Getting Started with StreamJIT

Jeffrey Bosboom Sumanaruban Rajadurai

June 9, 2014

Section 1 contains instructions for setting up StreamJIT. Section 2 describes how to autotune the standard benchmarks used in the paper to replicate the paper's results modulo hardware differences and autotuning randomness. Section 3 describes how to write and autotune a new StreamJIT program.

Section 4 describes the StreamJIT distributed backend.

This guide accompanies our OOPSLA artifact submission virtual machine, which is a VirtualBox (virtualbox.org) virtual machine. The username and password are both `oopsla`. **The version of this guide in the virtual machine image contains errors:** it has curly quotes and backquotes in place of straight quotes and backticks. Even in this corrected version, some PDF readers introduce garbage at line breaks when copying multi-line shell commands. Please take care when copying, or use the scripts provided in the VM image. (The errors were noticed when the authors were attending PLDI 2014, and the hotel Internet connection will not support uploading a new multi-gigabyte VM image.)

1 Requirements and Setup

If you're using the virtual machine from the OOPSLA artifact submission, the steps in this section have already been performed for you; just `cd ~/streamjit`.

StreamJIT requires a recent release of Java 7 or Java 8 (preferred). Early releases of Java 7 had performance issues (related to outdated inlining heuristics) and mid-life releases had correctness issues (related to the newly-introduced incremental inlining). StreamJIT also requires the Apache Ant build tool.

OpenTuner (`https://github.com/jansel/opentuner`) is expected to be checked out in `../opentuner` relative to the StreamJIT repository directory. See OpenTuner's readme for instructions on setting up OpenTuner.

1. In the StreamJIT repository directory, build StreamJIT with `ant -Dplatforms.JDK_1.8.home=`*`home`* `jar`, where *home* is the path to your JDK installation (e.g., `/usr/lib/jvm/java-7-openjdk-amd64/`). Use this command line even if you're using Java 7. (This is provided as `build-jar.sh` in the artifact submission VM.)

2. Run the sanity and regression tests with `java -cp `` `echo dist/*.jar lib/*/*.jar | sed 's/ /:/g'` `` edu.mit.streamjit.test.Benchmarker --include-attribute=SANITY --include-attribute=REGRESSION` (provided as `sanity-reg-test.sh` in the artifact submission VM). You should see 92 output lines like `Compiler2StreamCompiler (8 cores 1 mult) / BitReverse8RegTest / [01000): 1913 ms compile, 338 ms run; 1000 items output, 0.338000 ms/output`.

2 Autotuning Standard Benchmarks

Your autotuning results may vary based on your hardware (especially if running inside a virtual machine), the length of the tuning run, and the vagaries of stochastic search. If you don't have time or hardware, you can still get some sense for the search space by trying some configurations manually – see section 2.1.

1. Set the maximum number of threads StreamJIT will use when running the benchmark by modifying the argument in `new SubsetBiasAllocationStrategy(8)` on line 97 in `src/mit/edu/streamjit/impl/compiler2/Compiler2.java`, then rebuild StreamJIT as described in section 1, step 1.

2. If you need to use a `java` executable not on your `PATH` (perhaps because your system's default Java is too old), edit `lib/opentuner/streamjit/tuner3.py` line 134 to specify the path to the proper executable. (This is not necessary in the artifact submission VM.)

3. Start a tuning run for a benchmark with `lib/opentuner/streamjit/tuner3.py --program=`*`benchmark`* `--stop-after=`*`seconds`*, where *benchmark* is

paper name	internal name
Beamformer	Beamformer
BitonicSort	BitonicSort (N = 32, asc)
ChannelVocoder	ChannelVocoder 16, 64
DCT	DCT2
DES	DES2
FFT	FFT5
Filterbank	FilterBankBenchmark
FMRadio	FMRadio 7, 128
MPEG2	MPEG2
Serpent	Serpent
TDE-PP	TDE_PP
Vocoder	Vocoder

Figure 1: The benchmark names used in the paper and the internal benchmark names.

one of the internal names in figure 1, quoted if it contains spaces. The autotuner will print the result of each trial in the tuning run in the form of four numbers, the last of which is the *inverse throughput* in nanoseconds per output data item that the tuner seeks to minimize. (The paper inverts and scales this value to show throughput in outputs per second.) The tuner will also periodically print the best known configuration's score and which search technique discovered it. At the end of the tuning run, the autotuner will write the best-performing configuration (as a JSON blob) to the current directory.

2.1 Manual Tuning

You can manually experiment with different configurations on the command line by running `java -cp` `` `echo dist/*.jar lib/*/*.jar | sed 's/ /:/g'` `` `edu.mit.streamjit.impl.compiler2.Compiler2` *`benchmark threads multiplier`*, where *benchmark* is one of the internal names in figure 1 (quoted if it contains spaces), *threads* is the number of threads to use, and *multiplier* is the granularity multiplier (increasing values amortize the cost of synchronization, but use more cache). You can also turn fusion, removal and unboxing on or off by changing the strategies around line 97 in `src/mit/edu/streamjit/impl/compiler2/Compiler2.java` to `new`

`AlwaysStrategy()` or `new NeverStrategy()`, then rebuilding the StreamJIT JAR file. If you change the strategies, ensure you change them back and rebuild the JAR file before running the autotuner.

The output from manual tuning is the same as printed by the autotuner for each tuning run, described in section 2, step 3.

3 Writing New StreamJIT Programs

1. Write new worker classes, then compose them with pipelines and splitjoins. Details are in section 5.1 of my S.M. thesis, which is an expanded version of the paper's description of the StreamJIT API. (The thesis is included in the artifact submission VM as `~/jbosboom-sm-final.pdf`.) The tests and applications in `src/mit/edu/streamjit/test/{sanity,regression,apps}` may be useful as additional examples.

2. For debugging, compile the graph using `DebugStreamCompiler`. This will interpret the graph on a single thread without transforming the bytecode, allowing you to set breakpoints in `work` methods as with any other Java code. You can also use the `CheckVisitor` class to check your stream graphs for errors – in particular, unbalanced splitjoins.

3. For performance, compile the graph using `Compiler2StreamCompiler` (note the `2`). You can play around with a few parameters as described in section 2.1 above.

4. To enable autotuning, implement a `edu.mit.streamjit.test.AbstractBenchmark` subclass. Its `instantiate` method should return a fresh copy of your stream graph. Pass a `Dataset` with input suitable for your graph to the superclass constructor (if you provide multiple datasets, only the first will be used). Reference output is not required and will be ignored during tuning. If the input is small, the autotuner will loop over it to ensure each tuning trial runs long enough to provide accurate results. For best performance, the input should come from a random-access source, such as a file or `ArrayList`. Finally, annotate the benchmark subclass with `@ServiceProvider(Benchmark.class)` and rebuild the StreamJIT JAR file; our annotation processor will register your benchmark class in the appropriate `META-INF/services` file.

5. After creating a benchmark class, you can autotune your StreamJIT program just the same way you tune the standard benchmarks, passing the name of your benchmark (unless you specified otherwise to the superclass constructor, it's the name of the benchmark class) instead of the name of a standard benchmark. If your benchmark can't be found, open the StreamJIT JAR (`dist/jstreamit.jar` and ensure your benchmark class is listed in `META-INF/services/edu.mit.streamjit.test.Benchmark`. If it isn't, our annotation processor either didn't run or has a bug, so you'll have to register your benchmark class manually. (In particular, Eclipse seems to skip running the annotation processor.)

4 Distributed

Implementing the distributed backend in a timely manner (i.e., before the paper deadline) required breaking some abstractions, so it resides in a separate repository. In the artifact submission VM, the distributed backend is in `~/streamjit-distributed`.

These steps will start two StreamNodes in different Java virtual machines, and StreamJIT will connect with them via loopback TCP connection and distribute the application.

1. Build StreamJIT in the same way as non-distributed StreamJIT.

2. Export edu.mit.streamjit.impl.distributed.node.StreamNode as a runnable jar file with the name of "StreamNode.jar" and save it into streamjit-distributed folder. In Eclipse, this can be done by right click on 'project explorer' and select 'export', 'runnable JAR file'. (In the artifact submission VM, this is already done.)

3. Open any StreamJIT program's file and run it from Eclipse. e.g., edu.mit.streamjit.test.apps.fft5. FFT5, DES2, DCT2, Beamformer1, ChannelVocoder7 and FMRadio are ready to run in distributed mode.

4.1 Advanced steps

1. Automatically starting the StreamNodes can be stopped by turning GlobalConstants.autoStartStreamNodes off. In this case, user has to manually start the "StreamNode.jar".

2. If you want to distribute to more than 2 StreamNodes, then noOfStreamNodes can be passed as application argument when start the StreamJIT Application.

3. You can turn on the GlobalConstants.printOutputCount to see the number of outputs generated as a console print message.

4. Execution time is measured to generate GlobalConstants.outputCount amount of outputs at steady state execution. (Initial schedule execution outputs will be skipped from this measurement.) You can change GlobalConstants.outputCount at the main() method to change the amount of output used when timing execution. (Different StreamJIT programs may take different time to generate given amount of outputs.)