StreamJIT: A Commensal Compiler for High-Performance Stream Programming

Jeffrey Bosboom Sumanaruban Rajadurai
Weng-Fai Wong Saman Amarasinghe

MIT CSAIL
National University of Singapore

October 22, 2014

Modern software is built out of libraries

There's a C, Java and/or Python library for basically every domain.

ImageMagick	image processing	C
LAPACK/BLAS	linear algebra	C
CGAL	computational geometry	C++
EJML	linear algebra	Java
Weka	data mining	Java
Pillow	image processing	Python
NLTK	natural language processing	Python

If a library doesn't exist for our domain, we build one, then build our application on top of it.

Domain-specific languages are better

Domain-specific languages can exploit domain knowledge in ways general-purpose languages can't, providing

- clean abstractions
- domain-specific semantic checks
- domain-specific optimizations

Despite these benefits, domain-specific languages are rare.

The high-performance DSL recipe

- lexer, parser, type-checker/inference
- domain-specific semantic checks
- general-purpose optimizations (e.g., inlining, common subexpression elimination)
- domain-specific optimizations
- optimization heuristics and machine performance models
- code generation (C, JVM bytecode, LLVM IR)
- debugging, profiling and IDE support
- interface with other languages, or enough general-purpose features to do without

The high-performance DSL recipe: actual value

- lexer, parser, type-checker/inference
- domain-specific semantic checks
- general-purpose optimizations (e.g., inlining, common subexpression elimination)
- domain-specific optimizations
- optimization heuristics and machine performance models
- code generation (C, JVM bytecode, LLVM IR)
- debugging, profiling and IDE support
- interface with other languages, or enough general-purpose features to do without

The high-performance DSL recipe: what's left

- lexer, parser, type-checker/inference
- domain-specific semantic checks
- general-purpose optimizations (e.g., inlining, common subexpression elimination)
- domain-specific optimizations
- optimization heuristics and machine performance models
- code generation (C, JVM bytecode, LLVM IR)
- debugging, profiling and IDE support
- interface with other languages, or enough general-purpose features to do without

Embedded DSLs get us to here.

The high-performance DSL recipe: what's left

- lexer, parser, type-checker/inference
- domain-specific semantic checks
- general-purpose optimizations (e.g., inlining, common subexpression elimination)
- domain-specific optimizations
- optimization heuristics and machine performance models
- code generation (C, JVM bytecode, LLVM IR)
- debugging, profiling and IDE support
- interface with other languages, or enough general-purpose features to do without

Commensal compilers reduce effort to just the domain knowledge.

Commensal compilation

Commensal compilers implement domain-specific languages on top of managed language runtimes.[1]

Massive investment in optimizing JIT compilers.

Let the JIT compiler do the heavy lifting. Only do the missing domain-specific optimizations.

I'll talk about the JVM, but .NET provides similar features.

[1] In ecology, a commensal relationship between species benefits one species without affecting the other; e.g., barnacles on a whale.

I'll talk about two commensal compilers today.

- a matrix math compiler built around the EJML library, which has two APIs, a simple API and a high performance API; our compiler lets users code to the simple API without forgoing performance (not in the paper)
- StreamJIT, a stream programming language strongly inspired by StreamIt, which provides 2.8 times better average throughput than StreamIt with an order-of-magnitude smaller compiler

Simple API or high performance?

$y = z - Hx$

```
y = z.minus(H.mult(x));
```

$S = HPH^T + R$

```
S = H.mult(P).mult(
      H.transpose()).plus(R);
```

$K = PH^T S^{-1}$

```
P.mult(H.transpose().mult(
      S.invert()));
```

$x = x + Ky$

```
x = x.plus(K.mult(y));
```

$P = P - KHP$

```
P = P.minus(K.mult(H).mult(P));
```

Simple API or high performance?

$y = z - Hx$	`y = z.minus(H.mult(x));`	`mult(H, x, y);` `sub(z, y, y);`

$S = HPH^T + R$	`S = H.mult(P).mult(`
	`      H.transpose()).plus(R);`
		`mult(H, P, c);`
		`multTransB(c, H, S);`
		`addEquals(S, R);`

$K = PH^T S^{-1}$	`P.mult(H.transpose().mult(`
	`      S.invert()));`
		`invert(S, S_inv);`
		`multTransA(H, S_inv, d)`
		`mult(P, d, K);`

$x = x + Ky$	`x = x.plus(K.mult(y));`	`mult(K, y, a);` `addEquals(x, a);`

$P = P - KHP$	`P = P.minus(K.mult(H).mult(P));`	`mult(H, P, c);`
		`mult(K, c, b);`
		`subEquals(P, b);`

Domain knowledge is temporary matrix reuse, transposed multiplies, and destructive operations. Operations API is 19% faster.

Commensal EJML compiler user interface

The user codes against the simple API, then calls our compiler to get an object implementing the same interface and uses it as normal.

```
KalmanFilter f = new Compiler().compile(KalmanFilter.class,
    KalmanFilterSimple.class,
    F, Q, H, new DenseMatrix64F(9, 1), new DenseMatrix64F(9, 9)));

/* use f as normal */
DenseMatrix64F R = CommonOps.identity(measDOF);
for (DenseMatrix64F z : measurements) {
    f.predict();
    f.update(z, R);
}
```

Commensal EJML compiler passes

We'll compile the simple API to the complex one by

1. building an expression DAG from the compiled bytecode
2. fusing multiply and transpose
3. packing temporaries, using inplace operations when possible
4. building a method handle chain that calls the complex API

Users get both the simple API and good performance.

Building the expression DAG

```
String name = ci.getMethod().getName();
if (name.equals("getMatrix") || name.equals("wrap"))
    exprs.put(i, exprs.get(fieldMap.get(ci.getArgument(0))));
else if (name.equals("invert"))
    exprs.put(i, new Invert(exprs.get(ci.getArgument(0))));
else if (name.equals("transpose"))
    exprs.put(i, new Transpose(exprs.get(ci.getArgument(0))));
else if (name.equals("plus"))
    exprs.put(i, new Plus(
        exprs.get(ci.getArgument(0)),
        exprs.get(ci.getArgument(1))));
else if (name.equals("minus"))
    exprs.put(i, new Minus(
        exprs.get(ci.getArgument(0)),
        exprs.get(ci.getArgument(1))));
else if (name.equals("mult"))
    exprs.put(i, Multiply.regular(
        exprs.get(ci.getArgument(0)),
        exprs.get(ci.getArgument(1))));
```

58 lines to build expression DAG from SSA-style bytecode IR.

Fusing multiply and transpose

```java
private static void foldMultiplyTranspose(Expr e) {
    if (e instanceof Multiply) {
        Multiply m = (Multiply)e;
        Expr left = m.deps().get(0), right = m.deps().get(1);
        if (left instanceof Transpose) {
            m.deps().set(0, left.deps().get(0));
            m.toggleTransposeLeft();
        }
        if (right instanceof Transpose) {
            m.deps().set(1, right.deps().get(0));
            m.toggleTransposeRight();
        }
    }
    e.deps().forEach(Compiler::foldMultiplyTranspose);
}
```

Code generation

We want to generate code that reuses the JVM's full optimizations.

- Interpret the expression DAG
  - dynamism inhibits JVM optimization

Code generation

We want to generate code that reuses the JVM's full optimizations.

- Interpret the expression DAG
  - dynamism inhibits JVM optimization
- Linearize DAG, then interpret (command pattern)
  - dynamism inhibits JVM optimization

Code generation

We want to generate code that reuses the JVM's full optimizations.

- Interpret the expression DAG
  - dynamism inhibits JVM optimization
- Linearize DAG, then interpret (command pattern)
  - dynamism inhibits JVM optimization
- Emit bytecode
  - complicated; moves compiler one metalevel up

Code generation

We want to generate code that reuses the JVM's full optimizations.

- Interpret the expression DAG
  - dynamism inhibits JVM optimization
- Linearize DAG, then interpret (command pattern)
  - dynamism inhibits JVM optimization
- Emit bytecode
  - complicated; moves compiler one metalevel up

We can use method handles to easily generate optimizable code.

Method handles

Method handles are typed, partially-applicable function pointers.

`static final` method handles are constants, so are their bound arguments – so the JVM can inline method handle chains all the way through.

```
private static final MethodHandle UPDATE = ...;
public void update(DenseMatrix64F z, DenseMatrix64F R) {
    UPDATE.invokeExact(z, R);
}
```

Method handle combinators

```java
public static MethodHandle apply(MethodHandle f, MethodHandle... args){
    for (MethodHandle a : args)
        f = MethodHandles.collectArguments(target, 0, a);
    return f;
}
```

Method handle combinators

```java
public static MethodHandle apply(MethodHandle f, MethodHandle... args){
    for (MethodHandle a : args)
        f = MethodHandles.collectArguments(target, 0, a);
    return f;
}

private static void _semicolon(MethodHandle... handles) {
    for (MethodHandle h : handles)
        h.invokeExact();
}
private static final MethodHandle SEMICOLON =
    findStatic(Combinators.class, "_semicolon");
public static MethodHandle semicolon(MethodHandle... handles) {
    return SEMICOLON.bindTo(handles);
}
```

Commensal EJML code generation

We walk the expression DAG, asking each node to provide a method handle.

```java
final MethodHandle ADD = findStatic(CommonOps.class, "add", params(3)),
    ADD_EQUALS = findStatic(CommonOps.class, "addEquals", params(2));
public MethodHandle operate(List<MethodHandle> sources, MethodHandle sink) {
    if (sources.get(0) == sink)
        return Combinators.apply(ADD_EQUALS, sources.get(0), sources.get(1));
    else if (sources.get(1) == sink)
        return Combinators.apply(ADD_EQUALS, sources.get(1), sources.get(0));
    return Combinators.apply(ADD, sources.get(0), sources.get(1), sink);
}
```

Inlining all the way down

```
private static final MethodHandle UPDATE = ...;
public void update(DenseMatrix64F z, DenseMatrix64F R) {
    UPDATE.invokeExact(z, R);
}
```

`UPDATE` is a constant, so the JVM inlines it.

Inlining all the way down

```java
public void update(DenseMatrix64F z, DenseMatrix64F R) {
    this.z = z;
    this.R = R;
    for (MethodHandle h : HANDLES)
        h.invokeExact();
}
```

The `HANDLES` array is a constant, so the JVM can unroll the loop.

Inlining all the way down

```
public void update(DenseMatrix64F z, DenseMatrix64F R) {
    this.z = z;
    this.R = R;
    HANDLES[0].invokeExact();
    HANDLES[1].invokeExact();
    HANDLES[2].invokeExact();
    HANDLES[3].invokeExact();
    HANDLES[4].invokeExact();
    HANDLES[5].invokeExact();
    HANDLES[6].invokeExact();
    HANDLES[7].invokeExact();
    HANDLES[8].invokeExact();
    HANDLES[9].invokeExact();
    HANDLES[10].invokeExact();
    HANDLES[11].invokeExact();
    HANDLES[12].invokeExact();
}
```

The JVM can inline each array element method handle.

Inlining all the way down

```
public void update(DenseMatrix64F z, DenseMatrix64F R) {
    this.z = z;
    this.R = R;
    mult(MH, MH, MH);
    multTransB(MH, MH, MH);
    addEquals(MH, MH);
    invert(MH);
    multTransA(MH, MH, MH);
    mult(MH, MH, MH);
    mult(MH, MH, MH);
    mult(MH, MH, MH);
    subEquals(MH, MH);
    mult(MH, MH, MH);
    sub(MH, MH, MH);
    mult(MH, MH, MH);
    addEquals(MH, MH);
}
```

The argument-providing handles `MH` are constants, so the JVM can inline them.

Inlining all the way down

```java
public void update(DenseMatrix64F z, DenseMatrix64F R) {
    this.z = z;
    this.R = R;
    mult(this.H, this.P, t1);
    multTransB(t, this.H, t2);
    addEquals(t2, this.R);
    invert(t2);
    multTransA(this.H, t2, t1);
    mult(this.P, t1, t3);
    mult(t3, this.H, t2);
    mult(t2, this.P, t4);
    subEquals(this.P, t4);
    mult(this.H, this.x, t5);
    sub(this.z, t5, t5);
    mult(t3, t5, t1);
    addEquals(this.x, t1);
}
```

The JVM can continue to optimize just as with hand-written code.

Evaluation

730 non-comment lines of code; about a week of effort.

EJML Kalman filter benchmark:
Simple API: 1793ms
Complex API: 1503ms
Commensal-compiled simple API: 1529ms

StreamJIT

StreamIt is a synchronous dataflow stream programming language.
The StreamIt compiler emits C code for GCC.

The StreamIt compiler is 266,000 lines of Java,
including a 31,000-line Eclipse IDE plugin.

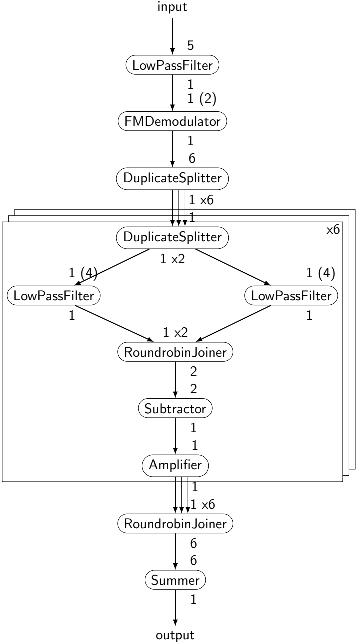
The StreamJIT commensal compiler is 27,000 lines of Java and Python – an order of magnitude smaller than StreamIt and smaller than StreamIt's IDE plugin alone.

StreamJIT achieves 2.8 times better throughput than StreamIt on StreamIt's own benchmark suite.

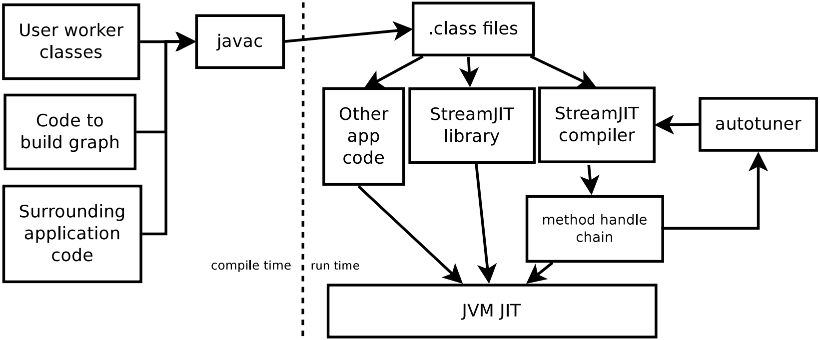
Synchronous dataflow

Synchronous dataflow programs are graphs of (mostly) stateless workers with statically-known data rates.

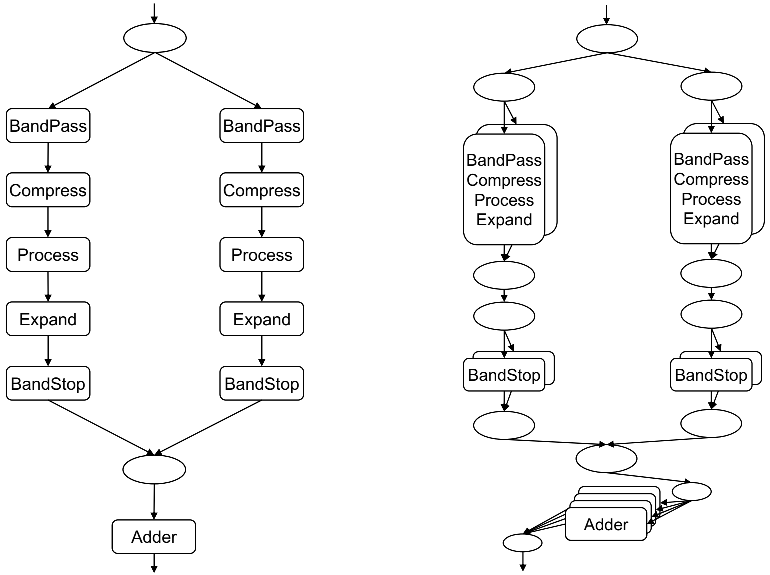
Using the data rates, the compiler can compute a schedule of worker executions, fuse workers and introduce buffers to remove synchronization, then choose a combination of data, task and pipeline parallelism to fit the machine.



StreamJIT Workflow



Fusion, data-parallel fission and splitter/joiner removal



IR is domain-level; mirrors stream graph, not worker bodies.

Problems with optimization heuristics

Optimizations themselves are easy. Hard part is deciding when to apply them based on the program, backend compiler, and machine.

We want to reuse the JVM as a black box, not model it.

Modeling hardware kills (performance) portability.

Models require maintenence as the JVM and hardware change.

Autotuning

We delegate our optimization decisions to the OpenTuner extensible autotuner, which decides

- an overall schedule multiplier (to amortize synchronization)
- whether to fuse workers
- whether to remove splitters and joiners
- how to allocate fused groups to cores

Code generation by method handles

Work allocation produces a schedule of worker executions per core.

We build a method handle chain that realizes a loop nest using custom combinators.

```
private static void _filterLoop(MethodHandle work, int iterations,
        int subiterations, int pop, int push, int firstIteration) {
    for (int i = firstIteration*subiterations;
            i < (firstIteration+iterations)*subiterations;
            ++i)
        work.invokeExact(i * pop, i * push);
}
```

Evaluation

benchmark	StreamJIT	StreamIt	relative perf
FFT	25,210,084	2,459,016	10.3
TDE-PP	12,605,042	2,357,564	5.3
DCT	23,622,047	6,434,316	3.7
DES	17,441,860	6,469,003	2.7
Beamformer	2,320,186	1,204,215	1.9
BitonicSort	9,771,987	6,451,613	1.5
FMRadio	2,272,727	2,085,143	1.1
ChannelVocoder	551,065	796,548	0.7
Filterbank	924,499	1,785,714	0.5
Serpent	2,548,853	6,332,454	0.4
MPEG2	32,258,065	-	-
Vocoder	406,394	-	-

2.8 times higher throughput (outputs/second) on 24 cores.

Conclusion

Commensal compilers reduce the cost of building domain-specific languages by reusing general-purpose languages and runtimes.

Thinking of adding a complex, abstraction-breaking, high-performance API to your library? Build a commensal compiler instead!

https://github.com/jbosboom/commensal-ejml
https://github.com/jbosboom/streamjit

Backup slides

StreamJIT source breakdown

User API (plus private interpreter plumbing)	1,213
Interpreter	1,032
Compiler	5,437
Distributed runtime	5,713
Tuner integration	713
Compiler/interp/distributed common	4,222
Bytecode-to-SSA library	5,166
Utilities (JSON, ILP solver bindings etc.)	2,536
Total (non-test)	26,132
Benchmarks and tests	7,880
Total	33,912

Vectorization limitations

```
float[] autocorr = new float[this.winsize];
for (int i = 0; i < this.winsize; i++) {
    float sum = 0;
    for (int j = i; j < winsize; j++)
        sum += peek(i) * peek(j);
    autocorr[i] = sum / winsize;
}
```