LLVM-Based IR for Distributed Dynamic Dataflow Runtime

Abstract

In this paper we present extensions to LLVM IR to represent task parallelism in applications in a target agnostic manner. The parallel IR uses metadata to represent the task-interaction graph of an application, much like DFGR **??**, enabling macro-level optimizations such as inter-task communication optimizations. It also offers the ability to express data distribution patterns, similarly as PGAS languages such as Chapel. We present an interface to plug in libraries that present parallel patterns as well as data distribution patterns, thus easing the use of parallel languages to target the underlying runtime. Effectiveness of the proposed IR is evaluated on the REDEFINE CGRA which has a dynamic dataflow runtime model with distributed memory.

I. Introduction

In order to achieve optimal performance, it is important to capture the intents of an algorithm in a program that fully exploits the underlying processing platform. Several programming languages approach the problem of expressing parallel algorithms via support for multithreading, multiprocessing etc. Programming models such as OpenMP and MPI provide extensions to existing languages to support parallel patterns, the newer versions of which support arbitrary task graphs (OpenMP 3.0**??**). PGAS programming languages such as X10**??**, Chapel **??** ease the programmers' burden by offering language constructs that spawn tasks and manage inter-task synchronization, also allowing the programmer to distribute data onto physically distributed memory with a shared address space. Compiling such languages onto the same target runtime can be a challenge, as is reusing high level optimizations due to custom compilation toolchains. There hence arises the need for a parallel IR which allows compiling various applications easily onto the same target, while enabling common optimizations among them. CnC is one such language designed to increase programmability that separates the concerns of expressing parallelism and tuning it to specific targets. Further, DFGR was proposed as an intermediate representation that assumes data race free execution of tasks. It extends CnC by supporting direct predication of a task by another and allowing communication of aggregate data objects such as ranges and regions between tasks.

In this work, we present an intermediate representation based on DFGR implemented as an extension of LLVM IR which eases static analysis and optimizations of applications at a macro level. The proposed IR captures the dynamic behavior of tasks that compose the application and further ensures deadlock freedom in dependences across tasks, as will be explained subsequently. The IR also comprises a subset of tuning directives to distribute input and output data onto various chunks of memory by allowing domain map definitions. Further, data parallelism is enabled exposing the underlying hardware resources as locales, an idea borrowed from Chapel programming language, similarly as *places* in X10. A subset of regularly occurring parallel patterns and data distribution libraries are provided that maybe enumerated into native graph description, enabling the compiler to perform target specific optimizations, such as offering locality in activation frames.

II. Description of Task-Interaction Graph

LLVM IR generates code using its virtual instruction set targeting unicore processors, making it unsuitable for representing parallel programs. State of the art parallel frontends are usually designed to suit the underlying hardware model (shared memory multithreading model assumed by OpenMP vs distributed memory model in case of MPI). A target agnostic representation of parallel programs based on DFGR[] is presented in this work that describes kernels in a graph-like manner. An application kernel described using DFGR comprises compute steps that represent scheduleable computations and item collections that represent data communicated between them. Compute step instances maybe spawned by other compute steps and the creator-created, controller-controllee relationships are captured in DFGR. This representation aims at easing the process of developing a new frontend for any parallel hardware target so long as there exists runtime support to execute the kernel on the target. DFGR also encompasses dynamic behavior of applications unlike the dataflow graph representation.

We propose realization of DFGR (in the current LLVM framework) by a parallel IR whose description is as follows.

A. Representing Computation

Compute steps of DFGR are represented as functions in LLVM IR. Statically enumerated compute steps are uniquely identified by the function's signature as the identifier. Dynamic instances of compute steps are identified by the function signature along with a tag, akin to the DFGR compute step tags. The tag is used to derive producer-consumer relationship between compute steps in our work and does not see a direct realization at runtime. To ease static analysis, we use limited set of functions defined on tags described next to define relationships between dynamic compute step instances.

Operands of a task are represented as arguments to the function. In a REDEFINE-esque landscape where a subset of arguments of a HyperOp maybe passed through a context frame, the arguments need to be pinned to registers, similarly as the GHC**??** by defining a custom calling convention. The choice of the set of arguments that need to be passed through context frames is a decision that the compiler makes and it is indicated by marking the arguments as *inreg*, implying that the arguments are passed in registers.

- $< id >$ is a self-referential tag of a compute step.
- $< suffix(id, i) >$ represents a step with suffix *i* of the self referential id.
- $< prefix(id, n) >$ represents the prefix of an id obtained by considering the first n entries of the self-referential id. Further, nesting of prefix function is allowed.

We observe that the aforementioned functions suffice for representing any kernel in our IR with the assumption that runtime manages instances independent of tag functions. Arbitrary tag functions make static analysis difficult but maybe easier to program. Hence, we support linear tag relationships to ease the programmer's burden. Any linear transformation of the form $a \times id + b$ is supported, where a and b are scalars and so is further nesting of tag functions (for example, transformation of the form $a \times prefix(id, i) + b$).

B. Representing Communication

Communication of data between compute steps of the parallel IR is represented using custom named metadata in LLVM IR that labels data with its consumer(s). Approaches proposed in [] realize communication via intrinsic extensions to LLVM and assume software support for communication. On the other hand, our approach makes no assumptions about the mode of realizing communication and hence can be compiled to esoteric targets(ASIC, FPGA) as well without performing machine level optimizations of removing communication instructions. The parallel IR defined as per the specification listed above can be compiled suitably onto any target and is open to local as well as graph optimizations and modifies the source IR to the least extent possible.

Communication in our parallel IR is asynchronous and the consumer is unaware of the source of received data. Communication between compute steps i.e., HyperOps are of three types namely *data, control, synchronization*. Interim data produced by a HyperOp (represented as a function) is annotated to define who the consumers are and what the data is used for (i.e., whether *ConsumedBy* or *Controls* or *SyncWith* consumer HyperOp(s)). In order to offer the flexibility of either using a single copy of the data that gets consumed by other HyperOps or to follow dynamic single assignment rule, data is not associated with instance ids (tags). This deviates slightly from DFGR where data is associated with instance ids uniformly. DFGR also assumes that compute steps are side-effect free. In our case, compute step instances that modify the global state are explicitly serialized by the compiler. Further, the data being communicated is marked as *static* or *composite* in order to assist the compiler in mapping data onto different modes of communication efficiently. Additionally, in case of the other two forms of communication i.e., *control* and *synchronization*, the runtime is free to realize data-based synchronization and control flow (like CnC) or may choose to support explicit control and synchronization primitives. The consumer HyperOps, when statically enumerated, are listed with the metadata linked with the data. A range of consumers may also be specified by extending the suffix function (described in the previous section) with vector extensions. Recall that $< suffix(id, i) >$ was earlier defined over scalar ids. These functions are extended to to support vector inputs in place of i which can now take a range of integer values specified as $[m : n]$, including m and n. n maybe variable but the lower bound m needs to be known. As mentioned previously, linear transformation defined over a range may also be used as the vector input to suffix function.

TODO: Topics of discussion here:
Disadvantages of the current IR implementation: Traditional optimizations may not be easily performed on the IR since metadata is ignored during compiler optimizations. CnC defines producer-consumer relationships at the boundaries of HyperOps and their compiler generates skeleton code for the programmer to fill with *get* and *put* calls assuming item collections to exist. This allows the communication to have a memory footprint and hence allows optimizations to go on since the memory operations change the global state. However, this hinders graph optimizations such as locality optimizations. Another approach proposed for PGAS languages on LLVM IR states that a shared address space in LLVM maybe written into or read from. I need to verify if a large number of address spaces can be created in LLVM and if they can be parametrically created and addressed. If this is indeed available, a pass that converts communication to writes and reads to address spaces can be implemented that allows standard optimizations to proceed. Another pass that eliminates these instructions and reintroduces the metadata maybe implemented. If feasible, this should be an ideal solution to our problems.

III. Tuning for the Underlying Architecture

In addition to expressing parallelism, a programmer should also be able to reason about performance of the application and tune it for optimality on the underlying hardware. As mentioned previously, we borrow the definition of *Locales* from Chapel[] programming language. Each locale represents a compute node and its associated chunk of distributed memory in the underlying hardware. The context of hardware in which the application runs is a set of locales, either composed with affinities defined by the compiler or specified by the user. Programmer may tune the application onto the underlying hardware in the following ways.

A. Locale Declaration

A set of locales may optionally be specified by the user via a local space declaration in the IR as follows:

- LocaleSet = $1:m, 1:n$ which specifies a rectangle of compute resources of size $m \times n$.
- LocaleSet = $x1, y1, x2, y2..$ where each xi, yi specifies coordinates of the resource in the underlying mesh.

If not declared, the compiler computes LocaleSet for optimal performance.

B. Data Layout and Distribution onto Locales

Data needs to be redistributed across Locales for locality guarantees in NUMA architectures. Explicit distribution onto locales identified via their x,y coordinates is useful in applications with statically enumerable graphs or certain dynamic but regular graphs or applications that don't follow known patterns across data-accesses. Data parallelism exhibited by irregular data dependent applications perform well when known distribution techniques are employed.

A multidimensional array to be distributed across the allocated locales has the following annotations:

- Cyclic, Locale= $LocaleSetIndex1, LocaleSetIndex2,../Locale = startindex1 : endindex1, startindex2 : endindex2$
- BlockCyclic, blocksize= $n1, n2, .., Locale = LocaleSetIndex1, LocaleSetIndex2, ../Locale = startindex1 : endindex1, start$

Specifying Locales in either case is optional when distributing data since the Locales may not be known at compile time. When the distribution is defined (Cyclic/BlockCyclic) without locales, an implicit iterator for LocaleSet is assumed (which increments along the column and then row when rectangular resource allocation scheme is employed or through the LocaleSet from left to right if the second declaration from III-A is used). *blocksize* need not be specified along all array dimensions.

Other than Cyclic and BlockCyclic data distribution patterns, we support Block (without cyclic distribution assumed) Random and Recursive Bisection as other patterns of distribution. **TODO: More work required in this to identify patterns and their suitability.**

C. Affinity of Computation onto Locales

Compiler defined distribution of tasks onto locales is easier to perform when the tasks are static instances, where a explicit locale is specified similarly as data layout locales. In case of dynamic instances, defining distribution of computations onto locales may either be explicit (using their x,y identifier) or maybe distributed cyclically where each task instance is assigned to a locale iterated over at runtime. The latter case proves useful in case of data parallel i.e., SIMD/SIMT applications when static analysis does not come in handy.

IV. Support for Parallel Compute and Data Distribution Patterns

The plan is to support these for a parallel frontend

Development of parallel algorithms and using them as building blocks in solving problems can be accelerated via support for parallel patterns of computation. Expressing such regular patterns embarassing parallelism is facilitated in the proposed compilation flow via support for parallel patterns in IR, which maybe generated from an explicitly parallel frontend. The parallel compute patterns are similar to macros/directives that are supported via annotations to loops representing the high level task-graph structure. The patterns come with an annotations processor interface that allows pattern specific optimizations (viz generation of equivalent task graph with larger span) to be performed and then expands the annotations onto the previously described LLVM IR format and resolves dependences between them. Supported patterns and their realization via annotations are listed in table I.

TABLE I. Supported Parallel Compute Patterns

Pattern	Annotation
Map, Stencil	parallel.for with blocksize (default size=1) over a loop nest
Fork-join, Reduction, Scan	parallel.for.deterministicreduction and parallel.for.reduction with predefined monotonic combiner functions (add, mul, min, max, and, or), parallel.scanreduction