 \times 10^7$), execution time decreases with more threads
- **Memory-bound nature persists:**

Rule of thumb

For memory-bound operations like AXPY, parallelism helps only when the problem size is large enough to amortize threading overhead and saturate memory bandwidth.

Exercises

- What does it change if we use **single precision** instead of **double precision**?
- Investigate weak scaling behavior by increasing problem size proportionally with the number of threads.
- Implement **C**ontinuous **I**ntegration (CI) for the repository using GitHub Actions.
- We could test **one BLAS** implementation **against another** (e.g., OpenBLAS vs Intel MKL) and one compiler against another (e.g., GCC vs Intel). How would you implement this? A good idea would be to look at spack.io for package management.

Summary and next-steps

- We have created a Fortran module wrapping BLAS AXPY routines with a clean interface.
- We implemented an OpenMP version of AXPY to explore parallelism.
- We analyzed performance using the roofline model, confirming AXPY is memory-bound.
- We studied strong scaling behavior by varying thread counts and problem sizes.

Next Steps:

- Look at inner products (DOT), norms and their parallel implementations.
- Start exploring Level 2 BLAS routines (matrix-vector operations).
- Look at more OpenMP pragmas and optimizations.