JavaOne™

BLISful Linear Algebra with Project Panama

Paul Sandoz, Oracle

JavaOne™

Overview

BLIS library

Panama and BLIS

2D Matrix API

MSET

BLIS linear algebra library

- High performance CPU-based library for dense linear algebra operations
  - Significant superset of the level 1-3 **B**asic **L**inear **A**lgebra **S**ubprograms (**BLAS**)
  - Especially noted is the level 3 performance e.g. **GE**neric **M**atrix **M**ultiplication (**GEMM**)
  - One of only 2 libraries to offer GEMM-like extensibility
- Developed by The Science of High Performance Computing Group at the University of Texas at Austin

BLIS Object API

- Defines a structure, called `obj_t`, that models a 2D matrix
  - Abstracts many details such as the element type and dimensions
- Defines operations that accept `obj_t*` as arguments
- It’s a well designed C API
  - But we can do even better binding to it in Java and wrapping it

Using the native BLIS Object API

```c
obj_t a, b, c;
bli_obj_create( BLIS_DOUBLE, 4, 5, 0, 0, &c );
bli_obj_create( BLIS_DOUBLE, 4, 3, 0, 0, &a );
bli_obj_create( BLIS_DOUBLE, 3, 5, 0, 0, &b );

obj_t* alpha = &BLIS_ONE;
obj_t* beta  = &BLIS_ONE;

bli_randm( &a );
bli_setm( &BLIS_ONE, &b );
bli_setm( &BLIS_ZERO, &c );

// c := beta * c + alpha * a * b, where 'a', 'b', and 'c' are general.
bli_gemm( alpha, &a, &b, beta, &c );
...
```

Panama

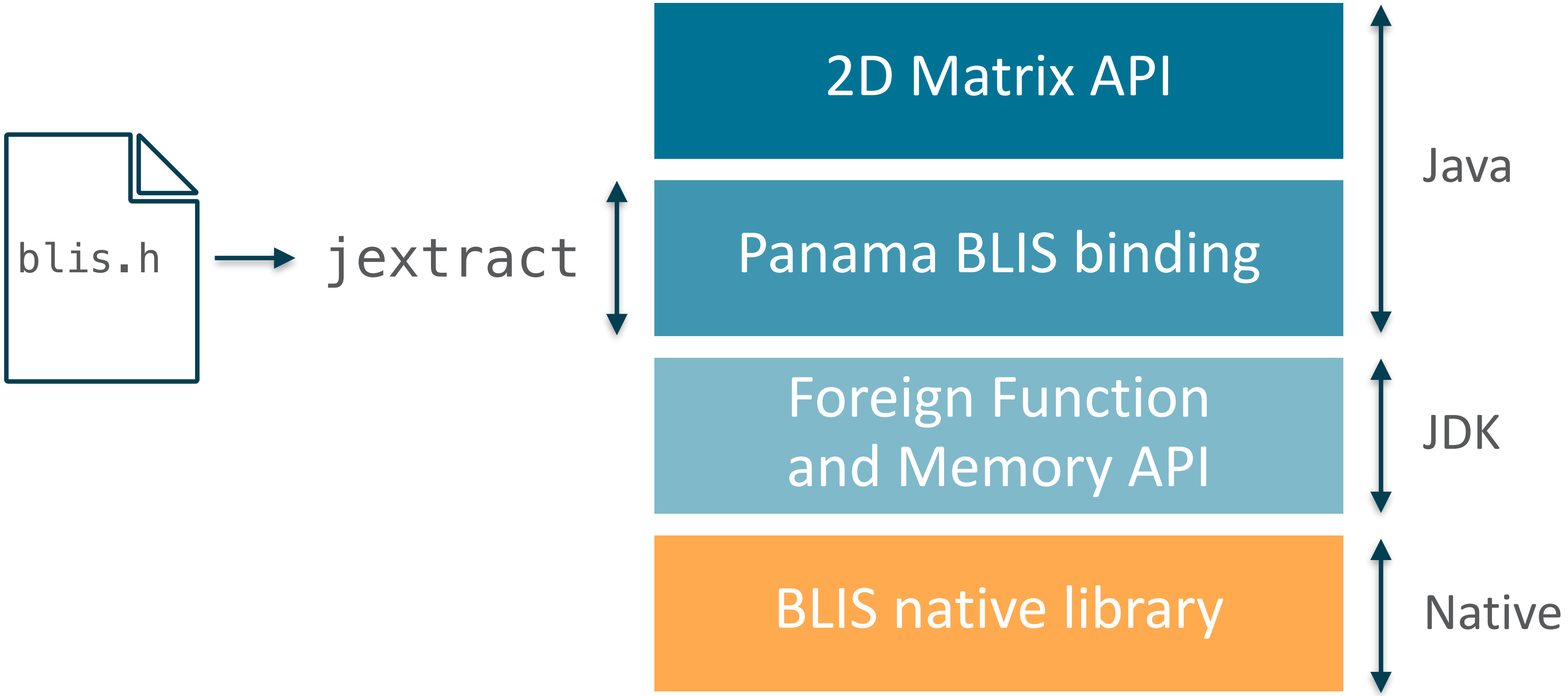
***F**oreign **F**unction & **M**emory (**FFM**) API and tooling*

- An API by which Java programs can interoperate with code and data outside of the Java runtime
  - Available as a preview API in JDK 19
- Enables Java developers to call native libraries and process native data without the brittleness and danger of **J**ava **N**ative **I**nterface (**JNI**)
  - Replaces JNI with a superior, pure-Java development model
- Provides tooling to generate pure-Java bindings to native C libraries
  - Autogenerate Java code from native library C header files

Foreign Memory API

- `MemorySegment`
  - Models a contiguous region of memory
  - Replaces direct `ByteBuffer`, overcoming its size limits and memory management constraints
- `SegmentAllocator`
  - A “malloc”-like abstraction for producing segments
- `MemorySession` (<: `SegmentAllocator`)
  - Manages the deallocation of segments it allocates
  - Controls access to the memory of a segment
    e.g., segment is inaccessible after deallocation

BLIS and Panama
Architectural overview



Using the Java binding to the native BLIS Object API

```
try (MemorySession s = MemorySession.openConfined()) {
    /* obj_t* */ MemorySegment a = obj_t.allocate(s),
    /* obj_t* */ MemorySegment b = obj_t.allocate(s);
    /* obj_t* */ MemorySegment c = obj_t.allocate(s);

    bli_obj_create(BLIS_DOUBLE(), 4, 5, 0, 0, c);
    bli_obj_create(BLIS_DOUBLE(), 4, 3, 0, 0, a);
    bli_obj_create(BLIS_DOUBLE(), 3, 5, 0, 0, b);

    /* obj_t* */ MemorySegment alpha = BLIS_ONE$SEGMENT();
    /* obj_t* */ MemorySegment beta = BLIS_ONE$SEGMENT();

    bli_randm(a);
    bli_setm(BLIS_ONE$SEGMENT(), b);
    bli_setm(BLIS_ZERO$SEGMENT(), c);

    // c := beta * c + alpha * a * b, where 'a', 'b', and 'c' are general.
    bli_gemm(alpha, a, b, beta, c);
    ...
}
```


2D Matrix API

- An idiomatic API for Java developers
  - Hides an API that is idiomatic for C developers
  - Manages the memory of the `obj_t` structure
- Matrix API and BLIS share the matrix structure and buffer of elements
  - No $2^{31} - 1$ size limit as with primitive arrays and `ByteBuffer`
  - Many level-1/2-like BLAS subprograms can be performed using pure Java
  - Level-3 BLAS subprograms can be performed natively using BLIS
- Higher-order operations over the elements using lambda expressions
  - Numpy-like with customized optimization using λ kernels

Using the 2D Matrix API

```java
try (MemorySession s = MemorySession.openConfined()) {
    DoubleMatrix c = Matrix.newDoubleMatrix(s, 4, 5);
    DoubleMatrix a = Matrix.newDoubleMatrix(s, 4, 3);
    DoubleMatrix b = Matrix.newDoubleMatrix(s, 3, 5);

    Matrix<?> alpha = Matrix.one();
    Matrix<?> beta = Matrix.one();

    BLI.randm(a);
    BLI.setm(DoubleMatrix.one(), b);
    // c's elements are already initialized to zero

    // c := beta * c + alpha * a * b, where 'a', 'b', and 'c' are general.
    BLI.gemm(alpha, a, b, beta, c);
}
```

Allocation

```
DoubleMatrix newDoubleMatrix(MemorySession scope, long rows, long cols) {
    MemorySegment buffer = scope.allocate(
        MemoryLayout.sequenceLayout(rows * cols, ValueLayout.JAVA_DOUBLE));
    // Allocate the obj_t struct and attach the buffer
    MemorySegment obj = obj_t.allocate(scope);
    blis_h.bli_obj_create_with_attached_buffer(
            // Element type
            blis_h.BLIS_DOUBLE(),
            // Shape
            rows, columns,
            // Pointer to elements
            buffer,
            // Row and column strides, column-major order
            1, rows,
            obj);
    return new DoubleMatrix(scope, obj, buffer);
}
```

Element-wise operations with lambdas

- Unary, binary, and ternary
  - Lambda expressions for the elemental operations
- Binary operation for matrixes A, B and C of the same dimensions

```
A.elementwise(B, C, (a, b) -> a + b)
```

$$C = A + B$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} a+j & b+k & c+l \\ d+m & e+n & f+o \\ g+p & h+q & i+r \end{bmatrix}$$

- What if B is a singular matrix, row vector, or column vector?
  - We can broadcast B into matrix B' of the same dimensions as A

Element-wise operations with broadcasting

- Broadcast scalar

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j \end{bmatrix} \equiv \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j & j & j \\ j & j & j \\ j & j & j \end{bmatrix}$$

- Broadcast row vector

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j & k & l \end{bmatrix} \equiv \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j & k & l \\ j & k & l \\ j & k & l \end{bmatrix}$$

- Broadcast column vector

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j \\ k \\ l \end{bmatrix} \equiv \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j & j & j \\ k & k & k \\ l & l & l \end{bmatrix}$$

Reduction operations with lambdas

- Reduce all elements
- Reduce all rows to produce a column vector
- Reduce all columns to produce a row vector

```
A.reductionColumn(m, (a, b) -> a + b)
m.elementwise(e -> e / A.rows())
```

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$m = \begin{bmatrix} (a+d+g)/rows & (b+e+h)/rows & (c+f+i)/rows \end{bmatrix}$$

Optimizing operations with λ kernels

- Higher-order operations are very expressive but may not reliably optimize
  - The operation does not know what the lambda expression does, and lambda expression does not know how matrix elements are arranged in memory
  - The compiler might not inline the lambda's body
- A λ kernel implements the operation's functional interface and the operation's λ kernel interface
  - Operates over memory segments, using a custom implementation that can fuse loops with the lambda expression
- Enables operating on elements in parallel
  - For example, on the CPU using thread-level parallelism over groups of columns using Fork/Join API, and data-level parallelism over a column using the Vector API
  - Or perhaps on a GPU?

Optimizing operations with λ kernels

- A λ kernel is passed to an operation in place of it's lambda expression

```
A.elementwise(rv1, rv2, B,
    // (a, v1, v2) -> { ... }
    MyTernaryKernel.INSTANCE)
```

- What if we could dynamically generate a λ kernel from the symbolic description of a lambda expression's body?
  - One potential solution to Fixing The Inlining "Problem"

MSET

- **M**ultivariate **S**tate **E**stimation **T**echnique (**MSET**) is a machine learning algorithm to determine if a system, producing time-series data from sensors, is operating normally or abnormally
  - Anomalies can be detected and resolved before they become critical problems (including sensor malfunction or manipulation rather than component malfunction)
- MSET was originally developed in 1996 by the US Department of Energy's (DoE) Argonne National Labs
  - Designed to monitor nuclear power plants and ensure they are safe and secure
  - Broadly applicable to many other areas, such as airplanes, cars, rollercoasters, datacenter

MSET2

- MSET2 is a proprietary enhancement to MSET
  - Can detect anomalies earlier with higher sensitivity and fewer false alarms than MSET
  - Superior than other machine learning approaches, such as neural nets and support vector machines, and comparatively more efficient

Core of the MSET algorithm

- Consider a system with m sensors and n observations under **normal** operation
- $X_i^T = [x_{i1}, x_{i2}, \ldots, x_{im}]$
  - The i'th normal observation for all sensors at time t_i, where $t_{i+1} > t_i$
- $D = [X_1, X_2, \ldots, X_n]$
  - D is a $m \times n$ matrix
  - Number of rows equals number of sensors
  - Number of columns equals number of observations
- D is commonly referred to as the design matrix

Core of the MSET algorithm

- Given D and current observation(s), X_{obs}, can we determine if the system behaving normally or abnormally?
- Given D and X_{obs}, compute X_{est}
  - The closest normal behavior
- Then, compute residual, $X_{res} = X_{est} - X_{obs}$
  - Make a decision based on difference

Ordinal least squares

- Estimate is a linear combination of weights
  - $X_{est} = D\omega_{est}$
  - $\omega_{est} = (D^T D)^{-1} D^T X_{obs}$
  - $X_{est} = D(D^T D)^{-1} D^T X_{obs}$
- However, systems are typically non-linear
  - Output is not proportional to change in input

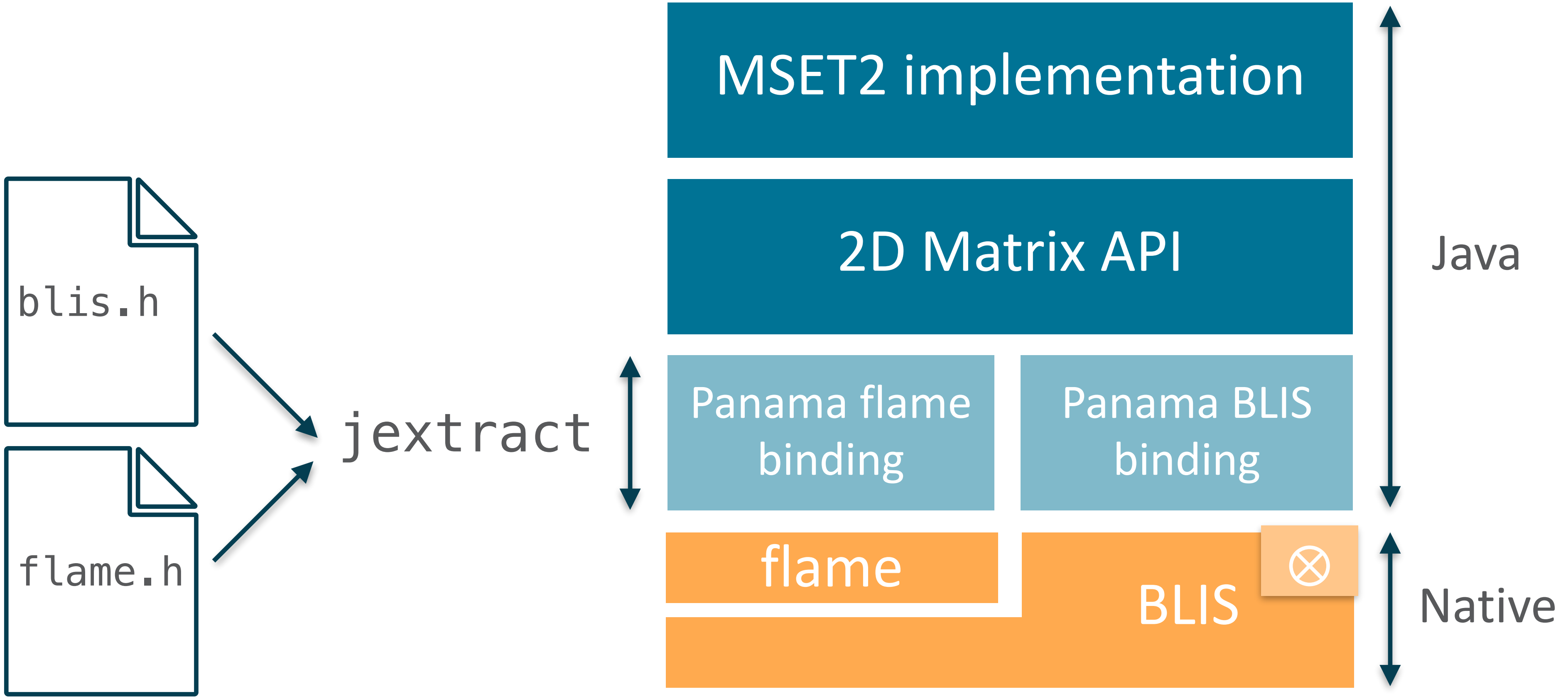
Core of the MSET algorithm

- Use a different level-3 operation, a similarity operation $\otimes$, that performs a non-linear comparison
  - Transforms from the observation space into a feature space, revealing the similarity between observations
- $$\begin{aligned}\omega_{est} &= (D^T \otimes D)^+(D^T \otimes X_{obs}) \\ &= D_{sim}^+(D^T \otimes X_{obs})\end{aligned}$$
  - $D^T \otimes D$, referred to as the similarity matrix D_{sim}, an $n \times n$ matrix
  - Compute pseudo-inverse of D_{sim}, D_{sim}^+

Requirements of MSET algorithm

- We can use the 2D Matrix API to compute ω_{est} but we require some enhancements to our architecture
  - We need the $\otimes$ operation and pseudo-inverse operation
- The $\otimes$ operation is implemented as a BLIS add-on operation
  - We take advantage of BLIS's extensibility and efficient GEMM infrastructure
- The pseudo-inverse operation is provided by the native flame library
  - Flame uses BLIS and provides functionality similar to LAPACK

MSET2 implementation
Architectural overview



In summary

- BLIS, Panama, and the 2D Matrix API with λ lambda kernels, enabled us to rapidly develop an efficient prototype of the MSET2 algorithm in Java
  - The efficiency of BLIS with the productivity of Java
- Leveraging a modern CPU (OCI BM.Standard.E4.128) gives GPU-like speeds and a hundred times the memory at a tenth the cost
  - MSET2 training and validation with 1,000 sensors and 100,000 observations took under 4 seconds
  - MSET2 estimation with 50,000 sensors and a 1,000,000 observations, requiring 4 terabytes of memory, took under 3 hours

JavaOne™

BLIS `obj_t` struct modeling a 2D matrix

```
typedef struct obj_s {
    ...

    dim_t          dim[2];  // Number of rows and columns

    ...

    void*          buffer;  // Pointer to elements

    inc_t          rs;      // Row stride
    inc_t          cs;      // Column stride

    ...
} obj_t;
```

Row-major and column-major order

$$m = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

m is a 2x3 matrix

Row−major order

$$rs = 3$$
$$cs = 1$$
$$buffer \rightarrow \begin{bmatrix} a & b & c & d & e & f \end{bmatrix}$$

Column−major order

$$rs = 1$$
$$cs = 2$$
$$buffer \rightarrow \begin{bmatrix} a & d & b & e & c & f \end{bmatrix}$$

$$index(i, j) = i * rs + j * cs$$

$$buffer[index(1,1)] =$$
$$buffer[1 * 3 + 1 * 1] =$$
$$buffer[4] = e$$

$$buffer[index(1,1)] =$$
$$buffer[1 * 1 + 1 * 2] =$$
$$buffer[3] = e$$