International Journal of Parallel Programming (2020) 48:713–728
https://doi.org/10.1007/s10766-020-00659-x

Towards High-Performance Code Generation for Multi-GPU Clusters Based on a Domain-Specific Language for Algorithmic Skeletons

Fabian Wrede[1] · **Herbert Kuchen**[1]

Received: 15 October 2019 / Accepted: 30 April 2020 / Published online: 22 May 2020

Abstract
In earlier work, we defined a domain-specific language (DSL) with the aim to provide an easy-to-use approach for programming multi-core and multi-GPU clusters. The DSL incorporates the idea of utilizing algorithmic skeletons, which are well-known patterns for parallel programming, such as map and reduce. Based on the chosen skeleton, a user-defined function can be applied to a data structure in parallel with the main advantage that the user does not have to worry about implementation details. So far, we had only implemented a generator for multi-core clusters and in this paper we present and evaluate two prototypes of generators for multi-GPU clusters, which are based on OpenACC and CUDA. We have evaluated the approach with four benchmark applications. The results show that the generation approach leads to execution times, which are on par with an alternative library implementation.

Keywords Algorithmic skeletons · Parallel programming · High-performance computing · Model-driven development · Domain-specific language

1 Introduction

Programming for multi-GPU clusters poses many challenges especially for inexperienced programmers. Data has to be distributed between multiple computing nodes and compute regions have to be offloaded to the accelerators, and data has to be distributed once again to utilize multiple GPUs.

✉ Herbert Kuchen
kuchen@uni-muenster.de

Fabian Wrede
Fabian.Wrede@wi.uni-muenster.de

[1] Department of Information Systems, European Research Center for Information Systems (ERCIS), University of Muenster, Leonardo-Campus 3, Muenster 48149, Germany

There are several approaches to make programming for GPU systems more accessible. For example, Thrust [1] provides containers for easier data transfers between devices and host, and OpenACC [2] standardizes a set of compiler directives and library routines for a non-invasive way to parallelize applications on GPUs. However, most of these approaches do not support both, distributed systems with multiple GPUs per node.

One approach to overcome these shortcomings are *Algorithmic Skeletons*. Algorithmic skeletons have first been proposed by Cole [3] and are reoccurring parallel patterns known from functional programming. These comprise data-parallel skeletons such as `map`, `fold/reduce`, and `zip` as well as task-parallel skeletons, such as `pipeline` and `farm`, and communication skeletons, such as `broadcast`.

There are multiple ways how to implement algorithmic skeletons. For instance, the Muenster Skeleton Library (Muesli), which has been introduced in 2002 [4], is a C++ library, which implements distributed data structures and algorithmic skeletons, which are implemented as member functions of these data structures. The skeletons support parallel execution on multi-core and multi-GPU clusters [5].

Another newer approach is the Muenster Skeleton Tool (Musket). It generates code based on an input model, which is expressed in a domain-specific language (DSL). This language has built-in support for distributed data structures and algorithmic skeletons. Since we look at the code generation process from the perspective of model-driven software development, we use the terminology of a DSL, model and generator, rather than programming language, program and compiler. From our perspective, there are two main advantages of using a model-driven development approach in the context of parallel programming with algorithmic skeletons. First, as known from the literature, a DSL can simplify and speedup the development process, since the language only focusses on necessary core features [6, 7]. For instance in our case, the musket DSL does not require to explicitely handle low-level details such as data transports between host and GPUs or between different GPUs. Consequently, the DSL code is much more concise than equivalent C++ code using e.g. CUDA. Second, it provides a good way for performing transformations on the model [6, 7]. In particular, the available model-driven development tools, such as xtext and xtend [8, 9], directly provide the abstract syntax tree (AST) of a model. Moreover this AST is rather small and easy to handle because of the concise syntax of the DSL. Thus, it is relatively easy to perform optimizations such as map fusion or fusing map and fold into a single mapreduce skeleton.

As the main contribution of the present paper, we describe the implementation of two new generators for Musket, which now offer support for multi-GPU clusters. Moreover, we evaluate the code produced by these generators experimentally based on four benchmarks. Up to now, there has only been a generator for sequential code and for clusters of multi-core systems (internally based on MPI and OpenMP). The new generators are based on the OpenACC standard [2] for programming GPUs and on CUDA [10]. While the most mature implementation of the OpenACC standard is currently provided by the PGI compiler for Nvidia GPUs, GCC includes most of the OpenACC standard version 2.5 in version 9. Therefore, we have decided to evaluate, whether OpenACC is a suitable tool for a portable back-end implementation and how the execution times compare to the CUDA implementation. In order to implement the new generators, we had to refactor the existing generator. In particular, we have split it into a general part, which can be reused in the new generators and a specific part, which needs to be replaced for each new generator and its corresponding destination platform.

Our paper is structured as follows: we begin with a brief introduction to Musket, followed by the description of the implementation and changes to the language. Afterwards, we evaluate the performance of the generated code. After the discussion of related work and topics, we will give an outlook on future work and conclude the paper.

2 Muenster Skeleton Tool

The Muenster Skeleton Tool (Musket) [11, 12] has been developed as an alternative to the library Muesli. There were two major points, which we aimed for with the approach. The first one is the usability. By defining a dedicated language the overall complexity can be reduced due to appropriate abstractions, which leads to shorter implementation times with less redundant code, better reusability, and better readability of models [6, 7]. The second point is the performance of generated code. While there are other alternatives as mentioned in Sect. 5, the generation approaches allows for optimizations such as rearranging the sequence of skeleton calls [13].

Excerpts from models expressed in the Musket DSL can be found in Listings 1, 3, 5, 7, and 8 in Sects. 3 and 4. After a general configuration, which is required to set the used generator and to provide information about the target architecture, the language is structured in three sections. First, all distributed data structures are defined. Information such as the name, size and distribution mode (copied or distributed among all processes) are required. At the moment, the data structures are the distributed array and the distributed matrix. Second, the user functions are implemented. These functions can be passed as an argument to a skeleton invocation. Third, the logic of the program is defined. This comprises the sequence of skeleton calls.

Skeleton calls include data-parallel operations, such as map and fold, as well as communication skeletons. The skeletons are invoked on data structures and the syntax follows the call of a member function in C++: `result = data_structure.skeleton(user_function())`. Arguments, such as the current element or the index of the element, are automatically passed to the user function. Moreover, the language includes built-in functions, such as `mkt::rand` to generate random numbers. These functions are for instance necessary, because the expression in the model has to be translated into more complex code in the generated C++ code. For the `mkt::rand` this might for example be the instantiation of random engines.

3 Implementation

In this section we will outline the generation process and the implementation of the GPU generators. We will also introduce aspects of the language and required changes for the GPU support.

The generation process is depicted in Fig. 1. The definition of the language is the basis, which is required to express the input model as well as to perform preprocessing and the generation. The transformations such as map-fusion are implemented as model-to-model transformation, which are performed by the preprocessor. Afterwards, the preprocessed model is passed to the generator, which generates the C++ code and additional project files, such as a CMakeLists.txt and build scripts.

3.1 Domain-Specific Language

The language is implemented with the Xtext framework [8]. Xtext's grammar language allows for defining a DSL in an EBNF-like style. From a language grammar, Xtext infers an EMF Ecore model, which is the meta-model of the language. While parsing textual input models, the Xtext parser automatically creates an in-memory object graph, which is the abstract syntax tree (AST). This object graph is the input for further steps, such as validation, preprocessing, and generation.

Xtext offers additional features to increase the usability of a DSL. For instance, it is possible to implement a custom syntax highlighting for a language. Additionally, validators can be implemented to signal warnings and errors to the user while developing a model. For example, the Musket DSL contains checks for the correct number of arguments for skeleton calls. These errors are already displayed during the modeling process in the IDE.

3.1.1 Configuration

The Musket DSL provides certain configuration switches, which are used by the generator. These are defined in the beginning of each model and guide the generation process.

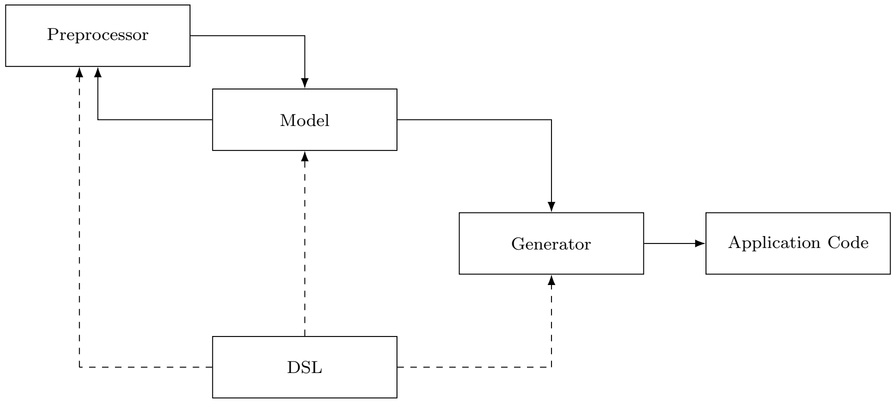


Fig. 1 Generation process

```
#config PLATFORM CPU GPU
#config PROCESSES 4
#config CORES 24
#config GPUS 4
#config MODE release
```

Listing 1: Configuration in Musket model

The `platform` variable determines which generators are triggered. Thus, in this example both the generators for multi-Core and for multi-GPU systems generate independent programs for each architecture. While the `processes` variable is used by both generators, other switches are ignored if they are irrelevant. Consequently, the CPU generator would ignore the value for available GPUs.

This example highlights the flexibility of the approach. If a new generator is implemented, as in this case the GPU generator, it is only necessary to specify the new platform and possibly newly required configuration variables, such as the number of GPUs. The remaining model can very likely remain without any changes. There have been two changes to the language, which will be discussed later. But even with those changes, original models were still working for the CPU generator.

3.1.2 Data Distribution

As mentioned above, Musket works with a two step data distribution approach, as it has also been the case for Muesli [14]. First, data is distributed among the compute nodes, and on each node the data is distributed among the available GPUs. The data distribution is implemented with two distributed data structures, the distributed array and the distributed matrix. Currently, there are two distribution modes called `dist` and `copy`. `Dist` means that the data is distributed, be it between nodes or GPUs and `copy` means that the data is available on each node or on each GPU.

```
CollectionType: ArrayType | MatrixType;

ArrayType: IntArrayType | DoubleArrayType | FloatArrayType |
    BoolArrayType | StructArrayType;

IntArrayType: 'array' '<' 'int' ',' size=IntRef ','
    distributionMode=DistributionMode (','
    gpuDistributionMode=DistributionMode)? '>';
```

Listing 2: Musket language definition of data structures expressed in Xtext

Listing 2 shows an excerpt from the definition of the data structures in the DSL implementation. The language defines the array and matrix as two distributed container types, which take the obligatory `distributionMode` and the optional `gpuDistributionMode` arguments. Consequently, Listing 3 shows a possible definition of an array in a model, which is distributed among the processes, but the data is copied on each GPU of that node.

```
array<int, 500000, 1, dist, copy> array_name;
```

Listing 3: Example of data structure in Musket model

The second parameter for the GPU distribution becomes only necessary with the new generators and this is also the reason why it is optional. To ensure that the parameter is set during modelling it is possible to implement a validator, which checks the defined data structures based on the chosen platform configuration.

3.1.3 Reduction Skeleton

With the new generators, we also included a shortcut for defining reductions. There has been a fold skeleton, which can take an arbitrary function and also requires the identity of the reduction operation as an argument. However, there are certain operations, which are commonly used to reduce elements, which are plus, multiply, minimum, maximum and logical operations. We decided to name the skeleton differently to have a clearer distinction. The definitions are rather similar as shown in Listing 4. Additional validation ensures that a reduction operation is only passed as an argument to the reduction skeleton. It is considered a best practice to keep the grammar concise and to use additional validation [15]. By using validators it is possible to provide better error messages and quickfixes.

```
FoldSkeletonVariants:
  {FoldSkeleton} 'fold' ('<' options+=FoldOption (','
      options+=FoldOption)* '>')? '(' identity=Expression ','
      param=SkeletonParameterInput ')' |
  {MapFoldSkeleton} 'mapFold' ('<' options+=FoldOption (','
      options+=FoldOption)* '>')? '('
      mapFunction=SkeletonParameterInput ','
      identity=Expression ',' param=SkeletonParameterInput ')';

ReductionSkeletonVariants:
  {ReductionSkeleton} 'reduce' '('
      param=SkeletonParameterInput ')' |
  {MapReductionSkeleton} 'mapReduce' '('
      mapFunction=SkeletonParameterInput ','
      param=SkeletonParameterInput ')';

SkeletonParameterInput: InternalFunctionCall | LambdaFunction
    | ReductionOperation;
```

Listing 4: Musket language definition of the reduction skeleton expressed in Xtext

3.2 Generator

The generator is implemented with Xtend, a Java dialect that offers additional features such as template expressions, which are especially useful for code generation [9]. The structure of the generator is similar to the CPU generator. In contrast to the first version of the CPU generator, which generated the complete code base, we decided to make use of a platform library for the GPU generators. For example, the distributed data structures are implemented as regular C++ templates and included in the generated project. In the same way, the CUDA kernels are implemented. This redesign facilitates a leaner implementation of the generators and allows for better maintenance of the C++ code in the library.

```
//[...]
array<Particle,500000,dist,dist> P;
array<Particle,500000,copy,copy> oldP;
//[...]
main{
  mkt::roi_start();
  for (int i = 0; i < steps; ++i) {
    P.mapIndexInPlace(calc_force());
    oldP = P.gather();
  }
    mkt::roi_end();
}
```

Listing 5: Musket model of N-body simulation benchmark

We want to give one more detailed example of how elements from the model are translated into C++ in the CUDA generator. The user functions from the model are generated as C++ functors. If another data structure is used within a user function, the GPU pointer is automatically passed to the function object. For example, in the N-body simulation model in Listing 5, the `calc_force()` user function is invoked on the `P` data structure but references values from the `oldP` array. The generated code is shown in Listing 6.

```
struct Calc_force_functor{
  Calc_force_functor(const mkt::DArray<Particle>& _oldP) :
      oldP(_oldP){}

  ~Calc_force_functor() {}

  __device__
  auto operator()(int curIndex, Particle curParticle){
    // [...]
  }

  void init(int device){
    oldP.init(device);
  }

  mkt::DeviceArray<Particle> oldP;
};

int main(int argc, char** argv) {
  // [...]
  mkt::DArray<Particle> P(0, 500000, 500000, 1, 0, 0,
      mkt::DIST, mkt::DIST);
  mkt::DArray<Particle> oldP(0, 500000, 500000, 1, 0, 0,
      mkt::COPY, mkt::COPY);

  Calc_force_functor calc_force_functor{oldP};
  // [...]
}
```

Listing 6: Generated C++ code for N-body simulation

Before the functor is generated, the body of the user function is filtered for accesses to data of other data structures. For each data structure a member of type `DeviceArray` is generated, which contains the pointers, number of elements and size of the data on the GPU. The `DeviceArray` is also a part of the static platform library. The `init` function is called within the skeleton function, so that the correct pointers are used in a multi-GPU scenario.

4 Evaluation

First, we describe the benchmark applications and explain the used models. Second, we discuss the results. The discussion begins with the evaluation of the OpenACC back-end, first, with automated parallelization and second, with an optimized back-end implementation. Further, we compare the results with an alternative back-end implementation, which uses CUDA. Since, the CUDA back-end leads to the best results, we compare the results of the benchmarks with an implementation with Muesli in order to determine whether the generation approach can compete with the library implementation in terms of performance.

4.1 Benchmarks

So far, we have evaluated the approach with four benchmark applications: (1) the simple calculation of the Frobenius norm of a matrix, (2) matrix multiplication, (3) N-body simulation, and (4) all-pairs of shortest paths in a graph. The first three benchmarks have been executed on a cluster where each node is equipped with up to four Nvidia K80 GPUs. As software we have used the PGI compiler in version 18.04, the Nvidia CUDA Toolkit 10.0, and gcc 7.3.0 as host compiler for CUDA. Moreover, we have used OpenMPI 2.1.2 for the benchmarks with OpenACC and OpenMPI 3.1.1 for the benchmarks with CUDA. The different OpenMPI versions have been necessary because of the dependency configurations of the cluster. Compiler optimizations have been enabled. We used the `O3` flag for nvcc and gcc, and `O4` flag for pgi. Moreover, we have specified the architecture using the `arch` and `code` (nvcc), `march` (gcc), and `tp` and `ta` (pgi) flags. Pinned memory has been used with PGI and CUDA in order to speedup data transfers between host and device. All benchmarks have been executed ten times and the mean average is presented. The execution times include all operations, such as creating data structures, data transfers, communication between nodes etc. Benchmark 4 has been run on a single Nvidia Titan Xp GPU. A drawback of this machine is that it is not equipped with the PGI compiler and we could hence not run the OpenACC back-end. Consequently, we could only compare the CUDA back-end and Muesli on this machine. This is not too bad, since the CUDA back-end turned out to be more efficient anyway.

The Frobenius benchmark uses one `mapReduce` skeleton. An excerpt of the model is shown in Listing 7. Within the `mapReduce` skeleton all values of the matrix *as* are squared and summed up. The `plus` argument of the `mapReduce` skeleton specifies the operation for the reduction step. In this case it is a shortcut for the plus operation. Afterwards, the square root of the sum is calculated to obtain the result. The functions `mkt::roi_start()` and `mkt::roi_end()` are translated into timer functions, which measure the execution time of the region of interest. We have used a 16384 × 16384 matrix with double precision values.

```
//[...]
mkt::roi_start();
as.mapReduce((double a) -> double {return a * a;}, plus);
fn = std::sqrt(fn);
mkt::roi_end();
//[...]
```

Listing 7: Musket model of Frobenius norm benchmark

The Matrix multiplication benchmark is an implementation of Cannon's algorithm with algorithmic skeletons. The algorithm is described in detail in [14]. For this benchmark we have used two 16384 × 16384 matrices with single precision values. The model is shown in Listing 8. The `shiftPartitions` skeletons move the data of the input matrices *as* and *bs* between processes. The *cs* matrix contains the result, which is calculated with the `mapLocalIndexInPlace` skeleton in line 5. The additional `LocalIndex` specifies that the user function `dotproduct` also takes the local indices of each element as an argument. The specifier `InPlace` means that the result of the map operation is written to the matrix on which the skeleton is invoked.

```
// [...]
  as.shiftPartitionsHorizontally((int a) -> int {return -a;});
  bs.shiftPartitionsVertically((int a) -> int {return -a;});
  for (int i = 0; i < as.partitionsInRow(); ++i) {
    cs.mapLocalIndexInPlace(dotProduct());
    as.shiftPartitionsHorizontally((int a)-> int {return -1;});
    bs.shiftPartitionsVertically((int a) -> int {return -1;});
  }
  as.shiftPartitionsHorizontally((int a) -> int {return a;});
  bs.shiftPartitionsVertically((int a) -> int {return a;});
}
//[...]
```

Listing 8: Musket model of matrix multiplication benchmark

The third benchmark is an N-body simulation performed on 500,000 particles over five time steps. This benchmark also uses single precision values and the model has been shown in Listing 5. The array *P* contains the particles and *oldP* contains the particles from the last iteration. The first distribution specifier `dist` and `copy` in lines 2 and 3 determine, whether the data is split between processes or whether the data is available on all processes. The second distribution specifier determines whether the data is split between the available GPUs per node or whether the data is available on all GPUs. After the particle system is updated in line 8, the data is copied from array *P* to *oldP*. Since this is a copy step from a distributed array to a copy distributed array, the `gather` skeleton is used.

The 4th benchmark computes all pairs of shortest paths in a randomly generated graph of 8192 nodes. The graph is represented by its distance matrix with integer distances. The implementation repeatedly uses a variant of Cannon's algorithm.

4.2 Results

Our first implementation has used OpenACC for programming GPUs. The rationale behind this choice has been that we wanted to focus on optimizations on the higher level of skeletons, such as skeleton fusion, rather than on low-level optimizations. Consequently, Musket performs transformations on the input model and generates C++ code, which is then compiled by an additional compiler such as nvcc, pgc++, or g++. In earlier work, we have found this to lead to better results than already incorporating techniques such as inlining into the generator, since these optimizations are performed anyway by the compiler later on. Thus, we include required files to build the generated project, such as a generated build script and CMake files. We wanted to analyze the performance of the generated code, letting the PGI compiler analyze the generated program and decide on the best way of parallelization. In theory, this would lead to a portable back-end implementation, which would support

Table 1 Overview of Musket's execution times with OpenACC back-ends (all measures in seconds)

Benchmark	Processes	GPUs	OpenACC (automated parallelization)	OpenACC (optimized)
Frobenius norm	1	1	1.2098	1.2257
	1	2	1.2526	1.2767
	1	4	1.4674	1.4992
	4	1	0.3486	0.3578
	4	2	0.4154	0.4115
	4	4	0.5837	0.5815
	16	1	0.5355	0.5225
	16	2	0.6065	0.6213
	16	4	0.8136	0.8174
Matrix multiplication	1	1	2252.5643	109.1062
	1	2	1133.3208	54.8244
	1	4	668.3012	28.4107
	4	1	556.8913	28.3133
	4	2	279.2927	17.1349
	4	4	164.3258	9.4570
	16	1	135.2462	8.1357
	16	2	66.9933	4.8964
	16	4	33.3739	3.2511
N-Body simulation	1	1	597.7319	109.0646
	1	2	302.4134	56.4951
	1	4	149.8264	30.1430
	4	1	150.8527	29.9718
	4	2	76.0810	16.8154
	4	4	38.3946	8.6539
	16	1	37.9842	8.4249
	16	2	19.2801	3.9035
	16	4	10.0177	4.0600

all platforms with a compiler that implements the OpenACC standard. However, the results, which are presented in Table 1: OpenACC (automated parallelization), have been rather unsatisfying.

While the execution times scale well, the absolute times are rather slow. For example, our generated code of a matrix multiplication for multi-core cluster with 4 nodes and 24 cores from another experiment takes 164.2069s for the execution [11], which is comparable with the utilization of 4 nodes with 4 GPUs. The reason is that the matrix multiplication is implemented with a `map` skeleton, which operates on the result matrix, and takes a user function that calculates the dot product. The PGI compiler decides to parallelize the outer loop on the gang level and the inner loop for the dot product calculation on the vector level, while it is the better decision to parallelize the outer loop and let the inner loop run sequentially. With the change to only parallelize the skeleton and not the user function, significant speed-ups can be achieved as presented in Table 1: OpenACC (optimized).

In order to determine the difference between CUDA and OpenACC, we decided to implement an additional alternative back-end. Since the back-end relies on CUDA for the acceleration with GPUs, the generated code is only usable on clusters with Nvidia GPUs. The results are presented in Table 2: Musket.

By using CUDA for the back-end implementation the execution times can even be further reduced compared to the OpenACC back-end implementation. One difference for instance is the selection of threads per block. For the benchmark, the PGI compiler chooses to use 128 threads per block, while the generated code used

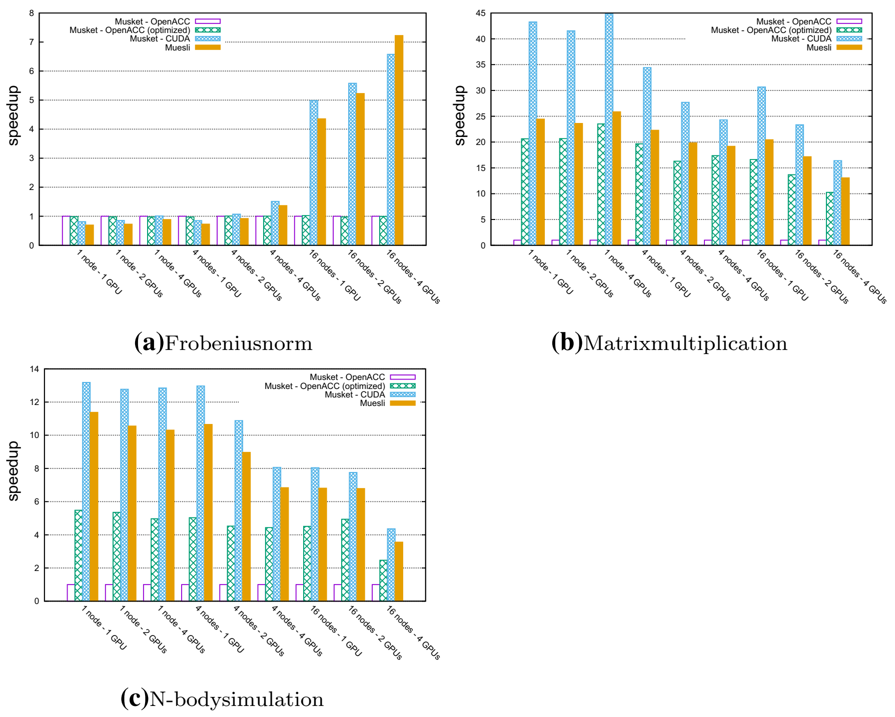
Table 2 Overview of Musket's and Muesli's execution times with their CUDA back-ends (all measures in seconds)

Benchmark	Processes	GPUs	Musket	Muesli
Frobenius norm	1	1	1.4848	1.7364
	1	2	1.4625	1.7209
	1	4	1.4519	1.6517
	4	1	0.4096	0.4778
	4	2	0.3872	0.4509
	4	4	0.3860	0.4269
	16	1	0.1074	0.1229
	16	2	0.1087	0.1160
	16	4	0.1237	0.1126
Matrix multiplication	1	1	52.0596	92.1485
	1	2	27.2736	48.0039
	1	4	14.9162	25.8381
	4	1	16.1760	24.9867
	4	2	10.0826	14.0779
	4	4	6.7614	8.5701
	16	1	4.4089	6.6124
	16	2	2.8685	3.9030
	16	4	2.0349	2.5534
N-Body simulation	1	1	45.3422	52.5339
	1	2	23.6784	28.6561
	1	4	11.6596	14.5347
	4	1	11.6300	14.1635
	4	2	6.9905	8.4848
	4	4	4.7615	5.6140
	16	1	4.7204	5.5778
	16	2	2.4854	2.8402
	16	4	2.2952	2.8144
All pairs of shortest paths	1	1	59.5139	89.2008

1024 threads per block, what seemed to be the more efficient setting. However, these results might not hold true for other benchmarks and architectures.

Next, we want to compare the execution times to benchmark implementations with Muesli. The results are included in Table 2: Muesli. Muesli is a skeleton library, which has been the reference for this approach. Consequently, both approaches incorporate very similar concepts, such as the distributed data structures. As already stated above, the instantiation of data structures, the execution of the skeletons as well as all data transfers and communication, such as the gather skeleton in the N-body simulation benchmark, are included in both execution times. The only difference in the execution is that Muesli uses 512 threads per block and Musket 1024 threads per block as mentioned above. While the results are not fully comparable in that regard, Musket certainly leads to good results compared to the original library implementation Muesli.

A graphical summary of the benchmark results is depicted in Fig. 2. The diagrams show the speedup of the Musket implementations with the different GPU back-ends and the Muesli implementation relative to the execution times of Musket with the unoptimized OpenACC back-end.



(a) Frobeniusnorm **(b)**Matrixmultiplication

(c)N-bodysimulation

Fig. 2 Speedups relative to Musket with the unoptimized OpenACC back-end

5 Related Work

In the following, we will mainly focus on related work in the domain of parallel programming with algorithmic skeletons or parallel patterns. The discussion incorporates other skeleton libraries, DSLs, and transformations of skeletons.

There have been various implementations of skeleton libraries and first, we want to mention selected examples. All the mentioned instances are libraries for C++, which is the first main difference compared to the generation approach followed by Musket.

FastFlow [16] is a skeleton library, which mainly focuses on task-parallel patterns and stream parallelism, but also provides support for data-parallel constructs, such as a parallel for-loop. It has also been extended for distributed systems and for offloading computations on accelerators [17, 18]. The main difference to our implementation is the focus on stream parallelism. For example the extension for distributed systems works in two tiers. The lower tier provides skeleton impelementations for a shared-memory architecture and the upper tier orchestrates the skeletons from the lower tier in the distributed architecture scenario. In contrast, we provide distributed data structures on which the skeletons operate.

SkeTo targets distributed memory architectures and by utilizing C++ template metaprogramming, transformations of skeletons, such as skeleton fusion, are accomplished [19]. Skeleton fusion is for example the fusion of two map skeletons into one, or of a map and a reduce skeleton into a single mapReduce skeleton instance. By applying those transformations, the number of skeleton calls and amount of data movements can be reduced. Instead of relying on template metaprogramming, Musket implements transformations on the skeleton level as model-to-model transformations based on the input model.

The last library we want to mention is SkePu [20]. In version 2, SkePu incorporates a pre-compilation step, which utilizes C++ annotations to generate required code, such as `__device__` for CUDA GPU functions. The implementation of the pre-compilation is based on Clang and LLVM. This follows a very similar concept as the model-driven approach by Musket with the difference that SkePu has standard C++ with annotations as input, while we defined a custom language.

In the following we want to touch upon the differences between internal DSLs, external DSLs, and general purpose languages. As mentioned in Sect. 2, the use of a DSL, in contrast to a general purpose language, can lead to multiple advantages. A DSL increases the level of abstraction and consequently reduces complexity and redundant code. This in turn leads to better readability of models, reduces the development time and increases software quality [6, 7]. For instance, the C++ standard includes parallel algorithms [21] and other concepts such as futures and possibly soon executors. However, it might still be worthwhile to implement a DSL to reduce complexity. In this scenario the Musket DSL provides a higher abstraction and focuses on necessary core features for parallel programming.

The difference between an internal and external DSL is that an internal DSL is embedded into another language. In a broad sense, even OpenMP and OpenACC could be interpreted as an internal DSL. The main advantage of internal DSLs is that they can be non-invasive. By using OpenMP or OpenACC pragmas, it might not be necessary to restructure the original code. Thus, if the pragmas are ignored, the program still contains valid sequential code. In the context of algorithmic skeletons, SPar [22] is an internal DSL for annotating a sequential program, which is then transformed into a FastFlow application. In contrast, Musket provides an external DSL. This reduces the complexity, since there is no need for annotations or pragmas to be embedded within another language.

Finally, we want to mention an approach for generating OpenCL kernels for GPUs. Steuwer et al. [23] proposed a DSL, which incorporates concepts from functional programming. The main aspect of the approach is a set of rewrite rules for algorithmic skeletons. By rewriting high-level expressions in a systematic way using these rewrite rules, it becomes possible to transform the high-level expressions into device-specific low-level implementations. From our perspective this approach follows a different objective. While its focus is on the low-level transformation of expressions for GPUs, we consider Musket being on a higher level with a general focus on usability for parallel programming on different architectures.

6 Conclusion and Future Work

In this paper we have introduced two new generators for the parallel-programming DSL Musket. The first implementation has used OpenACC, while the second implementation has used CUDA. We have described the implementation of the generators and evaluated the generated programs with four benchmark applications. The execution times have shown that our CUDA implementation outperforms OpenACC. Consequently, while it might be worthwhile to maintain a flexible generator based on OpenACC, the focus for Nvidia GPUs should be on a lower-level CUDA implementation.

There are many optimizations, which are not yet included in the CUDA generator, which should be explored in future work. For instance, direct communication between GPUs has not been considered. At the moment, all communication happens through the host main memory. With the increasing bandwidth offered by NVLink, it might be worthwhile to consider alternatives to the current 2-step-approach for distributing data between nodes and GPUs.

Acknowledgements Open Access funding provided by Projekt DEAL.

References

1. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems Jade edition, pp. 359–371. Elsevier, Amsterdam (2012)
2. OpenACC Organization. Openacc (2019)
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1991)
4. Kuchen, H.: A skeleton library. In: B. Monien and R. Feldmann, (eds.) Proceedings of the 8th International Euro-Par Conference on Parallel Processing, volume 2400 of Lecture Notes in Computer Science, Berlin, Heidelberg, pp. 620–629 (2002)
5. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw. **7**(2), 129–138 (2012)
6. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
7. Stahl, T., Völter, M.: Model-Driven Software Development. Wiley, Chichester (2006)
8. The Eclipse Foundation. Xtext documentation (2019)
9. The Eclipse Foundation. Xtend documentation (2019)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)
11. Wrede, F., Rieger, C., Kuchen, H.: Generation of high-performance code based on a domain-specific language for algorithmic skeletons. J. Supercomput. (2019). https://doi.org/10.1007/s11227-019-02825-6
12. Rieger, C., Wrede, F., Kuchen, H.: Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, New York. ACM, pp. 1534–1543 (2019)
13. Kuchen, H.: Optimizing Sequences of Skeleton Calls. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. Lecture Notes in Computer Science, vol. 3016, pp. 254–274. Springer, Berlin (2004)
14. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. Int. J. Parallel Program. **45**(2), 283–299 (2017)
15. Bettini, L.: Implementing Domain-specific Languages with Xtext and Xtend. Community experience distilled. Packt Pub, Birmingham, UK (2013)
16. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multicore. In: Pllana, S., Xhafa, F. (eds.) Programming Multi-Core and Many-Core Computing Systems. Wiley Series on Parallel and Distributed Computing. Wiley, Hoboken, pp. 261–280 (2017)
17. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting Distributed Systems in FastFlow. In: Caragiannis, I., Alexander, M., Badia, R.M., Cannataro, M., Costan, A., Danelutto, M., Desprez, F., Krammer, B., Sahuquillo, J., Scott, S.L., Weidendorfer, J. (eds.) Euro-Par 2012: Parallel Processing Workshops, Berlin, Heidelberg, pp. 47–56 (2013)
18. Buono, D., Danelutto, M., Lametti, S., Torquati, M.: Parallel patterns for general purpose many-core. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 131–139 (2013)
19. Matsuzaki, K., Emoto, K.: Implementing fusion-equipped parallel skeletons by expression templates. In: Morazán, M.T., Scholz, S. (eds.) Implementation and Application of Functional Languages, pp. 72–89. Springer, Berlin (2010)
20. Ernstsson, A., Li, L., Kessler, C.: SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. In: International Journal of Parallel Programming (2017)
21. ISO Standard.: Programming Languages—Technical Specification for C++ Extensions for Parallelism. In: Standard ISO/IEC TS 19570:2015, International Organization for Standardization, Geneva, Switzerland (2015)
22. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: SPar: a DSL for high-level and productive stream parallelism. Parallel Process. Lett. **27**(01), 1740005 (2017)
23. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP '15. ACM, New York, pp. 205–217 (2015)