Matlab to C compilation targeting Application Specific Instruction Set Processors

Ioannis Latifis*, Karthick Parashar†, Grigoris Dimitroulakos*, Hans Cappelle†, Christakis Lezos*, Konstantinos Masselos*, Francky Catthoor†
*University of Peloponnese, Department of Informatics and Telecommunications
Terma Karaiskaki, 22100 Tripoli, Greece
{*latifis, dhmhgre, lezos, kmas*}@*uop.gr*
†Interuniversity Microelectronics Centre (IMEC)
Kapeldreef 75, B-3001 Leuven, Belgium
{*Karthick.Parashar, Hans.Cappelle, catthoor*}@*imec.be*

***Abstract*—This paper discusses a MATLAB to C compiler exploiting custom instructions such as instructions for SIMD processing and instructions for complex arithmetic present in Application Specific Instruction Set Processors (ASIPs). The compiler generates ANSI C code in which the processor's special instructions are represented via specialized intrinsic functions. By doing this the generated code can be used as input to any C/C++ compiler. Thus the proposed compiler allows the description of the specialized instruction set of the target processor in a parameterized way allowing the support of any processor. The proposed compiler has been used for the generation of application code for an ASIP targeting DSP applications. The code generated by the proposed compiler achieves a speed up between 2x-30x on the targeted ASIP for six DSP benchmarks compared to the code generated by Mathworks MATLAB to C compiler. Thus the proposed compiler can be employed to reduce the development time/effort/cost and time to market by raising the abstraction of application design in an embedded systems / system-on-chip development context while still improving implementation efficiency.**

***Index Terms*—MATLAB, compilation, Application Specific Instruction Set Processor (ASIP), embedded systems, System-on-Chip (SoC)**

I. Introduction

MATLAB [1] is a popular language for algorithmic and system modelling with several million users worldwide both in industry and academia in different scientific and technical disciplines. In the context of embedded systems and Systems-on-Chip MATLAB code is used as a high level input specification for hardware and software development flows. Consequently, MATLAB compilation to code that can be implemented in an automated way to software or hardware has been made essential in the domain of embedded systems systems-on-chip. A number of tools [2], [3], [4], [5], [6], [7], [8] exist for automatically generating C or VHDL code from MATLAB sources.

Application Specific Instruction Set Processors (ASIPs) offer a very interesting trade-off between implementation efficiency (area and power for given performance) and flexibility/programmability (to implement different algorithms, for upgrading etc.) as compared to conventional CPUs and ASICs. The instruction set of an ASIP is customized to benefit specific set of applications. ASIPs are in most cases instantiated as components in Systems-on-Chip (also field programmable ones) in embedded systems. ASIPs are very popular for the implementation of DSP algorithms such as baseband signal processing and video processing and they can be designed using commercially available tools such as Processor Designer [9] from Synopsys.

The ever increasing application complexity creates the need for raising the design abstraction and for design automation in the form of compilers. To the best of our knowledge no tools generating vectorized C code for ASIPs are currently available. This paper presents a MATLAB to C compiler with a number of features favoring its use in ASIP based hardware platforms. MATLAB expressions matching the instruction set of the targeted processor are exposed in C code in the form of intrinsic functions that are exploited by the C compiler at a later stage. In this way the proposed compiler can be used with any C compiler such as the very popular LLVM/Clang and GCC but also the (retargetable) ASIP compilers (i.e. Target Suite Tool [9]). In this way the proposed compiler allows mapping specialized operations (in the form of intrinsic functions) to specific hardware modules usually present in ASIPs. The selection of the appropriate instructions is achieved with the aid of a parameterized processor model. This approach allows the proposed compiler to target any processor. Moreover, the compiler's backend may produce either scalarized or SIMD-style C depending on the target processor. SIMD support can be configured with respect to the blocks that are eligible for SIMD code generation and the preferred vector size.

The proposed compiler has been evaluated using six realistic fixed point DSP algorithms. The benchmarks have been mapped on an ASIP processor customized for DSP supporting SIMD processing that has been used for the implementation of complex wireless communication systems (baseband processing). The performance (speed) of the generated C code has been compared to that of the code generated by the Mathworks MATLAB to C compiler. Experimental results show that the proposed compiler achieves upto 30x on some benchmarks for the target ASIP.

*This work has been partially supported by the FP7 ALMA (FP7 ICT-2011. 287733) project, funded by the European Community.

The remainder of this paper is organized as follows. Section II summarizes the related work in the field. Section III describes the proposed compiler infrastructure. Section IV overviews the target architectures which were used for the experiments. Section V, presents the experimental results and finally, the conclusion of the paper and future work are discussed in section VI.

II. Related Work

Several approaches have been presented for the compilation of MATLAB to languages that provide a more efficient execution/implementation environment. One of the first approaches is FALCON [10], a MATLAB to FORTRAN90 compiler with main key contribution the static and dynamic inference mechanism supported by a sophisticated symbolic value propagation algorithm. MATCH [3] compiler target high-level synthesis and translate MATLAB code to a register transfer level HDL supporting fixed point arithmetic. The tool provides a framework of notations named as directives, that the user can insert in MATLAB code to bridge the gap between MATLAB source and the available computational structures. OTTER [4], and MENHIR [5] are MATLAB compilers producing SPMD-style C code for parallel code execution (MENHIR can produce C or FORTRAN) relying on libraries such as ScaLapack and MPI message-passing libraries. The MAT2C [6], a similar to Mathworks compiler [11], uses MAJICA [12] type inference tool attaining better performance from Mathworks compiler (MCC) [11]. MEGHA [7] uses a heuristic algorithm to map data parallel regions of the program (kernels) to heterogeneous processors (CPU and GPU). MATISSE [8] compiler focus on MATLAB to C efficient compilation performing optimizations and transformations on MATLAB code supported by LARA [13] aspect-Oriented programming language.

Compiling MATLAB to C for vectorized processor architectures using the Mathworks compiler and an auto-vectorization compiler (i.e. LLVM) wouldn't be a good solution. The information related to vectorization which should be represented in MATLAB would be partly counteracted or even eliminated (due to internal transformations and other optimizations) during translation to scalarized C code incapacitating the auto-vectorizer to fully exploit the vectorized MATLAB operations. Mathworks embedded coder [14] provides the user with an environment to compile MATLAB code targeting embedded systems. Embedded coder uses an architecture description model similar to the one used by the proposed compiler for the customization of the generated C code. The major disadvantage is the lack of support for vector operations since only instructions for scalars and arrays are supported and vectorized C code cannot be generated. Even if array (instead of vector) operations are used for the customization of the generated code, code including operations with indexing cannot be efficiently compiled. Furthermore, Embedded Coder generates code storing sub-array references to an intermediate temporary array and then the specialized function (corresponding to customized instruction) is called. Such code leads in most cases to a worse performance than the corresponding code generated from the Mathworks compiler [2].

On the contrary, the approach proposed in this paper compiles MATLAB to C and generates SIMD-style C code using a more appropriate representation in which array operands and variables' indices are expressed as vectors. This code is then directly suited as input for LLVM level vectorizing compilers (like Clang) or commercial C-level vectorizing compilers like the Synopsys ASIP design environment [9]. To the best of our knowledge no environments supporting MATLAB to C compilation producing customized and vectorized C code for any targeted processor (particularly useful in an ASIP implementation context) currently exists. The proposed approach introduces a multi-target MATLAB to C compilation framework extendable and flexible to cover any target processor while still generating customized and optimized code compatible with any C/C++ compiler. The proposed compiler uses a parameterized target processor model and exploits the entire instruction set of the processor. A set of declarative statements can be used by the developer to select between SIMD and scalarized C code as well as to select appropriately derived data types (floating point, integer or fixed point).

III. Compiler Infrastructure

Figure 1 depicts the proposed compiler's flow highlighting the innovative contributions. The inputs to the compiler are the annotated application MATLAB code and the instruction set architecture of the target processor in xml. Annotations of MATLAB code are introduced by the developer and given in the form of pragma functions to declare the data type, the array shape/size and the fixed point attributes (word and fraction length) in the case of fixed point variable. Pragma functions are also used for the selection of the parts of the input code (SIMD blocks) that will be translated in SIMD style allowing the developer to select the preferred vector size of the block's SIMD operations. Figure 2 shows the MATLAB

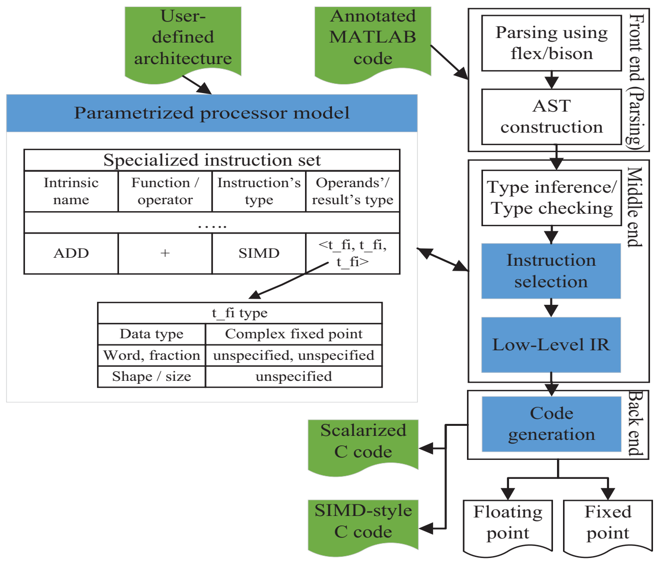

Fig. 1. Compiler infrastructure.

```
dec_vect('in','16','12',4,8);    %variable declerations
dec_vect('twd_fac','16','12',4,1);
dec_vect('out','16','12',4,8);
startSIMD('2');
for k=1:size(in,1) %stage 1
  out(:,k) = in(:,k) +in(:,k+4);
  out(:,k+4) = (in(:,k) - in(:,k+4)).* twd_fac;
end
... %code for stage 2 and 3
stopSIMD();
```

Fig. 2. FFT-32 stage 1, MATLAB code annotated for SIMD generation.

```
for(si=0; si < 4; si=si+2){ //SIMD block for-loop
  for(k=1; k < 5; k=k+1){
    out[(si+(k-1)*4)/2]=ADD(in[(si+(k-1)*4)/2],in[(si+(k+3)*4)/2]);
    tmp0[si/2]=SUB(in[(si+((k-1)*4))/2],in[(si+((k + 3)*4))/2]);
    out[(si+((k + 3)*4))/2]=VMUL(tmp0[si/2], twd_fac [si/2]);
  }
  ... //code of the rest FFT stages is continued here
}
```

Fig. 3. FFT-32 stage 1, SIMD C code.

code of the first stage of a 32-point FFT annotated for SIMD code generation. The *dec_vect* pragma functions declare vector variables of complex fixed point data type with word length of 16 and fraction length of 12 while the last two parameters are referred to the dimensions of the variables. The *startSIMD* and *stopSIMD* pragma functions define the SIMD block while the parameter (value 2) of *startSIMD* indicates the selected SIMD width.

The parameterized processor model describes the instruction set architecture of the target processor including specialized instructions. The model includes a list with information about the instructions. For each instruction the function name or operation type, the corresponding instruction name in C, the instruction type (scalar, SIMD or array) and the operands' (and result's) types are provided. For the instructions which can be mapped without the requirement of any of the type's attributes, the corresponding attributes can remain unspecified.

In the compiler's middle-layer, type inference analysis is performed using an approach similar to the one described in [10] only for compile-time type detection. In the next step, the instruction selection pass utilizes the parametrized processor model to map available specialized instructions to function calls and MATLAB operations. To achieve this, the traversal passes to the model, the operands/parameters types, the operation type or function name and information on whether or not the current function or operation is part of a SIMD block. After instruction selection, a separate pass is applied to transform Abstract Syntax Tree (AST) to a low-level intermediate representation (IR). This pass involves the decomposition of complex MATLAB array expressions to simpler ones retaining the vectorized form of the SIMD operands.

The compiler's back-end comprises of the code generation which generates the special instructions as intrinsic functions. The output depends on the developer's SIMD block declarations and may be scalarized where MATLAB array expressions (operations) are translated to C loop nests or SIMD-style C where each SIMD block is implemented as a C for-loop. The for-loops corresponding to SIMD blocks are composed by a for-loop condition depending on the array size of the SIMD block operands while the for-loop iteration step is the vector width has been specified in the SIMD block pragma function. The derived data types can be floating point, integer or fixed point. For the latter, extra C code is generated for handling fixed point arithmetic such as shifting to adjust the operand's fraction length. Figure 3 presents the generated SIMD code of the MATLAB code depicted in figure 2. The output includes the SIMD block for-loop with step 2 (defined SIMD width) and loop condition 4 as the operands' size for the specialized instructions. The complex fixed point addition, subtraction and multiplication have been mapped to *ADD*, *SUB*, *VMUL* intrinsic functions respectively, while flattening has been performed for the generation of the vector indices.

IV. Target Architecture

The ASIP approach provides a tradeoff between the less specialized general purpose processors on one hand and the rigidly defined and highly optimized ASIC platforms. ASIPs target application specific domains to sharpen the efficiency of implementation while keeping them programmable across various applications in the chosen domain. ADRES is an ASIP architecture template targeting applications in the streaming domain suitable for many signal processing applications. This template has been used in the past for multi-media and wireless signal processing applications and very high power and throughput efficiencies have been reported. It has also been able to prove that in some cases [15], that careful design choices while deriving ADRES can surpass the efficiency of ASIC and FPGA based designs in terms of performance achieved and more importantly, the development time. In this work, an instance of the ADRES template *BoT* has been chosen to experiment with.

The *BoT* is a 10 way VLIW instruction set processor. It has been envisioned specially for implementing wireless physical layer signal processing [16]. The *BoT* architecture consists of 4 essential parts as shown in figure 4. These parts are: the scalar data path which is used for managing the control flow, the shuffle unit that orders/ re-orders vector-type data in a vector register file, the load-store (LD/ST) unit for interaction with vector memory and the SIMD data-path that enables massive parallel computation. The vector paths are 3 with each of them capable of supporting 4 or 8 way SIMD complex data type with an aggregate width of 32 bits allocated as 16 bits each for imaginary and real parts. The choice of SIMD width between 4 and 8 is a design time choice that the template provides. The vector register file is also 32 deep and 256 bits (or 128 bits for 4 way SIMD) wide to hold vectored data serving

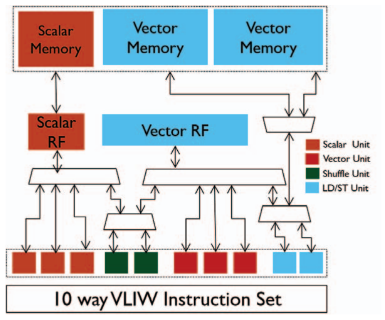

Fig. 4. BoT architecture.

any of the three Vector units. The vector instructions include Trigonometric functions like Sine, Cosine and inverse-Tangent functions, complex number operations such as absolute, multiply accumulate (MAC), multiple shift, real operations such as inverse and inverse square on vector data types.

The *BoT* architecture is coded using the Synopsys ASIP design tool [9]. In this tool the processor is described using the *nML* language. The tool generates all required compilation, debug and simulation infrastructure for the defined architecture. Benefitting from these facilities, the *BoT* architecture exposes several domain specific functionalities as C language intrinsics. These intrinsics directly map to one or more predefined group of instructions on the processor and can be called like a regular C-language function sub-routine.

V. Experimental Results

The proposed compiler has been used to generate code for a number of application benchmarks for the *BoT* ASIP discussed in previous section. Mathworks compiler [2] has been also used to generate code for the application benchmarks for the targeted processor to allow comparisons. In this case the benchmark application codes have been modified to add fixed point behavior in the source code using fi objects [17]. The C application codes generated by both the proposed compiler and the Mathworks compiler have been mapped to the targeted processor using the compiler of the Synopsys ASIP Designer.

The proposed compiler has been evaluated and compared against Mathworks MATLAB to C compiler using six fixed point DSP algorithms under different scenarios concerning the applications' input stream sizes. The experiments have been carried out using SIMD processing width of 8 except FFT algorithms, where SIMD processing width of 4 has been used. Using SIMD processing width of 8 on FFT is not meaningful since more data shuffling would be needed thus dramatically decreasing performance. Furthermore, due to the incompatibility of performance value ranges the experimental results referring to performance have been normalized per each application and input size. Figure 5 describes the cycles that correspond to a performance value of 1 for each application.

Fig. 5. Reference values corresponding to performance value 1.

FFT32	1443
FFT64	8032
FT128	17420
CFO32	11563
CFO64	22483
CFO128	45668
MEAN32	181
MEAN64	256
MEAN128	448
MEAN1024	3152
FIR32x256	724074

Figure 6 presents the normalized execution times per application achieved by both the proposed compiler and Mathworks compiler on *BoT* processor. Furthermore, the line added in the same figure depicts the execution speedup achieved with the proposed compiler over Mathworks MATLAB to C compiler. The comparison of the proposed compiler to Mathworks compiler proves that the proposed approach achieves a significant speed up between 2x-30x.

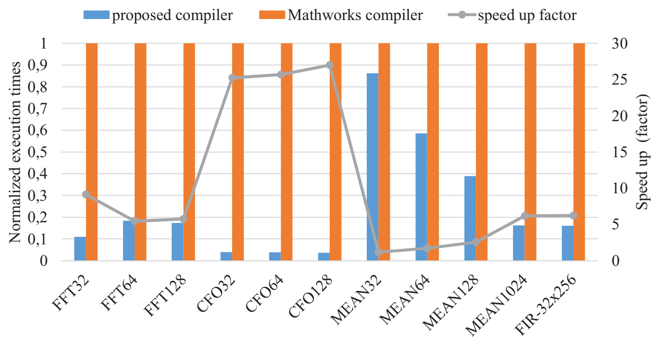

Fig. 6. Speed up comparing with Mathworks compiler.

VI. Conclusions and Future Work

In this paper a MATLAB to C compiler is presented targeting embedded systems and specifically Application Specific Instruction Set Processor architectures. A key feature of the proposed compiler is the parametrized processor model allowing the matching of MATLAB expressions with the available hardware modules of any targeted architecture. The compiler generates SIMD or scalarized ANSI C code representing the special instructions in the form of intrinsic functions. Experimental results show substantial better performance against the Mathwork MATLAB to C compiler.

Ongoing work concerns the use and benchmarking of the proposed compiler on other architectures such as ARM NEON general-purpose SIMD engine and GPU architectures.

References

[1] Matlab the language of technical computing. [Online]. Available: http://www.mathworks.com/products/matlab
[2] Matlab coder generate c and c++ code from matlab code. [Online]. Available: http://www.mathworks.com/products/matlab-coder
[3] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems," in *FCCM '00*.
[4] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao, "Preliminary results from a parallel MATLAB compiler," in *IPPS/SPDP '98*.
[5] S. Chauveau and F. Bodin, "Menhir - An Environment for High Performance Matlab," *Sci. Program.*, vol. 7, no. 3-4, pp. 303–312, Aug. 1999.
[6] P. G. Joisha and P. Banerjee, "A Translator System for the MATLAB Language," *Softw. Pract. Exper.*, vol. 37, no. 5, Apr. 2007.
[7] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors," in *PLDI '11*.
[8] J. Bispo, L. Reis, and J. M. P. Cardoso, "Multi-Target C Code Generation from MATLAB," in *ARRAY '14*.
[9] Synopsys - asip designer. [Online]. Available: http://www.synopsys.com/dw/ipdir.php?ds=asip-designer
[10] L. De Rose and D. Padua, "Techniques for the Translation of MATLAB Programs into Fortran 90," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 2, pp. 286–323, Mar. 1999.
[11] Matlab compiler - build standalone applications from matlab programs. [Online]. Available: http://www.mathworks.com/products/compiler/index.html
[12] P. G. Joisha and P. Banerjee, "The MAGICA Type Inference Engine for MATLAB," in *CC '03*.
[13] 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 *AOSD '12*.
[14] Embedded coder - generate c and c++ code optimized for embedded systems. [Online]. Available: http://www.mathworks.com/products/embedded-coder
[15] R. Fasthuber, F. Catthoor, P. Raghavan, and F. Naessens, *Energy-Efficient Communication Processors*. Springer New York, 2013.
[16] Bot- a low power processor for wireless baseband. IMEC. [Online]. Available: http://tinyurl.com/olasdj6
[17] Construct fixed-point numeric object - matlab fi. [Online]. Available: http://www.mathworks.com/help/fixedpoint/ref/fi.html