BONSAI Documentation

Release 17.5-alpha

Matthew Bachstein, Jakub Kurzak, Piotr Luszczek, Mike Tsai

Oct 04, 2017

Contents

1 Contents **3**
- 1.1 Installation . . . 3
- 1.2 Tutorial . . . 3
- 1.3 LANAI . . . 6
- 1.4 Release Notes . . . 8

BONSAI is Python based infrastructure for the autotuning of GPGPU computational kernels. BONSAI is based on *LANAI*, a Python based way of specifying the search space of parameters upon which to tune the kernel. These 'iterators' are used to create the valid parameter configurations. BONSAI will then take your templated C/C++ code and benchtest it with those configurations.

CHAPTER 1

Contents

Installation

Installation is straight forward, unzip the tarball from http://icl.cs.utk.edu/bonsai/software/index.html and be sure that Python 2.7 is selected as your Python interpreter.

Tutorial

This is a tutorial on how to get started with BONSAI. First, BONSAI requires:

- Python 2.7+ (Python 3 is not supported)
- GCC 4.4 (or similar) or later + make

There are three primary steps in an auto-tuning workflow,

- Generating and pruning the Search Space
- Tuning and selecting the proper kernel configuration
- Analyzing the generated profiles to inform the next tuning run

Variables in the Makefile are defined in make.inc for easy customization.

There are two top level make targets, *make space* and *make tune*. The first generates the search space, and the second performs a tuning sweep over the search space with the provided kernel.

Working examples are provided in the *examples* folder, one CPU and one GPU, of varying complexity. The CPU example is simple with one iterator for tutorial purposes. The GPU example requires CUDA and the CUDA toolkit be installed. The GPU example is rather complex, showing how BONSAI can be incorporated into complex kernels.

Space Generation and Pruning

The first step in the autotuning workflow is to generate the search space for the parameters. (make space) This is done using a *LANAI* iterator file. For this example, we will use the following (admittedly toy) iterator.

```
from lanai import *

max_sz = 32
# -----------------
dim_m = range(1,max_sz)
dim_n = range(1,max_sz)

@iterator
def blk_m(dim_m):
    return range(dim_m, max_sz, dim_m)

@iterator
def blk_n(dim_n):
    return range(dim_n, max_sz, dim_n)
#------------------
@condition
def only_even(dim_m, dim_n):
    return( (dim_m % 2) == 1 or (dim_n % 2) == 1)
```

dim_m and *dim_n* can be viewed as the total size of a matrix panel, and *blk_m* / *blk_n* the size of the blocks within the panel. In LANAI terms, *m* and *n* are called 'expression' or 'global' iterators; *blk_m* and *blk_n* are called 'deferred' or 'local' iterators since they depend on a global value. We also have one condition for these iterators, in that we only want to consider even numbers for the sizes. An important note, conditions are written as **what should fail**, not what should be true. (see *LANAI* for more)

After specifying an iterator file (ITER in make.inc), BONSAI will take it and generate a C source file (by default sharing the same name as the iterator file) that defines the search space. BONSAI then will then compile the generator file using the specified options, and output to the specified CSV file.

Integrating with the Runtime

The runtime is expecting two routines, the kernel to be tested and a driver for the kernel that sets the parameters. The routines can be in the same file. As an example, look at the dgemm C code below (this is from examples/gemm_cpu).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>

#ifndef FLOAT_TYPE
#define FLOAT_TYPE double
#endif

int main(){
  // Here, BLOCK_SIZE is an iterator that is defined within
  // the LANAI iterator file

  struct timeval start;
  struct timeval end;
  unsigned long diff;
  srand((unsigned) time(NULL));
```

```
  int i, j, k, i_b, j_b, k_b;
  int n = 1024;
  int blk = BLOCK_SIZE;

  FLOAT_TYPE* A = (FLOAT_TYPE*) malloc(sizeof(FLOAT_TYPE)*n*n);
  FLOAT_TYPE* B = (FLOAT_TYPE*) malloc(sizeof(FLOAT_TYPE)*n*n);
  FLOAT_TYPE* C = (FLOAT_TYPE*) malloc(sizeof(FLOAT_TYPE)*n*n);

  for (i=0; i < n*n; i++)
    A[i] = ((FLOAT_TYPE)rand() / RAND_MAX - 0.5)*2;
  for (i=0; i < n*n; i++)
    B[i] = ((FLOAT_TYPE)rand() / RAND_MAX - 0.5)*2;
  for (i=0; i < n*n; i++)
    C[i] = ((FLOAT_TYPE)rand() / RAND_MAX - 0.5)*2;

  gettimeofday(&start, NULL);

  for (i_b=0; i_b < n/blk; i_b++)
    for (j_b=0; j_b < n/blk; j_b++)
      for (k_b=0; k_b < n/blk; k_b++)
        for (i=i_b*blk; i < (i_b+1)*blk; i++)
          for (j=j_b*blk; j < (j_b+1)*blk; j++)
            for (k=k_b*blk; k < (k_b+1)*blk; k++)
              C[i + j*n] += A[i + k*n] * B[k + j*n];

  gettimeofday(&end, NULL);

  diff = 1000000 * (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec);
  printf("Time: %ld us\n", diff);

  free(A);
  free(B);
  free(C);

  return 0;
}
```

With the corresponding iterator file

```
from lanai import *


block_size = range(2, 66)


@condition
def only_even(block_size):
  return block_size % 2 == 1
```

Note how we have a correspondence between the iterator *block_size* and the assignment *int blk = BLOCK_SIZE*. The runtime is expecting the iterator names to be used like *#define* macros in the kernel/driver files. The runtime will construct the correct compiler statement for each configuration using the *-D* flag to define the macro values.

LANAI

LANguage for Autotuning Infrastructure

LANAI is a python based method for specifing the search space and pruning constraints for autotuning programs. This is the basis for the entire BONSAI infrastructure.

General Structure

The general structure for a LANAI specification file is

```
#imports
#iterator constants
#iterator specifications
#constraint constants
#constraint specifications
```

LANAI allows for outside imports in its iterator files for abstraction purposes. For example, say a file that contains various CUDA constants for different compute capabilities and a file with constants for a particular card. The top of the iterator file would then be

```
from lanai import *
from cuda_constants import *
from Telsa_K40c import *
```

Following that may come any other constants that we may wish to specify.

Iterators

LANAI supports 2 basic iterator types (expression and deferred) each corresponding to a specific scope (global and local).

Global/Expression Iterators

Global/expression iterators take the form of:

```
dim_x = range(1, max_dim_x+1)
```

using the standard Python `range` syntax. The 3 argument version of `range` is also supported. These are the most basic form of iterator expressions.

Local/Deferred Iterators

Local/deferred iterators are allowed to take other iterators and/or other constants as arguments. Local iterators must also be decorated with `@iterator` before the iterator definition. This takes the basic form of

```
@iterator
def blk_x(dim_x):
    return range(dim_x, max_dim_x+1, dim_x)
```

Deferred iterators are also allowed to contain more complex flow control statements as well. For example

```
@iterator
def vec_mul(dim_vec):
    if dim_vec == 1:
        return range(0, 1)
    else:
        return range(0, 2)
```

`dim_vec` is also an iterator, so `vec_mul` can take on different values depending on what value `dim_vec` is currently holding. It is probably worth mentioning that the iterator definition of `dim_vec` *must* come before any other definition where it is used as an argument.

Conditions

LANAI condition statements prune the search space generated by the iterators. Each condition statement must be decorated with a `@condition` before the definition as so:

```
@condition
def over_max_threads(threads_per_block):
    return threads_per_block > max_threads_per_block
```

Conditions are always written as a boolean statement that returns 'true' when the condition fails. Not following this can lead to unwanted behavior. As an example, look at the following two iterators and pruning condition, where we want both R1 and R2 to be greater than 0.

```
R1 = range(3)
R2 = range(3)

@condition
def greater_than_zero():
    return R1 > 0 and R2 > 0
```

Without the condition, we get the search space:

```
R1, R2
0 , 0
0 , 1
0 , 2
1 , 0
1 , 1
1 , 2
2 , 0
2 , 1
2 , 2
```

Adding the condition prunes the search space to:

```
R1, R2
0 , 0
0 , 1
0 , 2
1 , 0
2 , 0
```

This seems to be a bug. The problem lies with how the condition statement was defined. We specified the condition as a statement that we want to always hold, not the condition that we **want to fail**. To fix this, take the logical complement of the condition:

```
@condition
def greater_than_zero():
    return R1 <= 0 or R2 <= 0
```

This prunes the search to the space we expected:

```
R1, R2
1 , 1
1 , 2
2 , 1
2 , 2
```

Oddities

To declare a consant value, it still must be written as an iterator with size one, e.g.

```
tex_a = range(1,2) #
```

Release Notes

Release 17.10

This is a pre-alpha release. The features pertaining to the generation and pruning of the search space are useable. The parallel tuning runtime is still in early stages of development. A simple serial runtime has been included as a prototype, to demonstrate the method of integrating a kernel with the framework.