Automated Translation of MATLAB Code to C++ with Performance and Traceability

Geir Yngve Paulsen
and Stuart Clark
Simula Research Laboratory, Norway
Email: geirpa@gmail.com, stuart@simula.no

Bjørn Nordmoen, Sergey Nenakhov
and Aron Andersson
WesternGeco, Norway
Email: {nordmoen, snenakhov, AAnderson11}@slb.com

Xing Cai
Simula Research Laboratory, Norway
University of Oslo, Norway
Email: xingca@simula.no

Hans Petter Dahle
Fornebu Consulting, Norway
Email: Hans.Petter.Dahle@fornebuconsulting.com

***Abstract*—In this paper, we discuss the implementation and performance of *m2cpp*: an automated translator from MATLAB code to its matching Armadillo counterpart in the C++ language. A non-invasive strategy has been adopted, meaning that the user of *m2cpp* does not insert annotations or additional code lines into the input serial MATLAB code. Instead, a combination of code analysis, automated preprocessing and a user-editable metainfo file ensures that *m2cpp* overcomes some specialties of the MATLAB language, such as implicit typing of variables and multiple return values from functions. Thread-based parallelisation, using either OpenMP or Intel's Threading Building Blocks (TBB) library, can also be carried out by *m2cpp* for designated for-loops. Such an automated and non-invasive strategy allows maintaining an independent MATLAB code base that is favoured by algorithm developers, while an updated translation into the easily readable C++ counterpart can be obtained at any time. Illustrating examples from seismic data processing are provided in this paper, with performance results obtained on multicore Sandy Bridge CPUs and Intel's Knights-Landing Xeon Phi processor.**

Keywords–Code translation; Seismology; Image processing; MATLAB; C++.

I. INTRODUCTION

MATLAB® is a popular software for computational mathematics, particularly because of its accessibility for scientists and engineers as a high-level scripting language. This ease of use and a large library of toolboxes make MATLAB a good choice for testing and prototyping new algorithms. However, once the algorithms are tested, the ability to sufficiently optimise MATLAB code may become a key concern. MATLAB is a scripting language, so it cannot make use of compile-time based optimisations, such as latency-grouped instructions. While MATLAB offers multi-process based parallelisation, multi-thread optimisation can in many cases work better on current high-performance systems [1]. As a result of these issues, in situations for which optimisation is extremely important, rewriting the MATLAB code in another language, such as C++, could be a remedy. For that purpose, the Armadillo C++ library was developed to enable MATLAB-like syntax in a C++ setting [2][3].

Although Armadillo has adopted a MATLAB-resembling syntax for matrix and vector based computations, there are still distinctive syntax differences between Armadillo and MATLAB. For example, Armadillo's indexing of matrix and vector entries starts from 0, whereas indices in MATLAB start from 1. Another example is extracting columns or rows from a matrix, which has a drastically different syntax in MATLAB than in Armadillo. (An incomplete list of the syntax differences can be found in [3].) Manually translating a MATLAB code to its C++ counterpart in Armadillo is thus tedious and potentially error prone. Since there is an almost one-to-one mapping between the two high-level syntaxes, automated MATLAB-to-Armadillo translation is theoretically possible. However, MATLAB and C++ are two fundamentally different languages. Some inherent language features of MATLAB pose challenges to an automated code translator. Two such examples are implicit typing of variables and multiple return values from functions. To handle these challenges, some existing MATLAB translators ask the user to insert annotations of variable declaration and initialisation. Such an invasive approach is not very user-friendly, and may also cause problems if algorithm developers want to further change the MATLAB code.

Therefore, we aim for an automated MATLAB-to-C++ translator that adopts a non-invasive strategy. This is achieved by combining (1) code analysis enabled inside the translator, (2) a fully automated preprocessor that identifies the actual types of all variables used in a MATLAB program, and (3) an accompanying metainfo file that allows user editing and, if necessary, introduction of special translation rules designated for some of the MATLAB code lines. At the same time, we certainly do not want the auto-translated C++ code to lose the capability of parallel computing, which is already available through MATLAB's Parallel Computing Toolbox and Distributed Computing Server [4]. We have focused on parallelising designated for-loops (through MATLAB's `parfor` construct) as one inherent step of the auto-translation, making use of either OpenMP [5] or Intel's Threading Building Blocks (TBB) [6] library in a shared-memory setting.

This paper thus presents the design and implementation of *m2cpp*, an automated MATLAB-to-Armadillo translator. It has been substantially enhanced from its initial version (named *Matlab2cpp* [7]). Using illustrating examples from seismic data processing, we will show the capability of *m2cpp*. Performance of the auto-translated C++ programs is measured for both serial and parallel executions, compared with the original MATLAB versions. The tested hardware platforms involve multicore Xeon Sandy Bridge CPUs and Intel's second-generation Xeon Phi processor (Knights Landing). As a benefit of the readable C++ code, which retains the same structure and variable

 ISBN: 978-1-61208-599-9

names of the original MATLAB code, we also demonstrate a particular example of further performance optimisation of the auto-generated C++ code.

The remainder of the paper is organised as follows. Section II summarises the overall design and main features of the *m2cpp* translator. Section III uses four real-world examples to compare the performance of MATLAB, auto-translated and manually optimised C++ codes, in both serial and parallel settings. Section IV places the present paper in the landscape of existing relevant work, whereas Section V provides a few concluding remarks and some thoughts on future work.

II. Design and implementation

A. Overall Structure

As mentioned above, the aim of *m2cpp* is to facilitate an automatic and non-invasive translation from MATLAB code to the matching Armadillo counterpart in the C++ language. The non-invasiveness refers to that the user of *m2cpp* does not need to insert any annotation or extra coding into a serial MATLAB program before passing it to the translator. The *m2cpp* translator itself is written in the Python language, with a tailor made top-down recursive descent parser [8] that follows the same approach adopted by ANTLR [9]. The parser reads the input MATLAB code and internally sets up an abstract syntax tree, which is then subjected to a post-order tree walk [8] for necessary code analysis and translation to the resulting C++ code. Tasks of code analysis include, e.g., identifying MATLAB functions that return multiple values, which are translated as additional input arguments of the corresponding C++ functions. Another important task of code analysis is in connection with thread-based parallelisation of for-loops, where some variables have to be made private per thread to avoid race conditions.

B. Metainfo File

For the automatic translation of a MATLAB program to work correctly, *m2cpp* also relies on an accompanying metainfo file, which has the same name as the (principal) MATLAB input file but ending with `*.m.py`. This approach ensures that the *m2cpp* translator is non-invasive to the input MATLAB code, because the additional information needed for the code translation is provided in a separate file, easily editable by the user if needed. The metainfo file consists of three parts, where the first part is a list of all variables to be encountered, containing the name and type of each variable. This part of information can be automatically filled out by an automated preprocessor (see below). The second part of the metainfo file, marked as the *includes segment*, contains explicit C++ `include` statements needed by the Armadillo library, as well as necessary `include` statements when the entire source code is spread over several files. These `include` statements are figured out by the automated preprocessor and will later be inserted into the translated C++ code. The third part of the metainfo file, marked as the *verbatims segment*, is optional. Here, the user has the possibility of introducing special translation rules. That is, the user can dictate how a specific code line in the MATLAB program should be translated into C++, without following the general translation rules of *m2cpp*.

C. Preprocessing

There is a preprocessing functionality with *m2cpp*. The main purpose is to automatically prepare the metainfo file with respect to identifying the actual type of each variable, which is needed for the subsequent MATLAB-to-Armadillo translation. Automatic identification of variable types is achieved by autonomously running a copy of the MATLAB program with inserted `dump` function calls for recording all the state information, including the actual data type of all encountered MATLAB variables. The recorded data type information is then automatically extracted and inserted into the metainfo file. Even if *m2cpp* is used on a computer without a MATLAB installation, the preprocessor of *m2cpp* will automatically identify all the encountered MATLAB variables, while providing a reasonable guess of the variable types. The user can then make corrections to the variable type information in the metainfo file, if necessary.

In a typical code development scenario, where the input MATLAB code is repeatedly changed, an existing metainfo file can be reused provided that the changes on the MATLAB side do not introduce new variables or non-standard statements that require a special translation rule. Even if such changes take place on the MATLAB side, it is often more convenient for the user to directly edit the metainfo file, without having to re-run the preprocessor of *m2cpp*.

D. Parallelisation of For-Loops

The focus of *m2cpp* with respect to parallelisation is on MATLAB for-loops that have independent iterations. These for-loops are assumed to be already marked as `parfor` constructs in the MATLAB program. The *m2cpp* translator considers shared memory and adopts thread-based parallelisation of the designated MATLAB for-loops. More specifically, the user of *m2cpp* can freely choose between parallelisation enabled by the OpenMP [5] standard or Intel's TBB library [6]. For the case of OpenMP, a compiler directive `#pragma omp parallel for` is automatically inserted before each designated for-loop. For the case of TBB, the code lines shown in Figure 1, making use of C++11's lambda expressions, are automatically inserted for each designated for-loop.

```
tbb::parallel_for(
  tbb::blocked_range<size_t>(1,num_points+1),
    [&](const tbb::blocked_range<size_t>& _range) {
      // declaration of thread-private variables ...
      for(size_t i=_range.begin();i!=_range.end();++i)
      {
        // loop iteration body ...
      }
  }
);
```

Figure 1. An example of parallelising a for-loop in TBB.

Common for both parallelisation approaches, *m2cpp* is able to properly introduce temporary variables that are private to each thread, so that race conditions will not happen.

E. Limitations

It should be noted that *m2cpp* is not supposed to translate any MATLAB code. The automated translation of *m2cpp* is restricted to MATLAB programs that make use of matrix and vector computations that are covered by the functionality of

 ISBN: 978-1-61208-599-9

the Armadillo library. Nevertheless, we have included a couple of new C++ functions beyond the original functionality of Armadillo, so that typical plotting functions of MATLAB can also be automatically translated by *m2cpp*.

In addition, two special features of the MATLAB language can not be handled by *m2cpp*. First, MATLAB allows a variable to implicitly change its type within a program. This is fundamentally in contrast to the static typing rule of C++. Although for some cases, it is possible to introduce a new variable (having a different name) in the C++ code to resolve the implicit change of a MATLAB variable type, we have decided to not support this, due to the infrequent occurrence of variable type changes in MATLAB programs. The other special MATLAB feature is dynamic expansion of matrices and vectors. It typically happens with variables that are declared with empty storage but are dynamically expanded inside a loop. In principle, a detailed code analysis can deduce the final size of dynamically expanded matrices or vectors, so that the translated C++ code can declare the matrices or vectors with a correct size. But this requires a rather elaborate code analyser, not yet supported in *m2cpp*. Another cumbersome and inefficient option is to frequently insert a call to the `resize` function of Armadillo, which we deem non-viable. As a remedy, though, the user can introduce a special translation rule in the metainfo file to prescribe a correct size for each dynamically-expanded MATLAB matrix or vector variable.

III. RESULTS

A. SeismicLab

We have chosen the open-source MATLAB package SeismicLab [10], which concerns seismic data processing, for verifying the correctness of *m2cpp*-translated C++ code. Moreover, we want to measure the speed of the translated C++ code, in both serial and parallel computing settings, for a comparison with the original MATLAB code.

For this paper, four relatively computation-heavy demo programs from SeismicLab have been chosen. They are `parabolic_moveout_demo`, `radon_demo_1`, `moveout_demo` and `fx_decon_demo`. (Each demo program spans several `*.m` files.) We have enlarged the computation size for all the four demos by increasing the resolution of the original input data files with help of linear interpolation. For the first two demos, which share the same input data file, the new computation size is 4004×1568, whereas the new computation size is 4176×2432 and 2004×1600 for the last two demos, respectively.

The numerical results produced by the auto-translated C++ codes have been carefully compared with those from the original MATLAB codes, to ensure the correctness of the code translation done by *m2cpp*. In the remainder of this section, our focus is thus directed to the serial and parallel efficiency of the auto-translated C++ codes.

B. Example of Parabolic Moveout

Readability of the auto-translated C++ code is ensured by retaining the exact same coding structure and variable names as in the original MATLAB code. For instance, let us first show in Figure 2 the main computation segment from the original MATLAB code for the example of parabolic moveout.

```
for it = 1:ntau
  for iq = 1:nq
    time =  tau(it) + q(iq)*(h/hmax).^2 ;
    s = zeros(2*L+1,nh);

    for ig = -L:L;
      ts = time + (ig-1)*dt;

      for ih = 1:nh
        is = ts(ih)/dt+1;
        i1 = floor(is);
        i2 = i1 + 1;

        if i1>=1 & i2<=nt ;
          a = is-i1;
          s(ig+L+1,ih) = (1.-a)*d(i1,ih) + a*d(i2,ih);
        end;
      end
    end

    s = s.*H;
    s1  = sum( (sum(s,2)).^2);
    s2  = sum( sum(s.^2));
    S(it,iq) = s1-s2;
  end
end
```

Figure 2. The original MATLAB code of the computational core of the parabolic moveout example.

The MATLAB code segment shown in Figure 2 constitutes the computational core of the parabolic moveout example. It is in fact a nested for-loop of four layers. The corresponding code segment of the auto-translated C++ code is shown in Figure 3. We can see that the C++ code adopts the Armadillo syntax while maintaining the same readability as the original MATLAB code. (We remark that the `%` operator in Armadillo does element-wise multiplication.)

```
for (it=1; it<=ntau; it++) {
  for (iq=1; iq<=nq; iq++) {
    time = tau(it-1)+q(iq-1)*arma::square(h/hmax) ;
    s = arma::zeros<mat>(2*L+1, nh) ;

    for (ig=-(L); ig<=L; ig++) {
      ts = time+(ig-1)*dt ;
      for (ih=1; ih<=nh; ih++) {
        is = ts(ih-1)/dt+1 ;
        i1 = std::floor(is) ;
        i2 = i1+1 ;

        if (i1>=1&&i2<=nt) {
          a = is-i1 ;
          s(ig+L, ih-1) = (1.-a)*d(i1-1, ih-1)
                                +a*d(i2-1, ih-1) ;
        }
      }
    }
    s = s%H ;
    s1 = arma::as_scalar(
            arma::sum(arma::square(arma::sum(s, 1)))) ;
    s2 = arma::sum(arma::sum(arma::square(s))) ;
    S(it-1, iq-1) = s1-s2 ;
  }
}
```

Figure 3. The auto-translated C++ code of the computational core of the parabolic moveout example.

Auto-parallelisation of the outermost `it`-indexed for-loop can also be carried out by *m2cpp*, via either OpenMP or TBB as described in Section II-D. This merely requires adding a comment of form `%#PARFOR` above the target for-loop in the MATLAB input code. (The auto-parallelised C++ code is not shown.)

 ISBN: 978-1-61208-599-9

C. Time Measurements

To study the performance of the auto-translated C++ codes, we used two representative hardware platforms: a dual-socket 2×8-core Sandy Bridge server and a 68-core Knights-Landing (KNL) Xeon Phi processor. The hardware specification can be found in Table I. The compilation flags used for the C++ codes were `-Ofast, -xHost, -D NDEBUG, -D ARMA_NO_DEBUG, -lmkl_intel_lp64, -lmkl_core, -lmkl_sequential`. It should be noticed that Intel's Math Kernel Library (MKL) is invoked by the auto-generated C++ codes when applicable. This is fair with respect to the original MATLAB codes, which also internally invoke Intel's MKL when applicable. For the codes parallelised with TBB, the additional compilation flags `-std=c++11` and `-ltbb` were also used. Each time measurement listed in Tables II-V was obtained by running the code at least three times and reporting the fastest time.

TABLE I. HARDWARE SPECIFICATION OF THE TWO TESTBED PLATFORMS USED.

Platform	Sandy Bridge server	KNL
Processor model	E5-2670 (dual socket)	Xeon Phi 7250
Clock frequency	2.6 GHz	1.4 GHz
Core count	16 (2×8)	68
Compiler	icpc v17.0.1	icpc v17.0.0

Table II compares the serial performance of the auto-translated C++ code, i.e., executed on only one hardware core of each machine. Since MATLAB (version R2016a) is only available on the dual-socket server, time measurements of the original MATLAB code are thus only reported for that system. It is clear that the *m2cpp*-translated C++ versions run considerably faster than the MATLAB counterparts on the dual-socket server. (The only exception is the `fx_decon` example, for which the core computation is done using Intel's MKL for both MATLAB and C++ versions.) Moreover, the single-core C++ performance obtained on KNL is lower than that obtained on a single core of the Sandy Bridge CPU, because of a much lower clock frequency and the absence of an L3 cache.

TABLE II. SINGLE-CORE TIME USAGE (IN SECONDS) OF FOUR DEMO PROGRAMS FROM THE SEISMICLAB PACKAGE.

Code Platform	MATLAB Server	C++ Server	C++ KNL
Parabolic_moveout	261.7	59.5	133.2
Radon1	33.6	19.0	41.6
Moveout	3.0	0.9	1.6
Fx_decon	2.7	2.0	4.0

Regarding the parallel performance, Tables III-IV show that the auto-translated C++ programs (using either OpenMP or TBB) get speedup when the number of threads increases up to the same number as physical cores. It should be remarked that MATLAB's `parfor` only works for the `fx_decon` example, due to a rather conservative MATLAB runtime system, although the iterations are actually independent in the other three examples. Therefore, parallel MATLAB performance is only reported for the `fx_decon` example in Table IV, not the other three examples in Table III.

TABLE III. PARALLEL TIME USAGE (IN SECONDS) OF SEISMICLAB'S `PARABOLIC_MOVEOUT`, `RADON1` AND `MOVEOUT` DEMOS.

Parabolic_moveout				
	Dual-socket server		KNL	
# threads	OpenMP	TBB	OpenMP	TBB
1	59.3	59.9	126.4	128.5
2	30.1	30.0	84.6	66.1
4	15.5	15.3	44.2	32.9
8	8.1	7.9	22.1	17.8
16	4.4	4.4	11.4	9.4
32	4.7	4.2	6.1	5.0
68			3.0	3.6
Radon1				
	Dual-socket server		KNL	
# threads	OpenMP	TBB	OpenMP	TBB
1	18.4	19.1	41.1	42.0
2	10.8	10.6	22.9	24.0
4	6.5	6.4	13.6	13.8
8	4.4	4.4	8.3	8.8
16	4.0	3.5	5.6	6.5
32	3.8	3.6	4.3	4.9
68			3.8	4.5
Moveout				
	Dual-socket server		KNL	
# threads	OpenMP	TBB	OpenMP	TBB
1	0.91	0.85	1.56	1.78
2	0.56	0.54	1.03	0.98
4	0.39	0.38	0.61	0.54
8	0.31	030	0.35	0.35
16	0.30	0.27	0.25	0.25
32	0.29	0.26	0.20	0.20
68			0.18	0.19

TABLE IV. PARALLEL TIME USAGE (IN SECONDS) OF SEISMICLAB'S `FX_DECON` DEMO.

Fx_decon					
	Dual-socket server			KNL	
# threads	MATLAB	OpenMP	TBB	OpenMP	TBB
1	2.92	1.99	1.99	3.75	3.72
2	1.76	1.32	1.32	2.61	2.42
4	1.15	1.01	0.99	1.91	1.72
8	0.86	0.85	0.82	1.49	1.40
16	0.86	0.85	0.75	1.28	1.23
32				1.22	1.21
68				1.25	1.24

D. Further Manual Optimisations

A careful reader will notice that the original MATLAB code of the parabolic moveout example (shown in Section III-B) is not efficiently programmed. One major problem is that the `d` matrix is traversed in a row-major fashion, contrary to the underlying column-major data structure (in both MATLAB and Armadillo). The auto-translated C++ code is consequently also inefficient, even though it is much faster than the original MATLAB code (see Table II). Since the auto-translated C++ code has the same readability, it is possible for an experienced programmer to carry out further optimisations. Figure 4 contains an improved code segment that shows the result of such manual optimisations:

It can be seen that the manually optimised C++ code segment in Figure 4 has swapped the `ih`-indexed for-loop with the `ig`-indexed for-loop. Moreover, the `s` matrix and `ts` vector have now become obsolete and thus removed. When possible, the if-test is lifted out of the innermost for-loop, allowing the compiler to do auto-vectorisation. The hand-optimised code also uses two statically allocated arrays: `s_temp` and `H`, both

 ISBN: 978-1-61208-599-9

```
for (it=1; it<=ntau; it++) {
  for (iq=1; iq<=nq; iq++) {
    time = tau(it-1)+q(iq-1)*arma::square(h/hmax) ;
    memset(s_temp, 0, sizeof(s_temp));
    s2 = 0.;

    for (ih = 1; ih <= nh; ih++) {
       double is_start_double = time(ih - 1) / dt ;
       int is_start = std::floor(is_start_double);
       a = is_start_double - is_start;

       if (is_start-L >= 1 && is_start+L <= nt) {
           for (ig = -(L); ig <= L; ig++) {
              i1 = is_start + ig;
              ss = ((1.- a)*d(i1-1, ih-1) + a*d(i1, ih-1))*H[ig + L];
              s2 += ss*ss;
              s_temp[ig + L] += ss;
            }
       }
       else {
           for (ig = -(L); ig <= L; ig++) {
              i1 = is_start + ig;
              if (i1 >= 1 && i1 < nt) {
                ss = ((1.- a)*d(i1-1, ih-1) + a*d(i1, ih-1))*H[ig + L];
                s2 += ss*ss;
                s_temp[ig + L] += ss;
              }
            }
       }
    }
    s1 = 0;
    for (int i = 0; i < _countof(s_temp); ++i)
       s1 += s_temp[i]*s_temp[i] ;
    S(it-1, iq-1) = s1-s2 ;
  }
}
```

Figure 4. The further improved computational C++ kernel of the parabolic moveout example after manual optimisations.

of length `2*L+1`, where the latter replaces the unnecessary `H` matrix in the original MATLAB code and the auto-translated C++ code.

On a single core, the hand-optimised C++ code runs more than 5 times faster on both the Sandy Bridge CPU and the KNL Xeon Phi processor, as shown in Table V. Although some of the manual optimisations are also applicable to the original MATLAB code, the resulting performance gain is smaller because MATLAB is not a compiled language. Table V details the parallel performance of the hand-optimised C++ code. It remains to be investigated why the hand-optimised OpenMP version runs slower than the TBB counterpart on the KNL Xeon Phi processor (unless all the 68 cores are used).

IV. Related work

To our knowledge, with respect to automated MATLAB-to-C/C++ code translation, the only existing tools are MATLAB Coder [11] and MATISSE [12]. The former is MATLAB's commercial product and makes use of GUI-supported directives to address variable types and shapes, whereas the latter relies on an aspect-oriented programming language (LARA) for initialising variables and specifying their types and shapes. Both tools are thus, to a certain extent, invasive to the original MATLAB code. Moreover, MATISSE does not support parallelisation in the translated C code.

Compared with [7], the *m2cpp* translator discussed in the present paper has been considerably enhanced. For example, *m2cpp*'s preprocessor is now capable of automatically identifying the type of variables used in the input MATLAB code. Moreover, MATLAB functions that have multiple return values can now be handled by *m2cpp*. A very important new feature of *m2cpp* is its capability of adopting multiple threads to parallelise the independent iterations of a designated for-loop, with help of either OpenMP or Intel TBB. The responsibility of ensuring iteration independency lies with the user, who labels the designated for-loops of multi-threading by `%#PARFOR` in the MATLAB source. With respect to performance study, the present paper has included detailed time measurements of both original MATLAB codes and auto-translated C++ codes, using different core counts on a two-socket multicore CPU server and one second-generation Xeon Phi processor. An example of further manual optimisations of auto-translated C++ code has also been provided to show the readability and traceability of *m2cpp* output.

V. Conclusion

Indeed, as the authors of [13] have pointed out, translating MATLAB code to the C/C++ counterpart should be the last option for speeding up MATLAB programs. However, when efficient serial programming practices in the MATLAB context are insufficient or even not applicable, code translation can be the remedy. The four examples from SeismicLab show that the auto-translated Armadillo code in C++ has a clear performance

 ISBN: 978-1-61208-599-9

TABLE V. TIME USAGE COMPARISON BETWEEN ORIGINAL/AUTO-TRANSLATED AND HAND-OPTIMISED CODES FOR THE `PARABOLIC_MOVEOUT` DEMO.

Serial performance	MATLAB (on server)	C++ (on server)		C++ (on KNL)	
Original/auto-translated	261.7	59.5		133.2	
Hand-Optimised	119.1	10.8		26.5	
Parallel performance	N/A	OpenMP	TBB	OpenMP	TBB
2 threads		5.64	5.88	24.52	13.70
4 threads		2.97	3.07	12.79	7.00
8 threads		1.68	1.72	6.64	4.18
16 threads		1.03	1.09	3.67	2.46
32 threads		0.87	0.83	2.15	1.39
68 threads				1.24	1.17

advantage over the MATLAB counterpart, except when the computational core of a MATLAB code already internally uses multi-threaded and highly optimised math libraries such as Intel's MKL. The automated *m2cpp* translator is not only 100% non-invasive for serial MATLAB code, but also retains readability of the resulting C++ code, giving rise to traceability of every algorithmic structure and detail. This in turn allows further manual code optimisations if needed.

Applying *m2cpp* to more real-world examples will be the best way to uncover new limitations and/or inefficiencies of the MATLAB-to-C++ translator, thereby prompting further improvements of *m2cpp*. We thus hope that the open-source software of *m2cpp* [14] will encourage more testing, especially among industrial users. A future research topic that concerns parallelising *m2cpp*-translated C++ code, in addition to the currently adopted data-parallel approach, is how to automatically identify independent tasks in the input MATLAB code and thereafter insert task-parallel execution in the auto-translated C++ code.

ACKNOWLEDGMENT

Dr. Jonathan Feinberg is acknowledged for his important contributions to an earlier version of the MATLAB-to-Armadillo translator. The translator was developed within the EMC^2 project [15] – *Embedded multi-core systems for mixed criticality applications in dynamic and changeable real-time environments*. The research and development work has received funding from Research Council of Norway and ARTEMIS Joint Undertaking (JU) under grant agreement No. 621429.

REFERENCES

[1] H. Inoue and T. Nakatani, "Performance of multi-process and multi-thread processing on multi-core SMT processors," in 2010 IEEE International Symposium on Workload Characterization (IISWC), Dec. 2010, pp. 1–10.

[2] C. Sanderson, "Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments," NICTA, Tech. Rep., Oct. 2010.

[3] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," The Journal of Open Source Software, vol. 1, no. 2, 2016, p. 26.

[4] G. Sharma and J. Martin, "MATLAB®: A language for parallel computing," International Journal of Parallel Programming, vol. 37, no. 1, 2009, pp. 3–36.

[5] B. Chapman, G. Jost, and R. van de Pas, Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007.

[6] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, 2007.

[7] G. Y. Paulsen, J. Feinberg, X. Cai, B. Nordmoen, and H. P. Dahle, "Matlab2cpp: A Matlab-to-C++ code translator," in Proceedings of 11th System of Systems Engineering Conference (SoSE), 2016, pp. 1–5.

[8] L. Torczon and K. Cooper, Engineering A Compiler, 2nd ed. Morgan Kaufmann Publishers Inc., 2011.

[9] "ANTLR – ANother Tool for Language Recognition," URL: http://www.antlr.org/ [retrieved: July, 2017].

[10] "SeismicLab: a MATLAB seismic data processing package," URL: http://seismic-lab.physics.ualberta.ca/ [retrieved: July, 2017].

[11] "MATLAB Coder," URL: http://se.mathworks.com/products/matlab-coder/ [retrieved: July, 2017].

[12] J. Bispo and J. M. P. Cardoso, "A MATLAB subset to C compiler targeting embedded systems," Software: Practice and Experience, vol. 47, no. 2, 2017, pp. 249–272.

[13] S. W. Zaranek, B. Chou, G. Sharma, and H. Zarrinkoub, "Accelerating MATLAB algorithms and applications," URL: https://se.mathworks.com/company/newsletters/articles/accelerating-matlab-algorithms-and-applications.html [retrieved: July, 2017].

[14] "Convertion program from Matlab to C++ using Armadillo," URL: https://github.com/emc2norway/m2cpp [retrieved: September, 2017].

[15] "EMC^2 – Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments," URL: http://www.artemis-emc2.eu [retrieved: July, 2017].