The Bones Source-to-Source Compiler Manual

Cedric Nugteren

August 7, 2012

Abstract

Recent advances in multi-core and many-core processors requires programmers to exploit an increasing amount of parallelism from their applications. Data parallel languages such as CUDA and OpenCL make it possible to take advantage of such processors, but still require a large amount of effort from programmers. To address the challenge of parallel programming, we introduce Bones.

Bones is a source-to-source compiler based on algorithmic skeletons and a new algorithm classification. The compiler takes C-code annotated with class information as input and generates parallelized target code. Targets include NVIDIA GPUs (through CUDA), AMD GPUs (through OpenCL) and x86 CPUs (through OpenCL and OpenMP). Bones is open-source, written in the Ruby programming language, and is available through our website. The compiler is based on the C-parser CAST, which is used to parse the input code into an abstract syntax tree (AST) and to generate the target code from a transformed AST.

This document is meant as a manual for users of Bones. It includes usage instructions, an installation guide and pointers to further documentation. It furthermore contains an overview of the tool itself and the skeletons, a mandatory read for users that plan on modifying or extending the skeletons and/or targets currently available in Bones.

Contents

1 General

This section provides some general information on the usage of Bones, a list of available examples and the current known limitations.

1.1 Usage

Bones can be invoked from the command-line. Two arguments (`-a` and `-t`) are mandatory, others are optional. This is an example of the usage of Bones assuming the file '`example.c`' to be present:

```
bones -a example.c -t GPU-CUDA -c
```

The full list of options is as follows:

```
Usage:
    bones --application <input> --target <target> [OPTIONS]
With the following flags:
  --application, -a <s>:   Input application file
       --target, -t <s>:   Target processor (choose from: GPU-CUDA, GPU-OPENCL-AMD,
                           CPU-OPENCL-INTEL, CPU-OPENCL-AMD, CPU-OPENMP, CPU-C)
     --measurements, -m:   Enable/disable timers
          --verify, -c:   Verify correctness of the generated code
         --version, -v:   Print version and exit
            --help, -h:   Show this message
```

1.2 Examples

The best place to start experimenting with Bones is the '`examples`' directory. A large number of examples are available in this folder, grouped by algorithm class (either element, neighbourhood, shared or tile). The examples illustrate different kinds of coding styles and give a large number of different classes to work with. The folder '`benchmarks`' gives more examples, taken from the PolyBench/C benchmark set. Additionally, a folder '`applications`' is included, containing example complete applications. Currently, the following examples are available:

```
|-- element
|   |-- example1.c
|   |-- example2.c
|   |-- example3.c
|   |-- example4.c
|   |-- example5.c
|   |-- example6.c
|   |-- example7.c
|   |-- example8.c
|   |-- example9.c
|   |-- example10.c
|   |-- example11.c
|   \-- example12.c
|-- neighbourhood
|   |-- example1.c
|   |-- example2.c
|   |-- example3.c
|   \-- example4.c
|-- shared
|   |-- example1.c
|   |-- example2.c
|   |-- example3.c
|   |-- example4.c
|-- applications
|   \-- ffos.c
\-- benchmarks
    |-- 2mm.c
    |-- 3mm.c
    |-- adi.c
    |-- atax.c
    |-- bicg.c
    |-- cholesky.c
    |-- correlation.c
    |-- covariance.c
    |-- doitgen.c
    |-- durbin.c
    |-- dynprog.c
    |-- fdtd-2d-apml.c
    |-- fdtd-2d.c
    |-- floyd-warshall.c
    |-- gemm.c
    |-- gemver.c
    |-- gesummv.c
    |-- jacobi-1d-imper.c
    |-- jacobi-2d-imper.c
    |-- lu.c
```

```
|   \-- example5.c
|-- chunk
|   |-- example1.c
|   |-- example2.c
|   |-- example3.c
|   |-- example4.c
|   \-- example5.c
|-- ludcmp.c
|-- mvt.c
|-- reg_detect.c
|-- saxpy.c
|-- seidel-2d.c
|-- syr2k.c
|-- syrk.c
|-- trisolv.c
\-- trmm.c
```

All examples can be ran through Bones for a specific target using an automated Rake task. Executing '`rake examples:generate`' or simply '`rake`' will execute Bones for all examples for a given target. The target can be changed in the '`Rakefile`' found in the root directory of Bones. More information on Rake tasks is given in section 5.1 of this document.

1.3 Limitations

Bones takes C99 source code as input. However, several coding styles are unsupported as of now or might yield worse performance compared to others. The numerous examples provided should give the user an idea of the possibilities and limitations of the tool. A complete list of coding guidelines and limitations will follow as soon as Bones is out of beta. Currently, an initial list of major limitations and guidelines is given below. In this list, we use '`algorithm`' to denote an algorithm captured by an algorithm class.

- If the algorithm works on a N-dimensional data structure, use N-dimensional arrays (don't flatten it yourself, e.g. use '`array[i][j]`' instead of '`array[i+j*A]`') and specify an N-dimensional algorithm class.
- Write your while-loops as for-loops if possible. For-loops should have a unit increment, other loops (e.g. decrementing loops) must be rewritten.
- Loops can have affine bounds containing constants, defines and variables. Variables should not include loop variables of loops that are part of the '`algorithm`'.
- Function calls are not allowed within the '`algorithm`'. Some mathematical functions are allowed.
- Variables are allowed in the definition of an algorithm class. If they are used, the should also be used somewhere in the body of the '`algorithm`'.
- Bones is designed to work on a single input file with at least a function called 'main'. If your (to-be-accelerated) code spawns over multiple C-files, Bones could either be applied multiple times, or the code could be merged into a single file.

2 Installation procedure

Installation of Bones is a simple matter of extracting the Bones package to a directory of your choice or installing the gem ('`gem install bones-compiler`'). However, there are a number of prerequisites.

2.1 Prerequisites

Bones requires the installation of Ruby, the Rubygems gem package manager and two gems:

1. Any version of **Ruby 1.8** or **1.9**. Information on Ruby is found at http://www.ruby-lang.org.
   - [OS X]: Ruby is pre-installed on any OS X system since Tiger (10.4).
   - [Linux]: Ruby is pre-installed on some Linux based systems. Most Linux package managers (yum, apt-get) will be able to provide a Ruby installation. Make sure that the ruby development package ('`ruby-devel`') is also installed, as it is required by one of the gems.
   - [Windows]: Ruby for Windows can be obtained from http://rubyinstaller.org/.
2. The **Rubygems** gem package manager. Information on Rubygems can be found at http://rubygems.org.
   - [OS X]: Rubygems is pre-installed on any OS X system since Tiger (10.4).
   - [Linux]: Most Linux package managers will be able to provide a Rubygems installation by installing the package '`rubygems`'.
   - [Windows]: Rubygems for Windows is obtained automatically when installing from http://rubyinstaller.org/.
3. Bones requires two gems, **trollop** and **cast**. Both gems can be installed by calling Rubygems from the command line, i.e.: '`gem install trollop cast`'. The 'cast' gem currently contains a minor bug causing installation problems with Ruby 1.9.2. Until a fix is provided by the maintainer of the gem, it is recommended to install an older Ruby version (e.g. 1.8.7).

For example, all prerequisites can be installed as follows on a Fedora, Red-Hat or CentOS system:

```
yum install ruby ruby-devel rubygems
gem install trollop cast
```

For an Ubuntu, Debian or Mint system, the equivalent commands are:

```
apt-get install ruby ruby-devel rubygems
gem install trollop cast
```

2.2 Installing Bones

To install the compiler, simply extract the '`bones_x.x.tar.gz`' package to a directory of your choice. The Bones executable is found in the '`bin`' subdirectory. Including the path to the '`bin`' directory to your environmental variable '`PATH`' will make Bones available from any directory on your machine. The compiler can also be installed as a gem ('`gem install bones-compiler`').

3 Documentation

There are two ways to go to obtain more information regarding Bones. To obtain more information about the compiler itself, the ideas behind it and the algorithm classification, it is a good idea to read scientific publications. To get more information about the code structure, HTML documentation can be generated automatically using RDoc.

3.1 Code documentation

Code documentation can be generated automatically using RDoc. Navigate to the installation root of Bones and use Rake to generate documentation: '`rake rdoc`'. More information on using Rake is provided in section 5.5. Next, open '`rdoc/index.html`' to navigate through the documentation. The same documentation is also available on the web at '`http://parse.ele.tue.nl/tools/bones/rdoc/`'.

3.2 Scientific publications

Scientific publications related to Bones can be obtained from '`http://parse.ele.tue.nl/publications`'. Two publications are relevant:

- 'A Modular and Parametrisable Classification of Algorithms' [1], which provides details on the used algorithm classification. When referring to the algorithm classification in scientific work, you are kindly asked to include the following citation:

```
@TECHREPORT{Nugteren2011,
  author      = {Cedric Nugteren and Henk Corporaal},
  title       = {{A Modular and Parametrisable Classification of Algorithms}},
  institution = {Eindhoven University of Technology},
  year        = {2011},
  number      = {No. ESR-2011-02},
}
```

- 'Introducing Bones: A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons' [2], which introduces the Bones source-to-source compiler. When referring to Bones in scientific work, you are kindly asked to include the following citation:

```
@INPROCEEDINGS{Nugteren2012,
  author    = {Cedric Nugteren and Henk Corporaal},
  title     = {{Introducing 'Bones': A Parallelizing Source-to-Source Compiler
               Based on Algorithmic Skeletons}},
  booktitle = {{GPGPU-5: 5th Workshop on General Purpose Processing on
               Graphics Processing Units}},
  year      = {2012},
}
```

4 Modifying and extending Bones

This section contains an overview of the tool and the skeletons it uses. It is aimed at users that plan on modifying or extending the skeletons and targets which are currently available in Bones.

4.1 Overview of the Bones compiler

A high-level overview of the Bones source-to-source compiler is found in figure 1. The compiler performs 4 steps, which are explained below. The step numbers correspond to numbers given in the blue boxes in figure 1.

1. The pre-processor extracts the user supplied class (or: algorithmic species) information from the source code. Code in between the `#pragma species kernel` and `#pragma species endkernel` statements along with the class (or: species) name is stored as an '`algorithm`'. Multiple such algorithms can exist within one piece of source code.

2. The AST of the kernel code and the AST of the original code are analysed to obtain information on the used variables, e.g. are they dynamically or statically allocated, are they private to the kernel, of which dimension are they, are they used as input or output. The information on the variables is used later on to instantiate and populate the skeletons.

3. The kernel AST is transformed according to a per skeleton unique transformation list. These transformations are fairly basic, e.g. renaming of variables, removal of the outer loop. The transformation list contains integer codes referring to the different available transformations.

4. The skeletons are instantiated and populated. A common library supplies parametrised code which is equal independent of the algorithm class. This includes memory allocation and memory transfer mechanisms for example. The skeleton library supplies both parametrised host code and kernel code, unique to an algorithm class. The parameters are instantiated based on the extracted variable information and the supplied species information.

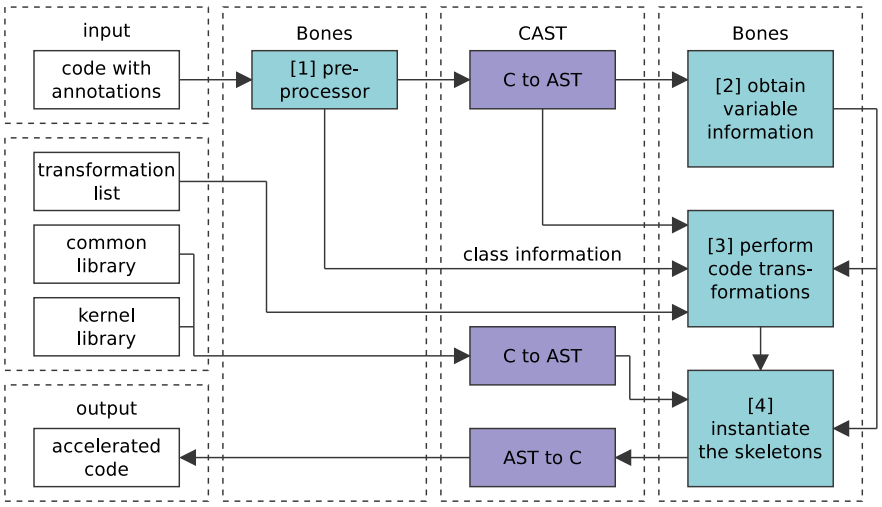

Figure 1: High-level overview of the Bones source-to-source compiler.

The Bones tool relies heavily on the AST format provided by the front-end and back-end of the CAST gem, as seen in figure 1. Because of this, multiple extensions have been made to the CAST gem to support the steps taken by the compiler. These extensions are part of the Bones tool and can be found in the '`lib/castaddon`' folder. The main (Bones) module is found in the '`lib/bones`' folder.

4.2 The structure of the skeletons

In order to extend or modify skeletons for Bones, it is paramount to understand the structure of the skeletons and the different output files. Bones takes a single C source file as an input (with the name '<name>.c'), and produces three or four output files. They are briefly introduced below, but discussed in more detail later:

- '<name>.c' — This file is a copy of the original input file, with some additions and alterations.
- '<name>_host.*' — This file contains the code used to set-up the accelerated kernel on the device. It might also contain a copy of the original code in case verification is enabled.
- '<name>_device.*' — This file contains the kernel which will be executed on the device.
- '<name>_verification.*' — This file contains the optional verification code (including the original code).

Following, the contents of the four files are explained in more detail. This is done according to colours as given in figure 2, which lists the contents of the three files. In this figure, strings between quotes denote a skeleton file, found either in the 'skeleton/verification' directory or in the 'skeleton/<target>' directory, in which '<target>' is the name of the selected target. Strings between square brackets denote (modified) original code parts. The contents are as follows:

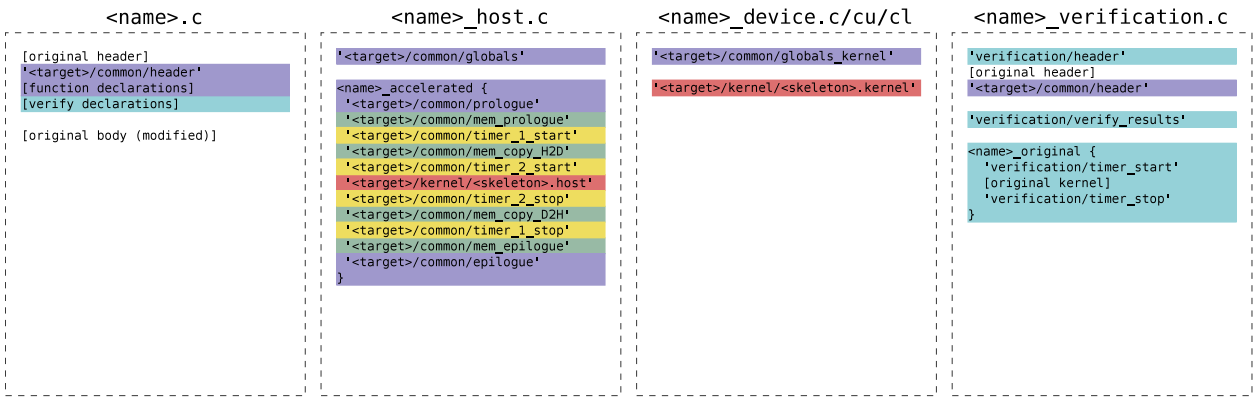

Figure 2: The structure of the skeletons.

red This is the class-specific part. It contains of a host part which sets-up the kernel and invokes it, and it contains the actual kernel code. This code is unique per target-class combination.

green This part contains the memory related code. It is target, but not class specific. It consists of four parts: the prologue (for all arrays), a host to device copy (for all input arrays), a device to host copy (for all output arrays) and an epilogue (for all arrays).

purple This is the general prologue, epilogue and header part, specific to a target. It contains headers for all files, including possible global variables or small functions. The header for the host function always contains an accelerated initialization function, although it might be empty for certain targets.

yellow These are the timer skeletons, which are only added when supplying the measurement flag.

blue This is the verification code, making it possible to run the original code as a reference for either performance comparison or result comparison (a check for correctness). This code is only generated when the verify flag is given.

4.3 Instantiating skeletons

The class-target specific skeletons as found in the '`skeletons/<target>/kernel/`' directory may contain a number of parameters. These parameters are instantiated by Bones during phase 4 (see figure 1). Parameters are delimited by angle brackets (< and >). A list of possible parameters is given below. In this list, an asterisk (*) is used to represent a positive integer, and brackets in combination with a vertical bar represent a selection possibility. The list is not fully explained, however, it is fairly straightforward to become familiar with by just trying out a few example parameters or by investigating existing skeletons. The list is as follows (comments start with a hash sign):

```
# Algorithm parameters (names, code, etc.)
algorithm_id
         _name
         _basename
         _filename
         _code*

# Parameters related to a specific input or output array. Some examples:
# 'in2_dimensions' gives the dimensions of the second input array (e.g. 16*32)
# 'in2_dimension1_sum' and 'in2_dimension2_sum' give the individual sizes (16 and 32 respectively)
# 'out1_type' might return int or float for example
(in*|out*)_type
          _name
          _devicename
          _devicepointer
          _dimensions
          _dimension*_to
                     _from
                     _sum
          _to
          _from

# Parameters related to tile and neighbourhood sizes
(in*|out*)_parameters
          _parameter*_to
                     _from
                     _sum

# List of global/local kernel IDs
(in*|out*)_ids
          _localids
          _flatindex

# These parameters represent different lists of all input or output arrays
(in|out)_names
        _devicenames
        _devicedefinitions
        _devicedefinitionsopencl

# These parameters represent different lists of all arrays (input and output combined)
names
devicenames
devicedefinitions
devicedefinitionsopencl

# Miscellaneous parameters
parallelism
factors
ids
verifyids
```

4.4 Species to skeletons and transformations

For each target, there is a mapping file: '`skeletons/<target>/skeletons.txt`'. The file contains a list of mappings from species to skeletons (see the file itself for more information), each with one or more 2-digit numbers at the end. The 2 digits represent settings for optimisations. The optimisations and the settings are as follows:

- The first digit enables ('1') or disables ('0') the use of local memory. It transforms the original array accesses such that it uses a different name and a different index, corresponding to the local memory of the target device.
- The first digit also enables a dimension swap for the first input ('2'), for both the first and the second ('3') or disables ('0') the swap. This swap is required for the skeletons implementing a swap of dimensions as a pre-kernel to enable better memory coalescing for example.
- The first digit finally also enables ('4') or disables ('0') thread-merging by a factor 4.
- The second digit enables ('1', '2', '3' or '4') or disables ('0') optimizations specific for reduction code. Details on this are in the '`transform_reduction`' function in the '`lib/castaddon/node.rb`' file.

A second (or third, or fourth, etc.) 2-digit number may be supplied. Again, it will perform certain optimizations (or not) depending on the given values. However, the resulting code is now saved in the parameter '`code2`' instead of '`code1`' (see paragraph on instantiating skeletons). In that way, skeletons which require multiple copies of the same code with different optimizations can be created.

4.5 Adding a new skeleton

Skeleton names can be made up arbitrarily by the user of Bones. However, it makes sense to follow the same conventions as used for the provided skeletons. Which species maps to which skeleton is defined in '`skeletons/<target>/skeletons.txt`'. There, Bones groups different species together to use a single skeleton. For example, $x|element \wedge x|element \rightarrow x|element$ uses the same (default) skeleton as $x|element \rightarrow x|element$. The exact rules are target-specific and found in the corresponding '`skeletons.txt`' files.

Adding a new skeleton is done in 3 steps:

1. Adding a host skeleton in the '`skeletons/<target>/kernel`' folder. The filename is created by taking the skeleton name, '`.host`' and the appropriate extension. The host skeleton does not start or ends with a function definition, as can also be seen in figure 2.
2. Adding a kernel skeleton in the '`skeletons/<target>/kernel`' folder. The filename is created by taking the skeleton name, '`.kernel`' and the appropriate extension. The kernel skeleton does contain a complete function. For some targets a forward declaration needs to be added to the host file. In that case, the declaration(s) can be added at the start of the kernel skeleton in between the '`/* STARTDEF`' and '`ENDDEF */`' markers.
3. Adding the skeleton name and a list of transformation settings to the '`skeletons.txt`'-file as described in the previous section.

4.6 Adding a new target

Adding a new target is as simple as copying an existing target's skeleton directory '`skeletons/<target>`' into the '`skeletons`' directory and giving it a new name. A Rake task has been added to so (see section 5.3), although it is a minor effort to do this manually. All files which are in the '`skeletons/<target>/common`' folder are required, but can be left blank if they are unused. Also, the '`skeletons.txt`'-file is necessary to ensure correct functioning of Bones. Individual skeletons can be added as described above.

5 Using Rake to automate your tasks

Rake is Ruby's Make. Similar to Make, Rake tasks are specified in a Rakefile. Bones's root directory provides such a Rakefile, which could help you automate a number of tasks.

5.1 Running examples

Examples can be ran through Bones for a specific target using a Rake task. Executing the task '`rake examples:generate`' or simply '`rake`' will execute Bones for all examples for a given target. The target itself is set in the Rakefile and can be modified accordingly. The Rakefile contains the following lines of code to determine a target:

```
TARGETS = ['GPU-CUDA','GPU-OPENCL-AMD','CPU-OPENCL-INTEL','CPU-OPENCL-AMD','CPU-OPENMP','CPU-C']
TARGET = TARGETS[5]
```

In the above example, the target is set to the 6th element in the array (5 starting at zero), which is 'CPU-C'.

Although the Rake task will run all examples in the examples directory per default, it can also be given a specific set of examples to run. Providing the task an argument (using square brackets) makes it possible to select one or multiple examples using wildcards (*). Below are a few examples:

```
rake examples:generate[examples/tile/example2.c]
rake examples:generate[examples/*/example1.c]
rake examples:generate[examples/element/example*.c]
```

5.2 Compiling and executing generated code automatically

Similar to the '`examples:generate`' task, there are the '`examples:compile`' and '`examples:execute`' tasks. They take filenames as an argument as well, in the same way as '`examples:generate`' does. These tasks are stub tasks (they are not filled in yet), because they are system-specific. Users can fill in these tasks to automate the compilation and execution of the examples, such that the Bones generated code is tested to compile and execute.

The task '`examples:verify`' combines the three tasks '`examples:generate`', '`examples:compile`' and '`examples:execute`', thus creating a complete test chain for the examples directory. The '`examples:verify`' target takes arguments similar to the other tasks in the '`examples`' namespace.

5.3 Adding a new target

Executing '`rake add_target[<target_name>,<base_target>]`' will create a new target with the name '`<target_name>`' based on the target '`<base_target>`'. An example is given below:

```
rake add_target[EXAMPLE-TARGET,CPU-OPENCL-INTEL]
```

5.4 Testing the Bones code

To test correctness of parts of the Bones code, tests have been created. Use '`rake test`' for testing.

5.5 Generating documentation

Code documentation can be generated automatically using RDoc. Navigate to the installation root of Bones and use Rake to generate documentation: '`rake rdoc`'. Next, open '`rdoc/index.html`' to navigate through the documentation.

5.6 Cleaning up

Rake provides a task to clean-up generated code: '`rake clobber`'. This will remove all the generated code in the sub-directories of the '`examples`' directory.

6 Questions

Questions can be directed by email. You can find contact details on the personal page of the author at http://parse.ele.tue.nl/cnugteren/ or on the project page at github.

References

[1] C. Nugteren and H. Corporaal. A Modular and Parameterisable Classification of Algorithms. Technical Report ESR-2011-02, Eindhoven University of Technology, 2011.

[2] C. Nugteren and H. Corporaal. Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons. In *GPGPU-5: 5th Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2012.