Nitro Autotuning System Tutorial

School of Computing, University of Utah

March 17, 2015

1 Introduction

Nitro is an automatic performance tuning system for GPU applications. It dynamically selects the optimal algorithmic variant to execute based on characteristics of the underlying architecture and input data set.

2 Installation

Nitro can be obtained from GitHub: `http://github.com/nitro-tuner/nitro`. Before trying to install Nitro, however, make sure the following software packages are installed:

- NVIDIA CUDA 6.5 or above
- Python 2.7 or above
- LibSVM 3.12 or above: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
- CUSP library (for the SpMV example): `https://github.com/cusplibrary/cusplibrary` [1]

The following system paths must be set. Here, `$NITRO_ROOT` refers to the directory where Nitro is installed:

- Add `$NITRO_ROOT/src` to `$PYTHONPATH`

[1] Tested with commit `93f6`

- Set `$LIBSVM_PATH` to directory where libSVM is installed [2]
- Add the libSVM installation directory to `$PATH` and `$LD_LIBRARY_PATH`
- Set `$CUSP_PATH` to directory where CUSP library is installed
- Set `$SHADER_MODEL` to the SM architecture of the GPU you have (eg.: 20 for Fermi-class GPUs)

3 General System Architecture

Nitro consists of two parts:

- A Header-Only C++ Template Library
- A Python-based Autotuning Interface

An existing application can be integrated with Nitro by (1) specifying code variants, features and constraints *within* the application code using Nitro's C++ library, (2) building a customized autotuner for the application using Nitro's Python interface, and (3) specifying training data and (optionally) testing data.

The Python interface allows users to customize various aspects of the tuning process such as toggling constraints, specifying which classifier to use, classifier properties etc.

3.1 Tuning Process

Given a set of code/algorithmic variants (hereafter referred to as code variants), and a set of features associated with each, Nitro selects the optimal (w.r.t some metric such as performance) variant for the given input and architecture. To accomplish this, Nitro builds a *classification model* from user-provided *training data*. This classification model is then used to predict the code variant to execute. Each feature is a user-defined function that describes a characteristic of the input. For example, the number of rows in the input matrix for the SpMV sample.

[2] Make sure you do a `make` and `make lib` in this directory

4 Tuning Applications Using Nitro

A Sparse-Matrix Vector Multiplication (SpMV) example application is included with this release to demonstrate how applications can be tuned using Nitro. It can be found at `$NITRO_ROOT/examples/spmv`. The following subsections walk through the process of integrating Nitro into the C++ part of the application and then customizing the tuning process using the Python-based tuning interface.

4.1 Setting up the Context

Before specifying variants and features in the target C++ application, an object of type `nitro::context` must first be created. This object sets up initial state and manages the coordination among the various code variants executing within the application.

4.2 Specifying Variants

4.2.1 Creating the `code_variant` object

Code variants are represented by the `nitro::code_variant` class. It takes in three template parameters:

- **Tuning Policy**: For a code variant named `variantX`, a class of the same name is generated automatically in the `nitro::tuning_policies` namespace in `nitro_config.h`. The class is generated in accordance with what's specified in the `tune.py` script (described in Section 4.3).
- **Argument Type Tuple**: The types of the arguments of the variant must be wrapped in a `thrust::tuple` type and specified here. For example, if the variant takes argument types `(float*, float*)`, then `thrust::tuple<float*, float*>` must be specified.
- **Device**: This can be either `nitro::sm` or `nitro::x86`, corresponding to a variant executing on a GPU and X86 CPU respectively.

Thus, an example instantiation of the `code_variant` object would be:

```
using namespace nitro;
```

```
typedef thrust::tuple<float*, float*> ArgTuple;
code_variant<tuning_policies::test_variant,
             ArgTuple, sm> v(cx, "test_variant");
```

This creates an object `v` of type `nitro::code_variant` that uses the tuning policy specified in `nitro::tuning_policies::test_variant`, takes in argument types `float*` and `float*`, and runs on a GPU.

Note that the string specified as the second argument to the constructor must exactly match the name of the variant specified in the `tune.py` script.

4.2.2 Defining Code Variants

A code variant in Nitro can take any types of arguments but must return a string representing the performance of the variant on the given input. There are two ways in which a variant can be defined:

- **Create a Functor**: The functor must derive from `nitro::variant_type<T1, T2, ...>` where each T_i is an argument type of the variant. Further, it must override the function call operator taking the correct argument types and returning a double. In our running example, this would be `double operator()(float*, float*)` { ... }.

- **Wrap Function Pointer**: In case the variant is defined as a function that accepts the relevant argument types and returns a string, then Nitro provides a function pointer wrapper class `wrap_variant` that can create the correct functor type. For our example, this would be (assuming `variant1` is one of the variants defined as a function): `nitro::wrap_variant<float*, float*> _variant1(variant1);`.

4.2.3 Adding Variants

Variants defined using the notation explained in the previous sub-section can be added to a `code_variant` object by using the `add_variant` function. It accepts a pointer to the function object representing the concerned variant and adds it to the internal list of variants maintained by the `code_variant` object.

4.2.4 Executing the Variant

Execution of the variant is accomplished by simply calling the `code_variant` object's function call operator with the required input arguments. Depending on the tuning context, Nitro automatically selects the correct internal variant to execute.

4.2.5 Specifying Input Features

Input features can be specified in a way similar to variants: using either a functor, or using the `wrap_feature` metafunction. They take in the exact same input as the variant will, and must return a real (double) value representing the value of the calculated feature for that input data. Here's an example:

```
double feature1(float* a, float *b) {
  ...
  return foo;
}


...
wrap_feature<float*, float*> _feature1(feature1);
test_variant.add_input_feature(&_feature1);
```

4.2.6 Specifying Constraints

For certain inputs, it's possible that a variant produces wrong results, or takes unacceptably long to execute. To handle such cases, Nitro supports the specification of *constraint functions*. Constraint functions can be added to code variants using the `add_constraint` function which accepts a constraint function and the specific variant for which it is valid. Constraints are automatically evaluated by Nitro and force the computation to revert to the default variant if a constraint fails in the deployed executable. In the SpMV example, the `dia_cutoff` constraint ensures that the DIA variant doesn't execute if the constraint evaluates to `false`.

4.3 Customizing the Autotuner

Tuning parameters may be specified using Nitro's Python interface. This includes specifying training inputs, customizing the machine learning algorithm, etc. For SpMV, the configuration file is `tune.py`. The final call to the `tune()` function begins the autotuning process.

4.4 Running the Autotuner

Running the tuning script using `python tune.py` invokes the autotuner. Training data is automatically collected, and variant selection models are placed in the `models/` folder.

4.5 Using the Tuned Application

Once the tuning process is complete, an adaptive executable named `spmv_tuned` is generated, which automatically queries the constructed model in the `models/` folder at run-time and selects the appropriate variant to execute.