EasyChair Preprint

№ 5597

Automatic Mapping of Parallel Pattern-based Algorithms on Heterogeneous Architectures

Lukas Trümper, Julian Miller, Christian Terboven and Matthias S. Müller

May 23, 2021

Automatic Mapping of Parallel Pattern-based Algorithms on Heterogeneous Architectures

Lukas Trümper[1,2][0000−0002−0961−7723], Julian Miller[2][0000−0001−5564−0775], Christian Terboven[2][0000−0003−2284−29578], and Matthias S. Müller[2][0000−0003−2545−5258]

[1] Huddly AS, Norway
[2] Chair for High Performance Computing, IT Center, RWTH Aachen University, Germany
lukas.truemper@rwth-aachen.de
{miller,terboven,mueller}@itc.rwth-aachen.de

Abstract. Nowadays, specialized hardware is often found in clusters to improve compute performance and energy efficiency. The porting and tuning of scientific codes to these heterogeneous clusters requires significant development efforts. To mitigate these efforts while maintaining high performance, modern parallel programming models introduce a second layer of abstraction, where an architecture-agnostic source code can be maintained and automatically optimized for the target architecture. However, with increasing heterogeneity, the mapping of an application to a specific architecture itself becomes a complex decision requiring a differentiated consideration of processor features and algorithmic properties. Furthermore, architecture-agnostic global transformations are necessary to maximize the simultaneous utilization of different processors. Therefore, we introduce a combinatorial optimization approach to globally transform and automatically map parallel algorithms to heterogeneous architectures. We derive a global transformation and mapping algorithm which bases on a static performance model. Moreover, we demonstrate the approach on five typical algorithmic kernels showing automatic and global transformations such as loop fusion, re-ordering, pipelining, NUMA awareness, and optimal mapping strategies to an exemplary CPU-GPU compute node. Our algorithm achieves performance on par with hand-tuned implementations of all five kernels.

Keywords: mapping · heterogeneous architectures · global transformations · parallel patterns · performance portability

1 Introduction

Advances in science and engineering are intrinsically linked to computing power. This demand is met with large-scale clusters with many compute nodes. However, these nodes are limited by their power draw, memory performance, and Instruction Level Parallelism (ILP) (cf. *three walls* [2]). Hence, current advances

in hardware are driven by specialization, which leads to clusters comprising heterogeneous architectures. To effectively utilize this quickly evolving landscape of architectures, domain scientists need to continuously adapt their software, which requires a significant development effort and experience.

To reduce these efforts, current parallel programming models introduce a *second layer of abstraction* to decouple the expression of parallelism from the hardware architectures. Thus, an architecture-agnostic source code can be maintained and the optimization for a specific architecture is delegated to the transformations supported by the programming model. E.g., *Kokkos* [13] and *RAJA* [7] utilize data layout rules and *Stateful Dataflow Multigraphs (SDFG)* [8] allows to interactively transform the dataflow of a parallel algorithm for the target architecture.

With increasing heterogeneity, effective utilization of the available processors however requires fine-grained mapping decisions. This mapping needs to match algorithmic properties with processor features, e.g., assigning a compute-intensive part of the algorithm to an appropriate accelerator. Furthermore, architecture-agnostic transformations are necessary to expose global parallelism and execute large parts of the algorithm on different accelerators simultaneously.

To automatize such mappings, this work provides a combinatorial optimization approach to globally transform and automatically map parallel algorithms to heterogeneous architectures. The approach leverages a hierarchical representation of parallel algorithms [22,21] based on *parallel patterns* [19]. This representation allows for the analysis of global properties like synchronization and dataflow through *algorithmic efficiencies* [22,21]. Based on these efficiencies and the *roofline model* [25], structural transformations and a cost-based optimization is derived. Thereby, this paper focuses on the re-ordering and delinearization of routines, the separation and fusion of large subflows within the dataflow as well as data affinity on a global level. Our key contributions are as follows:

- We introduce a static performance model based on algorithmic efficiencies that allows for global transformations of parallel algorithms and their mapping to heterogeneous architectures.
- We derive a transformation and mapping algorithm based on fine-grained splits of the algorithmic structure.
- We demonstrate the approach on five typical algorithms on a modern CPU-GPU architecture. The experiments show the mapping to target architectures with respect to algorithmic properties as well as the application of several global transformations such as loop fusion, re-ordering, pipelining, NUMA awareness, and target offloading.

The remainder of this paper is structured as follows: Chapter 2 briefly summarizes related work. Chapter 3 introduces the used model of algorithms and architectures and Chapter 4 proposes the static performance model. Chapter 5 derives the mapping algorithm and Chapter 6 evaluates the approach for typical data-parallel algorithms. The results of the evaluation and extensions of the approach are discussed in Chapter 7. At last, Chapter 8 provides a conclusion of the paper and summarizes future work.

2 Related Work

This paper focuses on the automatic transformation and mapping of parallel algorithms to heterogeneous architectures. The following sections briefly delineate this paper from related work.

Parallel Programming Models. Recent parallel programming models are designed on top of architecture-specific programming models like *OpenMP* [12], *MPI* [20], and *CUDA* [23]. The architecture-agnostic layer is typically represented by *parallel patterns* [19]. Kokkos [13] uses C++ software abstractions, where the mapping is specified by compile-time parameters. RAJA [7] follows a similar approach by expressing parallelism through loops. The SDFG framework [8] focuses on data parallelism and represents the patterns of a *python* program with graphs. Furthermore, there exists multiple skeleton-based models and interfaces [14]. These models minimize the necessary code changes for porting an application by automatically transforming the code. This paper extends these approaches by exposing additional parallelism, complex global transformations, and automatic mapping to a heterogeneous architecture. This work re-uses the pattern-based representation of algorithms found in the models above.

Transformations. Low-level transformations are applied to improve low-level and code-local performance properties for a specific architecture. Bacon et al. [3] provide an overview of loop transformations to improve the ILP. Many of these transformations are supported by the programming models discussed above and modern production compilers. *Structural transformations* are high-level manipulations of the structure of an algorithm. For instance, throughput-oriented processors typically require flat parallelism, which can be achieved by resolving nested parallelism [9] and nested loops [10]. Furthermore, skeleton-based libraries typically implement different transformation rules [17] like the fusion of skeletons. In contrast to such rule-based approaches, this work transforms and maps an algorithm based on a static performance model. This allows for structural transformations on the highest level of a parallel algorithm. By identifying substructures in the algorithm, fine-granular transformations such as the separation of a sparse dataflow and cache blocking on a routine-level are enabled.

Mapping. Various approaches for mapping a parallel algorithm on heterogeneous architectures exist. A theoretical foundation of the problem is the *MAKESPAN SCHEDULING* on unrelated machines for which different approximation algorithms were proposed [18]. Beaumont et al. [6] investigate the particular problem of matrix-partitioning on heterogeneous architectures and provide several approximation algorithms. The approaches above investigate the mapping within an isolated context, i.e., without a temporal dimension defined by subsequent routines and data dependencies. In this work, however, the mapping is part of an optimization problem over the global structure of an algorithm.

3 Parallel Algorithms

The approach proposes automatic transformations and mappings to heterogeneous architectures at compile time. The mapping algorithm thereby founds on a model of parallel algorithms consisting of a hierarchical decomposition of parallel patterns, which is introduced in [22,21]. This model is particularly applicable to the global analysis of algorithms as it combines two basic ideas making this combinatorially complex analysis feasible. First, it follows the idea of separating the structure of an algorithm from its executed function. This is mainly implemented by abstracting local routines like loops to respective parallel patterns. Second, the model introduces a two-level representation of the algorithmic structure, called the abstract pattern tree (APT). This representation contains local parallel patterns in their global context.

3.1 Parallel Patterns

A parallel pattern is an abstraction of local parallelism as often found in loops and other recurring structures. For this work, they are defined as a directed graph of operations and their data dependencies. An operation consumes and produces data and two operations o_1, o_2 are connected by a direct edge (o_1, o_2) iff o_2 consumes data produced by o_1. Data is thereby interpreted by its instant value, i.e., it does not refer to a memory location and is immutable. The only relevant data dependencies are therefore *true data dependencies* and the investigated graphs are acyclic. Local parallelism is then defined as follows:

- *Earliest-execution-time:* Let $s, o_1, \ldots, o_{n-1}, o$ be the longest directed path from some source of data s to operation o, then o is said to have *earliest-execution-time* n.
- *Parallel:* Two operations o, o' are parallel, iff there is no directed path connecting them and they have the same earliest-execution-time.

A *parallel pattern* comprises at least two parallel operations; the *serial pattern* is defined analogously. Figure 1 illustrates the concept with the example of a *map* pattern where $f_1, \ldots, f_4$ are parallel operations. In the following, local parallelism is assumed to be optimal, i.e., all true data dependencies are well-defined. For the scope of this paper, the three common parallel patterns *map*, *stencil*, and *reduction* as defined by McCool et al. [19] are considered. These patterns are data-parallel and are found in popular *Berkeley dwarfs* [2] like *dense linear algebra*, *spectral methods*, and *MapReduce*.

3.2 Abstract Pattern Tree

The abstract pattern tree (APT) is an internal representation of the structure of an algorithm over the execution order of parallel and serial patterns. Formally, an APT is an undirected graph of pattern nodes (parallel and serial patterns) and two types of meta nodes: The children of a *serial meta node* must be executed from left to right. The children of a *parallel meta node* can be executed

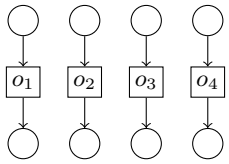

Fig. 1. The map pattern with $o_1, \ldots, o_4$ parallel operations.

in parallel. Furthermore, serial and parallel patterns may themselves be meta nodes to account for the concepts of sub-routines and nested parallelism. The term execution order corresponds to the order defined by the developer and is intentionally different from the data dependencies. As such, the developer may miss concurrency as shown by the linearization of nodes in the APT. However, it is assumed that the developer-provided code is correct, i.e., all true data dependencies are well-defined. An exemplary APT is illustrated in Figure 2.

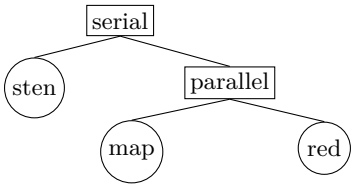

Fig. 2. An exemplary APT of a stencil (sten) and a subsequent parallel map and reduction (red).

The partitioning of operations into sets of parallel operations defines local *algorithmic steps*. Globally, algorithmic steps $STEP_1, \ldots, STEP_T$ combine the local steps according to their relation in the APT. In the following, the set of all operations is denoted $\mathcal{O}$ with $STEP_1 \dot\cup \ldots \dot\cup STEP_T = \mathcal{O}$.

4 Performance Modeling

The following section introduces the performance model, which guides the global transformations and mapping of parallel algorithms. This is based on *algorithmic efficiencies* introduced in [22,21], which are briefly summarized in the following. Algorithmic efficiencies are necessary conditions of performance defined over specific global properties of algorithms. The mapping of operations to processors must therefore be assessed within its global context defined through previous and subsequent data dependencies.

Synchronization. The *synchronization efficiency*'s purpose is to maximize the potential parallelism before mapping the operations to the processors. On the

global algorithmic level, this potential is mainly limited by unidentified concurrency, e.g., due to the linearization of independent parallel patterns.

Inter-Processor Dataflow. Formally, a mapping is defined as a function $M : \mathcal{O} \to \mathcal{P}$ from operations to processors, where each operation must be executed by exactly one processor. A *processor* thereby describes a homogeneous set of *cores* sharing the same processor-level cache. This corresponds to a socket of a CPU or the streaming processor of a GPU in practice. The set of processors of a cluster may be heterogeneous and can comprise different processors on the same device, node and within the same network on different nodes. Without loss of generality, the mapping can be decomposed into a sequence of stepwise mappings $M \to M_1^T$, where $M_t : STEP_t \to \mathcal{P}$ is the mapping of step t. The *inter-processor dataflow efficiency* defines the costs of a mapping through *execution costs* $E_t : \mathcal{P} \times 2^{STEP_t} \to \mathbb{R}$ and *network costs* $N_t : \mathcal{P} \times 2^{STEP_t} \to \mathbb{R}$. The criterion to be minimized by the mapping is defined as follows:

$$\sum_t \max_P \left\{ E_t(P, M_t^{-1}(P)) + N_t(P, M_t^{-1}(P); M_1^{t-1}) \right\} \to \min_{M_1^T} !,$$

where the network costs at step t may depend on previous steps and the execution costs only depend on the current step. The costs can thereby be modeled based on assumptions of existing performance models. In the following, the costs are adopted from the *roofline model* [25], where the assumption of the overlap of execution is relaxed:

- *Execution costs:* The execution of operations is captured by the number of floating point operations (Flops) divided by the peak performance π_P (clock frequency times Flops per cycle):

$$E_t(P, M_t^{-1}(P)) := \frac{\sum_o Flops(o)}{\pi_P}, o \in M_t^{-1}(P).$$

- *Network costs:* The network costs are defined as the slowest data transfer between two processors. A data transfer thereby bundles all bytes to be transferred from one processor to another to satisfy the data dependencies. The bandwidth $\beta_s(P', P)$ is determined by the slowest interconnect between these two processors and a latency penalty $\Gamma_s(P', P)$ is added:

$$N_t(P, M_t^{-1}(P)) := \max_{P'} \left\{ \frac{\sum_{(o',o)} BYTES((o', o))}{\beta_s(P', P)} + \Gamma_s(P', P) \right\},$$
$$o \in M_t^{-1}(P), o' \in M_{1\ldots t-1}^{-1}(P').$$

Intra-Processor Dataflow. Given a mapping M_1^T, the *intra-processor dataflow efficiency* seeks to optimize the execution of parallel operations assigned to the same processor. This includes the scheduling on its cores, the utilization of core-local caches, and transformations on the instruction-level to allow overlapping execution of operations of different steps via asynchronous techniques. It is a subsequent optimization after the mapping to the processors, and it is assumed to be optimized by downward compilers (cf. Chapter 2).

5 Mapping Algorithm

The mapping and transformation algorithm consists of two steps: First, the synchronization efficiency is optimized by re-ordering the APT's nodes to resolve false linearization. Second, the actual mapping is derived by sequential optimization over the algorithmic steps with respect to the inter-processor dataflow efficiency. An implementation of the presented algorithm is included in the supplemental material.

5.1 Step 1: FlatAPT

The serial meta nodes of the APT encode the algorithmic steps defined by the developer. To resolve false linearization, the children of a serial meta node need to be re-ordered according to their actual data dependencies. If two such children are parallel, the parallelism is encoded by re-inserting them as the children of a new parallel meta node into the respective sub-tree.

Due to potential parallelism across nested serial meta nodes, local optimality does not directly lead to global optimality. Instead, the APT is traversed in-order and every node is added to a new *FlatAPT*. This FlatAPT consists of only a single serial meta node, the main node, and parallel meta nodes at the second level, the algorithmic steps. A new pattern node is added as the child of a new parallel meta node to the end of the FlatAPT, which introduces a new (last) step. The node then *bubbles up* the algorithmic steps as long as there are no true data dependencies and becomes the child of an earlier parallel meta node. Since the FlatAPT consists of only a single serial meta node, the FlatAPT is constructed globally optimal. Note that the nested parallelism is not leveraged at this stage and, thus, the algorithm omits the traversal of sub-trees defined by a parallel pattern or parallel meta node.

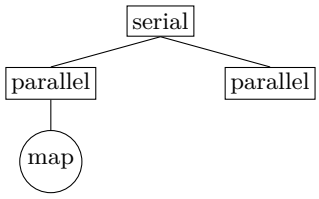

Fig. 3. Exemplary structure of a FlatAPT with a map inserted and an empty second step already created.

The worst-case computational complexity occurs if every node of the APT needs to be traversed and compared to every preceding node in the FlatAPT at insertion, i.e., every node bubbles up into the first step. Hence, the algorithm holds a quadratic worst-case complexity to the number of nodes. The feasibility and complexity of the actual data dependence analysis for each pair of nodes

depend on the static information provided by the programming language as described by Hennessy et al. [15] and Banerjee et al. [5].

5.2 Step 2: Step Mappings

The inter-processor dataflow efficiency is modeled as a sequential optimization problem over the sequence of step mappings M_1^T. The optimization is thereby restricted to the following mapping space:

- *Team:* A team reserves a number of cores on a processor. The set of all possible teams is denoted $\mathcal{T}$.
- *Pattern split:* A pattern split is a subset of the parallel operations of a parallel pattern, e.g., an interval of independent iterations of the same loop.
- *Step mapping:* A step mapping is refined as a function $M_t : SPLITS_t \to \mathcal{T}$ from pattern splits to teams. There is at most one active team per processor and step.

Accordingly, the recursive formulation of the efficiency is considered instead:

$$Q(M_1^t) = \min_{M_1^{t-1}} \left\{ d(M_t; M_1^{t-1}) + Q(M_1^{t-1}) \right\},$$
$$d(M_t; M_1^{t-1}) = \max_{P} \; E_t(P, M_t^{-1}(P)) + N_t(P, M_t^{-1}(P); M_1^{t-1}),$$

where $d(M_t; M_1^{t-1})$ is the inter-processor dataflow efficiency of only the step mapping M_t given the history M_1^{t-1}. The optimization problem associates three decision dimensions yielding the general structure of the optimization algorithm:

- *Teams:* At each step t, select k out of K teams.
- *Assignment:* Partition the N pattern splits into k sets.
- *Time:* Assess the step mapping within the temporal context and extend the current hypotheses of partial mappings.

However, this optimization problem does not admit *optimal substructure* to the end that a globally optimal mapping can be constructed from locally optimal step mappings. Because of arbitrary temporal dependencies in the network costs, series of suboptimal step mappings might enable particularly efficient step mappings in the last algorithmic steps. By considering the full search space, the global optimum can be obtained. However, the combinatorial complexity then grows exponentially in all three dimensions K, N, T. In the following, approximation techniques are proposed to reduce this complexity.

Time. Data dependencies are assumed to be limited to a maximum length of m steps. The mapping is therefore constructed greedily over time with a lookahead of m steps:

$$Q(t) = \min_{M_t, M_{t+1}, \ldots, M_{t+m}} \{ d(M_t; B(t-1)) + Q(t-1; B(t-2)) + \sum_{\tau=1}^{m} d(M_{t+\tau}; B(t-1+\tau)) \},$$

where $B(t)$ is a *traceback array* storing the actual step mappings of previous steps. The approximating assumption leverages the *principle of locality*. Furthermore, data dependencies refer back to the operation that initially created the data. In practice, the data may also be read by another team in the meantime and it is then transferred from this team. This effectively shortens the temporal length of dependencies.

Assignment. For a given set of teams, only the locally optimal assignments of pattern splits concerning the current step are considered. This transforms the optimization over step mappings M_1^T to one over teams U_1^T. The estimation of the locally optimal assignment is approached by the modeling as an instance of the NP-hard *MAKESPAN SCHEDULING* on unrelated machines [18]; teams are distinguishable in execution costs by hardware characteristics and network costs for accessing remote memory locations. The main difference to the original problem is that pattern splits may share data, i.e., data only needs to be loaded once from memory for two different splits. This invalidates the assumption of independent costs for the splits and the optimality bounds of existing heuristics for the original problem. The assignment is therefore based on a generic *branch-and-cut* algorithm as shown in the prototype implementation in the supplemental material.

Teams. At each step, the set of teams is determined by a variant of *local search*. The basic idea is to gradually extend an initial set of teams consisting of a single CPU team. The search compares different strategies for an extension, called moves, and follows the direction of the best move similar to *hill climbing*. Two phases of this search are distinguished that differ by the type of moves to be compared: At first, the search progresses by *local moves*. Such moves may only add new teams located on devices used by the current teams, e.g., another socket of a CPU. When a local optimum is reached, the set of teams is extended with new devices and nodes called *jump*. These two types of moves are necessary in order to avoid a collapse of hypotheses. The assignment may lead to a *subset selection* of teams by not assigning splits to some teams. This frequently occurs when extending to a new node and too few teams are added to compensate for the additional network costs. Thus, local optima of this search are always plateaus and the optimization space consists of different levels of plateaus. This variant of a local search is therefore denoted *stair climbing*.

Analysis. While the mapping based on the inter-processor dataflow efficiency is a general scheduling problem, the proposed algorithm cuts it into many local assignment problems by explicitly exposing the optimization over time and teams. Assuming the stair climbing algorithm needs to search the whole cluster, but each move doubles the number of teams on average and only two moves are compared at a time, the computational complexity of this outer optimization regarding the assignment is $T \cdot (K \cdot a(N))^{1+m}$, where $a(N)$ is the complexity of each assignment. Assuming $m = 1$, K to be constant and T to be linear in N, the complexity is $\mathcal{O}(a(N)^3)$.

6 Evaluation

In the following, the proposed optimization framework for global transformations and automatic mappings is evaluated regarding its approximation quality and achieved local optima. Since the approximation quality cannot be derived analytically for practical problems, an empirical analysis of typical data-centric algorithms such as linear equations, classifications, and neural networks is used. Each algorithm is implemented in a *baseline version* closely following its numerical definition. Furthermore, a *hand-tuned version* is derived by applying crucial transformations and mapping decisions as found in the literature and library implementations. The proposed framework transforms and maps the baseline versions, which is denoted *auto version*. The resulting optimizations are then compared to the hand-tuned version including a qualitative analysis of the optimization as well as a quantitative analysis of the estimated and actual runtimes. Once the implementation of the approach as a fully integrated tool is finished, a comparison with related approaches will be carried out.

Experimental Setup The experiments were executed on two typical CPU-GPU nodes connected via Intel® Omni-Path with 100 Gb/s. Each node features two Intel® Skylake® Platinum 8160 processors with 24 cores each, a base frequency of 2.1 GHz, disabled HyperThreading, and 192 GB of DDR4 RAM. Two NVIDIA® Tesla® V100 GPUs with 16 GB of HBM2 memory are connected via PCI-express. The experiments were repeated 30 times and median measurements are reported.

We implement the algorithms in a *Parallel Pattern Language (PPL)*, from which the APT can be parsed directly. After optimization of the APT, the resulting mapping is manually translated into C code using the parallel programming models OpenMP 4.5, OpenMPI 3.1.3, and CUDA 10.2; the baseline versions are directly implemented in C. The OpenMP versions were compiled with gcc 9.3.0 and compiler flags *-fopenmp -std=c99 -O2*, the MPI versions with gcc 9.3.0, OpenMPI 3.1.3, and compiler flags *-fopenmp -std=c99 -O2*, and the CUDA versions with the NVIDIA compiler 10.2, gcc 8.2.0 and compiler flags *-Xcompiler -fopenmp*. If the generated pattern and data splits are large on the GPU, the block size is reduced to 512 due to the limits of the CUDA framework. The work assigned to a specific processor is statically scheduled to its cores. The implementations of the parallel algorithms, as well as the mapping reports of the optimization framework, can be found in the supplemental material.

Parallel Algorithms. The following parallel algorithms are investigated. Depending on the problem size of the evaluated parallel algorithms, different hyperparameters for the optimization algorithm are chosen. They steer the granularity of the pattern and data splits and were derived experimentally.

The *Multi Filter* algorithm represents a typical image processing task, where different filters are applied to the same image independently. In detail, the task is to compute different image derivatives: A *Sobel* filter estimates horizontal derivatives on the upper half of the image and a *Prewitt* filter on the lower half.

Furthermore, a discrete Laplacian is applied to the whole image. The size of the image is 8194×8194. The baseline version processes the three filters sequentially. The used pattern and data split sizes are 4096 and 128 respectively.

The *Batch Classification* represents a typical classification task, where the elements of a batch are processed in multiple stages separately. The first two stages extract the features by first standardizing and then integrating over the absolute inputs. The third stage classifies based on the majority vote of 4096 thresholding classifiers. To highlight potential pipelining and fusion transformations, the baseline version processes the batch in reverse during the second stage. The size of the batch is 2^{19} and each input element is a vector of size 4096. The used pattern and data split sizes are 262144.

Three linear equation systems are solved with the *Jacobi method* sharing the same coefficient matrix A. The number of unknowns is 8192 in each linear system and the number of Jacobi iterations is fixed to 50. The baseline version solves the linear systems sequentially. The used pattern and data split sizes are 4096.

The *Neural Network* algorithm defines the forward pass of an eight-layer neural network for a batch of size 2^{18}. Each layer consists of 64 neurons. The respective matrix-matrix computations admit massive data parallelism and acceleration potential. The used pattern and data split sizes are 8192.

The *Monte Carlo Pi* algorithm defines a Monte Carlo method for approximating π by accumulating the area of a unit circle. The approximation is obtained by averaging over 96 independent estimations with 10^9 random draws each. The used pattern and data split sizes are 24 and 1 respectively.

6.1 Results

The cost estimations and runtimes are reported in Table 1 for the unoptimized baseline version and the automatically transformed and mapped version.

	Cost [s]		*Runtime [s]*		
Algorithm	*Base*	*Auto*	*Base*	*Auto*	*Transformations*
Multi Filter	0.013	0.010	0.038	0.030	Fusion, re-ordering, pipelining, NUMA awareness
Batch Classification	0.475	0.396	1.157	0.898	Fusion, pipelining
Jacobi	0.913	0.637	1.320	0.561	Fusion, re-ordering, pipelining
Neural Network	0.167	0.009	0.329	0.175	GPU offloading
Monte Carlo Pi	13.611	6.944	43.449	22.238	distributed computing

Table 1. The estimated runtime in seconds and the measured runtime in seconds for the baseline version (base) and the automatically transformed and mapped version (auto). Moreover, the applied transformations by the proposed framework are listed.

Multi Filter: The optimization identifies the independence of the filters and the shared input data. It fuses the Sobel operator and the Laplacian splits iter-

ating over the upper half of the image; the Prewitt and the remaining Laplacian are fused analogously. Hence, the halves of the images are pipelined and kept in the local caches across the different filters. The baseline version is estimated at 0.013 s and measured at 0.038 s. The estimated runtime costs of this automatic mapping are 0.010 s and measured 0.030 s. Due to the short runtime of the algorithm, the average of 10 repetitions with random input is used.

Batch Classification: Due to the integration over absolute values and the thresholding classifiers, the computation is dominated by branching. The framework accounts for this property by mapping to the two CPU teams of a single node with 24 cores each. Furthermore, the mapping identifies the pipelines for each input and fuses the extraction and classification stages. The estimated runtime costs for the baseline are 0.475 s while the measured runtime is 1.157 s. The estimated runtime of the automatic mapping is 0.396 s and the actual runtime is 0.898 s.

Jacobi: The framework maps the algorithm to the two CPU teams of a single node. It thereby integrates the three linear systems in a single Jacobi pass. Furthermore, the splits across the different linear systems comprising the same rows of A are assigned to the same teams. Accordingly, the same teams are used for corresponding splits over different Jacobi iterations. Hence, the respective rows of A are pipelined across the different linear systems and iterations. The baseline version is estimated at 0.913 s and measured at 1.320 s while the cost of the automatic mapping is 0.637 s and the runtime is 0.561 s.

Neural Network: The proposed framework offloads the massive data parallelism of the algorithm to a GPU. Due to the induced data transfers for a multi-GPU implementation, a single GPU is preferred by the performance model. The costs for the CPU baseline version is 0.167 s and its runtime is 0.329 s. The automatically mapped version is estimated at 0.009 s and measured at 0.175 s.

Monte Carlo Pi: The automatic mapping distributes the splits of the 96 pi estimations across four CPU teams located on two different nodes. The mapping thereby leverages the low communication costs for separating the sparse dataflow initially. The cost of the baseline version that only utilizes a single node is 13.611 s and the measured runtime is 43.449 s. The automatic version is estimated at 6.944 s and measured at 22.238 s.

7 Discussion

The paper introduces an automatic mapping algorithm utilizing a model of algorithmic efficiencies and cost-based combinatorial optimization. The evaluation investigates its approximation quality empirically on a representative set of parallel algorithms. The obtained optima consistently improve the runtime and their qualitative analysis shows that these optima comprise a rich set of global transformations as applied by an HPC expert. In the following, the results and next steps are discussed in detail.

Static Analysis. While the static analysis requires complete information about local parallelism, real-world problems often contain dynamic characteristics. To

overcome this challenge, the information about the exact sizes of data structures may be relaxed to approximative estimates. Furthermore, the definition of local parallelism via a parallel pattern is general enough to be defined over concrete structures of existing programming models like directives or C++ templates, from which the APT can be parsed. This information is available for a wide set of scientific algorithms implemented in modern parallel programming models.

Evaluation. The integration of the algorithm into a parallel programming model is the next step towards a thorough evaluation of the approach. This implementation enables a performance comparison on typical benchmarks such as NAS Parallel Benchmarks [4] with existing approaches and an evaluation on proxy and real-world applications regarding the approach's practicability. The critical part of the integration is thereby the optimal use of further architecture-specific transformations as applied by Kokkos, Raja, and modern compilers. For instance, current data representations and team definitions may be refined for compatibility with other models and devices, e.g., CUDA thread blocks.

Performance Modeling. The comparison of the estimated costs for the baseline and the automatic mapping shows that the mapping decisions are well-guided by the high-level performance model. The difference between measured and estimated runtime varies by a factor of $0.9-3.2$ except for the neural network, where the GPU mapping is underestimated by a factor of 19. While the model provides rich efficiency information, the implementation currently renounces architecture-specific features required for higher estimation accuracy. For instance, describing the peak performance solely via the number of cores oversimplifies the *SIMT* execution model of modern GPUs since it implies independent execution of threads on cores. The model also excludes architecture-specific acceleration features like *vector registers*, which could be included with the *LogCA model* [1]. Moreover, the overlap between execution and network costs is not investigated in detail and the modelled network costs assume that the communication of a single step occurs simultaneously in a single transfer. This latter assumption overestimates the bandwidth if many teams use the same interconnect simultaneously. This could be improved by utilizing the *LogP model* [11].

Computational Complexity. The computational complexity of the mapping algorithm is cubic with respect to the complexity of the assignment algorithm. Approaches for the assignment based on LP-relaxations and the simplex algorithm are themselves at least cubic on average [16]. Hence, efficient heuristics must be derived, which for instance group similar patterns and leverage the symmetry of modern architectures. Furthermore, the pattern and data split sizes allow to control the feasibility in practice. However, large split sizes may result in a coarse granularity of the analysis, where the degree of parallelism is reduced significantly and artificial data dependencies are introduced. To balance this, a pattern-specific split size could be applied instead of the global parameter. As this increases the number of hyperparameters, an algorithmic strategy for

determining these hyperparameters is an important extension. Evolutionary algorithms based on a parallel model like the *island model* [24] may be considered.

8 Conclusion

In this paper, we investigate the problem of mapping parallel algorithms to heterogeneous architectures. To this end, we introduce a static performance model that minimizes the global execution costs. This includes global transformations to improve the dataflow and expose parallelism as well as utilizing large clusters efficiently while minimizing the required data transfers. We show that the proposed algorithm delivers complex optimizations as carried out by HPC experts and execution performance similar to such hand-tuned versions for five typical parallel algorithms. Our proposed approach is extensible in several directions: The performance model could be extended with architecture-specific features to increase its accuracy on a wide set of applications and hardware platforms. Furthermore, the computational complexity of the optimization algorithm may be lowered by heuristics and auto-tuning methods. We will integrate the approach into existing parallel programming models to conduct a larger performance evaluation on a wide set of benchmarks.

Acknowledgement We thank Adrian Schmitz (RWTH Aachen University) for implementing the prototype language PPL and useful discussions. We also thank Huddly AS for supporting Lukas Trümper in this work.

References

1. Altaf, M.S.B., Wood, D.A.: LogCA: A Performance Model for Hardware Accelerators. IEEE Computer Architecture Letters **14**(2), 132–135 (2014)
2. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A View of the Parallel Computing Landscape. Commun. ACM **52**(10), 56–67
3. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler Transformations for High-performance Computing. ACM Comput. Surv. **26**(4), 345–420
4. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The nas parallel benchmarks summary and preliminary results. In: Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. pp. 158–165. IEEE (1991)
5. Banerjee, Shyh-Ching Chen, Kuck, Towle: Time and Parallel Processor Bounds for Fortran-Like Loops. IEEE Transactions on Computers **C-28**(9), 660–670
6. Beaumont, O., Becker, B.A., DeFlumere, A., Eyraud-Dubois, L., Lambert, T., Lastovetsky, A.: Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. IEEE Transactions on Parallel and Distributed Systems **30**(1), 218–229
7. Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.: RAJA: Portable Performance for Large-Scale Scientific Applications. In: 2019 IEEE/ACM International

Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 71–81
8. Ben-Nun, T., de Fine Licht, J., Ziogas, A.N., Schneider, T., Hoefler, T.: Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19, Association for Computing Machinery, New York, NY, USA (2019)
9. Chakravarty, M.M.T., Keller, G., Lechtchinsky, R., Pfannenstiel, W.: Nepal — Nested Data Parallelism in Haskell. In: Sakellariou, R., Gurd, J., Freeman, L., Keane, J. (eds.) Euro-Par 2001 Parallel Processing. pp. 524–534. Springer Berlin Heidelberg
10. Chen, C., Chame, J., Hall, M.: Chill: A framework for composing high-level loop transformations. Tech. rep., Citeseer (2008)
11. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: Logp: Towards a realistic model of parallel computation. In: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–12. PPOPP '93, ACM
12. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
13. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202–3216, domain-Specific Languages and High-Level Frameworks for High-Performance Computing
14. Gonzàlez-Vèlez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience **40**(12), 1135–1160
15. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fifth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., 5th edn.
16. Klee, V., Minty, G.J.: How Good is the Simplex Algorithm? Inequalities **3**(3), 159–175
17. Kuchen, H.: Optimizing Sequences of Skeleton Calls. In: Domain-Specific Program Generation, pp. 254–273. Springer (2004)
18. Lenstra, J.K., Shmoys, D.B., Tardos, É.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical programming **46**(1), 259–271 (1990)
19. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming - Patterns for Efficient Computation. Elsevier (2012)
20. Message-Passing Interface Forum: A Message-Passing Interface Standard, http://www.mpi-forum.org/
21. Miller, J., Trümper, L., Terboven, C., Müller, M.S.: A Theoretical Model for Global Optimization of Parallel Algorithms, Manuscript submitted for publication, RWTH Aachen University, Germany.
22. Miller, J., Trümper, L., Terboven, C., Müller, M.S.: Poster: Efficiency of Algorithmic Structures. In: IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC19)
23. NVIDIA, Vingelmann, P., Fitzek, F.H.: CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit
24. Sudholt, D.: Parallel Evolutionary Algorithms. In: Springer Handbook of Computational Intelligence, pp. 929–959. Springer (2015)
25. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM **52**(4), 65–76 (2009)