Multi-Target C Code Generation from MATLAB®

João Bispo, Luís Reis
Faculty of Engineering (FEUP)
University of Porto, Porto, Portugal
{jbispo, ei09030}@fe.up.pt

João M. P. Cardoso
Faculty of Engineering (FEUP)
University of Porto & INESC-TEC, Porto, Portugal
jmpc@fe.up.pt

Abstract

This paper describes our recent work on MATISSE, a framework for MATLAB to C compilation. We focus on the new optimizations and transformations, as well as on OpenCL generation. MATISSE is controlled with LARA, an aspect-oriented language, able to specify transformations to the input MATLAB code (e.g., insertion of code for variable initialization and for monitoring) and to express information concerning types and shapes of variables. We evaluate the compiler with a set of benchmarks when targeting both an embedded system and a desktop system. The results show that we were able to achieve a speedup up to 1.8× by employing information provided by LARA aspects. We also compare the execution time of the generated C code with the original code running on MATLAB, and we achieve a geometric mean speedup of 19×. The geometric mean speedup reduces to 12× when optimizing the MATLAB code with LARA aspects. Finally, we present a preliminary version of a fully-functioning pragma-based OpenCL generator, built over the MATISSE framework.

Categories and Subject Descriptors D.3.4 **[Programming Languages]**: Processors – Code generation, Compilers, *Optimization*, Retargetable compilers D.2.2 **[Software Engineering]**: Design Tools and Techniques

General Terms Performance, Experimentation, Languages

Keywords MATLAB-to-C, source-to-source compiler, Aspect-oriented programming, LARA, embedded systems, OpenCL

1. Introduction

MATLAB [1] is a *de facto* standard high-level programming language and interactive numerical computing environment in many domains in engineering and science, including embedded computing as it is ubiquitously used by engineers to quickly develop and evaluate their solutions. *MATLAB* is dynamically typed, and relies on interpretation (and/or JIT compilation) as the information about the types and shapes (i.e., number and size of matrix dimensions) of variables is only known at runtime. Due to advances in JIT compilation and the use of pre-compiled libraries for the most intensive functions, the *MATLAB* runtime environment currently exhibits acceptable performance.

ARRAY'14, June 11 2014, Edinburgh, United Kingdom

http://dx.doi.org/10.1145/2627373.2627389

In many embedded system settings, however, the use of a *MATLAB* runtime environment is infeasible, either because it is not available, or due to performance and/or resource constraints. To address this potential shortcoming, a typical solution relies on the development of an implementation in executable code written in an imperative language such as C/C++ once the base or original *MATLAB* code has been validated. This implementation must then in turn be validated against the output of the *MATLAB* code resulting on a lengthy and error prone process that further complicates the overall application development cycle and cost. The existence of two code specifications - the original prototypical *MATLAB* code and the reference C/C++ code - also exacerbates maintenance costs. Another solution is to rely on the automatic translation of MATLAB to the target programming language as provided, e.g., by the MATLAB Coder [2] and the Embedded Coder [3], which translate MATLAB to C code. Besides the inherent advantages, this, however, has typically the disadvantage of the low support to control and guide the code translation. The code generation is typically based on directives (GUI based in the case of the MATLAB Coder) addressing types, shapes, and target. When dealing with the myriad of target architectures and toolchains in embedded systems, this approach presents a low level of flexibility, e.g., as the style of the C code generator might need to be tuned to the toolchain as is the case when targeting C to hardware compilers. Instrumentation and code transformations, as well as an approach to express strategies for code transformations and instrumentation (e.g., by using a DSL) can be very important during the design process and may increase productivity. Furthermore, the target platform may require code generators to specific programming languages as is the case of the generation of OpenCL [4] when dealing with GPGPUs and/or GPU-based FPGA implementations [5].

Our approach relies on a compilation tool, named as MATISSE [6], which generates C code directly from *MATLAB*. Our approach explores the use of Aspect-Oriented Programming (AOP) [7][8] concepts, through the use of the LARA language [9][10] as a vehicle to convey information to the compiler (e.g., types and array shapes) and to express code specialization and code instrumentation strategies. The compiler uses the user-provided information complementing and checking its consistency against the information it can derive from its own analysis. MATISSE is being developed as a modular and flexible compiler framework, which includes custom Intermediate Representations (IRs) for the MATLAB and C code, keeping in mind the generation of C code from a higher-level programming language. In particular, the IR representing the output C code (C-IR) supports matrix types natively, and can be easily extended to support additional types and language constructs (such as the ones needed to generate OpenCL). The end result is a synergy between compiler analysis and the user that allows the compiler to generate very high-quality code from *MATLAB* specifications. It is also possible

to generate different versions of the C code, to better target different embedded systems, platforms, and/or toolchains. In this paper we focus on our recent MATISSE improvements.

The remainder of this paper is organized as follows. Section 2 presents the MATISSE compiler framework. In Section 3 we describe the C-IR, the internal representation we use to generate C code. Section 4 explains how the MATLAB code is transformed into C code and OpenCL with the help of LARA aspects. Section 5 shows some experiments performed using MATISSE. Section 6 describes related work and finally, Section 7 concludes this paper and describes ongoing work.

2. MATISSE Overview

MATISSE consists of a MATLAB-to-C compiler targeting embedded systems, and a LARA-controlled MATLAB *weaver* which allows transformations over MATLAB code. LARA [9] is a domain-specific language inspired by AOP concepts [7] and JavaScript semantics and constructs. LARA uses a declarative semantic to allow programmers to specify strategies for actions over application source code and/or compiler IRs (e.g., instrument code, extract information, explore transformations, apply compiler optimizations). LARA aspects are applied by a target language dependent *weaver*, such as the *weaver* for the MATLAB language integrated in MATISSE.

Figure 1 presents the overall flow of MATISSE. The input MATLAB files are parsed and translated to an abstract syntax tree (AST) based MATLAB IR. The IR is the input to *MWeaver*, which uses LARA aspects to modify and add information (e.g., variable types and shapes) to the MATLAB IR. MATISSE is being developed in a way to make easy the integration of code generators. At the moment code generators for MATLAB and C are already fully working and an OpenCL code generator is under development.

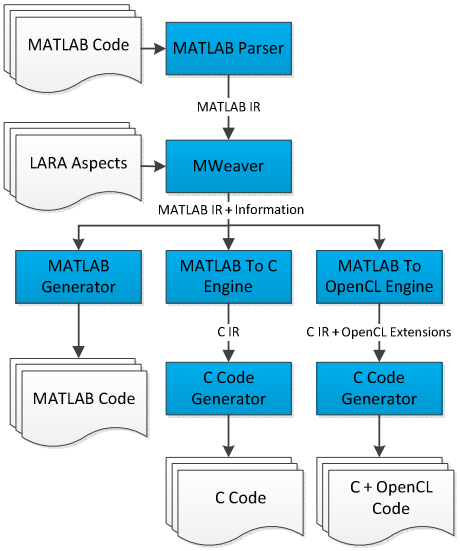

Figure 1. Overview of the MATISSE compiler framework.

MATISSE generates *MATLAB* code for validation, testing, monitoring, and specialization, and C code to be used by third-party design-flows targeting software/hardware systems. MATISSE is able to generate customized C code for a particular target without modifying the original MATLAB code. LARA aspects enable MATISSE to have fine-grained control over the generation of C code, and also allow the generation of different implementations from the same source code. A common example includes the restructuring of source code and the use of statically declared array variables to be compliant with the requirements of most hardware compilers.

MATISSE can be used as a source-to-source code transformation and instrumentation tool allowing developers to quickly and reliably generate reference C implementations, a key step in the deployment of embedded system applications. The transformation stage of the compiler performs weaving actions such as insertion of code, definitions of types and shapes, and code specialization based on default values.

3. A High-Level C-IR

Given the differences between MATLAB and C, we decided to use a C specific AST-based IR to represent the C code (C-IR). This simplifies the generation of C code and allows us to separate concerns related to C generation (e.g., include files, variable declarations) that are not considered in the MATLAB IR. This option also makes the MATLAB IR clean and independent of the specificities of the code generation to be applied.

3.1 Variable Types

The C-IR uses the *VariableType* interface to represent all the information needed about a data type (e.g., code needed to declare the variable, how to convert to another type, how to perform an addition between variables of this type). *VariableType* allows a seamless integration of several types in a modular way. For instance, we were able to add support for OpenCL native types through an implementation of *VariableType*, without changing the C-IR library. Figure 2 shows a subset of the hierarchy that starts with the *VariableType* interface. To add a new type, one needs to create a class that implements the *VariableType* interface. *Scalar* represents a single value, and *Matrix* represents a multi-dimensional array of elements of type *Scalar*. These two classes add contract methods that provide information specific to these types, such as number of bits, maximum and minimum values or signed/unsigned, for *Scalar*, or matrix shape and element type, for *Matrix*. *CNative* represents the native types of C (e.g., *int*, *float*, in the case of *Numeric*, and, e.g., *int32_t*, *uint8_t* in the case of *StdInt*).

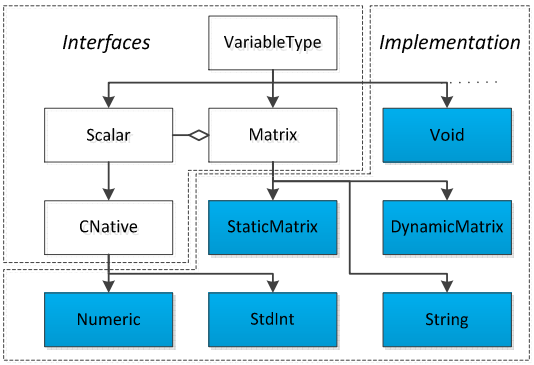

Figure 2. Subset of the *VariableType* hierarchy in the C-IR.

3.2 Matrix Types in C-IR

Currently, the C-IR has two Matrix implementations, *StaticMatrix*, which uses statically allocated C arrays (e.g., *int A[3]*), and *DynamicMatrix*, which uses a C structure to represent dynamically allocated arrays of a given element type. For dynamic arrays, MATISSE allows the option to use for each array access a func-

tion that performs array bounds checking (for debugging), or to use inlined code without checks (for performance).

MATISSE uses linearization of multi-dimensional arrays, whereby an element of the multi-dimensional array is accessed through a single pointer variable. Linearization has several benefits. Firstly, simple element-wise operations are compactly executed in a single loop rather than using a loop nested structure. Secondly, the single allocation of storage and corresponding boundary values also enables one out-of-bound condition check per array access rather than having to perform a verification per array dimension. Lastly, it also provides other advantages regarding the size of the code generated. When multi-dimensional arrays are generated without linearization, code may need to consider various pointers (when dimensions are not known at compilation time) in order to allocate the space needed for all dimensions.

4. Transforming MATLAB to C

MATLAB is a dynamically typed language. This is in stark contrast with C, which is statically typed and needs the types of all variables to be declared. When converting *MATLAB* to efficient C code, it is necessary to statically determine the types used. This can be a challenge, as the same *MATLAB* function can have very different C implementations, depending on the types of its variables. Fortunately, defining the types of the arguments/parameters of a *MATLAB* function is often enough to infer the remaining types of variables in the function. In the case of static arrays, usually it is also possible to determine the shape of the arrays at compile time (i.e., how many dimensions the array has, as well as the size of each dimension).

MATISSE uses an interface that represents all possible C implementations for a given named *MATLAB* function and selects one of them based on the received arguments and types. The specialization occurs at the level of the function call, instead of the function. This mechanism allows *MATISSE* to generate multiple versions of highly specialized C functions. For instance, if there are multiple function calls to a *MATLAB* function that accepts a matrix type as input, and it receives different *StaticMatrix* types (a type whose shape is known at compile time), it is possible to generate a function specialized to each specific shape.

4.1 Type and Shape Inference Analysis

The type inference uses a simple data-flow analysis approach [11], where type information is derived by processing each *MATLAB* statement, complemented with information provided by LARA aspects. In most cases only the types of function parameters are required for the compiler to achieve an efficient type inference. There are two general situations where type-inference is applied in MATISSE: 1) during a function call; and 2) during assignments. C-IR nodes representing function calls contain information about the signature of the functions (i.e., their input and output types), and are specialized according to the inputs of the call. The output types are usually determined by the object that creates the function call, and this means that each function can define its own rules regarding the type-inference of its output types. For the assignments, the variables on the left hand are usually bound to the type inferred in the right hand. Note that when in conflict, types of variables defined in a LARA aspect always override the inference mechanism. Besides the type, variables can carry other information, such as values and the shapes of matrices. This information is propagated and in many cases updated and extended by information determined in other assignments.

Consider the code in Figure 3. In the first line, the function *size* returns an array with the shape of the given variable. If the variable *H* is of the type *StaticMatrix*, it always contains information about its shape, and the values of *h1* and *h2* are known at compile time. If *H* is a *DynamicMatrix*, the values of *h1* or *h2* might not be known. *Size* is a supported MATLAB function, and MATISSE creates a *FunctionCall* node in C-IR, specialized to the type of *H*. C-IR nodes always carry information about the types they return. Thus, MATISSE assigns the types defined by the *FunctionInstance* of *size* to *h1* and *h2* (in this case, both are of type *int*). However, if the types of *h1* and *h2* are defined in a LARA aspect, MATISSE uses those types. If the shape of *H* is known at compile time, the values of *h1* and *h2* are also known.

In the second line in Figure 3, for operations such as + there is a default rule that chooses the first maximal fit type between the operands. The maximal fit is automatically determined using information obtained from the *Scalar* interface (i.e., the minimum and maximum possible values of the type). Although MATISSE infers types for constants (e.g., 1 is inferred as an integer, 1.0 as a double), for this rule the types of constants are not taken into account when inferring the type of the output (unless all operands are constants). In this case, the result of the addition will have the same type as *h1*. The operator / has special inference rules. By default, the output is assigned to a real type, to avoid losing precision. The flexibility of MATISSE allows, on the one hand, general rules available to all functions, and on the other hand, custom rules for a particular function. As previously referred, if a LARA aspect defines the type of any of the variables (i.e., *H*, *h1*, *h2* or *offset1*) the inference mechanism for that particular type is augmented with that information. For instance, if the type of the variable *offset1* is set to float, the operations on the right-hand of the assignment will also consider float as output. This way it is possible to address the limitations of a static type/shape inference analysis, and the usual cases where the user needs to force and evaluate data types not derived by type inference.

```
[h1,h2] = size(H);
offset1 = (h1+1)/2;
```

Figure 3. Code snippet from *conv2*.

4.2 C Code Generation Example

We now illustrate the application of the proposed approach to a *MATLAB* function implementing an FIR (Finite Impulse Response) filter (see Figure 4). This function takes as input two arrays, *vector_1d* and *coef*, and outputs an array named *output*. In the absence of information about the shape of input arrays, MATISSE generates C code that uses the *DynamicMatrix* type for the parameter as well as for the function's return value, which represents a structure with dynamically allocated memory. The definition of types and shapes for the function parameters in a LARA aspect (see Figure 5) enables the use of the *StaticMatrix* type.

The *MATLAB* code in Figure 4 uses the MATLAB built-in function *sum*. Currently, MATISSE supports a general version of *sum*, by using a description in MATLAB translated to C by MATISSE. With information about the types of the arguments of *sum*, we can apply transformations over the code. Figure 6 shows a possible transformation, which is applied as follows. The function *sum* is called with an expression as argument. By analysing the MATLAB-IR corresponding to the expression, MATISSE determines that it is composed only by element-wise operations (i.e., .*). Furthermore, the operands in the expression return one-dimension arrays (*coef* is used directly and we know the shape, and *vector_1d* is accessed linearly, returning a one-dimension array). The output of *sum* will be a scalar, and can be replaced with an accumulator variable (*sum_acc*) and a *for* loop. If there

were no ranges, we could iterate the loop over the size of any of the input matrices. As there is a range, we use it to control the *for* loop: (`i:-1:i-NTAPS+1`). If there were any other ranges, MATISSE would calculate the indexes outside the *for* and would access them with an induction variable. The matrix *coef* is used as parameter, so the compiler uses an induction variable (*matrix_i*) to access the elements of *coef* and incremented after each loop iteration. This is an example of a transformation applied with the current framework, at the MATLAB-IR level. In this case, the output of the transformation is a modified MATLAB-IR, but we can perform the same transformation to generate directly C-IR. This transformation in particular will slow down the code if executed in MATLAB, but can significantly help to generate efficient C code (see Figure 7).

```
function output=fir_1d(vector_1d, coef)
  NTAPS = numel(coef);
  N = numel(vector_1d);
  output = zeros(1, N);
  for i = NTAPS:1:N
    output(i) = sum(vector_1d(i:-1:i-NTAPS+1)
                .* coef);
  end
end
```

Figure 4. MATLAB *fir* code example.

```
aspectdef firSingle
  var typeDef = { // Type definition
    vector_1d    : "single[1][1024]",
    coefficients : "single[1][32]",
    output       : "single" };
 // Matrix sizes
 var matrixSizes = {output  : "1, N" };
 // Define types
 call defineTypes("fir_1d", typeDef);
 // Define matrix sizes
 call initMatrixes("fir_1d", matrixSizes);
 // Inline all functions MATISSE supports
 call matisseInline("true");
 // Define the matrix implementation as static
 call matisseMatrixImpl("static")
end
```

Figure 5. LARA aspect that defines the types for the *fir* example.

```
function output=fir_1d(vector_1d, coef)
  NTAPS = numel(coef);
  N = numel(vector_1d); output = zeros(1, N);
  for i = NTAPS:1:N
    sum_acc = 0; matrix_i = 1;
    for sum_i = i:-1:i-NTAPS+1
      sum_acc = sum_acc + vector_1d(sum_i)
                .* coef(matrix_i);
      matrix_i = matrix_i + 1;
    end
    output(i) = sum_acc;
  end
end
```

Figure 6. MATLAB *fir* after transformation.

4.3 OpenCL Generation

We are developing an OpenCL generator which uses OpenACC [12] based pragmas in the *MATLAB* code to decide what/how to parallelize. Sections to parallelize begin with the "parallel loop" pragma. The "end" pragma indicates the end of the code section the pragma applies to. Parallel loop sections can be parameterized with *copyin*, *copyout* and/or *reduce*. Two types of reductions are supported: sums and products.

We present the OpenCL engine to illustrate the overall flexibility of the framework, and of the C-IR. In a few months' work, a single Master student was able to develop an OpenCL engine capable of generating the OpenCL code for sections in a relevant subset of MATLAB annotated with pragmas, plus the necessary wrapper classes that perform the communication between the C and the OpenCL code (both codes are represented using the C-IR). The OpenCL engine replaces pragma occurrences with a call to a custom function. It then translates that function to OpenCL.

The OpenCL generation is still in a very early phase and the currently generated code is not optimized.

```
float* fir(float input[1024], float coef[32],
float output[1024])
{
 int NTAPS; int N; int i; float sum_acc;
 float sum_acc; int matrix_i; int sum_i;
 NTAPS = 32; N = 1024;
 zeros_f1x1024(output);
 for(i = NTAPS; i<=N; i = i+1){
   sum_acc = 0.0f; matrix_i = 1;
   for(sum_i = i; sum_i>=(i-NTAPS)+1; sum_i--){
     sum_acc = sum_acc+(vector_1d[sum_i-1] *
               coef[matrix_i-1]);
     matrix_i = matrix_i+1;
   }
   output[i-1] = sum_acc;
 }
 return output;
}
```

Figure 7. C code with static matrices for the *fir* function generated by MATISSE with information about shapes.

5. Experimental Results

We carried out a series of experiments to evaluate the impact of the information introduced by aspects on the performance of the generated C code, when executed on an embedded platform. We also compared the execution time of the generated code against the original MATLAB code, when using a desktop PC. Finally, we show some preliminary results obtained by an early version of the OpenCL generator.

5.1 Methodology

We use MATISSE to automatically derive C code corresponding to kernels written in *MATLAB*. We then compare the execution time after compiling the resulting code to two architectures: 1) A desktop PC with a 2.93GHz Core 2 Duo processor and Windows 7 32-bit, 3GB of RAM and an nVidia Quadro NVS-290; 2) A BeagleBoard-XM revB running Ubuntu 12.10 32bit, with a 1GHz ARM Cortex-A8 and 512MB of RAM.

We use as benchmarks a set of kernels we consider relevant for embedded systems. We include *subband* and *grid_it*, two critical functions from the *3D Path Planning* and the *MPEG audio encoder* applications [10]. In addition, we include an application to perform correlation using FFTs and with 3D matrices as input. This application named *cfd*, uses forward and inverse 2D FFTs provided by a *MATLAB* function able to perform N-dimensional FFTs (identified as *fft2d*), and a dot product between 3D matrices (identified as *cpx*). Table 1 contains all the benchmarks used.

For both architectures, the generated C code was compiled with gcc-4.6, using flag –O2 and the OpenCL code was compiled with the AMD APP SDK v2.9. Besides the code of the functions, MATISSE also generates main functions for testing purposes, when specifying a .M or .MAT file with the values of the input arguments of the function to test. All benchmarks output correct

results when compared with MATLAB original output, up to an error of 10E-6.

Table 1. Benchmark characteristics.

Benchmark	Input Sizes	MATLAB LoCs
cfd	256 × 256 × 3	50
conv	96 × 11	73
cpx	512 × 512 × 10	23
dilate	2048 × 2048 , 2	17
fft2d	256 × 256	124
fir1d	1M × 32	14
grid_it	32 × 64 × 16	38
latnrm	32K × 8	29
subband	128 × 64ki	35

5.2 Results

We consider two options for generating C code: aspects (*minimal*/*optimized*) and function *inlining* (enabled/disabled). *Minimal* aspect refers to the minimum information we have to provide to generate C code that outputs correct results (sometimes we have to define the type of the output, or of some intermediate variables), while *optimized* is an aspect that has been tailored to provide specialized code.

Figure 8 shows the impact on performance by each option. Function inlining provides the highest speedups, on average a speedup of 1.6× for the tested codes (with speedups up to 2.2× and 2.6×, for *grid_*it and *conv*, respectively). This was expected as inlining removes the overhead of calling a function and enables further optimizations. In the case of the inlining of get/set functions, we are trading array-bound checking for performance. Specialization achieved a more modest effect, on average a speedup of 1.2×, with a maximum of 1.8× for *subband*. When specializing with aspects, we are replacing certain types with potentially less expensive types (doubles with floats, floats with integers), and reducing the overhead of type casting. In architectures more sensitive to data types a larger impact is expected (e.g., when double precision is supported by software).

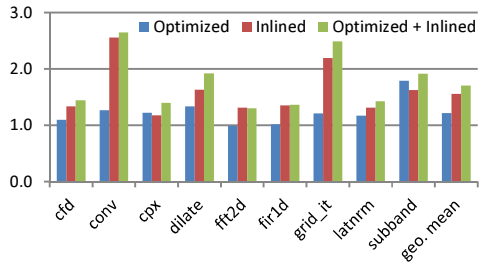

Figure 8. Speedups on the BeagleBoard of the C code when considering three options.

Figure 9 presents the speedups of the C code generated by MATISSE when compared over the execution of the original *MATLAB* code when using MATLAB 2012b, on the Core 2 Duo. Significant speedups were achieved for most cases, ranging from 5× to 9×. We measured a slowdown of 0.6× for *fft2d*. The slowdown is related to missing opportunities for optimizing the code of the function. We also achieved very high speedups. E.g., for *cpx*, the main reason is the pre-allocation of matrices inserted by the LARA aspect. If we compare to the execution time in MATLAB after the code is transformed according to LARA specifications, the speedup reduces from 105× to 5× (the geometric mean reduces from 19× to 12×). For *latnrm*, we consider that the MATLAB code does not use appropriate idiomatic constructs, and that slowdowns the execution in MATLAB. In the case of *fir1d* and *subband*, we are already using MATLAB built-in functions (e.g., *sum*), and the impact of pre-allocation does not completely explains the speedups (*fir1d* reduces to 63×; *subband* increases to 33×). The speedups come from the transformation presented in Section 4.2. We did an experiment where we implemented our own *MATLAB* version of *sum*, and used it to generate C code instead of doing the transformation. In this case, the *subband* speedup was reduced to 2× (MATISSE could not generate C code using the custom *sum* for the other benchmarks where *sum* is used, such as *fir1d*).

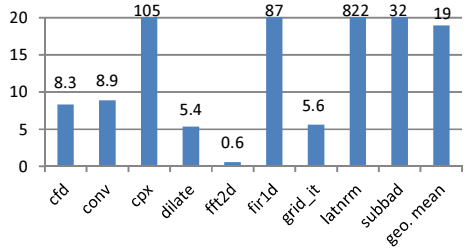

Figure 9. Speedups when comparing the execution of MATLAB code with the C code generated by MATISSE.

Table 2 shows the speedups achieved by the code generated with the OpenCL engine, when compared with the execution of the original MATLAB code, running on MATLAB, and the C version generated by MATISSE without OpenCL directives. From the set of benchmarks, we chose those whose loops could be parallelized and currently supported by the OpenCL generator. The OpenACC based pragmas were manually added to the MATLAB code. The generated OpenCL code run in the nVidia Quadro NVS-290 of the desktop PC.

We achieved a speedup of 1.37× on *dilate* over C-only code, after increasing the size of the inputs (from a 512×512 image to 1024×1024). For *cpx* and *subband*, the OpenCL implementation could not beat the pure C code currently generated by MATISSE. These results were expected as our current OpenCL generator is not able to take fully advantage of the target GPUs.

Table 2. Speedup of C+OpenCL compared with execution in MATLAB and execution of C-only MATISSE on Core 2 Duo.

Benchmark	MATLAB	C-only MATISSE
dilate 1024 × 1024	17.5	1.37
cpx	18.8	0.19
subband	1.9	0.05

6. Related Work

Given the importance of *MATLAB* there have been research efforts to improve the execution of JIT *MATLAB* compilers. A recent example is the compiler presented in [13] which performs function specialization based on the runtime knowledge of the types of the arguments of the functions. Given the widespread of the use of MATLAB to develop embedded systems and the hardware constraints of such systems that precludes the use of a runtime MATLAB environment, an important aspect is the automatic translation of *MATLAB* programs into equivalent C code.

DeRose and Padua developed the FALCON environment [14] that translates *MATLAB* to FORTRAN90 code. They leverage an aggressive use of static and type inference for base types (doubles and complex) as well as shape (or rank) of the matrices. Other researchers have explored the reuse of storage for array variables

across a *MATLAB* code thus reducing the memory footprint of the corresponding C reference code [15]. Joisha et al [16] focused on type and shape inference techniques. Researchers have also relied on a mix of type inference approaches and user's provided information. For instance, [17][18] use annotations to specify data types and shapes and simple type inference analysis and target VHDL code for hardware synthesis onto FPGAs. We specifically note that the focus of our approach is mostly on embedded implementations of the *MATLAB* programs. In this context, an efficient translation to an implementation language (mainly C) is needed. One of the possibilities is to consider a subset of *MATLAB* allowing feasible and efficient static compilation. Examples using such a subset are the Matlab Coder [2] and the Embedded Coder [3].

The popularity of the *MATLAB* language is also reflected in the similar languages that have been proposed. Examples of those languages are *Scilab* [19] and *Octave* [20]. A *Scilab* to C translator [21], named *Sci2C*, has been developed. *Sci2C* focus entirely on embedded systems, and is completely dependent on annotations embedded in the *Scilab* code to specify data sizes and precisions. Our compiler distinguishes from *Sci2C* as it is able to generate C code without polluting the original code. Furthermore, *Sci2C* requires that the size of arrays is fixed and statically known while our compiler also produces C code when those sizes are unknown. The use of user-specified rules and strategies for code transformations has been used to optimize Octave programs [22]. with loop vectorization and partial evaluation of types and values.

In this work we describe a mechanism for conveying information about types and shape/rank similar in spirit with the notion of Aspects [7]. Previous work has proposed aspect-oriented extensions to *MATLAB* and an aspect-oriented code transformation language for *MATLAB* [23]. Other authors have explored aspect-oriented approaches for *MATLAB* [24], but do not use aspects to specify complementary information that can be used by compilers to produce more efficient implementations.

7. Conclusion

This paper presented the current status of MATISSE, a compiler infrastructure for MATLAB. MATISSE relies on LARA aspects for specifying data types, shapes, and code instrumentation and specialization, and on the C-IR for type inference and C code generation. We presented the general flow of the tool, and described the possible transformations that can be applied and the optimizations performed by the compiler. The experiments reveal promising performance results, achieving a geometric mean speedup of 12× over execution in *MATLAB* when considering 9 benchmarks. Additionally, we described our first steps on OpenCL generation from *MATLAB*. Our OpenCL generator takes advantage of OpenACC-based directives to decide about the parallelization and about the MATLAB code sections to be mapped to the accelerator. Ongoing work is focused on further optimizing the C generator and on evaluating and optimizing the OpenCL generator.

Acknowledgments

This work was partially supported by *Fundação para a Ciência e a Tecnologia* (FCT) under FEDER/ON2 and FCT project NORTE-07-124-FEDER-000062.

References

[1] MATLAB – the Language of Technical Computing, http://www.mathworks.com/products/matlab

[2] MATLAB Coder: Generate C and C++ code from MATLAB code, © 2012 The MathWorks, Inc.

[3] Embedded Coder: Generate C and C++ code optimized for embedded systems, © 2014 The MathWorks, Inc.

[4] The OpenCL Specification, Version: 1.0, Doc. Rev.: 48, Khronos OpenCL Working Group, Editor: Aaftab Munshi, Last Rev. Date: 10/6/09.

[5] T. S. Czajkowski, et al., "From opencl to high-performance hardware on FPGAs," *22nd Int'l Conf. on Field Progr. Logic and Applications (FPL'12)*, Oslo, Norway, Aug. 29-31, 2012, pp. 531-534.

[6] J. Bispo, et al., "The MATISSE MATLAB Compiler - A MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SystEms," in *IEEE Int'l Conf. on Industrial Informatics (INDIN'13)*, Bochum, Germany, 29-31 July 2013, pp. 602-608.

[7] G. Kiczales, et al., "Aspect-Oriented Programming," In *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, Springer-Verlag, LNCS 1241, June 1997, pp. 220-242..

[8] J.D. Gradecki, and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, John Wiley & Sons, Inc., NY, USA, 2003.

[9] J.M.P. Cardoso, et al., "LARA: An Aspect-Oriented Programming Language for Embedded Systems," in Proc. Int. Conf. on Aspect-Oriented Software Development (AOSD'12), Potsdam, Germany, March 25-30, 2012, pp. 179-190.

[10] J.M.P. Cardoso, P. Diniz, J.G. Coutinho, and Z. Petrov (eds.), Compilation and Synthesis for Embedded Reconfigurable Systems, Springer, May 2013.

[11] A. Aho, J. Ullman, M. Lam, and R. Sethi, Compilers: Principles, Techniques and Tools, Addison Wesley, 2006.

[12] The OpenACC™ Application Program Interface, August 2013. Version: 2.0a, © 2011-2013 OpenACC-Standard.org.

[13] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge, "Optimizing MATLAB through Just-In-Time Specialization," in Int. Conf. on Compiler Construction (CC'10), March 2010, pp. 46–65.

[14] L. De Rose, and D. Padua, "Techniques for the Translation of MATLAB programs into Fortran 90," in ACM Trans. Program. Lang. Syst., 21, 2 (Mar. 1999), pp. 286–23.

[15] P. Joisha, and P. Banerjee, "Static array storage optimization in MATLAB", in *Proc. ACM Conf. on Prog. Language Design and Implementation (PLDI'03)*, June 9-11, 2003, San Diego, CA, USA, pp. 258-268.

[16] P. Joisha, and P. Banerjee, "An algebraic array shape inference system for MATLAB," in ACM TOPLAS, 2006; 28(5), pp. 848–907.

[17] A. Navak, M. Haldar, A. Choudhary, and P. Banerjee, "Parallelization of MATLAB Applications for a Multi-FPGA System", in *Proc. 9th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'01)*, Rohnert Park, CA, USA, May, 2001, pp. 1-9.

[18] P. Banerjee, at al., "Automatic Conversion of Floating Point MATLAB Programs", in Proc. 11th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'03), Napa, CA, USA, 2003.

[19] Scilab, http://www.scilab.org/

[20] The Octave Home Page. http://www.gnu.org/software/octave/

[21] Scilab 2 C - Translate Scilab code into C code, http://forge.scilab.org/index.php/p/scilab2c/

[22] K. Olmos, and E. Visser, "Turning dynamic typing into static typing by program specialization in a compiler front-end for Octave," in Proc. 3rd IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM'03), 26-27 Sept. 2003, pp. 141-150.

[23] J.M.P. Cardoso, et al., "A Domain-Specific Aspect Language for Transforming MATLAB Programs," in Domain-Specific Aspect Language Workshop (DSAL'2010), part of AOSD'2010, March 15-19, 2010, Rennes & Saint Malo, France.

[24] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren, "AspectMatlab: An Aspect-Oriented Scientific Programming Language", in Proc. Aspect Oriented Software Development Conference (AOSD), March 2010, ACM, NY, USA, pp. 181-192.