Parallel Pattern Language Code Generation

Adrian Schmitz
a.schmitz@itc.rwth-aachen.de
IT Center, RWTH Aachen University
Aachen, Germany

Julian Miller
julian.miller@rwth-aachen.de
IT Center, RWTH Aachen University
Aachen, Germany

Semih Burak
burak@itc.rwth-aachen.de
IT Center, RWTH Aachen University
Aachen, Germany

Matthias S. Müller
mueller@itc.rwth-aachen.de
IT Center, RWTH Aachen University
Aachen, Germany

Abstract

Memory and power constraints limit the current landscape of high-performance computing. Hardware specializations in clusters lead to heterogeneity, Non-Uniform Memory Architecture (NUMA) effects, and accelerator offloading. These increase the complexity of developing and optimizing scientific software.

To ease these challenges for domain scientists, the code generator for a prototype of the Parallel Pattern Language (PPL) is implemented, enabling its evaluation. The proof of concept uses parallel patterns to define parallelism and apply static global optimizations automatically. Most notably, an assignment between tasklets and the provided heterogeneous cluster architecture is calculated during compile time, the new code generator creates a source file combining shared-memory, distributed-memory, and accelerator offloading according to the generated mapping.

The prototype successfully optimizes and compiles most Rodinia benchmarks. Six Rodinia benchmarks already show significant speedups. The tools limitations include dynamic algorithms that are challenging to analyze statically and overheads during the compile time optimization.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; **Concurrent programming languages**; **Distributed programming languages**.

Keywords: parallel patterns, shared memory, gpu offloading, distributed systems, code generation

ACM Reference Format:
Adrian Schmitz, Julian Miller, Semih Burak, and Matthias S. Müller. 2024. Parallel Pattern Language Code Generation. In *The 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '24), March 3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3649169.3649245

1 Introduction

The memory- [29] and power-wall [7] restrict an HPC system's scaling, previously dominated by increasing core counts. To continue satisfying the increasing computational demands, systems are equipped with specialized hardware like GPUs to leverage higher throughputs and energy efficiency than generic CPUs. Similarly, multiple levels of memory are used to satisfy the ever-increasing data demands. The installation of the Frontier system at the Oak Ridge National Laboratory marks the first exa-scale system, utilizing a heterogeneous GPU system with NUMA CPUs [13]. Given this heterogeneity, the development of efficient and portable scientific codes is increasingly challenging. This challenge in heterogeneity is often approached by HPC performance engineers, introducing typical challenges in the knowledge transfer toward domain scientists. Thus, limiting the long-term maintainability and extensibility of an application by the domain scientist.

One approach for improving maintainability while providing performance and portability are hardware-specific compiler optimizations. The PPL provides such an approach with a high-level, descriptive pattern language. It statically applies application-wide (global) optimizations including the assignment of parallel workloads to the targeted architecture as introduced in previous work [32, 47]. To perform those global optimizations, the intermediate representation (IR) provides as much information as possible by applying parallel patterns. Previous work [43] already introduced a high-level IR, defining the semantic understanding of parallelism and data flow for specific parallel patterns.

This work introduces a code generator of the PPL tool completing the entire prototypical toolchain. The provided code generator follows a source-to-source approach generating optimized parallel source code for shared-, distributed-memory, and GPU offloading.

The completed prototype implementation is further evaluated against the Rodinia benchmark suite [8], covering typical HPC applications. The measurement setup consists of the original OpenMP version as a baseline, a single CPU node, and a distributed setup with CPUs and GPUs. Furthermore, a port of the LULESH [21] proxy-app is discussed as a benchmark for scalability. Based on the evaluation, the current state of the prototype is assessed highlighting issues regarding the current Linear Programming (LP) based global optimization and the code generator. The path towards a productized version is drafted.

The following main contributions are provided with this work:

- The implementation of a code generator for the PPL prototype tool toolchain.
- A prototypical implementation of inlining and loop unrolling for the Abstract Pattern Tree (APT).
- An evaluation of the tool/concept against nearly all kernels from the Rodinia benchmark suite.
- A port of the LULESH proxy-app to the PPL.
- An analysis of the current issues of the PPL prototype including possible solutions for a release version.

The remainder of this work is structured as follows. Section 2 provides an overview of related work regarding the PPL. A brief explanation on the existing toolchain is presented in Section 3. While Section 4 introduces the new code generator and the APT inlining, Section 5 evaluates the PPL toolchain against the existing OpenMP implementation of Rodinia. Finally, the results are concluded and an outlook on future work is provided in Section 7.

2 Related Work

Recurring structures for the parallelization of hotspots can be found in most parallel algorithms. Many well-known programming models make use of such patterns including OpenMP [39], MPI [34], and CUDA [11].

The automatic optimization of parallel programs is tackled in different ways by many. In most cases, they focus on specific fields for optimization, such as polyhedral compilers [15, 30, 33] to optimize polyhedral loop nests, Korali [28] for Bayesian uncertainty and stochastic, or Halide [35] for image processing. Especially GPU code is often optimized by specialized compiler, such as the work of Pai et al. [41] focusing on graph algorithms. The PPL approach targets a broader range of applications and hardware by separating the parallel semantics from the usable hardware while leaving the degree of parallelism and utilization to the optimization.

In contrast to the wide spread production code generators included in LLVM [23] and gcc [12] the PPL does not generate assembly directly. Instead the PPL tool is a source to source compiler, generating C++ code compatible with any C++17 capable compiler. Thus, combined with the preferred C++ compiler to generate assembly a toolchain is constructed.

The source to source transpilation is also a well known aspect of software engineering, simplifying the porting process to more recent technologies, such as C2Rust [45] which transforms C99 code into Rust. In contrast to the porting aspect of the transcompilers the PPL utilizes a custom Domain Specific Language (DSL) to enable specific optimization, instead of a syntactical transformation between two predefined languages.

Code generation and DSL frameworks are well established tools applied for simulations and are highly tuned for very specific application. For example, OpenSBLI [27] generates fluid dynamics code on unstructured grids for heterogeneous architectures. The work of Kempf et al.[22] provides a code generator for the discontinuous galerkin method to apply vectorization. The PPL aims to provide a much wider applicability, focusing on an entire application when generating optimized code. Lift [44] utilizes parallel patterns specified in OpenCL [38] to optimize and generate GPU code. In contrast, the PPL also provides support for shared memory and distributed memory systems. Tools like YASK [50] or the work of Li et al. [25] only generate code for optimized stencil kernels in their DSL approach. The PPL however supports the map, reduction, and stencil patterns allowing a broader range of applications to benefit from the approach. Furthermore, most of the applied code generation frameworks only generate individual kernels to be used in conjunction with existing applications for management. The PPL follows a more holistic approach with its code generation by generating the entire application.

Skeleton libraries [10] like Müsli [9] follow a similar approach of lifting parallelism as a reusable set of instructions. Libraries like Thrust [37] or the Intel thread building blocks [20] add to the vendor driven skeleton libraries. Microsoft parallel pattern library [48] also directly applies the notion of parallel patterns in their library. Kokkos [46] and Raja [24] pursue an approach covering a complete node. Such libraries often only consider a single node while the PPL targets a complete cluster system including shared memory, offloading, and distributed memory.

SDFGs and the DaCe tool [4, 5] follow a rule-based optimization strategy to apply possible optimizations to hotspots in the computation. These optimizations need to be defined by an HPC expert once and can then be performed either automatically or semi-automatically in the SDFG interface. By allowing more freedom during the optimization and mapping process, the PPL tool might find solutions not yet proposed. Especially application wide optimizations might be considered, while the SDFGs focus more on local optimizations.

The Legion framework [3] targets the same system configurations as the PPL, including CPUs, GPUs, and distributed memory. In contrast to the PPL, the focus of Legion is the framework itself and enabling user-defined mappings while this work tries to enable automatic global optimizations.

3 Parallel Pattern Language Prototype

The PPL prototype is a full toolchain to verify the applicability of the proposed pattern-based approach to real-world problems and environments. The authors have already shown that many algorithms can be represented using such parallel patterns [43]. The current prototypical toolchain is constructed as a modular source-to-source compiler with five stages:

1. Parsing: Evaluation of the user input.
   - DSL: Application sources using parallel patterns.
   - Hardware Language (HWL): JSON description of the target cluster hardware.
2. APT Generation: Statically generates the hierarchical APT representation of the algorithm.
3. Optimization: Perform global optimizations on the APT, e.g., reordering and explicit hardware mapping.
4. Abstract Mapping Tree (AMT) Generation: Generate an AMT from the optimization results to add necessary synchronization and data transfers.
5. Code Generation: Generate optimized C++ code from the AMT.

The first four stages are already discussed by the authors [32, 43, 47], and a brief overview is provided in this section. The new code generator and an implementation of function-inlining/loop-unrolling on the APT are introduced in Section 4. An overview is available on our public GitHub [1].

[1]https://github.com/RWTH-HPC/PPL/

```
1   //Pattern: multiply every element in input by x and store it
        in output
2   map multX([Int] input, Int x) : [Int] output {
3       output[INDEX] = input[INDEX] * x
4   }
5
6   //Pattern: subtract every element in input by x and store it
        in output
7   map subX([Int] input, Int x) : [Int] output {
8       output[INDEX] = input[INDEX] - x
9   }
10
11  seq main(): Int {//program entry point
12      var [Int] x = init_List([4096]) //Array with 4096 elements
13      var [Int] y = init_List([4096]) //Array with 4096 elements
14
15      y = multX<<<>>>(x, 2) //perform multX from x to y
16      x = subX<<<>>>(y, 1) //perform subX from y to x
17
18      ... //some other computations with x
19  }
```

Listing 1. Two data dependant map patterns performing a multiplication and a subtraction on each element of an array

3.1 Parsing

Defining a custom DSL is a relatively fast way of developing and testing an approach encompassing a global view on an application. Nevertheless, the DSL is capable of representing most applications with a few limitations to reduce the development effort [43]. The DSL follow a mathematic-algorithmic approach, applying a CUDA inspired notation to highlight the element-wise definition of parallel patterns. The frontend makes use of parallel patterns by integrating them directly into the language via two main components:

Parallel pattern calls define the entry point for a parallel pattern. Examples of parallel pattern calls are in Lines 15 and 16 of Listing 1.

Parallel pattern definitions define the actual computation that includes which parallel pattern is used. An example can be found in Lines 2-4 and 7-9 in Listing 1.

This in turn allows the exploration of an optimal partitioning of multiple parallel patterns based on the target hardware. The current DSL requires statically defined array sizes compared to typical malloc-based pointer arithmetic and focuses on pure functions. The static array lengths enable the PPL to derive the size of a parallel pattern since the DSL does not follow a loop-based structure like OpenMP. The limitation to pure functions simplifies the data dependency analysis significantly since actual dependencies cannot be hidden via aliasing or global variables.

Listing 1 depicts an example in the DSL frontend, computing $x = 2 * x - 1$ for each element in the array `x`. Variable `y` defines the result from the multiplication as an interim result for each array element (see Lines 15,16). The computations are defined in Lines 2-4 and 7-9 respectively. `INDEX` states that the computation is performed for every element in the given arrays.

The HWL allows a definition of execution units by the programmer. These execution units can be an arbitrary number of cores evenly dividing the original device. For example a device with twelve cores execution units spanning twelve, six, four, three, two, and one core are possible.

3.2 APT Generation

The APT is a high-level hierarchical representation of parallel code. All nodes fall into two distinct categories:

- Expressions: Defining fixed computations, e.g., $a = b + c$.
- Statements: Defining control-flow and spanning a local variable scope, e.g., loops. All nodes defined in this scope are descendants of this node.

As special cases, function calls count as expressions, referencing the sub-APT of a function to limit memory consumption. Pattern calls are statements defining a parallel control-flow. The corresponding partial APT can be expanded for further analysis.

The limitation toward linear array accesses within parallel patterns allows the compiler to derive fine-grained data dependencies statically. All data dependencies must be related to an expression, since only those nodes are able to perform computations. The available dependencies can be passed to its ancestors as long as the accessed data element is still within scope, allowing a dependency analysis on multiple levels.

The PPL derives the APT from an abstract syntax tree by lifting the semantic information on parallel patterns. The representation of parallel patterns allows optimization preprocessing, such as partitioning the patterns. The size of these partitioned patterns (tasklets) is currently a static parameter defined by the user.

3.3 Global Optimization

Global optimizations are changes applied on an application level, like loop fusion or data flow optimizations between devices [32]. The PPL uses algorithmic efficiencies, the *synchronization*, *inter-* and *intra-processor dataflow efficiencies*, to evaluate potential mappings and optimizations during compile time [31, 32].

1. *Synchronization efficiency:* Minimize dependency based synchronization.
2. *Inter-processor dataflow efficiency:* Minimize the runtime of tasks T on a set of execution units E.
   - Network Cost: Minimize data transfer time.
   - Execution Cost: Minimize execution time.
3. *Intra-processor dataflow efficiency:* Maximize data reuse on the same execution unit.

Each efficiency describes a step in the global optimization pipeline. The *synchronization efficiency* utilizes the dataflow to optimize the APT by reordering it into steps. Each node within Step $n + 1$ must have a data dependency towards a node in the prior Step n. Nodes without any input dependency, such as data initialization, are placed in Step 0. This ordering maximizes the amount of parallelism available with data-independent tasklets by removing the unnecessary sequential ordering.

The *Inter-processor dataflow efficiency* defines an optimization problem to minimize the codependent network and execution costs. The current implementation [47] approximates the complete problem by iterating over the steps generated by the *synchronization efficiency*. Each step is optimized based on the previous step and a lookahead over the next n steps. The execution cost is estimated using an adapted Roofline Model [49]. The network cost is estimated by the available bandwidth/latency and the corresponding data. The resulting makespan-optimization utilizes the static information provided by the PPL, tasklets, and hardware and defines a Mixed Integer Linear Program (MILP). To solve the corresponding MILP representation, the Gurobi solver [16] is applied. Gurobi is a prominent proprietary solver for linear-programming. Since MILPs are limited in their support for uncertainties, the support for nested parallel patterns is restricted to allow optimizations for a subset of problems.

The *Intra-processor dataflow efficiency* targets the mapping generated from the *Inter-processor dataflow efficiency.* Tasklets on the same execution unit are ordered, or fused if possible, to improve caching.

3.4 AMT Generation

The AMT extends the APT by including the optimizations and mapping data from the previous stage. Thus, the APT is a single CPU representation, while the AMT is a heterogeneous, distributed and concurrent representation. The assignments can be integrated as a parameter per tasklet and are a direct representation of the mapping results. The data structure is extended to include the representation of tasklets, assignments to specific groups of cores defined in the hardware language as cache groups, new nodes to represent synchronization and data transfers, and special nodes for offloaded parallel patterns and GPU memory management. Data transfer and synchronization extensions are derived from the dataflow and mapping information iteratively.

First, the global optimization may assign multiple tasklets stemming from the same parallel pattern onto the same GPU, by grouping them, the GPU utilization is improved and synchronization reduced. The data management and synchronization is generated iteratively by deriving the sequential order of the computations from the data flow. Depending on the "distance" between the execution units and the access type, three classes are derived:

1. **Device local dependencies** can be solved by only synchronizing, assuming shared memory systems, enabling pattern-independent read and write accesses.
2. **Remote write** accesses do not need any synchronization, since they do not risk a write-after-write data race on separated device memory (disregarding unified memory and RMA operations). In this case, it is sufficient to invalidate any copies of the written data, setting the only valid location the data available at to the current device.
3. **Remote read** accesses require synchronization and data transfers between the owner of a local copy and the device requesting the data. By creating a new copy of the data on another device, further read accesses can utilize a potentially closer device to access the same data. This, at the cost of an increased memory footprint, can reduce the network traffic.

Due to their limited memory capacity, GPUs only hold data required by subsequent computations.

4 Code Generator

The code generator has a significant influence on the performance of a source-to-source compiler. For the first complete proof of concept, the goal is not to derive the best runtime from the get-go, but rather to discover flaws and bottlenecks in the approach and enabling future improvements. Following the concept of globally optimizing code for a heterogeneous system, any possible implementation needs to support the following features to represent the optimization as closely as possible:

1. Support for shared, distributed memory, and offloading programming models.
2. Pinning of workload to individual execution units.
3. Data movement across all devices.
4. Synchronization between any two execution units in the system.
5. Implementation of all supported patterns for each target architecture.

The provided implementation is aligned to all of these features. The PPL prototype utilizes a combination of PThreads [36], MPI [34], and CUDA [11] to feature their respective programming models covering the first constraint.

The assignment of work to individual execution units required a complete thread pool implementation since both the individual assignments of tasklets and the explicit synchronization between changing sets of cores is not available in existing approaches sufficiently. For instance, in OpenMP it is not possible to assign tasklets to specific cores explicitly, which is required to fully support the designated optimizations. The thread pool runs on each node covering the shared and distributed memory domains. For GPU offloading, assigning work to individual cores works against the inherent structure of GPUs and therefore dropped. Instead, the GPU is always regarded as a single execution unit, limiting potential serialization and improving the hardware utilization for GPU tasklets.

To simplify the data transfers and avoid relying on specific CUDA-aware MPI implementations, data exchanges are separated into two levels: CPU-CPU and CPU-GPU. This separation allows independent implementations via MPI and CUDA respectively while enabling data transfers between distributed GPUs with a combination.

Similarly, synchronization needs to be handled on three different levels to cover the hierarchical structure of the hardware. Intra-device covers the synchronization on a single device. On the CPU the thread pool covers intra-device synchronization. On the GPU, except for the implicit synchronization with a reduction pattern, which required a complete implementation of the reduction in CUDA for all supported combiner functions. Intra-device synchronization is not necessary since the GPU is not divided into further execution units as stated before. The intra-node synchronization handles the synchronization of all accelerators with the host. The code generator employs CUDA and the thread pool to synchronize the CPU and GPUs on a single node. The generated code makes use of blocking end-to-end communication since the optimization only requires synchronization for data dependencies, implying a previous or subsequent data transfer. The implementation of each parallel pattern is supported using the thread pool and CUDA. MPI is not directly required for the calculation except the reduction pattern.

4.1 Shared Memory

To fully utilize the results provided by the global optimization, the code generator needs to be able to directly assign tasklets to an execution unit defined in the hardware language. These execution units can theoretically be chosen to cover anything between a single hardware thread up to the whole CPU. To properly evaluate the necessity of such a fine granular optimization, the code generator needs the ability to respect those degrees of freedom. A possible assignment of workload to a specific thread/core is required. Tasking distorts the evaluation of the proof of concept via dynamic load balancing and thus cannot be employed yet. The integration of dynamic load balancing might still be a reasonable approach to cover problems that cannot be properly analyzed during compile time, e.g., graph algorithms or adaptive mesh refinement.

To enable the five features defined at the beginning of this section, a wrapper for pthreads is necessary. This library developed for the code generator supports these five features as follows:

Programming model Shared memory parallelism is natively supported by using multiple POSIX threads. The amount used is defined in the HWL and corresponds to the number of cores specified, pinning the thread to its corresponding core. Hyper-threading is currently disregarded. The execution units are loosely grouped, by assigning parallel tasklets to the targeted subset of threads.

Individual assignments The following two conditions need to hold to allow for an assignment of tasklets to individual execution units. First, a workload or tasklet needs to be definable during compile time. Second, the defined tasklet needs to be assignable to a specific execution unit during compile time. To simplify these conditions and follow the general design of pthreads, a simple master-worker based thread pool is used, allowing a singular definition of the threads.

To be able to add tasklets with different configurations into the same queue, they are stored as lambda functions `std::function<void()>` for deferred execution. Due to the pureness of the parallel pattern, all input- and output-arguments can be added into the scope of the lambda function by reference, ensuring identical signatures. Since all data dependencies are known beforehand, data races across lambda functions are avoided via synchronization.

The assignment to CPU cores can be performed using the `pthread_setaffinity_np` unix kernel callback. This is done by iterating over all cores and pinning the corresponding to a `CPUSET` only containing the id of the current core. To ensure MPI progress and avoid costly context switches, the main thread is pinned to Core 0 by default. This thread handles the integration of all three programming models and computes sequential workloads. All other cores are propagated to the thread pool. Thus Thread 0 will run on Core 1. While this reduces the achievable performance slightly, the dedicated main thread sped up synchronizations by 3 - 50 times depending on the number of threads, compared to the main thread sharing a core with a worker thread. The generated lambda functions can then be added directly to the task queue of the targeted core, the assignment is being handled by a dedicated main thread, which also takes care of the integration towards offloading and distributed memory.

Data movement The data transfers are handled by the main thread to integrate offloading and distributed memory. Data transfers on the device are handled by the operating system.

Synchronization To support synchronization between an arbitrary set of cores on the CPU, barrier tasklets are employed. Such barrier tasks are possible because task migration is not supported and local task queues strictly adhere to a FIFO ordering. The individual barrier tasks are added to the queues of the corresponding threads. The `pthread_barrier_wait` function defined in the POSIX standard only takes the number of threads. Thus, ensuring that the thread fence targets only the required threads, potentially including the main thread.

Pattern implementation The map and stencil patterns are already decomposed into tasklets without any further dependencies, thus they do not require additional work. The reduction pattern is further decomposed into a map and reduce step. The map step is handled like a map pattern and performs all operations on a private copy of the reduction variable. In the reduce step, all threads participating in shared memory are synchronized and an additional tasklet combines all private results in a single value and atomically adapts the final output value. The atomic access is used to include a participating GPU on the same node without risking data races. The integration of distributed memory supports the five features as follows:

4.2 Distributed Memory

In the context of distributed memory, many relevant alternatives exist, like GASPI [1] or OpenSHMEM [42]. However, the most mature and versatile programming model in the context of distributed memory programming is MPI, which was shown capable of implementing the OpenSHMEM standard [17]. Therefore, MPI is selected for code generation.

Programming model By distributing the shared-memory thread pool and the accelerator offloading across multiple nodes, distributed memory in a hybrid MPI + X approach is supported.

Individual assignments The individual assignments are covered by assigning the MPI `rank` in the same order as they are defined in the HWL. For example, Rank 0 corresponds to the first node etc.

Data movement The data transfers are explicitly covered by MPI communication. Following the potential irregularity gained from the current optimization approach compared to a rule-based approach, the use of other collective communication patterns is not yet evaluated. Potential collective communication would need to be addressed via pattern matching, which is known as a feasible solution [19].

Synchronization The multi-level synchronization allows to first synchronize the main thread and other node local execution units. Afterwards, the main threads synchronize via MPI to ensure that all required data is available before progressing. The synchronization between main threads is implicitly provided by the blocking MPI communication.

Pattern implementation Only the reduction pattern requires explicit MPI support, since map and stencil are sufficiently supported by running distributed with the shared memory or offloading programming models. For the reduction pattern, the MPI reduction is reused in conjunction with the node local reduction

4.3 Accelerator Offloading

The current prototype makes use of CUDA and an adapted thread pool to simplify addressing multiple GPUs from a single process.

Programming model The current prototype explicitly separates the (de-)allocations and data movement from the kernel to exploit memory locality across multiple kernels, as intended by the global optimization. The used CUDA calls are wrapped and compiled in a different file to be linked at a later step, avoiding compiler errors with non-CUDA-aware MPI implementations. To simplify targeting specific GPUs, the shared memory thread pool provides a task queue per GPU. By using `cudaSetDevice()` the corresponding thread is mapped to a specific GPU. The kernels are also generated into the same file as the CUDA calls. The current implementation does not support different parallel patterns to be mapped to the same GPU at the same time. Concurrent execution of patterns can be realized with asynchronous data transfers and CUDA streams but would require additional synchronization.

Data movement The data management is handled by wrapping the functions `cudaMalloc`, `cudaFree`, and `cudaMemcpy`. Since a dedicated thread takes care of offloading, the data management and computations can be concurrent with CPU computations.

Synchronization To ensure the correct ordering of CUDA calls only a single stream is used per GPU. Thus, synchronization is only necessary in case of a data transfer. For data on the GPU that would entail read operations on the output data of the GPU, thus `cudaMemcpy` is sufficient to synchronize the GPU with its dedicated thread. To allow a general usage of the GPU data with other threads including the main thread, the corresponding shared memory threads in the thread pool need to be synchronized, using the shared memory synchronization.

Pattern implementation For the pattern implementation, the optimization decides explicitly how many iterations of the original pattern are assigned per core, called the execution range. The execution range defines the number of iterations each thread needs to perform, with an initial value of $\lfloor n_{iters}/n_{thread} \rfloor$. Each thread is identified by `tid = blockIdx.x * blockDim.x + threadIdx.x` commonly used in CUDA for work-sharing. The size of a user-defined pattern is not necessarily evenly divisible by the GPUs warp size. To account for the remainder, the execution range is incremented for the first n threads, where $n = n_{iters} \% n_{thread}$. Afterwards, the thread loops over all assigned iterations by replicating the original `INDEX` while accounting for the remainder. `INDEX` $= t * r + i + n$, where t is the thread id, r the execution range, i is the current local iteration, and n the remainder size, represents the current array element. For the first n threads, the remainder is covered in the execution range, thus removing the offset of n again. To avoid additional branch divergence, the execution range is always handled in the same branch: `if (tid < n)`.

The reduction pattern further requires additional code for the intra-block and inter-block reduction. Both of these are already solved problems that could be adopted for the code generator [18]. As explained in the shared memory section, the access to the reduction variable on the CPU is performed atomically to simplifying a single reduction parallelized across multiple devices on a single node.

4.4 Prior Optimizations

To allow for the generation of parallel code, the optimization has to recognize the potential parallelism first. But the optimization of parallel patterns needs to strictly adhere to the semantics of the MILP specification of the global optimization. To enable the optimization of parallel patterns nested into generic function calls or loops of fixed length, inlining and loop unrolling need to be implemented from within the APT, to allow a proper flattening of the hierarchical structure before the optimization. Algorithm 1 shows the general steps of the inlining process within the APT. Given a function call to be inlined, the tool first queries the corresponding function declaration and defines a list to store the nodes replacing the original call, here called `replacer`. To avoid potential scope overlaps of variables/parameters sharing the same name, they are extended by a hash generated in Line 3. In Lines 4-6, the arguments are copied. This enables concurrency between nested parallel patterns sharing the same data. For input data, only the reference to an array is copied to avoid performance degradation. Due to the strict read-only policy for function arguments, nested parallel patterns must write to a new local array, avoiding write-after-write and write-after-read dependencies. To enable the use of `return` not only as the last statement within a function, the algorithm also creates a jump label corresponding to the function call which can be referenced whenever a *ReturnNode* is encountered (cf. Lines 7 and 9-11). These jump statements ensure that code that is not encountered for specific parameters will not be executed, since it could change the state of the computation. The jump label node is also responsible for deallocating any local data. All other children of the function node are deep copied and references to local variables in all child nodes are replaced with the hashed version of the same variable in Lines 12 and 13. In the Lines 16 and 17, the jump label is placed and the variable replacing the function call is assigned. The `call.replacer` variable encapsulates the return values of the original function and is used to replace the inlined function call in expressions. Since the PPL defines function calls as individual nodes with an expression, it suffices to set the value of `call.replacer` to an initialized value. During the code generation, the tool can then either print the variable or the original call. The nodes in the replacer list are added as new children before the current call node.

Algorithm 1 Simplified APT function inlining.

```
1: $node$ ←getFunction(call)
2: $replacer$ ←[]
3: $call.inlineHash$ ←genHash()
4: for all $param$ ←node.parameters do
5:     $replacer$[ ] ←createCopyAndAssignment(param, call)
6: end for
7: $JumpLabel$ ←newJumpLabel(call)
8: for all $child$ ←node.children do
9:     if $child$ == ReturnNode then
10:        $replacer$[ ] ←JmpWithDealloc(JumpLabel, node.scope)
11:    else
12:        $copy$ ←deepCopy(child)
13:        $replacer$[ ] ←replaceVars(copy, node.scope, call)
14:    end if
15: end for
16: $replacer$[ ] ←newJumpLabelNode(JumpLabel)
17: $call.replacer$ ←assignLocalResult(node.result)
18: return replacer
```

For loop unrolling the algorithm can be adapted by repeating the loop in Line 8 N times. To avoid uncertainties in the data transfers, the algorithm should disregard loops containing any `break` or `continue` statement. Thus, the management of jump labels/statements can be omitted. Further, parameters and result values can be omitted as well. Only the loop body needs to be included N times for a loop with N iterations including the update of the loop variable after every inlined iteration. The loop variable must be initialized with the correct value beforehand. The deep copy and variable replacements work analogously.

5 Evaluation

The optimization and the code generator are the two major influences for the generated code. To this end, the evaluation is split into two parts. First, the evaluation of the code generator using five examples from prior work [47], comparing hand written with generated code for identical optimizations. Second, an evaluation of the full toolchain against the OpenMP version of the Rodinia benchmark suit [8], providing a good coverage of the Berkeley Dwarfs [2]. The evaluation setup and its results are detailed in the following.

Table 1. Average runtime in seconds over 40 runs of the optimization benchmarks. The parallel baseline is compared to hand-tuned and PPL generated versions.

	CPU batch	CPU jacobi	MPI monte	CPU multi	GPU neural
Baseline (parallel)	1.040	1.057	12.431	0.068	0.311
Hand-tuned	0.797	0.531	5.953	0.072	0.236
Generated	0.913	1.590	21.919	0.043	0.162

5.1 Environment

All measurements are performed on the CLAIX18 systems of the RWTH Aachen University [2]. The systems are equipped with a two socket Xeon Platinum 8160 24C 2.1GHz (Skylake) and 192 GB RAM for the CPU, each connected by an Intel Omni-Path 100G network fabric. The nodes run Rocky 8.9 as an operating system. For GPU resources, the nodes are additionally equipped with two NVIDIA V100 GPUs. All codes are compiled with the Intel®OneAPI C/C++ Compiler 2022.1.0, IntelMPI 2021.6.0, and CUDA 11.8 using the highest optimization level *-O3*.

To ensure the correctness of the measurements, global optimizations are performed without any measurements or artificial synchronization points since they significantly influence the optimization process by adding additional dependencies, mitigated utilizing a smaller set of execution units compared to the optimal solution. Thus, all time measurements are added by hand into the generated code. The average wall-clock times of 40 repetitions are used and represented in seconds.

Gurobi [16] further introduces instabilities into the scheduling process. As a result, mappings with large differences in the resulting cost are encountered for identical inputs. This effect stems from the random seed Gurobi uses to calculate an initial solution. By using multiple seeds, it is possible to mitigate the issue, in this paper five random seeds are used. The Rodinia benchmarks show no deviations, but two of the generator test cases show strong differences between Gurobi seeds.

5.2 Generator Results

To evaluate the generated code, three different versions of the same parallel code are compared for five different kernels. The kernels contain a *batch* classification with feature extraction (CPU only), a *jacobi* solver for three equation systems (CPU only), a *monte*-carlo estimation of pi (CPU + MPI), a *multi*-filter convolution for images (CPU only), and a *neural* network forward pass with eight layers (GPU). The comparison includes a naive CPU parallelization of the kernels as a baseline, a handwritten implementation of the optimizations applied by the PPL, and a version completely generated by the PPL toolchain. The *monte* and *neural* kernels generate unstable results in the MILP optimization, the other three kernels do not indicate such deviations.

Table 1 presents the measurements for all three versions of the five kernels. The *batch* kernel shows a slow down of 17% compared to the handwritten code and a speedup of 14% compared to the baseline, in case of the *multi* kernel even speedups of 58% could be achieved. Aside from the *jacobi* kernel where the kernel fusion is

[2]https://blog.rwth-aachen.de/itc/en/2019/06/25/claix-2018-spitzenplaetze/

Table 2. Average runtime in seconds over 40 runs of the OpenMP Rodinia benchmarks and a PPL port optimized for a CPU system and a system with two compute nodes additionally equipped with 2 GPUs each.

	original	*PPL*		Speedup
		CPU	CPU-GPU	CPU/CPU-GPU
backprop	0.0609	0.0049	0.0075	12.43/8.12
heartwall	3.1097	3.8238	<-	0.81
hotspot	0.0137	0.0114	<-	1.20
hotspot3d	0.2283	0.0840	<-	2.72
kmeans	0.4711	0.3676	<-	1.28
lavaMD	102.5557	29.4010	<-	3.49
nn	0.0013	0.0004	<-	3.25
particle	0.0373	0.5469	11.5939	0.07/0.01
srad	0.0145	0.0045	<-	3.22

not yet implemented, the CPU kernels show performance improvements compared to the baseline and competitive results compared to the handwritten code. For distributed kernels, a slowdown can be measured, mostly attributed to the implementation of the local reduction pattern. Intels MPI implementation is highly optimized, less optimized MPI implementations, like OpenMPI, show speedups of 15% - 20% for the generated code with runtime of about 20 seconds. By overlapping computation and communication, the generated code for the *neural* kernel could achieve additional speedups of up to 45%.

5.3 Rodinia Results

Rodinia [8] is a benchmark suite designed around the berkeley dwarfs [2], aiming to provide coverage over most HPC applications. Therefore, these benchmarks well suited to evaluate the applicability and coverage of the optimized and generated code. The runtime of the 19 Rodinia OpenMP benchmarks is compared to an adapted port for the PPL in previous work [43]. Both versions are changed identically to fix errors detected by the compiler and add timing measurements to all benchmarks. The largest provided data set/configuration defined by the scripts shipped with Rodinia is used for all available benchmarks. The PPL port is adapted to also fit those sizes. Since the Rodinia codes do not show a deviation between different seeds the first measurement is used. Both versions are tuned, the original to the optimal number of cores and the PPL version automatically by the toolchain. Furthermore, the handwritten kernels for the *kmeans* and *nn* benchmarks are optimized to better utilize the parallelism, achieving almost six and 57 times faster runtimes in the original Rodinia code. The sources and measurement results are added to the auxiliary files accompanying this work[3]. Table 2 depicts the nine Rodinia benchmarks optimized by the PPL, fields marked with "<-" show the same optimization and runtime as the CPU optimization.

Ten codes from the Rodinia OpenMP version are omitted from Table 2 and Figure 1 for five different reasons. The list below states why and how many codes are implicated for each reason:

Not supported (3): This class covers the test cases, which are either not ported to the PPL (*b+tree*, *mummerGPU*) or have their

[3]https://github.com/RWTH-HPC/PPL/tree/master/Samples/rodinia-ppl

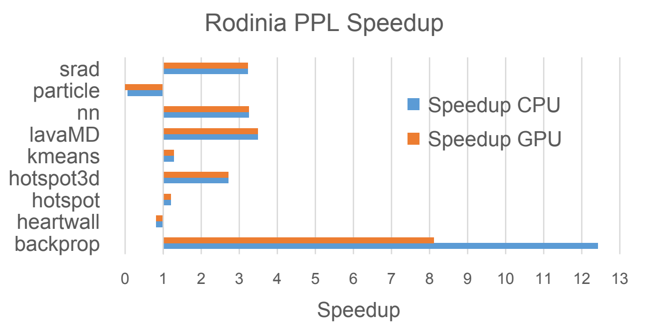

Figure 1. Visualization of the Rodinia PPL speedups compared to the parallel baseline of the original Rodinia C/C++ implementation.

kernel significantly altered such as the *leukocyte* benchmark splitting the kernel into separate applications. This limitation is caused by the current DSL frontend and the basic dependency analysis.

Dynamic control-flow (2): Benchmarks in this category could not be optimized properly, because their kernel is nested into a dynamic control structure (*bfs*, *streamcluster*), which cannot be represented by the LP implementation of the mapping algorithm.

Large search space (3): These benchmarks (*cfd*, *nw*, *pathfinder*) can be optimized in theory but the search space is significantly larger than other applications, e.g., the *cfd* solver needs to schedule at least 64,000 inter-dependent tasklets onto 324 execution units creating a search space too large to be addressed by MILPs in a reasonable time.

Dynamic workload (1): The LU-decomposition (*lud*) manages the creation of a triangular matrix with branches. This structure cannot be analyzed statically at the moment, causing large load imbalances for a static assignment assuming a homogeneous workload for each tasklet.

Memory overhead (1): The *myocyte* kernel requires to much memory per thread in the generated code.

For the *hotspot* benchmark, the previously implemented tiling is not possible in the DSL. Thus, the original code is rewritten into an element-wise iteration, and the corner cases for the stencil are excluded from their patterns. The static nature of the PPL combined with the extraction of corner cases avoided almost all branches within the kernel.

The global optimization and especially the scheduling can decide to only use a subset of the available hardware if the cost of data transfers and additional synchronization is larger than the benefit of additional parallelism. Due to the small sizes of the datasets, the PPL decides to utilize just the CPU resources of a single node for most applications. Only the *particle* benchmark makes use of GPUs if they are available. For multi-node parallelism, the *particle* and *backprop* kernels utilize a second node, when it is available.

Due to the comparable short runtime of many applications they mainly benefit from the elimination of synchronization, including the *backprop*, *hotspot*, *kmeans*, *nn*, and *srad* benchmarks. This static reduction reduction in runtime leads to speedups of 1.20 up to 12.43.

Hotspot3D can at most utilize eight threads due to the current parallelization of the array structure, i.e., only the outer most loop, with eight elements, is parallelized. The PPL does not enable the use of complex stencil borders, thus extracting them into separate kernels in the PPL version. Since the original could utilize at most eight threads, allowing for a parallel execution of edge cases and the main stencil leads to a speedup of 2.72.

The original *heartwall* benchmark is well optimized, leading to a 0.81 slowdown for the generated code.

The *lavaMD* benchmark shows a speedup of more than two for the code generated by the PPL. This speedup is induced by optimizations from the Intel®OneAPI compiler, without optimizations (-O0) both versions perform roughly the same with 290 seconds.

6 Discussion

The Rodinia benchmarks highlighted significant benefits of the approach in terms of performance. Especially, for smaller applications a reduction of static overhead is significant. This is especially prevalent with the *backprop*, *nn*, and *srad* benchmarks. In case of the *lavaMD* benchmark, the PPL could even enable subsequent compilers to apply much more aggressive optimizations reaching speedups of up to 3.5 times, as visualized in Figure 1.

The PPL approach shows great potential with its current workflow for static codes, enabling automatic, global optimization for heterogeneous architectures from a single source. A single source is especially productive for scientific applications that typically outlive the clusters they were originally developed for. Instead of porting the code to a new cluster, the PPL would only require changes in the hardware description of the new target system and potential tuning of the cache group and tasklet sizes.

Moreover, the proposed approach of abstracting the parallelism into parallel patterns removes typical boilerplate code, which improves the programming productivity and maintainability for domain experts, as shown in previous work [43]. In contrast to that, a significant amount of applications are already written in C/C++ or Fortran, having to port such applications is a significant effort that many developers will not want to shoulder. Three Rodinia benchmarks are not ported as a result of the limitations introduced in the DSL front-end. To address the reusability of source code and simplify the porting process, C/C++ support should be introduced. Annotations similar to OpenMP could be used to identify parallel patterns.

The code generator is already providing competitive performance for CPU and GPU programming. While a slowdown of 17% for the *batch* kernel, compared to hand optimized code, is significant. A speedup of 14% compared to the parallel baseline could be achieved. However, the *jacobi* and *monte* kernels do not achieve speedups. In case of the *jacobi* kernel, the relevant optimization (kernel fusion) is yet to be implemented. For the *monte* kernel, the custom reduction implementation, addressing potential shared-, distributed-memory, and offloading at the same time, is not optimal. Relying on existing shared memory/hybrid implementations of a distributed reduction would benefit the runtime, similar to the applied GPU reduction.

Codes with dominant dynamic behavior, e.g., *b+tree* and *bfs*, are not well suited for the static approach applied by the PPL, since they would need a way of adapting to different workloads during runtime. An extended data dependency analysis could help to identify dynamic workloads during compile time. SDFGs [4] already apply a more detailed dependency analysis. SDFGs enable aliasing elimination during compile time. Additionally, a dependency chain for all remaining variables can be extracted enabling the identification of runtime arguments in control flow structures and parallel patterns. To enable this analysis for the PPL, either the dependency analysis needs to be replicated or the a compatibility layer between SDFGs and the APT needs to be developed.

Once runtime-dependent applications can be identified statically, the code generation can be addressed accordingly. The dynamic solution needs to be able to react to different levels of uncertainty since not only the cost of a single pattern but also the number of patterns might be dynamic. Legion [3] would enable the execution of patterns/tasklets of unknown size but dynamic load-imbalances between patterns cannot be addressed. StarPU [14] additionally enable the migration of tasklets in distributed memory during runtime. Both StarPU and Legion enable the use of GPUs. Thus, generating code for StarPU would be the most feasible approach to support dynamic applications at the moment.

Three Rodinia benchmarks that could not be optimized in over 24 hours because of their exponentially growing search space. As a result, scalability of the tasklet scheduling is not feasible. A change in variable names, which are influenced by the random seed, can have a drastic effect on the duration of the solving process [26]. It is not reasonable for the user to find a good random seed. An alternative to LP-solvers would be a custom implementation of beam search to create more predictable duration for the scheduling and also eliminate the risk of unstable solutions. Beam search has been applied successfully to scheduling problems for a long time[40]. Applying beam search entails the implementation of pruning strategies required to limit the search space. The pruning strategies, e.g., n-best, provide means of controlling the compile-time while providing a stable and reproducible result. The work of Birgin et al. [6] applies beam search on a machine scheduling problem of consecutive production steps in a manufacturing process with multiple machines and products. Their approach is similar to the scheduling problem encountered during the global optimization, with one key difference, requiring a corresponding adaption. The distance between machines is not yet considered but can be added as a function depending on the source, destination, and amount of data, all of which is available during the optimization. The use of a more stable solving process further allows a potential integration of parallelism within dynamic control flow structures, such as loops and branches. This avoids the additional load of unrolling large loops and enables the parallelization of additional applications, including *streamcluster* and *cfd*.

The memory model of the code generator should also be improved to avoid local memory copies, impacting both the *particle* and *myocyte*. For normal functions, pureness is not a strict requirement and avoiding copies is therefore not trivial. A new global optimization step to eliminate local copies could be introduced. Depending on the size of the data and potential parallelism, the optimization could decide whether a local copy or an access by reference would be more beneficial.

To analyze the PPL toolchain on a larger application a port of the LULESH mini-app is available[4]. LULESH [21] is a physical simulation proxy application for a three dimensional wave propagation. The code is designed for weak scaling and is originally implemented as a hybrid MPI+OpenMP application. The code utilizes iterative solvers as an approximation for the underlying physics as well as an iteration over time steps for the entire wave propagation. While the LULESH port runs and the time stepping could be unrolled for a fixed number the iterative stencil kernels containing the propagation logic make up about 60% of the sequential runtime. Since the nested patterns would also require parallelism LULESH only highlights the same issues regarding dynamic structures already identified using Rodinia. While the LULESH port shows that the PPL can represent larger applications a detailed analysis of a larger application provides only a small benefit for the development while smaller less complicated applications already show the same dynamic behavior.

The global optimization approach is applicable to parallel programs in general, as indicated by the port of Rodinia in previous work [43]. Parallel patterns are used to reduce the analysis complexity. By adapting the analysis and view global optimizations could also be implemented for other parallel programming models like OpenMP.

7 Conclusion and Future Work

The implementation of the code generator enables the use of global optimizations [32, 47] and the intermediate representation [43]. The combined prototype provides a toolchain to automatically optimize and assign workloads to heterogeneous architectures based on a single, high-level source code. This significantly improves the developer's productivity and code maintainability, while targeting performance portability. The proposed prototype can compile and run most of the Rodinia benchmarks on cluster systems involving shared memory, distributed memory, and GPU offloading. By eliminating static overhead and utilizing additional parallelism, speedups of up to 12.43 could be achieved compared to the already parallel Rodinia benchmarks. The optimization prototype even revealed additional concurrency in the *kmeans* and *hotspot3D* benchmark resulting in speedups of 1.20 and 2.72 compared to the original OpenMP version.

In future work, the following limitations will be addressed. First, the code generator will be further improved to better utilize MPI. Especially the reduction implementation should rely more on existing solution to improve efficiency. Second, the memory overhead needs to be reduced by avoid/removing local copies. Next, the scalability of the scheduling step will be addressed by evaluating custom pruning strategies and scheduling algorithms like beam search. Support for C/C++ will be added to reduce the porting effort for applications and testing. Finally, dynamic applications could be addressed by extending the code generation towards existing dynamic load balancers.

References

[1] Thomas Alrutz, Jan Backhaus, Thomas Brandes, Vanessa End, Thomas Gerhold, Alfred Geiger, Daniel Grünewald, Vincent Heuveline, Jens Jägersküpper, Andreas Knüpfer, et al. 2013. Gaspi–a partitioned global address space programming interface. *Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing* (2013), 135–136.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. The landscape of parallel computing research: A view from Berkeley. (2006).

[3] Kihiro Bando, Steven Brill, Elliott Slaughter, Michael Sekachev, Alex Aiken, and Matthias Ihme. 2021. Development of a discontinuous Galerkin solver using Legion for heterogeneous high-performance computing architectures. In *AIAA Scitech 2021 Forum*. 0140.

[4] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the*

[4]https://github.com/RWTH-HPC/PPL/tree/master/Samples/lulesh-ppl

International Conference for High Performance Computing, Networking, Storage and Analysis. 1–14.

[5] Tal Ben-Nun, Tiziano De Matteis, Oliver Rausch, Carl Johnsen, Saurabh Raje, Andreas Kuster, Philipp Schaad, Manuel Burger, Neville Walo, Luca Lavarini, et al. 2019. *DaCe-Data Centric Parallel Programming.* Technical Report. SLAC National Accelerator Lab., Menlo Park, CA (United States).

[6] Ernesto G Birgin, João Eduardo Ferreira, and Débora Pretti Ronconi. 2020. A filtered beam search method for the m-machine permutation flowshop scheduling problem minimizing the earliness and tardiness penalties and the waiting time of the jobs. *Computers & Operations Research* 114 (2020), 104824.

[7] Pradip Bose. 2011. *Power Wall.* Springer US, Boston, MA, 1593–1608.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54.

[9] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. 2009. The münster skeleton library muesli: A comprehensive overview. (2009).

[10] Murray Cole. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA.

[11] Massimiliano Fatica. 2008. CUDA toolkit and libraries. In *2008 IEEE Hot Chips 20 Symposium (HCS)*. 1–22.

[12] Free Software Foundation. 2023. *GCC 13.2 Manual.* https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/ Accessed: 2024-01-02.

[13] Prometeus GmbH. [n. d.]. Top500 List (November 2022). https://www.top500.org/. Accessed: 2023-11-30.

[14] Maxime Gonthier, Loris Marchal, and Samuel Thibault. 2022. Locality-Aware Scheduling of Independent Tasks for Runtime Systems. In *Euro-Par 2021: Parallel Processing Workshops: Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers*. Springer, 5–16.

[15] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly: performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22 (2012), 1250010.

[16] Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[17] Jeff R. Hammond, Sayan Ghosh, and Barbara M. Chapman. 2014. Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*. Springer International Publishing, Cham, 44–58.

[18] Mark Harris et al. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 70.

[19] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. 2010. Scalable communication protocols for dynamic sparse data exchange. *ACM Sigplan Notices* 45, 5 (2010), 159–168.

[20] Intel Corporation. [n. d.]. *oneAPI Threading Building Blocks.* Intel Corporation. https://github.com/oneapi-src/oneTBB

[21] I Karlin, J Keasler, and J R Neely. 2013. LULESH 2.0 Updates and Changes. (7 2013). https://doi.org/10.2172/1090032

[22] Dominic Kempf, René Heß, Steffen Müthing, and Peter Bastian. 2020. Automatic code generation for high-performance discontinuous Galerkin methods on modern architectures. *ACM Transactions on Mathematical Software (TOMS)* 47, 1 (2020), 1–31.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[24] Lawrence Livermore National Laboratory. [n. d.]. *RAJA Performance Portability Layer.* Lawrence Livermore National Laboratory. https://github.com/LLNL/RAJA

[25] Mingzhen Li, Yi Liu, Hailong Yang, Yongmin Hu, Qingxiao Sun, Bangduo Chen, Xin You, Xiaoyan Liu, Zhongzhi Luan, and Depei Qian. 2021. Automatic code generation and optimization of large-scale stencil computation on many-core processors. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–12.

[26] Andrea Lodi and Andrea Tramontani. [n. d.]. Performance Variability in Mixed-Integer Programming. In *Theory Driven by Influential Applications*. INFORMS, 1–12.

[27] David J Lusher, Satya P Jammy, and Neil D Sandham. 2021. OpenSBLI: Automated code-generation for heterogeneous computing architectures applied to compressible fluid dynamics on structured grids. *Computer Physics Communications* 267 (2021), 108063.

[28] Sergio M Martin, Daniel Wälchli, Georgios Arampatzis, and Petros Koumoutsakos. 2020. Korali: a High-Performance Computing Framework for Stochastic Optimization and Bayesian Uncertainty Quantification. *arXiv preprint arXiv:2005.13457* (2020).

[29] Sally A. McKee and Robert W. Wisniewski. 2011. *Memory Wall.* Springer US, Boston, MA, 1110–1116.

[30] Sanyam Mehta and Pen-Chung Yew. 2015. Improving compiler scalability: Optimizing large programs at small price. *ACM SIGPLAN Notices* 50 (2015), 143–152.

[31] Julian Miller, Lukas Trümper, Christian Terboven, and Matthias S. Müller. [n. d.]. Poster: Efficiency of Algorithmic Structures. In *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC19)* (Denver, Colorado, USA, 2019-11).

[32] Julian Miller, Lukas Trümper, Christian Terboven, and Matthias S Müller. 2021. A Theoretical Model for Global Optimization of Parallel Algorithms. *Mathematics* 9 (2021), 1685.

[33] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12 pages.

[34] MPI Forum. [n. d.]. MPI: A Message-Passing Interface Standard, Version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Accessed: 2021-02-01.

[35] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[36] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. 1996. *Pthreads programming - a POSIX standard for better multiprocessing.* O'Reilly. I–XVI, 1–267 pages.

[37] NVIDIA Corporation. [n. d.]. *Thrust: The C++ Parallel Algorithms Library.* NVIDIA Corporation. https://github.com/NVIDIA/thrust

[38] OpenCL [n. d.]. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. Accessed: 2021-02-01.

[39] OpenMP Architecture Review Board. [n. d.]. OpenMP 5.1 Specification. https://www.openmp.org/wp-content/uploads/openmp-5-1.pdf. Accessed: 2021-02-01.

[40] Peng Si Ow and Thomas E Morton. 1988. Filtered beam search in scheduling. *The International Journal Of Production Research* 26, 1 (1988), 35–62.

[41] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–19.

[42] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. 2011. *OpenSHMEM - Toward a Unified RMA Model.* Springer US, Boston, MA, 1379–1391.

[43] Adrian Schmitz, Julian Miller, Lukas Trümper, and Matthias S Müller. 2021. PPIR: Parallel Pattern Intermediate Representation. In *2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 30–40.

[44] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85.

[45] C2Rust Development Team. 2023. *C2Rust Manual.* https://c2rust.com/manual/ Accessed: 2024-01-02.

[46] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33 (2022), 805–817.

[47] Lukas Trümper, Julian Miller, Christian Terboven, and Matthias S. Müller. 2021. Automatic Mapping of Parallel Pattern-Based Algorithms on Heterogeneous Architectures. In *Architecture of Computing Systems*. Springer International Publishing, Cham, 53–67.

[48] Tyler Whitney, Kent Sharkey, Next Turn, Colin Robertson, Mike Jones, Mike Blome, Gordon Hogenson, and Saisang Cai. [n. d.]. *Parallel Patterns Library (PPL).* https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl

[49] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[50] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK—Yet Another Stencil Kernel: A framework for HPC stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE, 30–39.