C and OpenCL Generation from MATLAB

João Bispo
Faculty of Engineering (FEUP)
University of Porto
Porto, Portugal
jbispo@fe.up.pt

Luís Reis
Faculty of Engineering (FEUP)
University of Porto
Porto, Portugal
ei09030@fe.up.pt

João M. P. Cardoso
INESC-TEC/FEUP
University of Porto
Porto, Portugal
jmpc@fe.up.pt

ABSTRACT

In many engineering and science areas, models are developed and validated using high-level programing languages and environments as is the case with *MATLAB*. In order to target the multicore heterogeneous architectures being used on embedded systems to provide high performance computing with acceptable energy/power envelops, developers manually migrate critical code sections to lower-level languages such as C and OpenCL, a time consuming and error prone process. Thus, automatic source-to-source approaches are highly desirable. We present an approach to compile *MATLAB* and output equivalent C/OpenCL code to target architectures, such as GPU based hardware accelerators. We evaluate our approach on an existing *MATLAB* compiler framework named MATISSE. The OpenCL generation relies on the manual insertion of directives to guide the compilation and is also capable of generating C wrapper code to interface and synchronize with the OpenCL code. We evaluated the compiler with a number of benchmarks from different domains and the results are very encouraging.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – Code generation, Compilers, *Optimization, Retargetable compilers*

General Terms Performance, Experimentation, Languages

Keywords MATLAB, source-to-source compiler, C programming language, embedded systems, GPUs, OpenCL

1. INTRODUCTION

MATLAB [1] is a high-level programming language supported by an interactive computing environment used in many domains in engineering and science. The language and its associated environment are ubiquitously used to quickly develop and evaluate models. *MATLAB* is dynamically typed, and is usually interpreted and/or JIT compiled. Ahead-of-time compilation of MATLAB is challenging because the information about the types and shapes (i.e., number and size of matrix dimensions) of variables might be only known at runtime. In order to improve performance, *MATLAB* has built-in support for pre-compiled and optimized functions and there have been efforts to add support for multicore and GPU (Graphics Processing Unit) architectures (see, e.g., [2][3]). In many high-performance and embedded system settings, however, the use of a *MATLAB* runtime environment might be 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 translation of the base or original *MATLAB* code/model to a programing language such as C/C++. This implementation must then in turn be validated against the output of the *MATLAB* code resulting in a lengthy and error prone process that further complicates the overall application development cycle and increases its cost. The existence of two source codes - the original model-driven *MATLAB* code and the implementation-driven C/C++ code - also exacerbates maintenance costs. An alternative solution is the automatic translation of *MATLAB* to the target programming language, e.g., by *MATLAB* to C code translators (e.g., [4]).

 ACM 978-1-4503-3196-8/15/04$15.00 http://dx.doi.org/10.1145/2695664.2695911

Despite their inherent advantages, typical automatic approaches have the disadvantage of providing low support to control and guide the code translation. The code generation is commonly based on directives (GUI based in the case of the *MATLAB* Coder) to specify variable types, shapes, and target domain/microprocessor. When dealing with the myriad of target architectures and toolchains in embedded systems, this approach presents a low level of flexibility. For example, the style of the generated C code might need to be tuned to the toolchain as is the case when targeting C code for hardware compilers. Furthermore, the target platform may require specific code transformations and/or specific programming languages such as CUDA [5] or OpenCL [6] when dealing with GPGPUs and/or FPGAs [7].

Our approach focuses on a compiler, named MATISSE [8][9], to generate C/OpenCL code directly from *MATLAB*. MATISSE is being developed as a modular and flexible compiler framework, which includes custom Intermediate Representations (IRs) for *MATLAB*, C and OpenCL code. 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. The result is a synergy between compiler analysis and the information provided by the user, which 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 describe our approach to translate *MATLAB* code to mixed C/OpenCL implementations. We evaluate this approach with a number of representative examples and the results achieved are very encouraging. This paper makes the following contributions:

- It proposes the use of OpenACC [10] based directives to partitioning the *MATLAB* programs between the GPP (General Purpose Processor) and the target GPU, and to instruct the OpenCL generator;

- It presents the current phases needed to allow an efficient OpenCL generation;
- It evaluates the approach with a number of *MATLAB* functions and targeting two distinct GPU architectures.

The remainder of this paper is organized as follows. Section 2 presents the MATISSE compiler and describes the proposed directives for OpenCL generation. Section 3 presents the OpenCL compilation phases. Section 4 shows some experiments performed using MATISSE. Section 5 describes related work and finally, Section 6 concludes this paper and describes future work.

2. THE MATISSE MATLAB COMPILER

MATISSE [8][9] is a *MATLAB* source-to-source compiler framework specially targeting embedded systems. Figure 1 presents the overall flow of the compiler. The input *MATLAB* files are translated to an abstract syntax tree (AST) based *MATLAB* IR (intermediate representation). This IR is then used for optimizations/transformations and for code generation. The compiler is being developed in a way to ease the integration of code generators. At the moment, it includes code generators for *MATLAB*, C, and OpenCL.

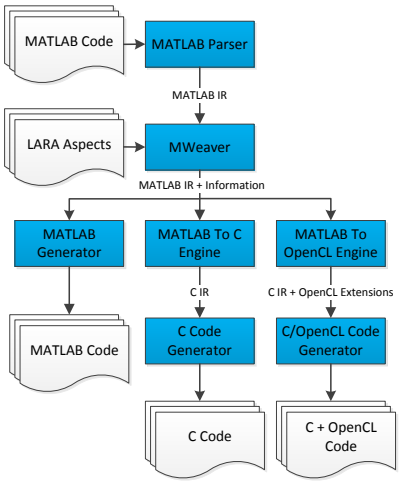

Figure 1. Overview of the MATISSE compiler framework.

MATISSE generates *MATLAB* code for validation, testing, monitoring, and specialization, and C/OpenCL code to be used by third-party design-flows targeting software/hardware systems. It is able to generate customized C code for a particular target without modifying a significant subset of original *MATLAB* code. Aspect files (using LARA [11]) allow users 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 supports the use of directives to guide the compilation flow and allow users fine-grained control over which portions of the *MATLAB* code are migrated to the OpenCL device and which are executed sequentially on a GPP. The current directives (see Table 1) are based on OpenACC [10] and are embedded in lines interpreted as comments by *MATLAB* environments (comments starting with `%acc` specify directives). This allows other *MATLAB* tools to accept *MATLAB* code with our directives (ignoring them).

Table 1. Directives currently supported by the compiler.

Directives	Description
`parallel loop`	Indicates that a *for* loop should be compiled to OpenCL and its iterations can be executed in parallel.
`loop`	Indicates that an inner *for* loop can be executed in parallel.
`ignore`	Indicates a code section to be ignored by MATISSE when generating OpenCL code.
`end`	Indicates the end of a `parallel`, `parallel loop` or `ignore` block.
`barrier`	Indicates a memory barrier. The compiler recognizes two different types of barriers, `barrier local` and `barrier global`. These are equivalent to the OpenCL barrier function when using the argument `CLK_LOCAL_MEM_FENCE`, and `CLK_GLOBAL_MEM_FENCE`.
`copyin(vars)`	A list of variables that should be copied for use by the OpenCL kernel. If only a portion of an array/matrix is used in a parallel loop region, then the range copy syntax (e.g., `A(1:100)`) can be used.
`copyout(vars)`	A list of variables that should be copied from the OpenCL device to the host device once the kernel finishes execution.
`Reduce (varname:operation)`	A list of scalar variables that are computed partially by multiple threads. Two operations (`+` for sum and `*` for product) are supported.
`local(vars)`	A list of matrix variables that are different for each local group. The values of these matrices are not copied to the device and changes to them are discarded once the kernel execution ends.
`local_id(vars)`	For each iteration, the value of these variables will be set to the respective local ID. The value of these variables remains the same outside of the `parallel loop` block.
`group_id(vars)`	Similar to `local_id`, but the variables store the respective group IDs.
`local_size(constants)`	The local size that should be used.

Erro! A origem da referência não foi encontrada. shows a simple *MATLAB* function to compute the element-wise square of a given matrix (equivalent to *MATLAB*'s `A.*A`). The code is extended with directives which indicate that the `for` block must be compiled to OpenCL, the *A* matrix must be copied to the OpenCL device (`readonly` because the input matrix itself is not modified), and *Y* should be copied back to the host once the GPU execution is finished.

```
function Y = elementwise_square_matrix(A)
  Y = zeros(size(A));
  %acc parallel loop copyin(readonly A) copyout(Y)
  for i = 1:numel(A)
     Y(i) = A(i) * A(i);
  end
  %acc end
end
```

Figure 2. MATLAB code example with directives.

3. COMPILATION PHASES

The current version of the compiler consists in multiple phases. We describe next the most important ones regarding the generation of OpenCL code. They are applied only on code between `%acc` pragmas.

3.1 Expression Decomposer

The expression decomposer is a compiler phase used to decompose expressions without changing the behavior of the program,

in order to simplify the subsequent phases. It introduces new temporary variables, e.g., the code in Figure 3(a) is transformed to the code in Figure 3(b). As a result of these changes, the following post-conditions are true:

- Every array access or function call is directly assigned to a variable (never used as part of another expression). This is important as it simplifies function inlining. In order to ensure that the above is true, `while(x)` loops become `while(1)` with an additional if statement that breaks the loop when the condition is false.
- All `for` loops become in the form `for varname=1:1:x`. This makes trivial to compute the number of iterations of any loop if `x` is known. This is useful for the OpenCL backend, which must be able to compute the number of iterations of every `parallel loop` before it can execute it.

3.2 Function Inliner

The function inliner replaces all function calls with the function body. Regarding the identification of the identifiers related to functions, name resolution is performed by the compiler and by the user. After the inliner has been executed, the code contains no function calls so the code generator can assume that every time `A(x)` appears it is a matrix access expression.

The inliner also needs to deal with the presence of return statements in the function to be inlined. These return statements are translated to code able to make the execution flow of previous return statements without returns (auxiliary variables and `if` and `break` statements are introduced). Figure 4(a) shows an early return. Before inlining, the compiler transforms the code such as in Figure 4(b). Finally, since the caller and the callee functions may use variables with the same names, callee variables must be renamed when they are inserted into the caller function.

(a)

```
for w = 1:2:x
  while w < 10
    y = y + f2(w); % f2 is a function
    w = w + 1;
  end
end
```

(b)

```
tmp_MaxValue1 = (x - 1) / 2 + 1;
tmp_Iterations1 =
floor(tmp_MaxValue1);
for tmp_LoopIndex1 =
1:1:tmp_Iterations1
  while 1
    if ~(w < 10)
      break;
    end
    w = (tmp_LoopIndex1 - 1) * 2 + 1;
    tmp_AccessCall1 = f2(w);
    y = y + tmp_AccessCall1;
    w = w + 1;
  end
end
```

Figure 3. Examples of decomposition of expressions: (a) original code; (b) decomposed code.

(a)

```
if x > 0
  y = 1;
  return;
end

y = 2;
```

(b)

```
k_return = 0;
if x > 0
  y = 1; k_return = 1;
end
if ~k_return
  y = 2;
end
```

Figure 4. Transformation before inlining: (a) original code; (b) transformed code.

3.3 Region Outlining

At this point, the compiler is nearly ready for generating OpenCL code. According to the directives, parts of the original code are translated to OpenCL code, other parts to C code, and wrapper C code is also generated so that the different parts can communicate and synchronize. To simplify, the code generation backend is selected on a per-function basis. However, the compiler may need to split functions with both sequential and parallel regions. This is instructed by the directive region outliner.

The outliner detects all directive region blocks and generates a new function for each one. The new functions are then compiled by the OpenCL backend and the main function (the caller) is compiled by the C backend. At the moment the `copyin`, `copyout` and `reduce` parameters are used to define which variables should be used as function arguments and which variables should be returned by the outlined functions.

3.4 OpenCL/C Code Generation

At this point, we have the MATLAB code organized in functions, ones for the C backend and others for the OpenCL backend. When the OpenCL backend generates code for a function, it outputs two functions: the OpenCL code, and the wrapper, which is a pure C function with OpenCL API calls to run the parallelized code and copy the data in and out of the OpenCL device.

The OpenCL backend compiles the code on a per-statement basis using a simple type inference engine that assumes each *MATLAB* variable remains of the same type for the full duration of its lifetime. Note that the information regarding types and shapes can be also provided by aspect files [8].

Table 2. Benchmarks.

Benchmark	Description	Input Size
dotprod	Dot product of complex 3D matrices [8].	2048×2048×20
dilate	Code from a Stereo navigation application [8].	2048×2048
matmul	Simple O(n^3) floating point matrix multiplication.	1024×1024
matmul_nv	OpenCL Matrix Multiplication code sample by NVIDIA [19], manually converted to MATLAB with directives.	1024×1024
matmul_nv2	matmul_nv modified to allow the OpenCL driver to perform a few extra optimizations, notably loop unrolling.	1024×1024
monte_carlo	Monte Carlo simulation based on MathWorks example [20], Modified to use pseudo-random number generators [21] and a Box-Muller transform [22].	100,000
rgb2yuv	RGB to YUV conversion.	2000×2000
subband	Based on MPEG2 encoder [8].	128×64k

4. EXPERIMENTAL RESULTS

We conducted an evaluation of the compiler in terms of its capacity to generate efficient C/OpenCL code from *MATLAB*.

4.1 Benchmarks and Environment

All tests were executed on a desktop computer running Windows 8.1 Enterprise Edition 64-bits with an AMD A-10 7850K CPU running at 4.10 GHz, 8 GB RAM. This computer has two GPUs (one integrated with the CPU and the other discrete). The inte-

grated GPU is a Radeon R7 Graphics and the discrete GPU is an AMD Radeon R9 280X. This computer is running the official AMD drivers, Catalyst version 14.4, Platform Version `OpenCL 1.2 AMD-APP`. All code was compiled on Windows with *gcc* 4.8.2 (64-bits MinGW distribution), and –O3.

In the experiments included in this paper, we use the benchmarks briefly described in Table 2. Although the compiler supports double precision floating-point numbers, the results presented herein consider single-precision, as the OpenCL double-precision extension (`cl_khr_fp64`) is not always available.

4.2 Code Size Impact

MATLAB code occasionally needs to be modified in order to be compiled by MATISSE. At the moment, the OpenCL generation requires the addition of directives and the replacement of matrix operations by the equivalent `for` loops, i.e., translating "idiomatic" code to a "non-idiomatic" version.

Figure 5 compares the size of the "non-idiomatic" *MATLAB* code to the equivalent "idiomatic" size, in terms of lines of code with directives and statements. The codes sizes of the matrix multiplication benchmarks (`matmul`, `matmul_nv` and `matmul_nv2`) are not compared because the "idiomatic" code for them is the built-in product operator, and there is a dependence to the matrix multiplication algorithm used. As Figure 5 shows, we add a significant number of lines (increase from 1.4× to 3.3× for the 5 functions) to adapt the codes for the OpenCL generator. For the `monte_carlo` benchmark, the "idiomatic" code excludes the Random Number Generation functions (except the initial seed definition), since *MATLAB* includes its own built-in pseudo-random number generators. If the same code for the random number generation was used on both versions, then the backend version would contain only 7.9% more code than the "idiomatic" version.

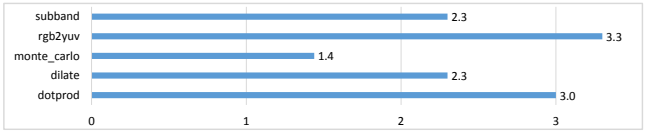

Figure 5. Relative increase in terms of lines of code of modified MATLAB compared to "idiomatic" MATLAB.

Most of the code modifications are due to limitations of the OpenCL generator (e.g., matrix operations need to be converted to `for` loops). These results highlight the need to extend the compiler with an additional OpenCL phase, which can insert pre-defined code templates for some of the *MATLAB* "idiomatic" operations. Note that the C backend supports most of the benchmarks in their original form.

4.3 Performance Comparisons

We have evaluated the performance of the C and OpenCL backends for the benchmarks in Table 2. For `matmul`, we prepared a special version, based on an OpenCL matrix multiplication example provided by Nvidia [19], which uses a significantly different algorithm. This version represents what we can do if we write custom code without being driven by *MATLAB* "idiomatic" goals, and optimizing for OpenCL by using implementation details such as barriers and local memory.

Table 3**Erro! A origem da referência não foi encontrada.** shows a comparison between the matrix multiplication versions, when considering total execution time including all overheads. We compared the speedup of code obtained from hand-optimized *MATLAB* code (`matmul_nv` and `matmul_nv2`) against a naïve version of the same algorithm, also obtained from *MATLAB* (`matmul`), and a manually written OpenCL version (`OpenCL`).

Table 3. `matmul_nv` speedups, compared with `matmul` and manual OpenCL.

Device	matmul_nv		matmul_nv2	
	matmul	OpenCL	matmul	OpenCL
CPU	1.98	0.39	3.94	0.78
Integr. GPU	0.90	0.06	3.66	0.25
Discrete GPU	0.35	0.14	0.66	0.26

Our original conversion of the Nvidia matrix multiplication example to *MATLAB* (`matmul_nv`) resulted significantly worse (2.5× on the CPU to 16.5× on the integrated GPU) than the original code. We examined the code generated by the OpenCL driver and concluded that this was due to some optimizations (notably loop unrolling) not being performed for the OpenCL generated code. We then developed a modified version (`matmul_nv2`) that more closely matches the Nvidia version and performs better than `matmul_nv` on all targets we tested. Considering `matmul_nv2` and comparing with a naïve implementation (`matmul`), the performance of the OpenCL code based on the optimized *MATLAB* code increases significantly when the code is executed on the CPU (3.9×) and on the integrated GPU (3.7×). However, it has a 1.5× slowdown when executed on the discrete GPU. This slowdown is not observed in the original Nvidia version, suggesting there is still margin for improvement. The first `matmul_nv` was actually slower than the naïve version on the discrete GPU, as the lack of loop unrolling meant that the reduced memory accesses came at the cost of a very significant increase in the number of executed vector instructions. Although the memory accesses are faster on the discrete GPU than on the integrated GPU, the reduction in memory access time was unable to compensate for the increased number of computations. The manual OpenCL code is faster than the OpenCL code generated from *MATLAB* `matmul_nv2`, and the performance gap for the matrix multiplication for this example goes from 1.3× (CPU) to 4× (Discrete GPU) slower, depending on the target device.

Although the MATISSE OpenCL backend allows the programmer to manually specify explicitly the local size for each loop, we let the OpenCL driver to automatically decide the local size to use for the benchmarks mentioned in this section, except for `matmul_nv` and `matmul_nv2`.

We have measured two variants of the OpenCL code. Each test was executed 30 times and the results were averaged. The speedups reported in Figure 6(a) include the overhead of data transfer and other driver calls such as setting kernel arguments and obtaining the OpenCL kernel instance. The execution time was measured using the `QueryPerformanceCounter` Windows API function. The speedups reported in Figure 6(b) are relative to the time spent *only* on kernel execution, using the built-in OpenCL profiling capabilities.

We were able to significantly speedup four of the six MATLAB benchmarks we tested. Although the kernel execution times showed speedups in all except `subband`, the overhead from data transfers significantly worsens most results, causing severe slowdowns in two cases (`dotprod` and `subband`). The highest

speedups were achieved by `matmul` and `monte_carlo`, 1,010× and 976×, respectively, using the discrete GPU. For these two benchmarks, the performance achieved by the discrete GPU is higher than with the integrated GPU. This is explained by the small amount of data transferred compared to the computations that need to be performed for these two examples. Additionally, by tuning the `local_size` directive in the `monte_carlo` benchmark, we were able to achieve a performance improvement of 17% relative to the automatic local size. For the remaining benchmarks, the impact of setting the `local_size` parameter was negligible.

Currently, the MATISSE OpenCL backend does not take advantage of concurrent execution between OpenCL devices and GPPs. When an OpenCL program is executed, the generated code waits for it to be finished before proceeding with the remaining computations, even if the results of the GPU computations are not needed yet. Exploring concurrency at this level may significantly reduce the overhead of data transfers on performance.

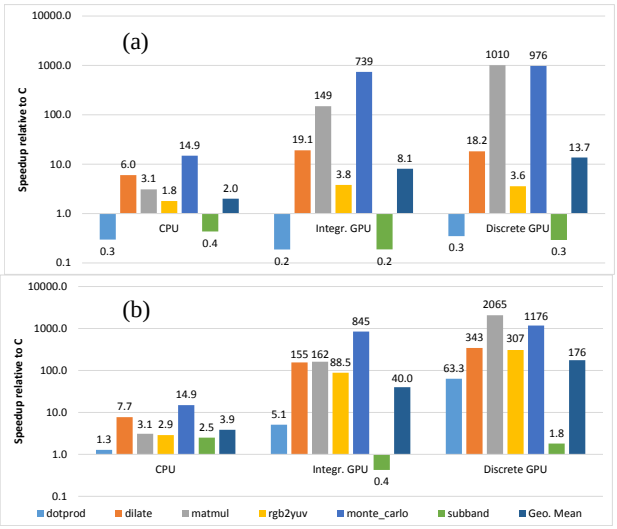

Figure 6. Speedup of OpenCL over C: (a) including overheads; (b) kernel time only.

Another limitation is related to when two or more GPU computations need to be performed in sequence. Currently, the generated code waits for the first computation to be finished, then loads the results to the CPU, transfer them again to the GPU, and only then runs the second computation. A future version of the tool would focus on eliminating unnecessary data transfers.

Regardless the lack of more advanced analysis and optimizations, our current results are very promising and show significant performance boosts from exploiting GPU parallelism, in some cases with negligible changes to the original *MATLAB* source code.

Finally, we have tested the same OpenCL code on an Odroid-XU+E board, which has an Exynos 5410 processor with ARM's big.LITTLE configuration and a PowerVR SGX544MP3 GPU. We run the OpenCL code for the same six benchmarks on the GPU, except for the modified matrix multiplications, which used a block size (i.e., 16) incompatible with Odroid (only supports block size 1). The results showed a small speedup of 1.01× (`monte_carlo`) and several slowdowns, up to 13× (`subband`), and with a geometric slowdown mean of 2.6×. We attribute these results to the significantly different resources available on the PowerVR GPU (e.g., much smaller shared memory). They highlight the need to deal more efficiently with the different characteristics of embedded platforms, and motivate us to invest in a multi-target specialization of the OpenCL code generator.

5. RELATED WORK

The translation of *MATLAB* code to programming languages more amenable to be efficiently compiled to the target architectures has been the focus of many research efforts. De Rose and Padua developed the FALCON environment [12] 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 [13]. Joisha and Banerjee [14] focused on type and shape inference techniques. Researchers have also relied on a mix of type inference approaches and user's provided information. 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 Embedded Coder [4].

The popularity of the *MATLAB* language is also reflected in the similar languages that have been proposed. Examples of those languages are *Scilab* [15] and *Octave* [16]. A *Scilab* to C translator [17], named *Sci2C*, has been developed. *Sci2C* focuses entirely on embedded systems, and is completely dependent on annotations embedded in the *Scilab* code to specify data sizes and precisions. MATISSE 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 MATISSE 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 [18] with loop vectorization and partial evaluation of types and values. The use of GPUs in the context of *MATLAB* has been mainly approached by the use of APIs (see, e.g., [23]). The most relevant work to our C/OpenCL approach are the compilers that translate *MATLAB* to CUDA. MEGHA [24] is a compiler which processes *MATLAB*/Octave scripts and generates CUDA and C++ code. MEGHA uses heuristics to decide which portions of the code should be executed on the CPU and which should be offloaded to the GPU. The authors report a geometric mean speedup of 12.2× over *MATLAB* for a number of benchmarks. Chun-Yu Shei et al. [25] presents another *MATLAB*/Octave to CUDA compiler. It generates both C++ and CUDA code. However, unlike MEGHA, portions of the resulting code remain in *MATLAB*. The reported speedups range from 1.5× to 17×.

6. CONCLUSIONS

This paper presented our approach to translate *MATLAB* code to C/OpenCL code. We described the general flow, the transformations/optimizations performed by the compiler. Our current approach to generate OpenCL code uses OpenACC-based directives to guide the partitioning process between the GPP and the GPU and to instruct the generation of OpenCL code. The experiments with six benchmarks reveal promising performance results.

The execution of the C/OpenCL generated implementations in a GPU achieved a geometric mean speedup of 14× and 176× over the generated C code implementations, for total execution times and kernel-only times, respectively. We also compared code for matrix multiplication generated from optimized *MATLAB* with a manual version written originally in OpenCL, and the performance gap was between 1.3× and 4×, depending on the target device, in favor of the manual version. Finally, preliminary results using an embedded device show a slowdown geometric mean of 2.6× and reveal the need for multi-target specialization of OpenCL code. Ongoing work is focused on enhancements of the C and OpenCL generators. Future work will address OpenCL generation aware of the target GPU architecture. Finally, as FPGA manufacturers are now supporting OpenCL, we will evaluate the performance of the OpenCL generated by MATISSE and the specific optimizations and code styles needed to improve FPGA results.

7. ACKNOWLEDGMENTS

This work was partially supported by Project NORTE-07-0124-FEDER-000062, funded by the North Portugal Regional Operational Programme (ON.2), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by National funds through FCT.

8. REFERENCES

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

[2] N.T. Bliss, J. Kepner, H. Kim, A. Reuther, "pMATLAB: Parallel MATLAB Library for Signal Processing Applications," in IEEE Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP'07), Vol. 4, April 2007, pp. 15-20.

[3] S. Samsi, V. Gadepally, A. Krishnamurthy, "MATLAB for Signal Processing on Multiprocessors and Multicores," in IEEE Signal Processing Magazine, Vol. 27, Issue 2, March 2010, pp. 40-49.

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

[5] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors," in SIGPLAN Not., 46(6):152–163, June 2011.

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

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

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

[9] J. Bispo, L. Reis, and J.M.P. Cardoso, "Multi-Target C Code Generation from MATLAB," in ACM/SIGPAN Int'l Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY'2014), Edinburgh UK, June 2014.

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

[11] J.M.P. Cardoso, P.C. Diniz, J.G.F. Coutinho, and Z. Petrov, ed. Compilation and Synthesis for Embedded Reconfigurable Systems: An AspectOriented Approach. Springer, 2013.

[12] 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.

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

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

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

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

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

[18] 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'l Workshop on Source Code Analysis and Manipulation (SCAM'03), 26-27 Sept. 2003, pp. 141-150.

[19] NVIDIA. NVIDIA OpenCL SDK Code Samples, 2014.

[20] MathWorks. Using GPU ARRAYFUN for monte-carlo simulations – MATLAB & Simulink example. 2014.

[21] E.W. Weisstein. "Linear Congruence Method," From MathWorld – A Wolfram Web Resource, 2014.

[22] E.W. Weisstein. "Box-Muller Transformation," from MathWorld – A Wolfram Web Resource. 2014.

[23] MathWorks. MATLAB GPU computing support for NVIDIA CUDA-enabled GPUs. 2013.

[24] A. Prasad, and R. Govindarajan, "Compiler optimizations to execute MATLAB programs on memory constrained GPUs," In 1st Asia-Pacific Prog. Lang. and Compilers Workshop (APPLC'2012), Beijing, China, 14 Jun. 2012.

[25] Chun-Yu Shei, A. Yoga, M. Ramesh, and A. Chauhan, "MATLAB parallelization through scalarization," In 15th Work. on Inter. between Compl. and Comp. Arch. (INTERACT'11), San Antonio Texas, USA, Feb. 2011. (pp. 44-53).