Automatic Generation of Parallel C Code for Stencil Applications Written in MATLAB

Johannes Spazier Steffen Christgau Bettina Schnor

Institute for Computational Science, August-Bebel-Straße 89, 14482 Potsdam, Germany
{spazier,christgau,schnor}@cs.uni-potsdam.de

Abstract

High-level programming languages such as MATLAB are widely used in scientific domains to implement prototypes based on mathematical models. These prototypes are finally often re-implemented in low-level languages to reach execution times required for the operational use. In order to exploit latest hardware architectures additional effort is necessary to add parallelism to the applications. This paper presents performance results of an automatic translation from a MATLAB subset into efficient parallelized C code for different architectures: multicores, compute clusters, and GPGPUs. We present the first compiler that generates native MPI code from MATLAB source and thereby showing significant performance improvements. The evaluation is done for two stencil applications which use different communication patterns, a Game-of-Life application and a Tsunami simulation. For the Game-of-Life application, the generated parallel code shows nearly optimal speedup. The generated parallel code of the Tsunami simulation reaches the performance of the available parallel reference implementations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Code generation, Compilers, Optimization, Retargetable Compilers; D.2.2 [*Software Engineering*]: Design Tools and Techniques

General Terms Languages, Performance, Measurement

Keywords MATLAB, source-to-source compiler, stencil applications, High Performance Computing, Message Passing Interface, OpenMP, OpenACC

1. Introduction

High-level programming languages such as MATLAB are widely used in scientific domains to implement prototypes based on mathematical models. Such languages provide straight-forward mathematical syntax for complex operations (e.g. matrix and vector operations) which are tedious or complex in more low-level languages, like C. Developed prototypes are often re-implemented in low-level languages to reach execution times required for the operational use. In order to exploit current hardware architectures additional effort is necessary to add parallelism to the applications. This paper presents performance results of an automatic translation from a MATLAB subset into parallelized C code. Different parallelization approaches were applied to utilize shared memory systems with OpenMP, distributed memory systems with MPI, and accelerator devices with OpenACC.

Applications often involve stencil computations, e.g. due to the numerical methods applied in them (Christen et al. 2011b). However, most of these computations are inherently data-parallel. Thus, the focus of this paper is on the parallelization of such applications. The evaluation of the prototype StencilPaC is done for two stencil applications which use different communication patterns, a Game-of-Life application and a Tsunami simulation. The performance is not only compared against the given MATLAB implementation, but also against hand-written parallel reference implementations if available.

The next section presents the characteristics of the two stencil benchmarks. Before StencilPaC generates parallel code, the MATLAB code is translated into sequential C code which is described in Section 3. The underlying concepts of the parallelization approach are presented in Section 4, 5 and 6. Related work is surveyed in Section 7. Finally, Section 8 concludes the paper.

ARRAY'16, June 14, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4384-8/16/06...$15.00
http://dx.doi.org/10.1145/2935323.2935329

2. Stencil Applications

In order to ensure the correctness and to evaluate the quality of the compiler generated code, various benchmarks are performed throughout the paper. This chapter introduces the characteristic properties of the two stencil applications and their parallel reference implementations.

The two applications differ in their stencil (three-point versus 9-point-stencil) and their memory demands. While the Tsunami simulation uses data from seven matrices to perform the stencil update, the Cellular Automaton needs only one matrix. Therefore, the Tsunami simulation shows a memory-bound behavior (Christgau et al. 2015). The Cellular Automaton on the other hand is compute-bound which results in a very good scaling for the parallel version (Christgau et al. 2011). Further, the MATLAB code of the Cellular Automaton makes use of MATLAB's `arrayfun` function.

2.1 EasyWave

The first example is a real-world application called *EasyWave* that is used in the field of early warning to simulate the generation and propagating of tsunamis. The algorithm computes shallow water equations in linear approximation allowing a prompt access to the simulation results. *EasyWave* was developed at the German Research Center for GeoSciences (GFZ) in Potsdam (Babeyko 2012). The computation is based on a two-dimensional grid consisting of multiple variables. These variables reflect the physical properties of the grid cells which are updated over time according to a given stencil pattern.

```
while it <= time_limit
  % height update
  h(j,i) = h(j,i) - cR1(j,i) .*
           ( fM(j,i) - fM(j,i-1) + fN(j,i) .*
             cR6(j+1,i) - fN(j-1,i) .*
             cR6(j,i) );

  hmax(j,i) = max(h(j,i), hmax(j,i));

  % flux update
  fM(j,i) = fM(j,i) - cR2(j,i) .*
            ( h(j,i+1) - h(j,i) );
  fN(j,i) = fN(j,i) - cR4(j,i) .*
            ( h(j+1,i) - h(j,i) );

  it = it + dt;
end
```

Figure 1. Main loop of EasyWave from the MATLAB implementation.

The computation carried out on the grid is repeated in a time-loop and divided into two parts: update of wave heights and update of reverse fluxes. Due to the employed numerical scheme the wave and flux updates require a three-point stencil, i. e. the update of a grid cell in a time step requires the current data of the grid cell and other data of two of its eight neighbors (Moore neighborhood). Additionally, the stencils differ in which cells are included in the computation. For wave update, the upper and left neighbors are used, whereas the flux update includes the lower and right neighbor cells (see Figure 1).

There exist different *EasyWave* implementations which serve as reference implementations: a sequential C++ version and different parallel versions (OpenMP, MPI, CUDA, OpenACC). Algorithmic details can be found in (Christgau et al. 2015) and (Christgau et al. 2014) including benchmark results on different hardware architectures. Previous work in (Christgau et al. 2015) has shown that *EasyWave* provides low operational intensity and thus is clearly memory bound. The scaling of the parallel versions is therefore likely limited by the memory bandwidth. Although the manual implementations are carefully optimized, they do not involve the advanced cache-aware algorithm presented in (Christgau et al. 2015).

The main loop of *EasyWave* from the MATLAB implementation, which consumes most of the compute time and which is parallelized by StencilPaC, is shown in Figure 1.

Two distinct scenarios are considered in the benchmarks of this paper. Their grid size amounts to 2701×2446 which represents a realistic input set for the application. The two scenarios differ only in simulation time, which is between 60 and 180 minutes. The runtimes of the MATLAB code for these two scenarios are 375 s and 1180 s respectively.

2.2 Cellular Automaton

The second application which is considered simulates a Cellular Automaton and is used for teaching MPI programming (Sanders and Worsch 1997). It is based on a two-dimensional grid of binary cells which are either set to 0 or 1. The state of a cell changes over time according to a predefined update rule, which is implemented with a lookup table. The update involves states of all cells from the Moore neighborhood, i.e. from all eight surrounding cells and the central cell (9-point-stencil). The sum of all neighboring cells is used as an index to an array containing the next cell state. The MATLAB code (see Figure 2) uses the `arrayfun` function that applies a function to each element of an array. Further, the application assumes cyclic boundary conditions which virtually turn the grid into a torus.

```
while it <= its
  % torus boundary exchange
  grid(yall, 1)     = grid(yall, xsize-1);
  grid(yall, xsize) = grid(yall, 2);
  grid(1, xall)     = grid(ysize-1, xall);
  grid(ysize, xall) = grid(2, xall);

  % update
  grid(y,x) = arrayfun(@lookup,
    grid(y-1,x-1)+grid(y-1,x)+grid(y-1,x+1)+
    grid(y,  x-1)+grid(y  ,x)+grid(y,  x+1)+
    grid(y+1,x-1)+grid(y+1,x)+grid(y+1,x+1));

  it = it + 1;
end
```

Figure 2. MATLAB source code of Cellular Automaton's kernel.

The default MATLAB implementation omits a data type specification for the grid cells. Thus, the current type deduction of the compiler infers the grid cells to type `int`. However, since the cells only store binary values, it is sufficient to use the smallest available type to represent the grid values, which is `char`. For this reason, a slightly adapted MATLAB version is additionally considered which explicitly forces type `char` for the underling grid. Reducing the size of the grid data type leads to a four times smaller memory consumption at run-time and thus lowers the required memory bandwidth significantly. In the following, only the `char` version of the automaton is considered.

For comparison, a sequential version of the cellular automaton, written in C, is available. In addition, this version was manually parallelized with OpenMP.

3. Automatic Generation of C-Code

The StencilPaC compiler supports a subset of the MATLAB language which it translates into the C language. The supported subset includes basic language elements, such as boolean and numeric scalar expressions, but also covers MATLAB specifics, like ranges, vectors, matrices as well as index accesses that allow easy access to vectors and (sub)matrices respectively. Besides control structures, user-defined functions and a selection of predefined MATLAB functions are supported as well. To implement the proof-of concept compiler, ANTLR (Parr 2007) is used in combination with Xtext (Bettini 2013).

3.1 Sequential Code Generation

The code generation in general is based on a traversal of the abstract syntax tree that is annotated with type attributes. In addition, a code template is attached to each node of the tree. A template contains placeholders for source code provided by the child nodes. During tree traversal, the placeholders are resolved such that the final result is the translated C source code of the MATLAB input program.

The dynamic typing of the MATLAB language requires the deduction of data types in order to perform a valid translation into statically typed C code. Furthermore, the possibility that a MATLAB variable may change its type at runtime must be overcome. To achieve this, a source variable of the MATLAB code is uniquely mapped to multiple target identifiers in the generated C code based on the actual data type. However, the prototype implementation of StencilPaC handles only one- and two-dimensional matrices. Higher dimensional matrices are subject to future work.

While the generation of code is trivial for scalar assignments and control structures, matrix expressions require more sophisticated actions since C does not provide an according built-in data type.

As a result, the compiler defines internal data structures for matrices that contain size information and the matrix data. Vectors are considered as a special instance of matrices. The matrix data is allocated dynamically in a column-major data layout using a one-dimensional array of the matrix' basic data type.

When a matrix expression is traversed in the abstract syntax tree, the compiler emits C code that translates two-dimensional matrix accesses from MATLAB into one-dimensional accesses on the matrix' data array. In addition, assignments or numerical matrix operations are translated into nested for-loops. The loop boundaries are based on the index expressions in the original MATLAB code which can be constant or variable expressions. As a result, the resulting C source code performs element-wise operations on each element of a matrix as illustrated in Figure 3

The compiler also resolves data dependencies in the generated C code. If overlapping accesses on matrix elements are detected (or must be assumed if the index expressions are not constant) a temporary copy of the accessed matrix is created and destroyed after matrix access.

For the generation of sequential source code, the compiler also performs critical optimizations to avoid overhead in the generated C code. The applied techniques are well-known in the field of functional array languages. Therefore, they are mentioned only briefly for the sake of completeness. For example, a matrix assignment in a loop with data-dependencies causes frequent temporary buffer allocations and releases. To avoid this, the compiler generates code that allocates a temporary buffer in front of the loop in case of a dependency.

In addition, a sequence of matrix assignments can be merged into a single loop if the boundaries and properties of the generated loops match. This can result in better cache utilization if matrices reoccur in the sequence. Further, for constant matrices in the MATLAB source code, the compiler generates according constant C arrays.

3.2 Performance Results

Table 1 shows the results for *EasyWave* running a simulation of the 60 minutes scenario and the Cellular Automaton with a field of size $2^{14} \times 2^{14}$ performing 250 iterations. The results were obtained on the same environment as in Section 4.2.

For *EasyWave*, the automatically generated C version without further optimizations delivers a speedup of 4.96 in comparison to MATLAB. With optimizations enabled the execution time is even further reduced leading to a total speedup of 7.59. This indicates the importance of optimizations performed by StencilPaC. Compared to the MATLAB runtime, the given C reference implementation of *EasyWave* achieves a speedup of 7.18. Thus, the generated C code can compete with the hand-written implementation. The runtime difference between both versions is probably caused by deviating optimizations applied by the underlying C compiler.

The lower half of Table 1 shows the performance of the Cellular Automaton using the `char` data type for the cell state. Compared to the MATLAB version, a reduction of the execution time by a factor of 1319 resp. 1717 after optimizations is observed when the code is compiled to C. The reason for this immense speedup lies in the usage of `arrayfun` in the MATLAB code. Although it is a convenient mean for programming, its performance impact on the application is disadvantageous. In contrast, StencilPaC is able to generate efficient code for the lookup.

3.3 Parallelization

As shown in the previous subsection, the compiler generates code that runs significantly faster than in MATLAB. However, to take advantage of parallel hardware architectures additional support has to be provided by the StencilPaC compiler.

program version	runtime/s	speedup
EasyWave MATLAB (60 min.)	375.44	–
EasyWave StencilPaC C	75.75	4.96
EasyWave StencilPaC Optimized C	49.45	7.59
EasyWave C Reference	52.31	7.18
CA MATLAB	625440	–
CA StencilPaC C	473.98	1319.54
CA StencilPaC Optimized C	364.06	1717.95
CA C Reference	394.03	1587.29

Table 1. Runtime and speedups of the two example applications.

The element-wise matrix expressions from the original MATLAB source provide inherent data-parallelism and are the primary target of the parallelization approach. In addition, *map* functionality on matrices is supported with MATLAB's `arrayfun` function. *Scan* and *reduce* operations, on the other hand, are subject to future work. Furthermore, MATLAB loops, which are mainly used for temporal iteration, are currently not considered for automatic parallelization. All supported *data-parallel statements* are converted into code for the targeted parallel architectures. Regardless of the programming model, other statements are not considered for further optimizations or parallelization. As a result they are replicated on each unit of execution. However, I/O operations, are executed on only one of these execution units in the current prototype implementation. Within the next sections, the parallelization approaches used by StencilPaC for different architectures are described in more detail.

4. Shared Memory Parallelization

The first approach is based on the OpenMP standard (OpenMP Architecture Review Board 2013) which defines a collection of compiler directives, library routines, and environment variables to utilize multiple threads on shared memory systems. The standard enables a portable implementation for different hardware architectures and supports data parallelism. It therefore offers a high-level method that is well suited for the parallelization of matrix expressions.

With OpenMP, an application follows the fork-join model and is executed by a single master thread. Multiple threads are spawned (fork) whenever a parallel region is opened, and terminated (join) at the end of such region. Parallel code regions are annotated by compiler directives. For data parallelism, for-loops can be annotated accordingly. At runtime, the loop body is assigned to the started threads having each thread computing individual iterations. Thus, OpenMP is used to divide the element-wise operations of a matrix expression over the available threads.

4.1 Concept

As described in Section 3, matrix expressions are translated into nested loops. The sequential code generation already ensures that the loop iterations are independent of each other and can be executed in any order. A temporary buffer is introduced if necessary to guarantee the correctness of the results like it is required for the Cellular Automaton. In order to obtain a parallel execution it is then sufficient that StencilPaC inserts an OpenMP directive in front of the loop structure as shown in Figure 3. This instructs the underlying C compiler to parallelize the loop iterations.

With the shared memory paradigm, all variables which are declared outside of a parallel region are shared by all threads, except the outer loop index (`tmp_x` in Figure 3). Writing these variables in parallel leads to race conditions and unpredictable results. In order to avoid such undefined behavior local copies of internal variables (generated by the StencilPaC compiler) are provided for each

```
#pragma omp parallel for \
 private(tmp_y) firstprivate(lv_i)
for( tmp_x = 1; tmp_x <= xdim; tmp_x++ ) {
  for( tmp_y = 1; tmp_y <= ydim; tmp_y++ ) {
    /* Operate on single values. */
  }
}
```

Figure 3. Example of a generated OpenMP code fragment for a matrix operation.

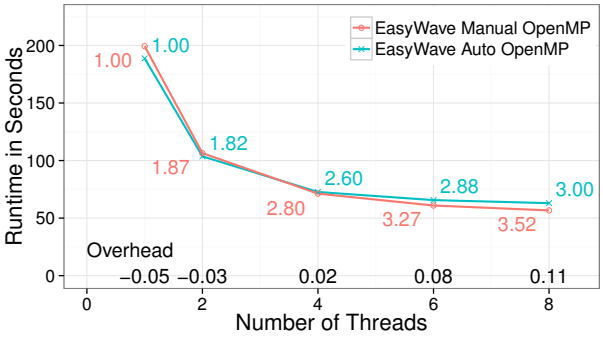

Figure 4. Scaling of the OpenMP version for an EasyWave simulation of 180 minutes.

thread. To achieve this, it is sufficient to specify them in an automatically generated `private` clause within the `parallel` directive. Each thread can then safely modify these variables independently without causing race conditions.

Nevertheless, read-only access on shared variables may decrease the performance due to dereferenced pointer accesses and cache conflicts (Sun Microsystems, Inc 2004). Therefore, StencilPaC generates local copies even for read-only variables $\texttt{lv}_i$. Since, their initial value (from the original MATLAB code) has to be retained, the `firstprivate` clause has to be emitted for every variable accessed inside the loop (see Figure 3).

4.2 Evaluation

The performance of the generated parallel OpenMP code was evaluated for the two benchmark applications. The experiments were conducted on a dual-socket Intel Xeon E5520 (2.26 GHz) machine with a total of eight cores and 48 GB of RAM. HyperThreading was disabled in all experiments. The code was compiled using the GCC 4.9.1 C compiler with optimization enabled (O2). Programs were executed with the libgomp OpenMP runtime under a Linux kernel 2.6.18. OpenMP threads were bound to the CPU cores, such that thread migration was avoided.

The results are shown in Figure 4 and 5 for up to 8 threads utilizing the entire node. As apparent from the diagrams, the scaling of the generated versions is similar to that of the manual implementations. The speedups shown in the figures are relative to the single threaded runtime of each version to illustrate the possible scaling of the individual approaches.

In case of EasyWave, a maximal speedup of 3 is reached when utilizing all cores of the node. However, the reason for the poor scaling is given by the low operational intensity as explained in Section 2.1. Regarding the Cellular Automaton, the scaling is much better and close to the ideal speedup for both of the compared versions. The automatically generated code remains faster with an overall improvement of 5%. The runtime could finally be reduced

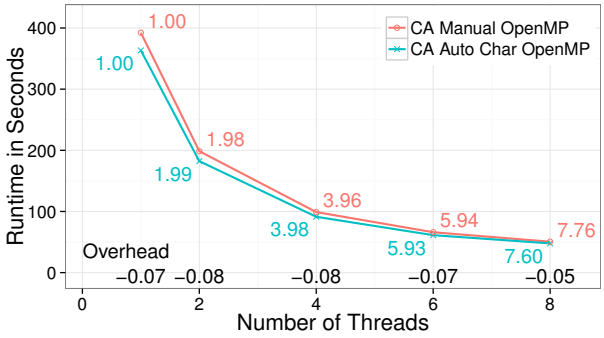

Figure 5. Scaling of the OpenMP version for the Cellular Automaton using data type *char*.

from 363.34 to 47.80 seconds using 8 threads. This corresponds to a speedup of 7.60.

The results show that the generated C code containing shared-memory parallelism can again compete with the hand-written counterparts.

5. Parallelization for Distributed Systems

The Message Passing Interface (MPI) (Message Passing Interface Forum 2015) is a specification especially designed for distributed memory systems like compute clusters. The programming model follows the *Single Program, Multiple Data* concept. That is, a single MPI program is executed by multiple processes which operate on separate address spaces. During execution, each MPI process is identified by a unique numerical rank. Its value is used to assign different workloads to processes, e.g. to compute different subsets of a compute domain. To exchange data, the MPI standard defines multiple methods, like sending and receiving messages or one-sided communication.

5.1 Concept

For MPI parallelization of data-parallel MATLAB code, the StencilPaC compiler has to a) distribute the data structures (domain decomposition) and b) must add communication to the application where required. However, this affects only the data-parallel matrix statements. On the other hand, the generated C code of the remaining statements can be reused and is not required to be parallelized.

The implementation of StencilPaC uses a one-dimensional decomposition along the x-dimension to distribute the elements of a matrix. In particular, each process gets a continuous block of columns assigned as outlined in Figure 6 for three parallel processes. Each block corresponds to a contiguous memory region since matrix elements are stored column-wise. The size of the blocks is determined dynamically depending on the number of started MPI processes. The benefit of a one-dimensional decomposition compared with a two-dimensional decomposition is that the communication overhead is reduced since every process has only two communication partners.

Besides the local portion of a matrix, each process has to provide additional memory in order to load external columns from remote processes. This memory is divided in two ghost zones representing remote columns to the left and to the right of the local part of the matrix. Each ghost zone can have an arbitrary size but needs to cover a continuous block of columns as outlined in Figure 6. The size of the ghost zone is derived from the MATLAB source code. If this is not possible, space for the entire compute domain is allocated by each process.

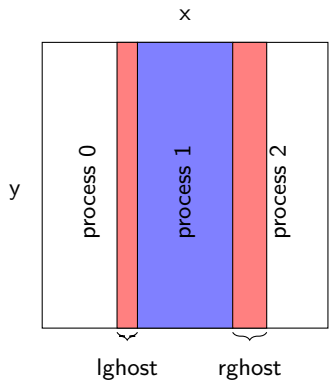

Figure 6. Ghost zone handling for a distributed matrix.

Matrix expressions can then be parallelized based on the fundamental concepts presented above. First of all, a base matrix needs to be determined which serves as a reference in the translation process. The choice is arbitrary, such that it is sufficient to take the first matrix found in the parse tree of a data-parallel expression. This corresponds to the destination matrix in case of a matrix assignment. Afterwards, each rank is responsible to process its local portion of the selected base matrix. This yields a parallel execution of the entire expression since the parts of the matrix are evenly distributed over the processes. A process must then ensure that external data, which is required to perform operations on the local part, is loaded into the local memory before processing the elements. This requires communication with other processes due to the separate address spaces. This communication step is called *ghost zone exchange*. After that, all ranks iterate their local parts and execute the embedded statements as usual.

For the ghost zone exchange, StencilPaC uses the one-sided communication API (Message Passing Interface Forum 2015, Chapter 11) of the MPI standard. It allows communication parameters, e.g. size, destination and addresses of the communication buffers, to be specified by only one side of the communicating processes. By using this API, the generated source code of StencilPaC is able to retrieve the required ghost zone data with `GET` operations without involving the two neighbor processes that provide the ghost zone data. This relieves the generated code to redundantly calculate the communication parameters in all communicating processes which would increase the overhead.

However, the ghost zone exchange must be performed at the right time, i.e. when the required source data of the ghost zone was updated but not overwritten by the producing process' computation. For that purpose, the fence synchronization scheme of MPI is employed. It synchronizes the processes at the beginning and at the end of the exchange. In consequence, the accessing process knows when ghost zone data is ready and all other processes know when the access was performed.

5.2 Evaluation

According to the programming model, the evaluation of the MPI parallelization was carried out on a cluster computer. It consists of the same nodes used in Section 4.2. These are connected with a 20 Gb/s InfiniBand Network via an Mellanox MTS3600R-1UNC switch. Open MPI 1.8.2 was used in the experiments as the underlying MPI implementation. The remaining parameters are identical to Section 4.2. However, no shared memory parallelization (using OpenMP) was performed.

Figure 7 shows the results for the EasyWave example in comparison to the manual MPI implementation, which is based on explicit point-to-point communication. With the automatic version, the sequential runtime could be reduced from 206.28 to 7.16 seconds while utilizing a maximum of 96 cores. This corresponds to an overhead of only 8% compared to the best manual runtime. Even though the compiler generated version cannot completely reach the performance of the manual counterpart, the scaling is quite similar.

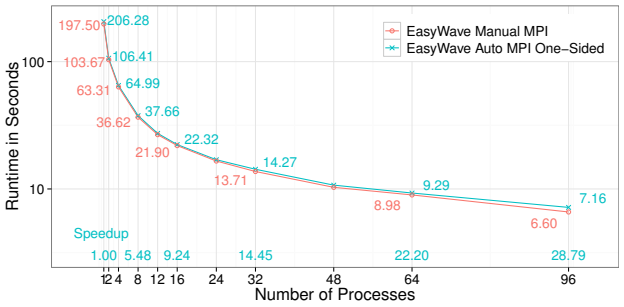

Figure 7. Measurement results for the MPI/OSC approach applied to EasyWave (180 min).

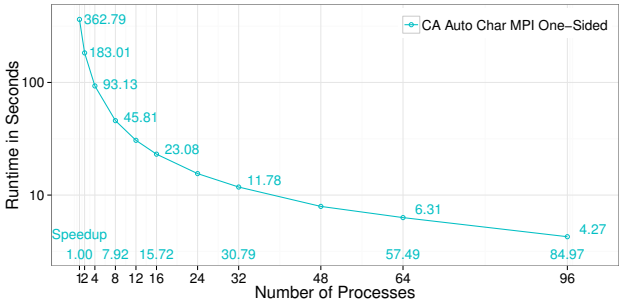

Figure 8. Measurement results for the MPI/OSC approach applied to the Cellular Automaton.

The measurements for the Cellular Automaton are illustrated in Figure 8. The automatic implementation scales almost linearly for up to 24 processes reaching an efficiency of more than 97%. However, the scalability slightly decreases for a larger number of processes. Thus, a maximum speedup of 84.97 is obtained using all 96 cores. This still corresponds to an efficiency of 88.5%.

6. OpenACC

A further extension of the StencilPaC compiler covers the utilization of accelerator cards such as GPUs. This approach is based on the OpenACC Application Programming Interface to enable the offloading of compute intense regions to an accelerator device. Compiler directives are used to mark code sections for automatic parallel execution on an attached device similar to OpenMP. The OpenACC (OpenACC-Standard.org 2013) standard offers a portable way to address various types of accelerators. Compared to CUDA, this high-level approach simplifies the implementation but limits the control over the generated device code. Therefore, the quality of the final executable depends on the capabilities of the underlying C compiler. However, upcoming hardware architectures may automatically be supported without adaptations to the StencilPaC compiler. Further, different groups report good or even the same performance for the OpenACC application compared with a native CUDA implementation of their application (Wienke et al. 2012; Christgau et al. 2014).

6.1 OpenACC Approach

An accelerator device usually consists of separate memory and has no direct access to the host memory. Therefore, it is necessary to copy required data into the memory of the accelerator before running a task on the device. The results must be transferred back after the computation is completed in order to access them on the CPU again. Thus, the presented approach has to manage the data flow between CPU and accelerator. To this end, the host and device memories are considered in a symmetric memory model.

With the OpenACC approach, a statement is either executed on the CPU (host statement) or offloaded to the accelerator (device statement). Thus, matrix elements may be updated both on the host and the device which results in potential inconsistencies between the memory regions. The application must therefore take care of synchronizing the host and device regions of a matrix. To keep track of inconsistencies, the matrix structure is extended by a `sync` attribute which states whether the elements are consistent or outdated on one side. The field can attain one of the values listed in Table 2 to reflect the current synchronization state.

`MEM_INIT`	The matrix was not yet used on the device.
`MEM_SYNC`	The matrix values are consistent.
`MEM_DEVICE`	The matrix was updated on the accelerator since last synchronization.
`MEM_HOST`	The matrix was updated on the host since last synchronization.

Table 2. List of possible synchronization states.

The compiler must ensure that the state is updated properly during program execution. To achieve this, synchronization code is inserted in front of each statement. With this approach inconsistencies are resolved as late as possible. Therefore, matrices can be kept on the accelerator as long as no host accesses occur. This eliminates expensive memory transfers whenever possible. Matrices which are never used inside a device statement even occupy no device memory at all.

In contrast to matrices, scalar variables get no explicit device memory assigned. They are kept in host memory only and are always passed to the device on entry of a parallel region. The copy overhead should be negligible and can even be beneficial since the values may directly be placed into registers. Scalar variables are either used read-only or exclusively inside a device execution and thus need not be transfered back to the host. Thus, no explicit allocation of device memory is required.

The current implementation supports offloading of matrix expressions to the accelerator. Since a matrix expression is translated into a nested loop structure, the entire loop construct is marked for offloading. This is achieved by automatically inserting a `parallel loop` directive in front of the outer `for` loop as illustrated in Listing 9. Thus, the loop iterations are executed in parallel on the device. Scalar variables are implicitly private for an executing thread and need not be specified in a `private` clause in contrast to the presented OpenMP solution. However, the accessed matrices and their associated memory pointers must be passed in a `present` clause to omit a memory exchange at region entry. The `collapse` clause instructs the OpenACC compiler to parallelize the inner loop as well. This is useful to fully utilize devices like GPUs which contain thousands of cores.

A special treatment is required to handle function calls in a device statement. In particular, a function must be compiled for the accelerator in order to make it accessible inside of a parallel region. This can be achieved with the `routine` directive in front of the function definition (OpenACC-Standard.org 2013). All user-defined functions are marked for offloading to allow function invocations within the matrix expressions.

```
#pragma acc parallel loop collapse(2) \
 present(M1, ..., Mn)
for( tmp_x = 1; tmp_x <= xdim; tmp_x++ ) {
  for( tmp_y = 1; tmp_y <= ydim; tmp_y++ ) {
    /* Depends on language entity. */
  }
}
```

Figure 9. Offloading of matrix expressions to the accelerator device.

6.2 Evaluation on GPU

The experiments were run on a NVIDIA Tesla K40m GPU with 2880 Cores. A 64 Bit Linux with base kernel 3.0.101 is installed on the GPU systems. The PGI compiler in version 15.10 is used to build the OpenACC implementations.

First, we present results for *EasyWave*. We compare the performance of the automatically generated OpenACC version with a hand-written GPU version from (Christgau et al. 2014). The hand-written GPU version of *EasyWave* is compiled with the CUDA toolkit which is available in version 7.0.

While the given MATLAB code runs about 1180 seconds, the automatically generated OpenACC code needs only 6.30 seconds. The hand-written CUDA code for *EasyWave* yields the fastest execution with 5.38 seconds, but this performance gain is neglectable compared with the impressive performance boost achieved by using StencilPaC.

The same performance benefit was observed for the cellular automaton. While the given MATLAB code has a runtime of about 625440 seconds, the automatically generated OpenACC code needs only 9.04 seconds to execute. A hand-written version for GPUs was not available for this benchmark.

7. Related Work

Christen et al. presented *PATUS*, a framework to utilize stencil computations on multicore CPUs and CUDA-capable GPUs (Christen et al. 2011b,a). *PATUS* is based on a domain specific language which can be used to define custom stencil kernels in a C-like syntax. Different strategies are available to automatically optimize and parallelize these kernels. However, there is no support for the MATLAB language.

Bispo et al. developed a MATLAB-to-C Compiler named MATISSE (Bispo et al. 2013), targeting embedded systems. MATISSE was further enhanced and optimized in (Bispo et al. 2015b). For the evaluation, the authors compare the speedup of the C code generated by MATISSE and by MEGHA, the underlying MATLAB-to-C compiler from (Prasad et al. 2011), against the MATLAB code for 9 benchmarks. Their approach delivers a geometric-mean speedup of 8.1.

While there exist other approaches from the field of functional array languages, see for example SAC (Grelck and Scholz 2006) or Obsidian (Svensson et al. 2011), which also generate C or even CUDA code in case of Obsidian, these are hardly comparable, since our focus is to support the more prominent MATLAB language.

In the following, we will summarize prior work on automatic generation of parallel C code for MATLAB.

7.1 Parallel Code for Distributed Memory Systems

Kim et al. introduce two libraries in (Hahn Kim, Julia Mullen and Jeremy Kepner) that enable the parallel programming with MATLAB. The code is developed at the Massachusetts Institute of Technology and freely available. The first library, called *MatlabMPI*, implements a subset of the MPI standard in pure MATLAB and makes it available to the MATLAB environment. This allows to run

multiple MATLAB instances in parallel which can communicate with each other using that library. The communication is performed through the file system with basic I/O functionality. The MPI functions can be used to apply a manual parallelization which requires the coordination of processes as well as an explicit communication handling. This results in expensive rewriting of existing applications. To address this problem, the authors present a second library named *pMatlab* that provides an abstraction layer to *MatlabMPI* allowing a higher-level parallelization. Although the effort in porting existing applications is significantly reduced, manual adaptations are still required. Hence, there are two substantial differences to the approach presented in this paper. In particular, neither *MatlabMPI* nor *pMatlab* provide a full automatic parallelization. Furthermore, both libraries are built on top of the MATLAB environment and do not allow a native execution.

Quinn et al. present an approach for C code generation that exploits data parallelism with MPI (Quinn et al. 1998). They provide a runtime library including parallelized C code and translate source expressions into library calls. But embedded parallel code is only generated for element-wise matrix operations. Apparently, element-wise operations are restricted to entire matrices and single values. Subdomains, which are required to define stencil patterns for grid-based applications, are not taken into account.

7.2 Automating GPU Computing

There exist GPU libraries for MATLAB that hide the complexity of programming GPUs (see for example (ArrayFire; Ryoo et al. 2008)). In order to make use of such libraries, the MATLAB program has to be adapted accordingly.

Shei et al. present an approach where this task is done automatically (Shei et al. 2011). MATLAB code is automatically translated into MATLAB code which makes use of the GPU library *GPUmat* (GPUmat) for MATLAB.

In contrast, the approach presented in this paper depends not on additional libraries.

Bispo et al. also generate code for GPUs from MATLAB code (Bispo et al. 2015a). Again the drawback of their approach is that the MATLAB code has to be adapted since it has to be annotated to generate C and OpenCL code.

7.3 Commercial Products

MathWorks itself distributes a series of commercial products which are related. The *Parallel Computing Toolbox* (The MathWorks, Inc. 2015c) provides different high-level concepts that can be used to add parallelism to existing MATLAB applications. The toolbox offers MPI support and enables, along with the *MATLAB Distributed Computing Server* (The MathWorks, Inc. 2015b), the utilization of clusters and grids. Furthermore, GPU-execution is available for CUDA-capable devices. The *Parallel Computing Toolbox* covers a broad subset of the MATLAB language supporting over 240 functions. However, manual modifications are again required to utilize multiple processes. In addition, except of the GPU code, the application is still executed in the MATLAB environment.

Another product line of MathWorks offers the *MATLAB Coder* (The MathWorks, Inc. 2015a) that automatically generates readable C/C++ code based on a MATLAB program. It supports a large set of functions but is incompatible with the *Parallel Computing Toolbox* and thus does not provide MPI functionality. Altogether, there is no known commercial product that offers a fully automated parallelization into efficient C code. Nor does any of the tools support OpenACC to address dedicated and upcoming architectures automatically (i.e. offloading to a XeonPhi).

8. Conclusion

We presented the underlying concepts of the StencilPaC compiler which automatically translates a given MATLAB program into parallel C code for different architectures. The approach supports a MATLAB subset important for stencil applications which are based on matrix operations. The benefit of the presented approach is that the original MATLAB source code serves as input. No adaptations or annotations of the MATLAB code are necessary.

Over the last years, several efforts have been made to facilitate programming of parallel hardware architectures, such as multicores and graphic cards. Interfaces like OpenMP and OpenACC use annotations to give compiler hints for the parallelization. We have shown that using these parallelization APIs also MATLAB code can gain an enormous speedup. For GPGPUs, the runtime of the Tsunami simulation *EasyWave* was reduced from 1180 seconds in MATLAB to 6.3 seconds for the generated OpenACC code. On a multicore system with 8 cores, the runtime of the generated OpenMP code is about 60 seconds.

StencilPaC also supports distributed memory architectures like compute clusters. For these systems, the de facto standard MPI was chosen as API for the parallelization. An automatic distribution of matrix elements and a dynamic data exchange, on the other hand, are key concepts of the MPI approach. For communication, the one-sided communication approach was chosen since it reduces the administration overhead required to manage the dynamic data exchange automatically at runtime and simplifies code generation.

For the Tsunami simulation, the runtime was reduced from 1180 seconds in MATLAB to 7.16 seconds for the generated MPI code on a 96 core compute cluster.

Further, the scaling of the parallel code was investigated for the shared and the distributed memory approaches. The speedups of hand-written parallel reference implementations and the automatically generated parallel codes are very close. This shows that there is no additional parallelization overhead introduced by the automatic approach of StencilPaC.

The results of this work have shown that an automatic compilation and parallelization based on a MATLAB-related source language can successfully accelerate a wide range of applications with common stencil patterns.

Future work will focus on enhancements of the StencilPaC prototype. These include, for example, the support for higher dimensional matrices and common generic operations such as *scan* and *reduce*.

References

ArrayFire. GPU Library for MATLAB . `http://arrayfire.com/why-arrayfire/`. Accessed: 2016-04-09.

A. Babeyko. EasyWave: fast tsunami simulation tool for early warning. `ftp://ftp.gfz-potsdam.de/pub/home/mod/babeyko/easyWave/easyWave_About.pdf`, Feb. 2012.

L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. ISBN 1782160302, 9781782160304.

J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J. M. P. Cardoso, and P. C. Diniz. The MATISSE MATLAB compiler. In *11th IEEE International Conference on Industrial Informatics, INDIN 2013, Bochum, Germany, July 29-31, 2013*, pages 602–608. IEEE, 2013. ISBN 978-1-4799-0752-6.

J. Bispo, L. Reis, and J. a. M. P. Cardoso. C and OpenCL Generation from MATLAB. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1315–1320, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3196-8.

J. Bispo, L. Reis, and J. a. M. P. Cardoso. Techniques for efficient matlab-to-c compilation. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Program-*

ming, ARRAY 2015, pages 7–12, New York, NY, USA, 2015b. ACM. ISBN 978-1-4503-3584-3.

M. Christen, O. Schenk, and H. Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Computer Science - R&D*, 26(3-4):205–210, 2011a. doi: 10.1007/s00450-011-0160-6. URL `http://dx.doi.org/10.1007/s00450-011-0160-6`.

M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 676–687. IEEE, 2011b. doi: 10.1109/IPDPS.2011.70. URL `http://dx.doi.org/10.1109/IPDPS.2011.70`.

S. Christgau, S. Kiertscher, and B. Schnor. The Benefit of Topology-Awareness of MPI Applications on the SCC. In D. Göhringer, M. Hübner, and J. Becker, editors, *3rd Many-core Applications Research Community (MARC) Symposium*, pages 47–51, Karlsruhe, 2011.

S. Christgau, J. Spazier, B. Schnor, M. Hammitzsch, A. Babeyko, and J. Waechter. A comparison of CUDA and OpenACC: Accelerating the Tsunami Simulation EasyWave. In *27th International Conference on Architecture of Computing Systems (ARCS), 2014*, Feb 2014.

S. Christgau, J. Spazier, and B. Schnor. A Performance and Scalability Analysis of the Tsunami Simulation EasyWave for Different Multi-Core Architectures and Programming Models. Technical Report TR-2015-01, University Potsdam, Institute of Computer Science, Potsdam, Germany, 2015. ISSN: 0946-7580.

GPUmat. GPU Library for MATLAB . `https://sourceforge.net/projects/gpumat/`. Accessed: 2016-04-09.

C. Grelck and S.-B. Scholz. SAC – A Functional Array Language for Efficient Multi-threaded Execution. *Journal of Parallel Computing*, 34: 383–427, 2006.

Hahn Kim, Julia Mullen and Jeremy Kepner. Introduction to Parallel Programming and pMatlab v2.0. MIT Lincoln Laboratory, Lexington, MA 02420.

Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1. `http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, April 2015.

OpenACC-Standard.org. The OpenACC Application Programming Interface Version 2.0. `www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf`, June 2013.

OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. ISBN 0978739256.

A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of matlab programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993517. URL `http://doi.acm.org/10.1145/1993498.1993517`.

M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary Results from a Parallel MATLAB Compiler. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 81–87, Mar 1998. doi: 10.1109/IPPS.1998.669894.

S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7.

P. Sanders and T. Worsch. *Parallele Programmierung mit MPI - ein Praktikum, Programmtexte im Internet*. Logos Verlag, 1997. ISBN 978-3-931216-76-4.

C.-Y. Shei, P. Ratnalikar, and A. Chauhan. Automating GPU Computing in MATLAB. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 245–254, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2.

Sun Microsystems, Inc. OpenMP API User's Guide - Performance Considerations. `http://docs.oracle.com/cd/E19059-01/stud.10/819-0501/7_tuning.html`, 2004. Accessed: 2015-10-10.

J. Svensson, M. Sheeran, and K. Claessen. *Implementation and Application of Functional Languages: 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*, chapter Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors, pages 156–173. Springer Berlin Heidelberg, 2011.

The MathWorks, Inc. MATLAB Coder. `http://de.mathworks.com/products/matlab-coder/`, 2015a. Accessed: 2015-11-11.

The MathWorks, Inc. MATLAB Distributed Computing Server. `http://de.mathworks.com/products/distriben/`, 2015b. Accessed: 2015-11-11.

The MathWorks, Inc. Parallel Computing Toolbox. `http://de.mathworks.com/products/parallel-computing/`, 2015c. Accessed: 2015-11-11.

S. Wienke, P. Springer, C. Terboven, and D. Mey. OpenACC — First Experiences with Real-World Applications. In C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 859–870. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32819-0.