A MATLAB-to-Target Development Workflow using Sourcery VSIPL++

Stefan Seefeld, Faheem Sheikh, Brooks Moses

{stefan_seefeld, faheem_sheikh, brooks_moses}@mentor.com

Mentor Graphics, Inc.

***Abstract* - A hybrid MATLAB/C++ programming model for high performance embedded computing is presented. It is shown how the use of a common data model and API can help not only to speed up the development process, but also to keep the original MATLAB model in sync with the evolving C++ code, and thus allowing it to remain a gold standard for the project as it evolves.**

***Keywords:* multi-language, scripting, VSIPL++, prototyping, signal- and image-processing**

1. Introduction

Prototyping is a common modeling technique used in the early stages of the development cycle to validate the feasibility of algorithmic/design choices. Implementors of signal and image processing, radar, control and communication systems frequently rely on interactive scripting environments like MATLAB[1] or Python[2] to prototype their designs. Once validated, the prototype is converted to a development environment suitable for a target device. Typically, this requires a change in the memory model and the data organization, and / or the adaptation to a new set of APIs. In addition to being an error-prone process, this transition often loses a lot of valuable information, as the original prototype can not be maintained as a reference, i.e., a gold standard, outside the development environment. Since it is difficult to maintain and test two separate versions of the system, the prototype version typically becomes stale and isn't kept current with the evolving development version. Keeping both the prototype and development versions current would be desirable. Even more beneficial would be to keep the prototype involved as a reference for verifying the development version.

Sourcery VSIPL++[3] is a high-level C++ implementation of the VSIPL++ specifications. It provides a portable high performance computing library that supports a wide range of platforms, including x86 and Power Architecture CPUs, NVIDIA CUDA GPUs, and Cell Broadband Engine processors. In addition to optimized implementations of signal and image processing algorithms, it offers a simple development workflow, making it easy to develop code on a workstation and then recompile with minimal effort for embedded platforms. Development productivity can be further improved by closely integrating this workflow with the interactive scripting environments that were used for prototyping.

In this paper we present a workflow that bridges the above gap by integrating the two environments closely, allowing a more seamless transition from early prototyping to development to deployment. It even becomes possible to write hybrid programs that combine code written in MATLAB with code written in C++.

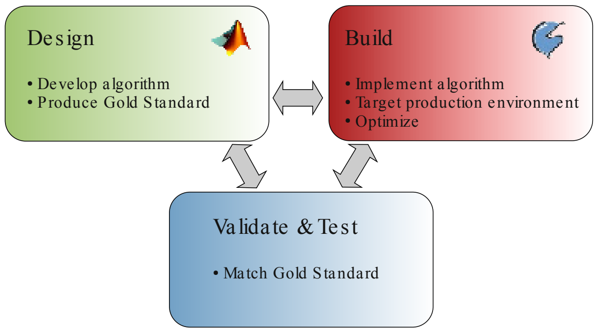

Figure 1. Typical development workflow

The MATLAB VSIPL Toolbox is an extension of the MATLAB scripting environment. The toolbox allows binding directly to the Sourcery VSIPL++ library, using an API that resembles the VSIPL++ interface. This feature results in the prototype developed with the toolbox looking very similar to 'pure' MATLAB scripts as well as the implemented code. Consequently, the cost (effort) of translating the code from the scripting to production code is significantly reduced.

Due to MATLAB's popularity and ubiquitous presence in the scientific community, there have been many attempts in the past to assist developers through the use of automatic code generation. For instance, the MATLAB compiler for heterogeneous computing system [5] generates C code based on user-supplied directives. It also uses algebra schemes to overcome type ambiguity. Other efforts to create an integrated development environment with MATLAB include interfacing of a TI DSP simulator to user scripts via MEX files [6]. In this environment users can produce DSP software directly from prototypes. Another similar approach but in a different domain has been adopted in [7] where an

AUTOSAR-compliant automatic code generation tool, called Processor Expert (Freescale micro-controllers) is integrated into Simulink. Programming of a NVIDIA GPU within the MATLAB environment using MEX files has been explored in [8] and has shown to increase the simulation performance. Finally a multi-language programming model has been presented in [9] where user applications written in C++ are linked with a third party library (FFTW) and MATLAB external interfaces for SONAR signal processing applications.

The above-mentioned works that attempt to bridge the prototyping and implementation gap either complicate the problem (, like writing a new compiler) or offer solutions that are only practical for domain-specific scenarios. Our motivation for this work comes from hybrid programming models such as Boost.Python [4].This particular model supports combining C++ with Python in a single development and deployment environment.

Along similar lines we present an object-oriented interface to MATLAB scripting using the same data model and API as VSIPL++. This not only makes the transition from MATLAB to C++ seamless and natural, it also allows one to combine the two environments into a single application, which has useful applications from testing to visualization and profiling.

Section 2 introduces a simple pulse compression algorithm that is used as reference algorithm throughout the paper. It also shows how the initial "raw" MATLAB code is converted to use the MATLAB VSIPL Toolbox. Section 3 talks about the process of rewriting the same algorithm in C++ , reusing bits from scripts to help in testing. Section 4 discusses how to bind the MATLAB VSIPL Toolbox API to the implementations from the Sourcery VSIPL++ library. Section 5 then concludes with an outlook into what else may be possible as this methodology is further refined.

2. Prototyping Using MATLAB And The VSIPL Toolbox

2.1. Reference Algorithm: Radar Pulse Compression

Pulse compression is a technique used to reach a compromise between a radar's transmitted signal energy and its range resolution. A moderately long pulse is emitted to keep the signal-to-noise ratio low. Frequency modulation then allows to improve the resolution by overlapping multiple pulses. Pulse compression then consists of correlating the measured signal with the initial signal, which allows to identify objects at a much higher resolution than the pulse length would normally support.

A linear frequency-modulating pulse -- also called a chirp signal -- is often used to achieve these pulse compression objectives. It can be expressed as

$$p(t) = \cos\left(2\pi f_c t + \beta \frac{t^2}{2}\right) \quad 0 < t < T \tag{1}$$

where T is the pulse duration, β is the rate of instantaneous frequency change and f_c is the base frequency of the pulse.

Assuming the above pulse $p(t)$ is reflected by a number of objects at different locations l_i with attenuations κ_i, the measured signal $r(t)$ is a sum of shifted and scaled pulses $p(t)$:

$$r(t) = \sum_i \kappa_i p\left(t - \frac{l_i}{c}\right) + \xi \tag{2}$$

where ξ corresponds to additive noise. To discover the echos (l_i, κ_i), the measured signal $r(t)$ is cross-correlated with the time-reversed original pulse $h(t) = p(-t)$. For efficient computation, a frequency-domain implementation of the correlation is used for this purpose:

$$c(t) = F^{-1}[H(\Omega) R(\Omega)] \tag{3}$$

where $c(t)$ is the pulse-compressed signal, and $R(\Omega)$, and $H(\Omega)$ are the frequency domain representation of $r(t)$ and $h(t)$ respectively.

F^{-1} represents the inverse Fourier transformation. If the sampling frequency is known, the exact location of pulses can be determined using a simple threshold detection scheme as follows:

$$\begin{aligned} d[n] &= 0 \quad c[MT] > \tau \\ d[n] &= 1 \quad \text{otherwise} \end{aligned} \tag{4}$$

Here, τ is the detection threshold obtained by averaging the sampled data.

```
function symbols = \
  pulse_compress(signal, chirp, M)

  filter = flipud(chirp')';
  corr_fft = fft(filter,N).*fft(signal);
  out = abs(ifft(corr_fft));

  %ignore the lag
  filtered = zeros(1,N);
  filtered(1:N-L) = out(L+1:end);

  sampled = filtered(1:M:end);
  symbols = sampled >= mean(sampled);
```

Figure 2. Pulse compression script in MATLAB

Figure 2 shows a code snippet expressing (3) and (4) in a MATLAB script, operating on simulated inputs.

For this work, a MATLAB simulator has been developed that generates test inputs like the ones used in the snippet above and also models the transmitter, channel and receiver of a radar.

The pulse-compression simulator models ten pulse durations in the life span of a particular radar detection session. A test script supplies sample radar echo locations, corresponding attenuation factors for these echoes and a signal-to-noise ratio (SNR) parameter to configure the Additive White Gaussian Noise (AWGN) channel. A second script is responsible for synthesizing input measurements. It produces a reference linear frequency modulated signal with a center frequency of 1MHz, a pulse duration of 1 microsecond and an instantaneous frequency sweep rate of 3.1013. Based on echo locations and attenuation factors, a time-multiplexed signal of echoes (each of which is a shifted and scaled copy of the reference signal) is generated (See Figure 3, top graph). The multiplexed signal is processed through an AWGN channel to produce a noisy signal, where the variance of the noise is configured from the given SNR parameter. The middle graph shows a signal with a SNR of 10dB below the multiplexed signal.

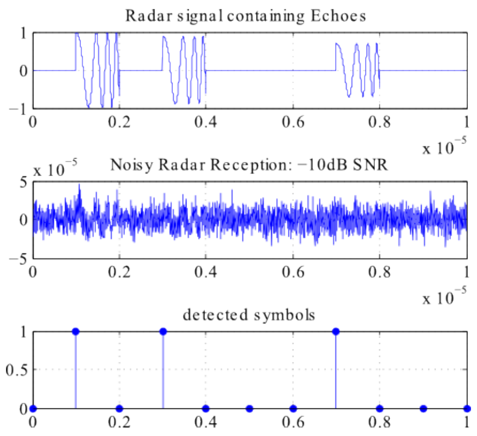

Figure 3. Pulse compression simulation results for -10dB SNR

2.2. The VSIPL Toolbox API

To bring the scripting and implementation environments close together, one really needs to establish a common data model and API that can be used across language boundaries. The VSIPL Toolbox provides an object-oriented API that is directly derived from VSIPL++, and only differs in certain areas to respect MATLAB-specific idioms. For example, MATLAB indices start with 1, while in VSIPL++ indices start with 0, etc.

At the heart of this toolbox are common view types for vectors, matrices, and tensors, which provide the same high-level data model as their VSIPL++ counterparts.

```
classdef Vector
  %  View which appears as a one-dimensional,
  %  modifiable vector.

  properties
    size
    block
    ...
  end

  methods
    % constructors
    function self = Vector(size, type) ...
    ...




    % accessors
    function r = subsref(self, s) ...
    ...
    % support functions
    function r = disp(self) ...
    function r = plot(self) ...
    ...
    % elementwise operations
    function r = plus(self, other) ...
    ...
  end % methods
end % classdef
```

Figure 4. Excerpt from the Vector class of the VSIPL Toolbox

The pulse compression script can thus be rewritten to use vsip.Vector and vsip.Matrix objects instead of builtin MATLAB arrays.

```
function symbols = \
  pulse_compress(signal, chirp, M)

  L = chirp.size;
  N = signal.size;

  filter = vsip.Vector(N, 'double', 0);
  filter(1:L) = chirp(L:-1:1);
  fft = vsip.Fft(N,'double',1);
  inv_fft = vsip.Fft(N,'double',0);
  corr_fft = fft(filter).*fft(signal);
  out = vsip.mag(inv_fft(corr_fft));

  %ignore the lag
  filtered = vsip.Vector(N,'double', 0);
  filtered(1:N-L) = out(L+1:end);

  sampled = filtered(1:M:end);
  symbols = sampled >= vsip.meanval(sampled);
```

Figure 5. Pulse compression script using the VSIPL Toolbox

The complete logic to test this algorithm requires functions to generate the chirp pulse, as well as synthesize the measured signal that would result from a specific set of targets. Using the VSIPL Toolbox, these functions all operate on vsip.Vector and vsip.Matrix arguments. The test application thus takes a set of target locations, together with attenuation coefficients, and returns the detected target locations. The detected target locations are compared to the expected result to validate the algorithm.

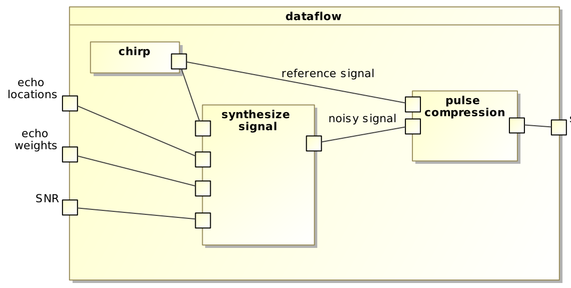

Figure 6. Composite structure of a test application for the pulse compression algorithm

3. Translating The Prototype Into C++

Once the algorithm prototype in MATLAB has stabilized, it is time to translate it into C++. The required effort is considerably reduced by the fact that a common data model and API are being used. Not only is it easier to translate an algorithm, but the similarity down to the syntactic levels really makes this an almost trivial exercise. A lot of statements can be translated almost literally:

```
Vector<float>
pulse_compress(Vector<float> signal,
               Vector<float> chirp, length_type M)
{
  typedef Fft<const_Vector, float,
    complex<float>, 0> fwd_fft_type;
  typedef Fft<const_Vector, complex<float>,
    float, 0> inv_fft_type;

  length_type L = chirp.size();
  length_type N = signal.size();

  fwd_fft_type fft(N, 1.0);
  inv_fft_type inv_fft(N, 1.0 / N);

  Vector<float> filter(N,0.0);
  filter(Domain<1>(0,1,L)) =
    chirp(Domain<1>(L-1,-1,L));

  Vector<float> out =
    mag(inv_fft(fft(filter) * fft(signal)));

  // Ignore the lag
  Vector<float> filtered(N,0.0);
  filtered(Domain<1>(N-L)) =
    out(Domain<1>(L, 1, out.size()-L));

  Vector<float> sampled =
    filtered(Domain<1>(0,M,N/M));

  return sampled >= meanval(sampled);
}
```

Figure 7. Pulse compression using the VSIPL++ API

Once the algorithm itself has been rewritten, the test logic needs to be translated too. It is at this point that the two sides typically start to diverge. Once the algorithm is translated, it is very difficult, if not impossible, to go back and compare the C++ version to the original MATLAB version. Thus, as the C++ code evolves further, the original MATLAB prototype is neglected and becomes obsolete.

Here we follow a different route, to keep the MATLAB code engaged. Using the original MATLAB code as a reference during testing avoids having to reimplement even the testing logic. It also bridges the above gap, encouraging the developer to keep the two in sync even as the algorithm evolves further.

Using a common data model with language bindings for C++ (VSIPL++) as well as MATLAB (the VSIPL Toolbox described here), it becomes possible to transfer objects across language boundaries. For example, the following code embeds a MATLAB session within a C++ application to plot a VSIPL++ vector:

```
    // define a vector...
vsip::Vector<> vector = ...

matlab::Engine engine;
// ...pass it into Matlab...
engine.define("v", vector);
// ...and plot it.
engine.eval("plot(v);");
engine.eval("title('Demo vector plot');");
engine.eval("xlabel('index');");
engine.eval("ylabel('value');");
```

Figure 8. Executing MATLAB from within C++

A similar approach can be used to compare a C++ algorithm with the equivalent gold standard reference implementation from MATLAB, in order to test the C++ implementation. Taking the data flow for the pulse compression prototype (Figure 6), this corresponds to the following test logic:

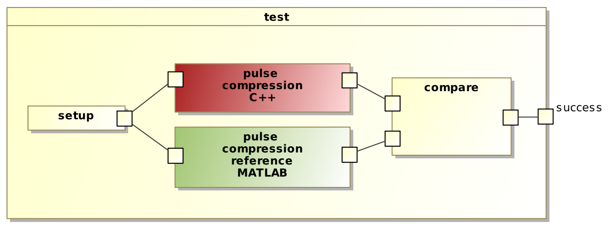

Figure 9. Testing of the new C++ implementation against the gold standard

In fact, even large complex algorithms can be tested or further prototyped using such a hybrid technique, i.e., where C++ and MATLAB building blocks are combined freely.

A slightly different approach is being used in [10], where the gold standard of an algorithm is componentized. Individual components of their Scalable Node Architecture (SNA) platform are prototyped in MATLAB, and then compiled into a shared library using the MATLAB compiler.

Thus for a given component there exist two versions: one implemented using the MATLAB VSIPL Toolbox, the other directly implemented with Sourcery VSIPL++. As both meet the same abstract component requirements, they can be used interchangeably.

4. Integrating MATLAB With C++

By using the above technique it allows one to embed MATLAB scripts into C++ applications. It is also possible to do the inverse, and embed C++ code into MATLAB. Specifically, the VSIPL Toolbox we have been using to prototype VSIPL++ applications is in fact also implemented on top of Sourcery VSIPL++. Vector and Matrix objects are thin wrappers around Sourcery VSIPL++ views and blocks, and all major operations map to implementations from the Sourcery VSIPL++ library.

MATLAB makes this possible by providing a C API to define operations. Figure 10 shows a skeleton of such a "MEX-function" that executes a binary operation implemented in Sourcery VSIPL++:

```
void
mexFunction(int nlhs, mxArray *plhs[],
            int nrhs, mxArray const *prhs[])
{
  if (nrhs != 2)
    mexErrMsgTxt("two arguments expected");

  if (!mxIsClass(prhs[0], "vsip.Vector") ||
      !mxIsClass(prhs[1], "vsip.Vector"))
    mexErrMsgTxt("wrong argument types");

  mxArray *a1 =
    mxGetProperty(prhs[0], 0, "block");
  mxArray *a2 =
    mxGetProperty(prhs[1], 0, "block");
  mxArray *lhs;
  // evaluate binary operation on
  // a1 and a2 into lhs
    ...
  // construct a View wrapping the new array
  mexCallMATLAB(1, plhs, 1, &lhs, "vsip.Vector");
}
```

Figure 10. MEX-function implementing a binary operation

Such a setup provides many advantages. Being able to execute the same code as the development environment significantly reduces risks associated with porting the code from MATLAB to C++. In addition, Sourcery VSIPL++ provides implementations for accelerated platforms (including GPUs), which the MATLAB VSIPL Toolbox can take advantage of.

Sourcery VSIPL++ also provides various profiling and diagnosing capabilities to monitor the execution and performance of applications. Combining these with interactivity offered by a scripting environment, such as MATLAB, can be invaluable, for example when certain algorithmic choices may depend on detailed knowledge of how the code maps to the available hardware.

5. Conclusion And Future Directions

In this paper, we have described the development of a new MATLAB VSIPL Toolbox providing an API modeled after the VSIPL++ specification, implemented with bindings to Sourcery VSIPL++. It is being used successfully in a hybrid environment where the interactive and rapid prototyping capabilities of MATLAB are combined with the performance and portability of Sourcery VSIPL++. The unified modeling environment not only helps to transition the prototype into ready to deploy software, it also allows developers to keep the original MATLAB version alive as a gold standard for validating the target C++ version.

There are a number of directions in which to explore this work further. For example, When the VSIPL++ model for parallelism was originally developed, the same paradigm was used to provide parallel computing capabilities for MATLAB, resulting in the pMATLAB package[11]. The above work on a VSIPL Toolbox provides a unique opportunity to reimplement the pMATLAB package with Sourcery VSIPL++, to integrate parallel compute capabilities into the VSIPL Toolbox.

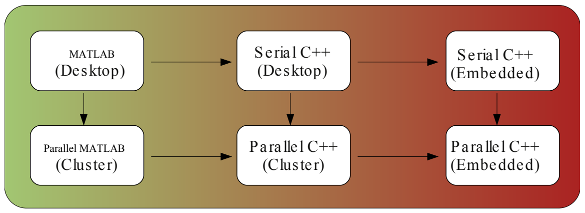

Figure 11. Integrated development workflow from prototyping to deployment, using MATLAB and Sourcery VSIPL++

Another area of possible improvement is that of automated code generation. While the use of a unified data model and API vastly simplifies the task of transitioning from prototype to final code, the process still involves manually rewriting code. It may eventually be possible to automate this transition, and generate optimized C++ code using the Sourcery VSIPL++ API.

References

[1] URL: http://www.mathworks.com.

[2] URL: http://www.python.org.

[3] URL: http://go.mentor.com/vsiplxx.

[4] David Abrahams and Ralf W. Grosse-Kunstleve. *Building Hybrid Systems with Boost.Python*. 2003-03-19. URL: http://www.boostpro.com/writing/bpl.html.

[5] P.Banerjee, N.Shenoy, A.Choudhary, S.Hauck, C.Bachmann, M.Haldar, P.Joisha, A.Jones, A.Kanhare, A.Nayak, S.Periyacheri, M.Walkden, D.Zaretsky, "A MATLAB compiler for distributed heterogeneous, reconfigurable computing systems" *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on* , pp. 39-48, 2000

[6] David P.Magee, "Matlab extensions for the development, testing and verification of real-time DSP software" In *Proceedings of the 42nd annual Design Automation Conference (DAC '05).* ACM, New York, NY, USA, 603-606

[7] R. Bartosinski, Z. Hanzalek, P. Struzka, L. Waszniowski, "Integrated Environment for Embedded Control Systems Design" *IEEE International Parallel & Distributed Processing Symposium*, WPDRTS07, March 26–30, USA, 2007

[8] M. Fatica, W. Jcong, "Accelerating MATLAB with CUDA" , *IEEE HPEC 2007*

[9] I. Aleksi, D. Kraus, Z. Hocenski, "Multi-language programming environment for C++ implementation of SONAR signal processing by linking with MATLAB External Interface and FFTW", *ELMAR, 2011 Proceedings*, pp.195-200, 14-16 Sept. 2011.

[10] DDS vs. DDS4CCM, Teton SNA Core Team, Northrop Grumman

[11] Nadja Travinin, Robert Bond, Jeremy Kepner, Hahn Kim *pMatlab: High productivity, high performance scientific computing*, 2005