Habbakuk Documentation

Release 0.1

Andy Porter

Aug 10, 2018

Contents

1 Introduction **3**

2 Getting going **5**
- 2.1 Download . . . 5
- 2.2 Dependencies . . . 5
- 2.3 Installation . . . 5
- 2.4 Running . . . 6
- 2.5 Testing . . . 7

3 Using Habakkuk **9**
- 3.1 Performance Predictions . . . 9
- 3.2 DAG Output . . . 9

4 CPU Configuration **11**
- 4.1 Instruction Cost . . . 11
- 4.2 Mapping of Instructions to Execution Ports . . . 11
- 4.3 Instruction Overlapping . . . 12
- 4.4 Support for Fused Multiply-Add Operations . . . 12
- 4.5 Cache-line Size . . . 12
- 4.6 Clock Speed . . . 12

5 Indices and tables **13**

Contents:

CHAPTER 1

Introduction

Habakkuk is a tool intended for analysis and performance prediction of code fragments (kernels) written in Fortran. Habakkuk is written in Python and makes use of fparser (https://github.com/stfc/fparser), a Fortran parser.

CHAPTER 2

Getting going

2.1 Download

Habakkuk is available from the Python Package Index (pypi). The project itself is hosted on github (https://github.com/arporter/habakkuk).

2.2 Dependencies

Habakkuk is written in Python and so needs Python (either 2.7 or 3.6+) to be installed on the target machine. It also requires the (Python) fparser and six packages. In order to run the test suite you will require py.test.

2.3 Installation

2.3.1 Using pip

The recommended way of installing Habakkuk is to use pip. This will obtain the package from pypi as well as any required dependencies:

```
$ pip install Habakkuk
```

By default, pip attempts to perform a system-wide installation which requires root privileges. Alternatively, a user-local installation may be requested by specifying the *–user* flag:

```
$ pip install --user Habakkuk
```

This will install the package(s) under *${HOME}/.local*. Depending on your linux distribution, you may need to add *${HOME}/.local/bin* to your *PATH* and *${HOME}/.local/lib/pythonX.Y/site-packages/* to your *PYTHONPATH*. (X.Y is the version of python your system is running, e.g. 2.7.)

2.3.2 From tarball

If pip is not available then tarballs of each of the releases of Habakkuk are available on github (https://github.com/arporter/habakkuk/releases). Once the tarball has been downloaded and unpacked, change to the resulting habakkuk directory and do:

```
$ python setup.py install
```

If you do not have root access then, as with using pip (above), you can specify the prefix for the install path like so:

```
$ python setup.py install --prefix ${HOME}/.local
```

2.4 Running

Habakkuk is run from the command line. The *-h/–help* flag will produce a list of the various available options:

```
$ habakkuk -h

Usage: habakkuk [options] <Fortran file(s)>

Options:
  -h, --help            show this help message and exit
  --no-prune            Do not attempt to prune duplicate operations from the
                        graph
  --no-fma              Do not attempt to generate fused multiply-add
                        operations
  --rm-scalar-tmps      Remove scalar temporaries from the DAG
  --show-weights        Display node weights in the DAG
  --unroll=UNROLL_FACTOR
                        No. of times to unroll a loop. (Applied to every loop
                        that is encountered.)

  Fortran code options:
    Specify information about Fortran codes.

    --mode=MODE         Specify Fortran code mode. Default: auto.
```

Habakkuk analyses Fortran source files, provided as arguments on the command line, e.g.:

```
$ habakkuk my_fortran_file1.f90 my_fortran_file2.F90
```

If all is well, you should see output similar to the following:

```
$ habakkuk tra_adv.F90
Habakkuk processing file 'tra_adv.F90'
Wrote DAG to tra_adv_loop1.gv
Stats for DAG tra_adv_loop1:
   0 addition operators.
   0 subtraction operators.
   0 multiplication operators.
   1 division operators.
   1 FLOPs in total.
   8 array references.
   8 distinct cache-line references.
   Naive FLOPs/byte = 0.016
```

(continues on next page)

(continued from previous page)

```
  Whole DAG in serial:
    Sum of cost of all nodes = 8 (cycles)
    1 FLOPs in 8 cycles => 0.1250*CLOCK_SPEED FLOPS
    Associated mem bandwidth = 8.00*CLOCK_SPEED bytes/s
  Everything in parallel to Critical path:
    Critical path contains 4 nodes, 1 FLOPs and is 8 cycles long
    FLOPS (ignoring memory accesses) = 0.1250*CLOCK_SPEED
    Associated mem bandwidth = 8.00*CLOCK_SPEED bytes/s
Schedule contains 1 steps:
            Execution Port
      0    1    2    3    4    5
0   /    None None None None None (cost = 8)
  Estimate using computed schedule:
    Cost of schedule as a whole = 8 cycles
    FLOPS from schedule (ignoring memory accesses) = 0.1250*CLOCK_SPEED
    Associated mem bandwidth = 8.00*CLOCK_SPEED bytes/s
  Estimate using perfect schedule:
    Cost if all ops on different execution ports are perfectly overlapped = 8 cycles
  e.g. at 3.85 GHz, these different estimates give (GFLOPS):
  No ILP   |  Computed Schedule   |  Perfect Schedule | Critical path
   0.48    |          0.48        |        0.48       |    0.48
 with associated BW of 30.80,30.80,30.80,30.80 GB/s
```

2.5 Testing

The Habakkuk source contains a test-suite written to use py.test. In order to run it you will need to obtain the Habakkuk source - either by downloading a tarball of one of the [releases](https://github.com/arporter/habakkuk/releases) or by cloning the git repository. Assuming you have Habakkuk and py.test installed you can then do:

```
$ cd habakkuk/src/habakkuk/tests
$ py.test
```

CHAPTER 3

Using Habakkuk

3.1 Performance Predictions

Note: To be written.

3.2 DAG Output

Note: To be written.

CHAPTER 4

CPU Configuration

In order to predict performance on any given CPU, Habakkuk must be configured with certain parameters that describe that CPU. Currently, only parameters for the Intel Ivy Bridge micro-architecture are supplied. In this section we describe the various parameters that Habakkuk uses.

4.1 Instruction Cost

The number of cycles that a given arithmetic operation requires is fundamental to constructing a performance estimate of a kernel. For floating-point operations on Intel architectures, Agner Fog provides comprehensive data. However, since Habakkuk processes high-level Fortran code, it must also account for Fortran intrinsic operations such as SQRT and COS. The cost of these operations has been estimated by running simple micro-benchmarks (dl_microbench) on the target CPU. All of this data is provided to Habakkuk in a dictionary, *OPERATORS*. The keys in this dictionary are the Fortran symbols for the various arithmetic operations (e.g. "*" for multiplication) and the names of Fortran intrinsics in uppercase (e.g. "SIN"). Each entry in the dictionary is itself a dictionary with keys "cost" and "flops". The entries for these keys are integers giving the number of cycles and number of floating-point operations, respectively, associated with the operation.

4.2 Mapping of Instructions to Execution Ports

In the Intel Ivy Bridge micro-architecture, instructions are despatched to different "execution ports", depending on their type. For example, floating-point multiplication and division is handled by port 0 while addition and subtraction are sent to port 1. Provided there are no dependencies between them, operations that are mapped to different ports may be executed in parallel.

The number of different execution ports is specified in *NUM_EXECUTION_PORTS* and the mapping of instructions to them is supplied to Habakkuk as a dictionary, *CPU_EXECUTION_PORTS*. The keys in this dictionary are the same as those in *OPERATORS* and the associated entries are simply the integer index of the corresponding execution port.

4.3 Instruction Overlapping

In addition to the instruction-level parallelism offered by having instructions despatched to different execution ports, the differing cycle count of the various f.p. operations provides further scope for overlapping them. In particular, f.p. division on Ivy Bridge takes eight times longer than e.g. multiplication and addition/subtraction. Even though multiplication and division operations are mapped to the same execution port, the results of micro-benchmarks show that the hardware is able to perform several multiplications while a single division is in progress. Similarly, multiple addition/subtraction operations may be performed on port 1 while a single division is in progress on port 0.

Whether or not this overlapping is supported by the CPU is configured by setting *SUPPORTS_DIV_MUL_OVERLAP* to *True* or *False*, as appropriate. If overlapping is supported then further data is required on the degree of overlapping permitted and the cost in cycles.

MAX_DIV_OVERLAP is a dictionary with keys "*" and "+". The entries are the maximum number of each of those operations that may be overlapped with a single division.

div_overlap_mul_cost(overlaps) is a routine that returns the cost of a division (in cycles) as a function of the number of other operations that it is overlapped with. *overlaps* is a dictionary holding the number of each type of operation ("*" and "+") that has been overlapped with the division. (Subtraction operations are counted as addition operations here.)

4.4 Support for Fused Multiply-Add Operations

Some micro-architectures (but not Intel Ivy Bridge) have support for fused multiply-add instructions. i.e. `a*x + b` can be performed in a single operation. *SUPPORTS_FMA* must be set to *True* or *False*, as appropriate.

4.5 Cache-line Size

The number of bytes contained in a single cache line must be supplied in `CACHE_LINE_BYTES`. This is used when estimating memory-bandwidth requirements.

4.6 Clock Speed

A representative clock-speed for the CPU being modelled is supplied in *EXAMPLE_CLOCK_GHZ*. This is used to generate concrete performance figures. Note that on CPU cores with frequency-stepping and turbo boost enabled, there is no single clock speed!

CHAPTER 5

Indices and tables

- genindex
- modindex
- search