Combining Scripting Environments and Sourcery VSIPL++ for Rapid Prototyping

Stefan Seefeld, Brooks Moses, Don McCoy, Justin Voo
CodeSourcery, Inc.
{stefan, brooks, don, justin}@codesourcery.com

Introduction

A common methodology for developing signal- and image-processing applications is to use a rapid-prototyping environment such as Matlab for algorithm development, and then to reimplement the algorithm with domain-specific tools and languages such as VSIPL++.

Thus, code is typically written (and tested) twice: once in the prototyping environment, and once in the production environment. This process is laborious, error-prone, and thus cost-intensive.

In this poster, we demonstrate some methods by which Sourcery VSIPL++ can be combined with a rapid-prototyping environment to simplify this process and reduce the duplicated effort. Sourcery VSIPL++ provides a portable environment, supporting algorithm and program development on a traditional workstation, with minimal effort required to recompile for embedded platforms. Adding scripting capabilities is a logical next step.

Development Environments

Scripting environments such as Matlab [1] or Python's Scipy [2] are very useful for prototyping numerical applications, due to their fast turn-around between writing and executing. This supports a highly incremental approach, where algorithmic modifications can be made quickly and tested immediately. However, due to the interpreted nature of the environments, performance of such scripts is often limited, and typically the portability to embedded environments is minimal. For example, Matlab is only available for x86 processors.

Meanwhile, library solutions such as Sourcery VSIPL++ [3] provide high performance through the use of traditional compiled languages like C++ in combination with fine-tuned architecture-specific implementations, and are appropriate for deployed systems. The VSIPL++ API supports productivity by allowing the user to express complex operations in compact mathematical notation, which reduces the amount of code that needs to be written, and thus the amount of code that needs to be tested. However, the code still needs to be recompiled after each modification, making such an environment less amenable to rapid prototyping.

In a typical development environment [4], the prototyping environment is used to produce a reference "Gold" standard, which is then fully reimplemented for the production environment, and comparisons of the results from the two implementations are used for testing.

Combining Scripting and Sourcery VSIPL++

Programmer productivity can be improved by integrating these two development environments. In this poster, we illustrate several methods.

First, the scripting environment can be adapted to provide bindings directly to the Sourcery VSIPL++ library, using an API that resembles VSIPL++'s C++ API. By scripting directly using the same API that is used in the production environment, the cost of translating the code from the scripting implementation to the production implementation is significantly reduced, and the risk of introducing bugs in the translation is minimized.

In addition, Sourcery VSIPL++ provides a profiling functionality that can be used to characterize the performance of VSIPL++ applications. This can be accessed through the scripting interface to measure the runtime of the VSIPL++ functions, and thereby provide estimates of the time budget at the prototyping stage.

The VSIPL++ API also provides functionality that is typically not present in scripting environments – for example, VSIPL++ provides a method for distributing data blocks across multiple compute nodes using user-supplied data maps. Likewise, Sourcery VSIPL++ provides extensions to the VSIPL++ API for executing user-written coprocessor kernels. Including bindings for these within the scripting environment allows for this functionality to be developed and tested within that part of the prototyping process.

Finally, a common data model between the scripting language and VSIPL++ library allows for a hybrid programming approach where the boundary between the two environments is more fluid, and parts of the application can still be in the prototype stage while being linked to other parts that are already stable and compiled.

Example Use Cases

As examples of how these combinations may be used, we show a couple of use-cases.

Use-Case 1: Scripting distributed algorithms.

- Define distributed maps and corresponding distributed data blocks in the scripting environment.
- Perform computations with VSIPL++ operations on these data blocks, and use the profiling features to measure performance and record data movement.

- Adjust the data distribution and iterate to optimize performance
- Translate the implementation directly into a C++ VSIPL++ program for production use.

Use Case 2: Prototyping CUDA user kernels.

- Write CUDA kernel code. Compile it "just in time" within the scripting session.
- Execute the kernel code through the Sourcery VSIPL++ user-kernel layer, test for correctness, and measure performance.
- Iterate to optimize performance.
- Save the kernel code to be compiled directly into the production Sourcery VSIPL++ application.

References

[1] The MathWorks. Matlab. [online] http://mathworks.com/.

[2] ScyPy.org. ScyPy. [online] http://www.scipy.org/.

[3] CodeSourcery, Inc. Sourcery VSIPL++. [online] http://www.codesourcery.com/vsiplplusplus.

[4] Julia Mullen and Matthew Alexander. "From Algorithm to Real-Time Embedded Software: A Controlled, Stepwise Software Development Methodology." HPMCP Users Group Conference, 2008.