Techniques for Efficient MATLAB-to-C Compilation

João Bispo
Departamento de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto, Porto, Portugal
INESC-TEC, Porto Portugal
jbispo@fe.up.pt

Luís Reis
Departamento de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto, Porto, Portugal
INESC-TEC, Porto Portugal
ei09030@fe.up.pt

João M. P. Cardoso
Departamento de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto, Porto, Portugal
INESC-TEC, Porto Portugal
jmpc@fe.up.pt

Abstract

MATLAB to C translation is foreseen to raise the overall abstraction level when mapping computations to embedded systems (possibly consisting of software and hardware components), and thus for increasing productivity and for providing an automated model-driven design-flow. This paper describes recent work developed in the context of MATISSE, a MATLAB to C compiler targeting embedded systems. We introduce several techniques to allow the efficient generation of C code, such as weak types, primitives and matrix views. We evaluate the compiler with a set of 9 publicly available benchmarks, targeting both embedded systems and a desktop system. We compare the execution time of the generated C code with the original code running on MATLAB, achieving a geometric mean speedup of 8.1×, and qualitatively compare our results with the performance of related approaches. The use of the new techniques allowed the compiler to achieve performance improvements of 46% on average.

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 Algorithms, Performance, Languages.

Keywords MATLAB; source-to-source compilers; embedded systems

1. Introduction

MATLAB [1] is a *de facto* standard matrix-oriented high-level dynamic programming language and interactive numerical computing environment in many domains in engineering and science, including embedded systems, as it is ubiquitously used by engineers to quickly develop and evaluate their solutions. In many embedded system settings, however, the use of a MATLAB runtime is infeasible, either because it is not available, or due to performance and/or resource constraints.

To address this problem, a solution is to rely on the automatic translation of MATLAB to the target programming language as provided, e.g., by the MATLAB Coder [2], which translates MATLAB to C code. Despite the inherent advantages of this approach, it typically has the disadvantage of 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 [2]) addressing types, shapes and target microprocessor. 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 generated might need to be tuned to the toolchain as is the case when targeting C to hardware compilers (e.g., Vivado HLS, Catapult-C). A MATLAB compiler framework aware of both the target computing platform and the toolchain when generating C code can be very important during the development process and may increase productivity and significantly reduce maintenance costs.

Our work is focused on addressing these aspects by providing a multitarget/multichain MATLAB compiler framework. Our approach relies on MATISSE, a compilation tool firstly presented in [3], which generates C code directly from MATLAB. We explore the use of Aspect-Oriented Programming (AOP) [4][5] concepts, using the LARA language [6][7] as a vehicle to convey information to the compiler. For instance, in previous work we have used LARA to help MATLAB compilation (e.g., to define the types and shapes of variables [3]). In this work we show that specializing C code for a given target can have a significant impact, and we see LARA as a way to communicate target-specific information without modifying the original code.

This paper describes the application of a number of techniques, such as weak-types, primitives and views with pointers in order to improve the performance achieved with the C code generated by MATISSE.

The remainder of this paper is organized as follows. Section 2 briefly presents the MATISSE compiler and describes some of the techniques and optimizations applied during MATLAB-to-C translation. Section 3 shows the experimental results, Section 4 presents the related work, and finally, Section 5 concludes the paper.

2. The MATISSE Compiler

MATISSE is a MATLAB compiler framework, which consists of a MATLAB-to-C compiler targeting embedded systems, and a LARA-controlled MATLAB *weaver* which allows transformations over MATLAB code and communication of user information. It can be used as a source-to-source code transformation and instrumentation tool (e.g., one can use LARA to insert and transform arbitrary MATLAB code) allowing developers to quickly and reliably generate reference C implementations, a key step in the development of embedded applications. The transformation stage of the

ARRAY'15, June 13-14 2015, Portland, OR, USA

ACM 978-1-4503-3584-3/15/06 $15.00

compiler performs weaving actions such as insertion of code, definitions of types and shapes, and code specialization based on default values. The knowledge regarding data types and shapes, provided by LARA aspects, allows MATISSE to generate highly customized C code that conforms to the requirements of a specific target.

2.1 Weak Types

On our initial experiments, we observed that the performance of the generated code could be significantly improved if we could reduce the number of cast operations. For instance, consider the MATLAB expressions in Figure 1(a) and suppose that variable *s* has been previously inferred as a single. When inferring the type of constant *c* we need some kind of default type. In the case of MATLAB, it uses double as the type of all numeric constants (MATISSE distinguishes between integer and real constants, and uses user-defined defaults). According to MATISSE semantics, the result of an addition between a single and a double should return a double. Considering the type of *c* is a double, we obtain the code `r = c + (double) s;` where one cast is introduced. Note that if we generate the C code `c + s`, it still has an implicit cast operation introduced by the compiler.

In the previous case, it will be beneficial if the constant has the same type of *s*. According to the type of *s*, the compiler should generate the code in Figure 1(b) or in Figure 1(c). We introduce the concept of *weak type* to refer to types that can be changed by type inference to lower or higher precision according to the needs. In Figure 1(a), by marking the type of the constant *c* as a weak double, the type inference system will determine that it is better to be a single (Figure 1(c)) or a double (Figure 1(b)), and back-propagate the type. In our current approach the compiler coalesces the different types that might be inferred for each variable into a single type (e.g., single and weak double into single). Weak types are being used to help the compiler do type coalescing. Generally, the types of constants in MATISSE are considered to be weak-types (this includes the types of functions that returns constants, such as *zeros* or *ones*).

```
c = 1.5;
r = c + s;
```
(a)

```
double c, r, s;

c = 1.5;
r = c + s;
```
(b)

```
single c, r, s;

c = 1.5f;
r = c + s;
```
(c)

Figure 1. Example to explain weak types: (a) MATLAB code; (b) C code resultant when *s* was inferred as a double; (c) C code resultant when *s* was inferred as a single.

However, weak-types should not always be ignored. For instance, consider that variable *s* was inferred as an integer. If *c*, a constant of a floating-point type, is ignored, variable *r* will be inferred as an integer, which probably is not what is intended. Although a weak type does not provide guarantees about the type, it does have some information about what is expected of the type, and when doing type inference that information should be taken into account. For instance, in this case the type of *c* has the property *Real*.

Figure 2 shows an algorithm for choosing which types should be considered for type inference for each expression. It starts with a list containing all the types of the expression (e.g., the input types of the addition, plus possibly a suggestion for the output type) and then splits the types into weak types and non-weak types. If one of the lists is empty, then all types are considered for type-inference. Otherwise, the algorithm collects the properties for all non-weak types. For each weak type, it then checks if it has properties that should be considered, in comparison with the properties of the non-weak types; if the answer is affirmative, the weak type is used for type inference. MATISSE currently uses two properties, *Real* and *DynamicallyAllocated*, and we are currently working on the property *Range*. The same type can have more than one property, and the type inference is enough flexible to deal with other properties that might be added.

```
// Input: list of types from an expression
List exprTypes
// Output: list of types to be used in type in-
ference for the expression
List inferenceTypes
// Select non-weak types
List nonWeakTypes = exprTypes.NonWeakTypes
// Select weak types
List weakTypes = exprTypes.WeakTypes
// When there is no mix of weak and non-weak
if nonWeakTypes.isEmpty || weakTypes.isEmpty
  inferenceTypes.add(exprTypes)
  return
// Properties of non-weak types
Map nonWeakPr = getProp(nonWeakTypes)
// All non-weak types are used in inference
inferenceTypes.add(nonWeakTypes)

for each weakType in weakTypes
  Map weakTypePr = getProp(weakType)
  if considerWeakType(weakTypePr, nonWeakPr)
    inferenceTypes.add(weakType)
```

Figure 2. Algorithm to choose which types to use in type inference.

2.2 MATISSE Primitives

When generating code for a given core functionality set, such as some built-in and toolbox functions, MATISSE uses templates written in C or in MATLAB. However, sometimes it is not possible to write code that will be translated efficiently to C using pure MATLAB. Consider that we want to initialize a matrix using information from other matrix (in MATLAB we can do it using the code in Figure 3 (a)). Also, before they are read (e.g., output of an element-wise mapping operation – see Section 2.4). In this case, one can just allocate the memory and avoid initialization. Furthermore, there is a call to the function `size` (which creates a matrix) that could be avoided, since we can directly use the data in the matrix that is given as input.

To enable the generation of more efficient code from MATLAB templates, MATISSE internally uses *primitive functions* that can be called from MATLAB code. Primitive functions are MATLAB functions that MATISSE understands and usually translates to more low-level C constructions.

For instance, assuming we are using dynamic matrices, the MATLAB code in Figure 3 (a) is translated to the C code in Figure 3 (b). Note that it allocates a new array for size, and calls a function that initializes the values to zero. If instead we use a strategy that maps the function `zeros` to the MATISSE primitive `matisse_new_array_from_matrix` (Figure 3 (c)) the C code generated will use a C function to allocate memory to a dynamic matrix (Figure 3 (d)). Also, it will directly read the shape information of matrix A, instead of allocation a new matrix with the shape of A.

Examples of other primitives are *matisse_change_shape*, which changes the shape of a matrix directly, instead of creating a new matrix with the new shape, and *matisse_to_real*, which casts an element to the default real type defined in MATISSE. In addition

there are primitives for debugging and exploration. For instance, *matisse_probe* accepts an arbitrary MATLAB expression and statically reports the output type inferred for that expression.

MATISSE primitives are extensively used in the internal MATLAB templates that implement certain functions, and can also be mapped with LARA strategies and woven by MATISSE before generating the C code (i.e., it is possible to use them directly in any .M file, for instance during transformations with LARA aspects). MATISSE also provides a compatibility library with MATLAB implementations of the primitives, so that MATLAB code which uses primitives can still run in MATLAB (for instance, to test the outputs).

(a)	`C = zeros(size(A));`
(b)	`zeros_double(size_d(A, &temp_m0), C);`
(c)	`C = matisse_new_array_from_matrix(A);`
(d)	`new_array(A->shape, A->dims, C);`

Figure 3. Example of a MATISSE primitive function: (a) original MATLAB code; (b) C code generated; (c) MATLAB code representing the mapping of function zeros to the MATISSE primitive *matisse_new_array*; (d) C code generated using the MATISSE primitive.

2.3 Matrix Views

A *view* is a section of a matrix, which is accessed using the colon operator (e.g., `A(N:M)`). Such accesses are very common in MATLAB, and the most straightforward way to implement them is to create a temporary matrix with a copy of the values in the range specified by the operator. In the other hand, if the range has a step of 1 we can avoid copying the values and just use a pointer to the beginning of the range (i.e., a *matrix view*).

However, it is not always possible to use matrix views, for instance, when the potential view is on the right hand of an assignment, over a local variable, and the recipient is an output of the function (e.g., `output_var = local_var(N:M)`). The problem with this case is that when using a view with pointers, `output_var` will point to the values in `local_var`. However, after the function ends, `local_var` is freed and `output_var` will point to nothing. Although currently MATISSE detects this case and uses a view by copy, we are considering the use of a reference-counting mechanism to avoid copies in cases like this.

2.4 Transforming Element-Wise Mapping Operations

Certain MATLAB operators and functions (e.g., addition, subtraction, square root) perform element-wise mapping operations. These operations output a matrix with the same size as the input matrix or matrices, and each element of the output depends on the element of the input matrices that is on the corresponding position. In the presence of chained-operations, allocating a new matrix for each operation is inefficient. Thus, if MATISSE identifies a chain of function calls as being mapping operations, it rewrites them to *for* loops (see Figure 4). The *for* loop is more efficient to implement in C and it also exposes parallelism.

Note that some functions are mapping operations depending on the types of input. For instance, the multiplication operator (*) is element-wise if one of the operands is a scalar, but not if both are matrices.

2.5 Algebraic Analysis for Ranges Determination

Figure 5 shows a segment of code which uses arithmetic expressions to determine a range of values for matrix *A*. To be able to generate code using static memory allocation, we need to know the size of all matrices at compile time, which in the case of matrix B, implies knowing the number of elements in the range for matrix *A*.

The number of elements in a range of values is given by the equation `end - start + 1`, which for the range in matrix *A* corresponds to the expression `(N×M×i) - (N×M×(i-1) + 1) + 1`. MATISSE generates the expression and then applies an algebraic solver (it currently uses the library Symja [8]) to calculate the number of elements in the range (`N×M`, in this case). If *N* and *M* are statically known, MATISSE then substitutes the expressions to constants, and is able to generate a static version of the code.

(a)	`R = sqrt((A .* A) + (B .* B));`
(b)	`sqrt_e((add_e(((mult_e(A, A, temp_m0))), ((mult_e(B, B, temp_m1))), temp_m2)), R);`

(c)	`for(i = 0; i < numel_A; i++){`
	`  R[i] = sqrt(A[i]*A[i] + B[i]*B[i]);`
	`}`

(d)	`sqrt_e((add_e(((mult_e(A, A, &temp_m0))), ((mult_e(B, B, &temp_m1))), &temp_m2)), R);`

(e)	`new_array_helper_f(A->shape, A->dims, R);`
	`for(i = 0; i < A->length; i++){`
	`  (*R)->data[i] = sqrt(`
	`  A->data[i]*A->data[i] +`
	`  B->data[i]*B->data[i]);`
	`}`

Figure 4. Element-wise example: (a) a MATLAB statement of element-wise operations; (b) the equivalent C with static allocation when element-wise transformation is disabled and (c) enabled; (d) the equivalent C with dynamic allocation when element-wise transformation is disabled and (e) enabled.

```
B = A(N*M*(i-1)+1:N*M*i);
```

Figure 5. Example of code that uses expressions to determine ranges.

2.6 Using Third Party Libraries

Using third party libraries is important for generating high-performance code. For instance, the benchmark *closure*, which spends most of its execution time in matrix multiplication operations, can run orders of magnitude slower if instead of BLAS [9] we use a simple matrix multiplication implementation. It is also important to control when libraries and which ones should be used or not. For small matrix sizes the use of BLAS can be less efficient than a simple matrix multiplication version. MATISSE addresses these aspects using LARA strategies and including support to interface to third-party libraries (at the moment the support allows the use of BLAS). Further improvements will focus on the specification of library interfaces to make MATISSE more flexible.

3. Experimental Results

We present here an evaluation of the MATISSE compiler using a set of MATLAB programs obtained from various sources. We consider three platforms for our experiments:

– PC – a desktop PC with a 2.66 GHz Core 2 Quad processor, Windows 7 64-bit, 12 GB of RAM;
– ODROID – an ODROIDXU+E board, with an Exynos 5410 SoC with an 1.6 GHz ARM's big.LITTLE configuration, 1 GB of RAM;
– BeagleBoard – a BeagleBoard-XM revB running Ubuntu 12.10 32-bit, with a 1 GHz ARM Cortex-A8 and 512 MB of RAM.

We use GNU gcc 4.8 on all platforms, MATLAB R2014a on the desktop, and OpenBLAS v0.2.12 as the BLAS library. We have also made available an online version of MATISSE [10].

3.1 MATLAB vs C Results

Figure 7 compares the performance between C code generated by MATISSE and C code generated by MEGHA [11]. Although MEGHA focuses on generating CUDA-enabled code, they also present good C performance. We tried to replicate in MATISSE the experimental setup used by Prasad et al. [11], by using input matrices with the same sizes, and adjusting inputs in certain cases so that we could get approximate absolute MATLAB execution times for the same benchmarks. Although it is not possible to directly compare the results, since the hardware and the software used in the experiments are different, the relative performance to MATLAB can give an indication of the performance of the C code. Also, consider that the C code currently generated by MATISSE does not target multicore architectures, and one of the planned steps is to generate C code with OpenMP directives.

In general, MATISSE achieves speedups in line with the ones published for MEGHA for C code (8.1× and 5.6× of geometric mean for MATISSE and MEGHA, respectively). The case where there was a greater difference between MEGHA and MATISSE, when considering the biggest inputs, was *editdist* (a 2.8× in performance). We think this might have to do with *editdist* being the only example that can be implemented using only integers, a case that MATISSE can optimize aggressively using aspects.

The *closure* example is dominated by matrix multiplication and since both MATLAB and MEGHA use BLAS to perform matrix multiplications, the performance is roughly the same.

Since the Core2Quad is relatively an old processor (i.e., 2008), we ran the same benchmarks using a processor with a newer architecture, to validate the data in more modern PC processors (i.e., an AMD A10-7850K 4.10GHz, from 2014). Although there were differences in speedups of more than 160% on some of the individual benchmarks with larger datasets (either faster or slower), we measured a similar geometric mean speedup of 8.0×.

3.2 Matrix Views Results

We used a synthetic benchmark (*copy_vs_ptr*) to test the potential of using views with pointers instead of views by copy (see Section 2.3). When we enable the use of views with pointers, we obtain speedups of 1.8×, 1.4× and 1.5× for the PC, ODROID and BeagleBoard, respectively. We associate the higher speedups on the PC as in this system there is a bigger penalty for copying data. The PC also achieved better performance on other benchmarks after enabling views with pointers, such as *closure* (1.6×), *fdtd* (1.3×), and *nbody3d* (1.3×).

3.3 Third-Party Libraries and Target-Specific Parameters

The benchmark *closure* spends most of its execution time in matrix multiplication operations, and can run orders of magnitude slower if instead of BLAS we use a simple, straight forward, version of matrix multiplication (e.g., 94× times slower on PC when *N* is 1,024). Figure 6 shows the impact on performance speedup of the BLAS version of the *closure* benchmark over a simple implementation for different input sizes. Using the library for certain input sizes can cause slowdowns. However, the threshold to decide if it is more advantageous to use BLAS changes with the target platform: on the PC, the use of BLAS provides clear speedups when the input size is 16, but on the BeagleBoard, for an input size of 32 it is still slower to use BLAS. The speedup using BLAS in the PC decreases from 2.6× to 1.3× for values of N between 16 and 32. When we observe the behavior of both implementations separately, the behavior of the simple algorithm is more predictable, while the behavior using BLAS is more dependent on the data sizes. BLAS is a very complex and highly optimized package, whose internals can change from version to version.

Another benchmark using matrix multiplications is *nbody3*. In this case, if one sets a low enough threshold MATISSE will use BLAS in all cases of matrix multiplication and have a 9% performance degradation as result, for the tested input sizes.

MATISSE allows the specification of strategies for statically deciding about the use of a certain implementation (future work will consider the runtime selection of implementations). The number of elements of a matrix is an example of a parameter that can be possibly discovered by auto-tuning, and that can be defined externally as a target-dependent optimization in a LARA aspect. We take the pragmatic approach of treating this kind of libraries as a black-box, and our future work will focus on how to automatically discover the parameters which better suit a given target.

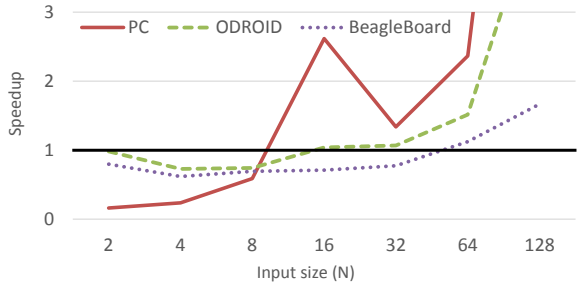

Figure 6. Speedups of *closure* using BLAS over the use of simple implementations for matrix multiplication according to data sizes.

3.4 Element-Wise Mapping Operations

If a program spends a significant amount of time using element-wise operations, transforming such operations to *for* loops can yield significant improvements (see Table 1). The benchmark *hypotenuse* (see Figure 4) spends almost all of its runtime doing element-wise operations, and can be used as an indicator of the maximum improvement we can obtain from such transformation. In the case of the PC and ODROID, we achieved a speedup of 3.7× and 3.2×, respectively. In the BeagleBoard the benchmark achieved a significantly lower gain, 1.6×. Inspecting the assembly of this benchmark, we concluded that the speedups come from the C compiler being able to vectorize the code automatically after transforming the operations to *for* loops, using special instructions available in the processors (SSE in the case of PC, and NEON in the ARM Cortex-A15 used in the Exynos 5410 of the ODROID board). The higher overheads in function calling, the worse code locality and the lack of support to vectorization in the BeagleBoard processor contributed to the lower speedups.

3.5 MATISSE Primitives

On the PC the use of the MATISSE primitives has some impact in the performance, while in the other platforms used there are virtually no difference (see Table 2). We think this happens because improvements of current primitives are related to avoiding setting newly allocated memory (*matisse_new_array_from_matrix*). Since in the PC the cost of memory operations is higher when compared with the other tested platforms, this kind of optimizations are more noticeable.

Table 1. Speedups achieved when applying the element-wise to *for* transformation.

Benchmark	Platform		
	PC	ODROID	BeagleBoard
capacitor	1.3	1.0	1.0
hypotenuse	3.7	3.2	1.6
nbody1d	1.8	1.1	1.2

Table 2. Speedups obtained when using MATISSE primitives.

Benchmark	Platform		
	PC	ODROID	BeagleBoard
capacitor	1.5	1.04	1.04
crnich	1.3	1.02	1.02
hypotenuse	1.3	1.14	1.01
nbody1d	1.3	1.06	1.02

4. Related Work

The popularity of the MATLAB language is also reflected in the similar languages that have been proposed (e.g., Octave [12]). 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. Yet, the need to avoid a MATLAB runtime system in most embedded systems has led to the development and research on how to best translate MATLAB programs into equivalent C code.

The translation of MATLAB to other programming languages is not recent. For instance, DeRose and Padua developed the FALCON environment [14][15] 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 [16]. Joisha et al. [17][18] 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, [19][20] use annotations to specify data types and shapes and simple type inference analysis and target VHDL code specification for hardware synthesis onto FPGAs.

The Sable Lab at McGill University have done extensive work around the MATLAB language [21], including an AOP-extension [22], a virtual machine [23] and code generators for several languages, such as Fortran95 [24] and X10 [25]. One of their focus is on having formal descriptions of MATLAB semantics. Although this is out of MATISSE scope, they describe several analysis and transformations that are relevant to our current work, and that we want to implement in the future (e.g., kind analysis [26], language simplification [27]).

MATLAB Coder [2] is Mathworks solution for MATLAB to C compilation. They support a large subset of MATLAB, and allow the code to be customized through directives and options. However, the customization is fairly coarse-grained, and finer control (e.g., addressing types, shapes, and target) is supported by direct modification of the MATLAB source code. 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.

MATISSE differs from the previous approaches in several aspects. First, its focus is on exploring several C implementations for multitarget solutions, unlike other approaches which have a larger focus on the MATLAB language itself and how to generate a single correct implementation [21]. In the multitarget context, it is very important to have a high degree of customization of the generated code. As an example, MATISSE allows defining the type of any variable in the code, unlike Coder, which only allows to set the types of the function inputs [2]. MATISSE main goal is to customize the output code without modifying the source code, hence the use of AOP concepts to control the customization. Furthermore, since MATISSE mainly targets embedded systems, it considers that the generated code will be run in environments where a MATLAB-compatible runtime might not be available, unlike hybrid approaches [28]. Orthogonally to the work presented in this paper, the MATISSE compiler is being extended with OpenCL generation [29], and to the best of our knowledge, it is the first tool that generates OpenCL kernels from MATLAB.

5. Conclusion

This paper presented recent improvements in MATISSE, a multitarget/multichain compiler framework for compiling MATLAB to low level programming languages such as C. MATISSE relies on LARA aspects for specifying data types, shapes, and code instrumentation and specialization. The compiler includes a type and shape inference stage and is able to generate MATLAB and C code. We presented some transformations and optimizations performed by the compiler. The results reveal that transformations such as detecting element-wise operations and operations such as matrix multiplications justify the use of a higher abstraction such as MATLAB as a starting point for specialized implementations in C. The experiments reveal promising performance results, achieving a geometric mean speedup of 8× over execution in MATLAB when considering a set of benchmarks previously used in literature. These results indicate a possible improvement when compared with a related state-of-the-art MATLAB compiler.

Ongoing work is focused on further optimizations for generating C code and on enhancing the flexibility of the code generators for efficient multitarget/multichain, such as generating C tailored for hardware generation through High-Level Synthesis tools, and on enhancing the OpenCL backend for GPUs and FPGAs.

Acknowledgments

This work was partially funded by project REAL TIME NORTE-07-0124-FEDER-000062 which was financed by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, *Fundação para a Ciência e a Tecnologia* (FCT). Reis acknowledges the support provided by FCT through grant PD/BD/105804/2014.

References

[1] *MATLAB – the Language of Technical Computing*. The MathWorks, Inc., http://www.mathworks.com/.

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

[3] J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J. M. Cardoso, and P. C. Diniz, "The MATISSE MATLAB compiler," in *11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 602–608.

[4] G. Kiczales, "Aspect-oriented programming," *ACM Comput. Surv. CSUR*, vol. 28, no. 4es, p. 154, Dec. 1996.

[5] J. D. Gradecki and N. Lesiecki, *Mastering AspectJ: aspect-oriented programming in Java*. NY, USA: John Wiley & Sons, 2003.

[6] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: an aspect-oriented programming language for embedded systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, Potsdam, Germany, 2012, pp. 179–190.

[7] J. M. P. Cardoso, P. Diniz, J. G. Coutinho, and Z. Petrov, *Compilation and Synthesis for Embedded Reconfigurable Systems*. Springer, 2013.

[8] *Symja - Java Computer Algebra Library*. .

[9] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, and others, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, 2002.

[10] *MATISSE*. http://specs.fe.up.pt/tools/matisse/.

[11] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors," in *ACM Sigplan Notices*, 2011, vol. 46, pp. 152–163.

[12] *Octave*. http://www.gnu.org/software/octave/.

[13] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge, "Optimizing MATLAB through just-in-time specialization," in *Compiler Construction*, 2010, pp. 46–65.

[14] G. Almasi and D. A. Padua, *MaJIC: A MATLAB just-in-time compiler*. Springer, 2001.

[15] L. A. De Rose, "Compiler Techniques for Matlab Programs," Citeseer, 1996.

[16] P. G. Joisha and P. Banerjee, "Static array storage optimization in MATLAB," in *ACM SIGPLAN Notices*, San Diego, CA, USA, 2003, vol. 38, pp. 258–268.

[17] P. G. Joisha and P. Banerjee, "A translator system for the MATLAB language," *Softw. Pract. Exp.*, vol. 37, no. 5, pp. 535–578, 2007.

[18] P. G. Joisha and P. Banerjee, "An algebraic array shape inference system for MATLAB®," *ACM Trans. Program. Lang. Syst. TOPLAS*, vol. 28, no. 5, pp. 848–907, 2006.

[19] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Parallelization of Matlab applications for a multi-FPGA system," in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, Rohnert Park, CA, USA, 2001, pp. 1–9.

[20] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, "Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design," in *Field-Programmable Custom Computing Machines, 2003. 11th Annual IEEE Symposium on*, Napa, CA, USA, 2003, pp. 263–264.

[21] A. Casey, J. Li, J. Doherty, M. Chevalier-Boisvert, T. Aslam, A. Dubrau, N. Lameed, A. Aslam, R. Garg, S. Radpour, and others, "McLab: an extensible compiler toolkit for MATLAB and related languages," in *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, 2010, pp. 114–117.

[22] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren, "AspectMatlab: An aspect-oriented scientific programming language," in *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, NY, USA, 2010, pp. 181–192.

[23] N. A. Lameed, "Dynamic Compiler Optimization Techniques for MATLAB," McGill University, 2013.

[24] X. Li, "Mc2For: A MATLAB to FORTRAN 95 Compiler," McGill University, 2014.

[25] V. Kumar and L. Hendren, "MiX10: Compiling MATLAB to X10 for high performance," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 617–636.

[26] J. Doherty, L. Hendren, and S. Radpour, "Kind analysis for MATLAB," in *ACM SIGPLAN Notices*, 2011, vol. 46, pp. 99–118.

[27] A. W. Dubrau and L. J. Hendren, *Taming Matlab*, vol. 47. ACM, 2012.

[28] C.-Y. Shei, A. Yoga, M. Ramesh, and A. Chauhan, "MATLAB Parallelization through Scalarization," in *15th Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, 2011, pp. 44–53.

[29] João Bispo, Luís Reis, and João M. P. Cardoso, "C and OpenCL Generation from MATLAB," in *30th ACM Symposium on Applied Computing (SAC 2015)*, Salamanca, Spain, 2015.

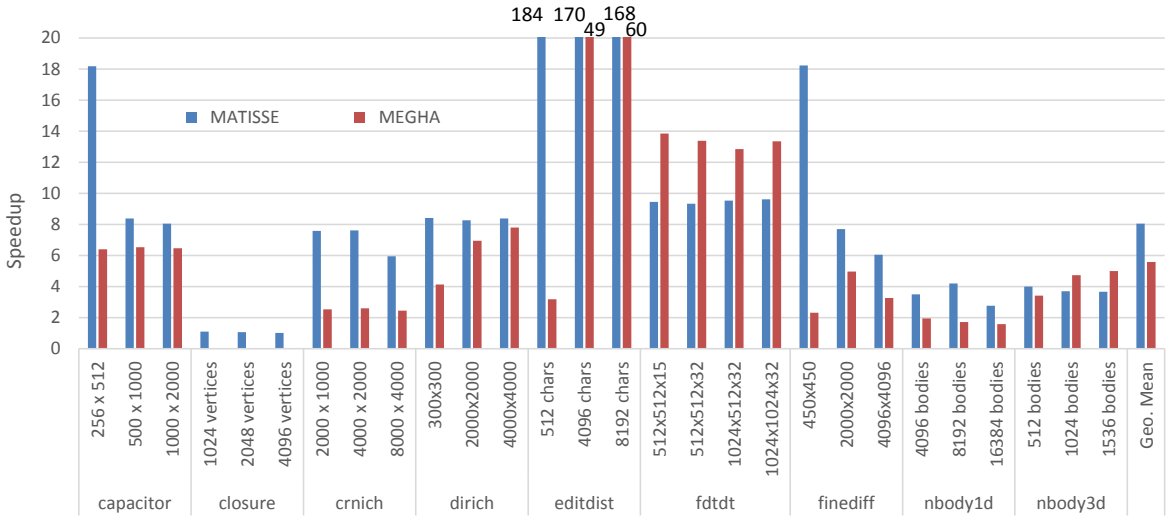

Figure 7. Speedup of C code vs MATLAB, for C generated by MATISSE and MEGHA [11].