1 Introduction

1.1 What is it

Stream aims to be a software framework for the implementation of large scale online learning algorithms. Large scale, in this context, should be understood as something that does not fit in the memory of a standard desktop computer. As the objective is to deal with large amounts of data, a lot of effort is being made to optimize the implementation for speed, sometimes at expense of conciseness - for example, using standard C I/O libraries instead of C++ stream ones.

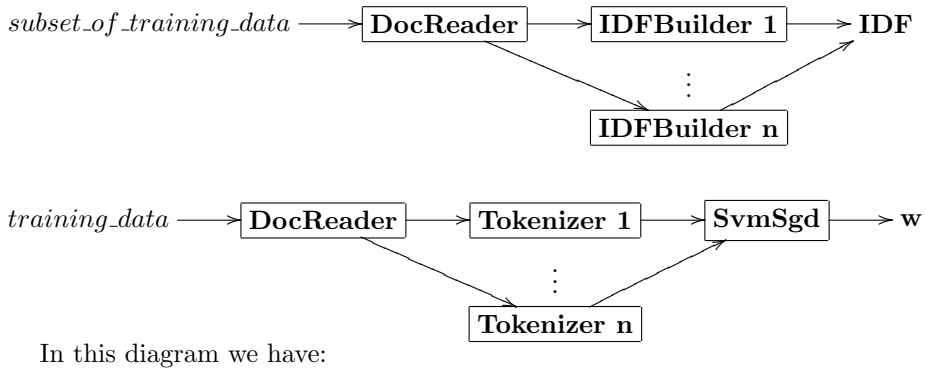
So far *stream* has been optimized for Intel multi-core processors, using Intel's Thread Building Blocks and Math Kernel libraries. As future development we are planning to add support to general-purpose computing on graphics processing units (GPGPU), using NVidia cards and the CUDA software development kit.

1.2 Architecture

The basic building block of *stream* is a *filter*. A *filter* is responsible for processing a unit of information: it receives an object, processes it and sends the result to the next *filter*.

Filters are connected to form a *pipe*, and a group of *pipes* is what we call a *task*, which is the code that uses these blocks to do some useful work.

To illustrate this, lets give an example of a simple task - *Reuters*:



In this diagram we have:

1. two groups of input files: *subset_of_training_data* and *training_data*
2. four different filters: **DocReader**, **IDFBuilder**, **Tokenizer** and **SvmSgd**
3. two output variables: **IDF** and **w**

So this is how it works:

1. in the first pipe we have two filters:
   - **DocReader** allocates memory for a buffer, reads data from *subset_of_training_data*, pre processes it (file unzipping, XML parsing, etc.), fills the buffer with a list of documents (text and class) and sends it to the next *filter*.

- **IDFBuilder** receives the buffer and calculates the number of documents each word appears on, storing the results in a variable (IDF). As this is the last *filter*, the buffer is freed. Since we are using a variable that supports concurrent access, we can run several instances of this *filter* in parallel. As long as there are available resources (CPUs), Intel's Thread Building Blocks (TBB) library will take care of this automatically.

2. the second pipe is similar:

   - **DocReader** allocates memory for a buffer, reads data from *training_data*, pre processes it (file unzipping, XML parsing, etc.), fills the buffer with a list of documents (text and class) and sends it to the next *filter*.
   - **Tokenizer** receives the buffer and, using the IDF variable from the last pipe, builds vectors with the TF/IDF of the documents, and passes them to the next *filter*. Again, this can be ran in parallel.
   - **SvmSgd** receives these vectors and uses them to train a simple stochastic gradient descent algorithm. The end result is a weight vector (w).

1.3 File Hierarchy

doc:	Documentation.
stream.tex	Source for this document.
src:	Source files.
config.txt	Example configuration file.
Makefile	Main makefile.
build_tests	Directory used to build unit tests (see 4.1)
bin	CxxTest executables (see 2.2.1)
data	Test data for the unit tests (see 4.1)
feature_extractors	Filters to process data and extract features (see 1.2)
libs	External libraries (see 2.2)
cxxtest	CxxTest (see 2.2.1)
FLENS-RC1	FLENS (see 2.2.4)
loss	Loss class
models	Model class
readers	Filters to read and pre process data (see 1.2)
scripts	Some scripts (pre processing, etc)
solvers	Filters to run algorithms (see 1.2)
tasks	Tasks (see 1.2)
utils	Utility functions/libraries

2 Installation

2.1 Obtaining stream

Stream is part of the Elefant project, and can be obtained here. You can check out the latest version of the source code with this command:

```
svn co http://elefant.developer.nicta.com.au/local/repos/trunk/elefant/stream
```

2.2 Prerequisites

Stream uses the following external packages:

1. CxxTest.
2. Intel's Thread Building Blocks library (TBB).
3. Intel's Math Kernel library (MKL).
4. Flexible Library for Efficient Numerical Solutions (FLENS)

Some of these need to be installed, others are included in the source code. Let's see this in more detail now.

2.2.1 CxxTest

CxxTest) is a unit test framework for C++. All the necessary files are included in the source code, so nothing has to be done here. You will need, however, a working python or perl interpreter. If you are going to use perl you'll have to edit `src/Makefile` - look for `cxxtestgen.py` and change to `cxxtestgen.pl`.

2.2.2 TBB

TBB is the base of our filter/pipe structure. You will have to follow the instructions here to download and install it. All our tests were made with the 20080605 stable release.

After you install it, you may have to change the TBB path variables in `src/Makefile` (`tbb_base`, `tbb_include`, etc.).

2.2.3 MKL

MKL is a collection of math routines optimized for Intel CPUs, taking advantage of multiprocessor cores were applicable. In our tests we have been using version `10.0.1.014`. Again, you'll have to follow the instructions on the site to download and install it. Don't forget to set the environment variables after the installation is complete, as detailed on chapter 4 (Configuring Your Development Environment) of MKL's user's guide.

Note that MKL is not strictly necessary. Any BLAS implementation should be ok, as long as you update FLENS configuration to use it.

2.2.4 FLENS

FLENS is a very efficient BLAS and LAPACK C++ wrapper. In *stream* we use FLENS to interface with MKL's BLAS implementation.

We have done some modifications to FLENS to add support for sparse vector objects. These are not yet part of FLENS' distribution (and may never be), so FLENS' source code with these modifications is also included here. All you need to do is:

1. go to FLENS dir:

```
cd src/libs/FLENS-RC1/flens
```

2. compile it:

```
make
```

3. copy the resulting library (`libflens.so`) to a directory that is in `LD_LIBRARY_PATH` (i.e. `/usr/lib`)

2.3 Compiling stream

With all prerequisites installed, compilation of *stream* is just a matter of running make:

```
cd src/
make
```

That should create a *stream* executable.
Beyond this, there are three other make targets:

1. `make clean`: removes all object files and executables
2. `make distclean`: same as clean, but also removes dependency files
3. `make utests`: builds and run unit tests (see 4.1)

All make targets accept these optional arguments:

1. `dbg=1`: build for debugging - changes compilation flags and uses the debug version of TBB library
2. `prof=1`: same as above, but also enables profiling information generation for use with `gprof`
3. `vtun=1`: build for intel's `vtune` - optimizations enabled, but with debugging info

3 Using stream

Stream reads its configuration at run time from an external file, called `config.txt`. All parameters are described in this file, so I will not repeat them here.

Parameters can also be set in the command line using this syntax:

```
./stream param1 value1 param2 value2 ...
```

Command line values take precedence over `config.txt`.

4 Expanding stream

The source code is the best documentation, so there isn't much to be added here. As a starting point, you can follow the code for the task described in 1.2 looking for `do_reuters()` in `src/tasks/main.cpp`.

If you add any new functionality to the code it is good practice to also add some unit tests for it (see 4.1).

4.1 Unit tests

Unit tests are implemented using the CxxTest framework, which was chosen for its easiness of use and low verbosity.

To run the unit tests type

```
make utests
```

The Makefile will automatically search for all files starting with `test_` and ending in `.h`, add these to the unit test sources list, compile the tests and run them:

```
make utests
././build_tests/utest
Running 12 tests............OK!
```

To add a new test just create a new `test_xxx.h` file following the model of the existing ones. For more details see the CxxTest documentation.

5 License

Stream is licensed under the Mozilla Public License (MPL).