Performance-portable Parallel Programming in Computer Science Education with Kokkos and C++

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

Abstract— Kokkos is a generic, performance-portable parallel programming model for host- and accelerator architectures in C++. It consists of a pure C++ interface, a specification and a programming library. The programming library exposes patterns and types and maps them to an underlying abstract machine model. This machine model offers a generic view of parallel hardware by abstracting from architecture- and vendor-specific programming models. In academia, this can contribute to a generic eduction of performance-portable parallel programming. In this work we present Kokkos from the perspective of educators, highlight key concepts and give an instruction to examples suited for incremental teaching of parallel programming in class.

Index Terms—Parallel programming, accelerator programming, Kokkos, C++

I. Introduction

Parallel and distributed computing moved into focus at universities and teaching institutions. The interest is twofold. Firstly, given their role, educators adapt to trends in research and industry and include parallel computing in their curricula, and secondly, parallel computing is an attractive field for department research and industry collaborations.

While industry-driven collaborations often predefine technological frameworks such as programming environments and targeted architectures, academic teaching leaves this mostly undefined. In this case, educators are exposed to a variety of solutions, approaches and trends in parallel computing. Which particular set to choose for teaching in class is not trivial. While decision factors differ, maximizing the subject relevance by teaching generic principles of parallel programming generically is among the most important ones.

Consequently, generic parallel programming models are needed that target mainstream programming languages and allow a vendor- and architecture independent, performance-portable expression of concurrency. In this way, a parallel program written in a popular language becomes capable of running efficiently on parallel computer architectures that conform to a common parallel machine model specification. We would like to compare this to the support of Von Neumann architectures in programming languages today. We define an abstract machine model later in this work. The lack of such parallel programming models motivated us to develop Kokkos.

Kokkos[1] [?], is a performance-portable, parallel programming model for C++ developed at Sandia National Laboratories (SNL). It consists of a programming model specification and a library. The specification defines execution and memory model and an application programming interface (library API). The runtime library exposes programming primitives, namely, patterns and types and maps them to the underlying abstract machine model. The machine model creates a generic view of compute hardware and memories. The Kokkos library maps execution primitives to vendor-specific programming models, libraries, and memory layouts. Kokkos has been successfully used in many large-scale projects at SNL, LLNL and other research centers [?]. We list its key properties as follows.

- Parallel programming model for C++
- Targets an abstract machine model
- Implements parallel patterns and tasking paradigms
- Offers abstract data types for performance portability
- Pure C++ implementation through template metaprogramming
- Aligns with C++ standardization efforts for parallel programming
- Vendor- and architecture independent and open-source
- Includes exercises and teaching material

The objective of this paper is to present Kokkos from two perspectives. Firstly, we would like to present our thoughts that accompanied the design and development of the Kokkos parallel programming model. These insights may be considered useful in parallel programming classes that introduce programming models and reason about design choices.

Further, we would like to present Kokkos to the educational community as one representative parallel programming model for a trend towards architecture-independent and performance-portable programming of CPUs and accelerators. We show a set of types that implement patterns and encapsulate data with enough

[1] from Greek for grain or seed

semantic information to produce performance-portable, parallel code on current architectures. Further, we highlight its alignment and contributions to standardization efforts towards supporting parallel programming in C++.

Secondly, we would like to promote Kokkos to students as a platform to understand and experiment with parallel patterns and types. Type specialization through templating allows to experiment with memory layouts and investigate their performance implications. Further, its modular code organization and open access under the BSD license allow students to peek behind the scenes and explore the mappings of abstractions to a variety of vendor-specific implementations. We believe that this makes Kokkos a valuable programming model for students which allows them to reason about vendor-provided programming models and to see them as particular implementations of higher abstractions in a parallel programming landscape.

The rest of this paper is structured as follows. The next Chapter gives an overview on parallel programming concepts and presents the notion of semantic capture. Chapter III presents the abstract machine model used to derive required semantic information for performance portability in Kokkos. Chapters IV and IV-B present the Kokkos programming model and its back-end support. Chapters VI and VII show the alignment to C++ standardization efforts and dicuss related work and lastly Chapter VIII concludes this work.

II. Towards Generic Parallel Programming

A programming model is an abstract representation of a computer. It defines a paradigm and a programming language. The programming language is used by the programmer to write source code, a human readable, textual representation of instructions and data. Source code is a sequential listing of statements evaluated from the top to bottom, one line at a time. While such ordering seems natural to describe an algorithm as it corresponds to sequential processing (both, reading code and progressing the program counter following a time line), it does not include the notion of concurrency.

Parallel programming models add the notion of concurrency to a sequence of instruction of a program. This is commonly achieved either by adding semantic value to the base programming language, often referred to as programming with pragma annotations, through library calls or new languages. All three approaches are conceptually suitable to represent programming paradigms such as tasking, parallel patterns and others. Well known representatives are the OpenMP programming model [?] or threading libraries respectively. In general, it is up to the parallel programming model to expose language constructs or library interfaces (APIs) such that they are convenient to the programmer and capture enough semantic information to allow the programming model to map the program execution to the parallel computer hardware efficiently and correctly. Questions of which semantic information is needed, what paradigm to present and by what syntactical means span a design space worth taking a closer look at. Figure 1 shows the three areas that constitute a parallel programming model and its design. We call it the semantic capture.

Semantic information
Paradigm
Language

Fig. 1: Semantic capture - a triplet of language, paradigm and semantic information defined by a parallel programming model allow the programmer to express concurrency and the compiler and runtime system to understand and run applications on a parallel machine.

Semantic information provided by the programmer to the parallel programming model can be grouped into information to express intent and properties. They address the question of what, where and how. In the context of parallel programming, these correspond to defining which code portion to parallelize, where to run and access and how to run that parallel code. Synchronization primitives may be considered as part of the what and information on the execution properties such as data placement or memory access type as the how. Semantics are expressed following a paradigm and a programming language.

The parallel programming paradigm is an abstract representation with the purpose of facilitating the programmer's understanding of programming rules and program behavior. Which programming paradigm to chose depends on several considerations. Parallel-patterns allow to express concurrency for commonly occurring programming patterns such as loop constructs. Tasking is a paradigm that support the expression of concurrent loops as well as irregular algorithms. Distributed and correctness-oriented programming models may implement actor-based programming. In this programming model, each unit of execution represent an actor who communicates over predefined communication interfaces. This paradigm eliminates accesses to shared state and align well with message passing programming (MPI). The execution model and memory model of a programming model defines behavior. That is, it defines the relationship between abstract concepts and program behavior on the given architecture.

Lastly, a language defines the syntax of the parallel semantic. Declarative languages add semantic information to the base language through pragma annotations while imperative languages rely on programming interfaces to capture information. Their semantic is defined in the API specification. Figures 2 and 3 show examples of two sample applications using the aforementioned parallel programming language types. While both programming models implement the same programming paradigm (parallel patterns), Figures 2 extends the base programming

```
# pragma model parallel for
for ( size_t i = 0; i < N; ++i) {
 /* loop body */
}
```

Fig. 2: Declarative parallel programming provides uses pragma annotations to capture semantic information.

```
model::parallel_for (N, [=] ( const size_t i) {
  /* loop body */
});
```

Fig. 3: Imperative parallel programming uses a base language and relies on programming interfaces to capture information and a documentation that describe their semantics.

language through pragma annotations. Both, pragma annotations and an API call as shown in Figure 3 have a similar semantic of describing a parallel loop construct. It is up to the programming model specification to define the execution and memory model in both cases.

Both examples provide the semantic information on what to parallelize, however, they do not provide information on how to map the parallel execution on modern parallel architectures. In order to define what further information is needed to express concurrency on modern computer architectures, it is important to define an abstract machine model first.

III. The Kokkos Machine Model

The Kokkos machine model defines abstractions that represent hardware capabilities for processing and data access. The Kokkos parallel programming model exposes these abstractions to the programmer through C++ and the Kokkos API. By abstracting from physical hardware, the machine model ensures that applications written in programming models targeting this machine model are generic and performance-portable on current and future hardware. The Kokkos parallel programming model is one particular instance of a programming model that builds on top of that machine model. We would like to point out that it is this conceptual differentiation that allows scenarios where the underlying machine model is instantiated in other languages beyond C++, yet the algorithmic specification as well as performance characteristic and portability remain the same.

The Kokkos abstract machine model targets a design of a future shared-memory computing architectures. The design is shown in Figure 4a. It is characterized by multiple latency-optimized execution units and off-die bandwidth-optimized accelerators. Both compute device types can have disjoint memory address spaces with unique performance properties. In such an architecture, execution units might be hierarchically organized with multiple levels of achievable parallelism with different memory access characteristics and coherence properties of caches. In order to ensure performance portability on such a wide range of configurations, an abstraction of compute resources and available memories are required. For this purpose we introduce the abstract concept called space.

An instance of an execution space is an abstraction over an execution resource to which a programmer can target parallel work. For example, an execution space can be used to describe a multi-core processor. In this case, the execution space contains several homogeneous cores organized into arbitrary logical groups. In a parallel programming model that implements this machine model, an instance of such an execution space would be made available to the programmer to run kernels. Adding more logical groups or accelerators simply increases the number of available execution spaces to the programmer.

Memory and memory types are exposed through memory spaces. Each memory space provides storage capacity at which data structures can be allocated and accessed. Different memory space types have different characteristics with respect to accessibility from execution spaces and performance.

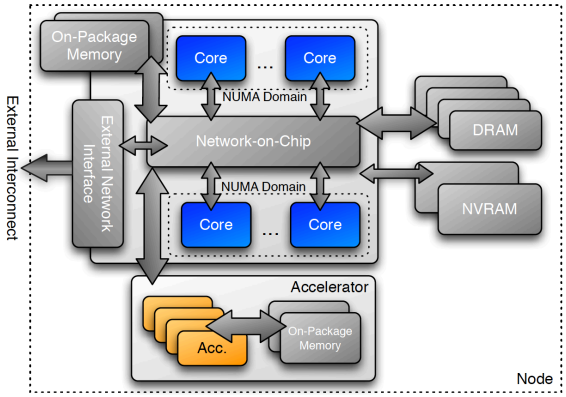
An instance of a memory space provides a concrete method for the application programmer to request data storage allocations. Following the architecture as discussed earlier, the multi-core processor may contain multiple memory spaces that abstract on-package memory, slower DRAM and non-volatile memories. Accelerators can provide an additional memory space through its local on-package memory. An example of such an architectural layout is shown in Figure 4b. The programmer is free to decide where each data structure is allocated by requesting the corresponding memory space. Programmatically this is achieved by instantiating the according memory space. Kokkos provides the appropriate abstraction of memory allocation and data management. We believe that an abstract representation of execution- and memory devices is a key property towards performance-portable parallel programming.

How a machine model is exposed to the programmer and what are the design considerations towards defining an appropriate semantic capture are discussed in the next section.

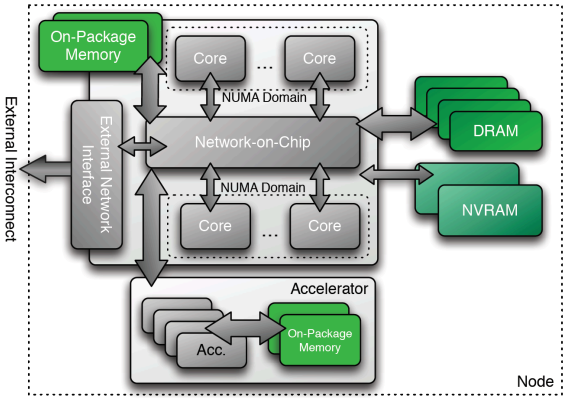
IV. The Kokkos Programming Model

The Kokkos programming model specifies a language, programming paradigm and an API. The API exposes the abstract machine model to the developer and defines a set of pattens and types that capture intents and properties of execution. This completes the semantic capture, discussed in Chapter II, We detail all three components as follows.

Kokkos defines a pure interface for C++ and uses C++ as a programming language. This has multiple reasons. It is our experience that application developers of scientific and high-performance computing applications at Sandia National Laboratories express the demand for C++ and



(a) Execution space is an instantiation of a space that represents a processing device to which the programmer can target parallel code.



(b) A memory space is an instantiation of a space that represents a memory location on which the programmer can allocate data.

Fig. 4: Spaces represent conceptual building blocks of the abstract machine model in Kokkos.

a generic parallel programming model to support their codes. Following their estimates, porting efforts towards adapting to vendors, programming models, APIs and software releases represents a burden worth addressing. Further, the possibility to maintain pure C++ codes is appealing from the code development and debugging perspective. Lastly, from the programming model perspective, C++ offers template metaprogramming which is well suited to implement generic APIs and libraries. In this case, class specialization and templating allow for compile-type generated types and optimizations for a given hardware architecture. It is worth to mention that Kokkos avoids the use of pragma annotations for the sake of treating concurrency as a first-class concept in C++.

Using the programming language, a parallel programming paradigm can be applied to simplify the process of thought. Kokkos supports the parallel patterns and task paradigms. Parallel patters covers for, scan and reduction. This allows the expression of concurrency over iterative, for-loop computable algorithms. To cover the class of while-loop computable algorithms (irregular algorithms), Kokkos implements the tasking paradigm. Tasks encapsulate work into units that may be executed in parallel to other tasks or sections of the program. Equipped with a language and set of paradigms, a particular set of abstractions can be defined to expose the abstract machine model and captures the intent of the programmer.

A. Kokkos Abstractions

To capture semantic information, that is the what, the where and the how, Kokkos introduces six abstractions: execution spaces, execution patterns, execution policies, memory spaces, memory layout and memory traits. These abstractions specify semantic information that enable to capture the programmer's intent and allow the runtime to efficiently map the program to any underlying hardware architecture. We list them as follows:

- An execution space is a place where code can be executed. On current hardware architectures this correspond to accelerators and CPUs and can include any compute device in the future. This abstraction supports remote compute devices in distributed memory scenarios as remote execution spaces.
- Execution patterns expose the parallel programming paradigm. Supported patterns are parallel_for loop that executes the loop body in any order a specified amount of times, the parallel_reduce which combines a parallel_for with a reduction operation, parallel_scan which combines a parallel_for operation with a prefix or postfix scan, and task which executes a single function potentially in parallel in respect to other tasks or code sections.
- Execution policy shape the iteration space of a loop pattern. A simple execution policy is a range policy. It specifies that the loop body is executed once for each element in a range.
- Memory spaces are the places where data resides. They specify physical locations of data as well as access characteristics. Different physical locations correspond to different device types such as high bandwidth memories, on-die scratch memories or non-volatile bulk storage. Different logical memory spaces allow for concepts such as memory in the CUDA programming model, which is accessible from the host and the CUDA accelerator. Memory spaces, similarly to execution spaces, conceptually support remote memory locations in distributed-memory scenarios. Furthermore, they encapsulate functionality such as consistency control and persistence scopes.
- Layouts express the mapping from loop indices to address offsets. By adopting appropriate layouts for memory structures, an application can optimize data access patterns in a given algorithm. If an implementation provides polymorphic layouts (i.e. a data structure can be instantiated at compile or runtime

with different layouts), architecture-dependent optimizations can be performed.
- Memory traits specify how a data structure is accessed. Traits express usage scenarios such as atomic access, random access and streaming loads or stores. This allows the programming model to optimize load and store operations.

```
Semantic information: Patterns (intent), Spaces, Layouts,
Policies and Traits
Paradigm: Parallel patterns and tasking
Language: C++
```

Fig. 5: Semantic capture defined by the Kokkos programming model.

WIP: Figure 6 shows definitions of the parallel for and parallel reduction interfaces. Both interfaces take a set of template parameters and function arguments. ExecPolicy shapes the iteration space for a particular device with varying degrees of hierarchical parallelism. A valid template parameter for ExecPolicy is a RangePolicy. This type defines a 1-dimensional iteration range that is templated. This allows to consider the distribtion of iterations into hierarchical orghanization of parallel processing units. In this case, ExecPolicy is and FunctorType and a set of function arguments.

Expressing concurrency with patterns and abstractions allows the compiler to generate optimized transformations. Further, a runtime library to map the execution to the parallel hardware. In the case of parallel patterns, the execution is unspecified and only promise deterministic results for the reduction and scan operations. This gives the freedom to the underlying abstraction layers to implement different mapping patterns on different hardware such as assignment of iterations to threads or vector lanes. We believe that these abstraction and their descriptive semantic represent key programming primitives and the way forward for native language support for parallel programming.

```
template <class ExecPolicy, class FunctorType>
Kokkos::parallel_for(const std::string& name,
 const ExecPolicy& policy,
 const FunctorType& functor);
template <class ExecPolicy, class FunctorType, class ReducerVarNonConst>
Kokkos::parallel_reduce(const std::string& name,
 const ExecPolicy& policy,
 const FunctorType& functor,
 ReducerVarNonConst& reducer);
```

Fig. 6: The Kokkos parallel_for and parallel_reduce class show how template metaprogramming allows to specialize types. Its descriptive semantic offers the freedom to optimize execution on modern architectures with multiple degrees of concurrency where an Execution policy shapes the iteration space accordingly.

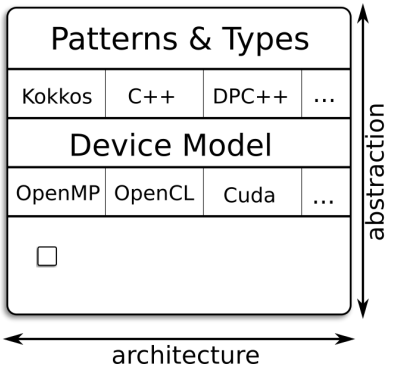


Fig. 7: Landscape

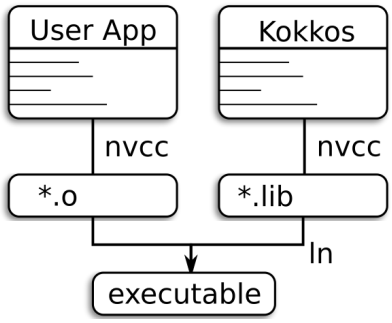


Fig. 8: Building workflow

Figure 5 shows the semantic capture defined by the Kokkos programming model. Figure 7 shows abstraction layers in Kokkos and and how it fits in a parallel programming landscape. The next section shows an example application written in Kokkos and provides an insight into the runtime library. For the interested reader, the complete programming model specification can be accessed on-line [?].

B. Back-end Support

The important consideration here is that the method of compiling code for different execution spaces and the dispatch of kernels to instances is abstracted by the Kokkos model. This unburdens application programmers from writing algorithms in hardware specific languages.

V. Tutorial Application

Describing here the tutorial app and the example apps that are included either in the repo or in the Kokkos distribution.

VI. Towards Parallel Programming in C++

VII. Related Work

VIII. Conclusion

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.