Polyhedral compilation formalism

Siddharth Bhat

September 14, 2018

Contents

1 Memory

A chunk (`chunk`) is conceptually contiguous section of memory, usually associated with that of an array or a struct. It is mathematically represented as a mapping from naturals to a value domain, $\mathbb{V}$.

Memory (`mem`) is conceptually a mapping of chunk-IDs (`chunkid`) to chunks, where each chunk-ID is mathematically represented by a natural number.

$$\texttt{chunk} : \mathbb{N} \to \mathbb{V}$$
$$\texttt{mem} : \texttt{chunkid} \to \texttt{chunk}$$

Two chunks are considered to *never alias*. Two memory accesses alias iff they access the same chunk and the same index within the chunk.

1.1 `array`

$$\texttt{array} \equiv (arrname : ID) \times (arrdim : \mathbb{N}) \times (arrsizes : \mathbb{N}^{arrdim})$$

An array contains a unique identifier $arrname \in ID$, and a dimensionality $arrdim \in \mathbb{N}$.

Each array also has an associated *array space*, which is an n-dimensional space,

$$arrspace(A) = [0 \ldots arrsizes(A)[0]] \times [0 \ldots arrsizes(A)[1]] \times \ldots [0 \ldots arrsizes(A)[n]]$$

1.2 `memacc`

A memory access is a mapping from a `timepoint` to an array index. More formally, it maps points in the iteration space of the scop to points in the array space of a given array.

A memory access can either be a read access or a write access.

1.2.1 Read accesses

$$\texttt{readmemacc}(S, A) \equiv (tag : \texttt{readtag}) \times (ixfn : \texttt{iterspace}(S) \to \texttt{arrspace}(A))$$

Every read access has a *tag*, telling the abstract name of the read value, an $ixfn$, telling the index of the array which is read from at a given timepoint.

1.2.2 Write accesses

$$\texttt{writememacc}(S, A) \equiv (valfn : \texttt{iterspace}(S) \times (\texttt{readtag} \to \mathbb{V}) \to \mathbb{V}) \times (ixfn : \texttt{iterspace}(S) \to \texttt{arrspace}(A))$$

Every write acces has a $valfn$, which given the timepoint and all the previous read values computes the value to write, and an $ixfn$, telling the index of the array which is written to at a given timepoint.

1.3 `schedule`

$$\begin{aligned} \texttt{vivspace} &\subset \mathbb{Z}^n \\ \texttt{scatterspace} &\subseteq \mathbb{Z}^m \\ \texttt{schedule} : \texttt{vivspace} &\to \texttt{scatterspace} \end{aligned}$$

A schedule is a mapping of virtual induction variables ($\texttt{vivspace} \subset \mathbb{Z}^n$) into points in the scatter space ($\texttt{scatterspace} \subseteq \mathbb{Z}^n$). In general, any polyhedral object is said to be *scheduled* if it is mapped into the scatter space, and is arranged according to lexicographic-ascending order in this scatter space.

In this formalism, we *schedule statements*.

1.4 `stmt`

A scop statement *stmt* is a set of `readmemacc` and a set of `writememacc` , where intuitively, all the **reads happen atomatically**, sequenced by a single write, which uses the data read by the reads. with semantic preservation.

The atomicity property is useful, since it allows us to not represent certain dependences between multiple reads or multiple writes, which we would otherwise be forced to do.

Intuitively, a scop statement represents a sequence of reads followed by a single write, which occur atomically.

2 `SCoPs`

A `SCoP`(Static Control Part) is mathematical representation of programs which have control-flow that can be analyzed statically.

A `SCoP` S is a *set* of `stmt`, *stmts*(S), along with a *set* of `arrays`, *arrs*(S).

3 Execution / dynamic semantics

To execute a `SCoP`, we need memory `mem`, and an enviroment $\texttt{env} : \texttt{readtag} \to \mathbb{V}$ (usually denoted by Γ), which maintains the values that were "last read".

Next, we need to know what it means to *execute* a memory access, a statement, and finally a scop.

3.1 Executing memory accesses

The execution of a memory access is a function from $\texttt{mem} \times \texttt{env} \to \texttt{mem} \times \texttt{env}$