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 multi-core heterogeneous architectures being used on embedded systems to provide high performance computing with acceptable energy/power envelopes, 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 programing language, embedded systems, GPUs, OpenCL

1. INTRODUCTION
MATLAB [1] is a high-level programing 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 programing language, e.g., by MATLAB to C code translators (e.g., [4]).

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 programing 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 GPU (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.

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.

<table>
<thead>
<tr>
<th>Directives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel loop</td>
<td>Indicates that a for loop should be compiled to OpenCL and its iterations can be executed in parallel.</td>
</tr>
<tr>
<td>loop</td>
<td>Indicates that an inner for loop can be executed in parallel.</td>
</tr>
<tr>
<td>ignore</td>
<td>Indicates a code section to be ignored by MATISSE when generating OpenCL code.</td>
</tr>
<tr>
<td>end</td>
<td>Indicates the end of a parallel, parallel loop or ignore block.</td>
</tr>
<tr>
<td>barrier</td>
<td>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.</td>
</tr>
<tr>
<td>cop(yin)(vars)</td>
<td>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(i:i+100)) can be used.</td>
</tr>
<tr>
<td>copyout(vars)</td>
<td>A list of variables that should be copied from the OpenCL device to the host device once the kernel finishes execution.</td>
</tr>
<tr>
<td>Reduce</td>
<td>A list of scalar variables that are computed partially by multiple threads. Two operations (+ for sum and * for product) are supported.</td>
</tr>
<tr>
<td>var-name-opera</td>
<td>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.</td>
</tr>
<tr>
<td>local(vars)</td>
<td>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.</td>
</tr>
<tr>
<td>local_id(vars)</td>
<td>Similar to local_id, but the variables store the respective group IDs.</td>
</tr>
<tr>
<td>local_size(const)</td>
<td>The local size that should be used.</td>
</tr>
</tbody>
</table>

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.

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

![Figure 4. Transformation before inlining: (a) original code; (b) transformed code.](image)

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 outline.

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, one 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.](image)

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.](image)

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 | Code Size Impact
<table>
<thead>
<tr>
<th>Device</th>
<th>Code Size Impact</th>
<th>Code Size Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Increase</td>
<td>Increase</td>
</tr>
<tr>
<td>Integr. GPU</td>
<td>0.99</td>
<td>0.39</td>
</tr>
<tr>
<td>Discrete GPU</td>
<td>0.35</td>
<td>0.14</td>
</tr>
</tbody>
</table>

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 naive 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 naive 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.

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 Ondroid-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 Ondroid (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 Sc2C, has been developed. Sc2C 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 Sc2C as it is able to generate C code without polluting the original code. Furthermore, Sc2C 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 GPU 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.

Figure 6. Speedup of OpenCL over C: (a) including overheads; (b) kernel time only.
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 code performance, in favor of the manual version.

The National Strategic Reference Framework (NSRF), through the European Program (ON.2), under the National Strategic Reference Framework (FEDER), 7.

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