Vector Pack in VeGen

1 What's in a Vector Pack

A vector pack is not a collection of vectors, but rather a collection of instructions, each of which computes a scalar value.

2 Kinds of Vector Packs

2.1 General

To create a general vector pack, we need a `VectorPackContext`, an array of `Matches`, two bit vectors of elements and their dependencies, an intrinsic that is the producer of this pack, and the LLVM `TargetTransformInfo`.

2.2 Phi

To create a Phi pack, we do not need `Matches` as we do for creating a general pack.

2.3 Load

To create a Load pack, we do not need `Matches` since we already set it to be the load operation. Additionally, we need a condition pack to prevent unwanted reads from improper addresses and a flag to handle non-consecutive loads.

2.4 Store

Same as Load.

2.5 Reduction

2.6 GEP

GEP stands for "get element pointer".

2.7 Gamma

A Gamma pack is also called a Gated Phi Pack. It is a Phi node with incoming blocks replaced with explicit control conditions.

2.8 Cmp

3 Construction of Vector Packs

To construct a vector pack, we need to perform three steps in common, in addition to filling the vector pack context.

3.1 computeOperandPacks

This step has two sub-steps: compute and canonicalize. The compute sub-step gathers matched values into an array, mainly using a structure named OperandPack. This structure stores the vector type and the producers of the operand. The canonicalize sub-step wraps the OperandPack with a unique pointer and uses a map to guarantee uniqueness.

3.2 computeOrderedValues

For a general pack, it checks the `Matches` and filters out the unmatched operands, setting them to `null`. For Load, Store, Phi, GEP, and Cmp, it simply copies values from the vector pack variants' own data structure to `OrderedValues`, creating a starting point for later processing. Reduction has only one value, and Gamma places only Phi nodes contained in it to the `OrderedValues`.

3.3 computeCost

This step is self-contained. The cost is either read from an intrinsic guide or estimated using LLVM `TargetTransformInfo` (primarily for load and store).

4 Vector Pack Context

A vector pack context is a data structure that maintains a bidirectional map between values and integers, enabling the use of a bitmap to record a set of values. It is an intra-function analysis.

5 Vector Pack Set

A vector pack set is an abstraction that manages a set of compatible vector packs and is responsible for lowering a set of packs to LLVM IR.

6 Packer

6.1 Packer Fields

The Packer class has many fields, most of which are pointers to other classes or data structures. Some of the fields of the Packer class are:

- → F: a pointer to the function.
- → VPCtx: an object of type VectorPackContext.
- → DA: an object of type GlobalDependenceAnalysis, which is a class that performs a global analysis of the memory dependencies between instructions in the function.
- → CDA: an object of type ControlDependenceAnalysis, which is a class that computes the control dependence graph of the function.
- → VLI: an object of type VLoopInfo, which is a class that represents a loop in the function and its vectorization properties.
- → TopVL: an object of type VLoop, which is a subclass of VLoopInfo that represents the top-level loop in the function.
- → MM: an object of type MatchManager, which is a class that manages the matching of expressions.
- → SecondaryMM: an optional object of type MatchManager, which is used to store a secondary match manager for the case when the primary match manager fails to find a suitable pattern. It is an optional object, which is a wrapper that may or may not contain a value of a given type.
- → BO: an object of type BlockOrdering, which is a class that defines an ordering of the basic blocks in the function.
- → SE: a pointer to an object of type ScalarEvolution, which is a class that provides information about the evolution of scalar values in loops, such as induction variables and trip counts.
- → DT: a pointer to an object of type DominatorTree, which is a class that represents the dominance tree of the function, which is a tree that shows the dominance relation between the basic blocks.
- → PDT: a pointer to an object of type PostDominatorTree, which is a class that represents the post-dominance tree of the function, which is a tree that shows the post-dominance relation between the basic blocks.
- → LI: a pointer to an object of type LoopInfo, which is a class that provides information about the loops in the function, such as their nesting structure and their headers and exits.
- → LoadDAG & StoreDAG: two objects of type ConsecutiveAccessDAG, which are classes that represent directed acyclic graphs (DAGs) of memory accesses that are consecutive in the same array or pointer. A DAG is a graph that has no cycles, meaning that there is no path from a node to itself.
- → Producers: a map object of type DenseMap, which is a class that implements a hash table that maps keys to values. The keys are pointers to objects of type OperandPack, which are classes that represent a pack of operands that are used by a vector pattern. The values are objects of type OperandProducerInfo, which are classes that store information about how the operands are produced.

→ BlockConditions: a map object of type DenseMap, which maps basic blocks to ControlCondition, which are classes that represent a condition that controls the execution of a block, such as a branch or a switch.

→ EdgeConditions: a map object of type DenseMap, which maps pairs of pointers to basic blocks to pointers to objects of type ControlConditions, which represent a condition that controls the transition from one block to another.

→ SupportedInsts: The elements are pointers to InstBinding.

→ LazyValueInfo: a pointer to an object of type LazyValueInfo, which is a class that provides information about the possible values of an instruction at a given point in the function, such as whether it is a constant or a range of values.

→ TTI: a pointer to an object of type TargetTransformInfo, which is a class that provides information about the target architecture, such as the instruction costs, the register pressure, and the vectorization capabilities.

→ BFI: a pointer to an object of type BlockFrequencyInfo, which is a class that provides information about the relative frequencies of the basic blocks in the function, based on a branch probability analysis.

6.2 Packer Construction

The packer's constructor derives BlockConditions and EdgeConditions from ControlDependencyAnalysis (CDA), which uses LoopInfo, DominatorTree, and PostDominatorTree. BlockConditions and EdgeConditions act as a cache of CDA's methods: getConditionForBlock and getConditionForEdge. The latter method is required when the first instruction of a basic block is a PHI node. The constructor also collects all load and store instructions and assigns them to the Loads and Stores fields. Lastly, it defines a local helper variable, EquivalentAccesses, which is associated with the buildAccessDAG method and AccessLayoutInfo.

6.2.1 buildAccessDAG

AccessDAG, or ConsecutiveAccessDAG, is a type that maps an Instruction to a set of instructions. To construct this DAG, the following steps are performed: (1) group all access instructions by their type and the loop depth of their parent basic block; (2) for each group of access instructions, invoke the findConsecutiveAccesses method in ConsecutiveCheck to obtain a list of pairs of consecutive accesses; (3) add the consecutive accesses to the DAG.

6.2.2 AccessLayoutInfo

AccessLayoutInfo assigns linear numbers to load and store instructions that access the same memory object at different offsets. It is used to identify consecutive memory accesses and optimize them.

- The class has a nested struct called AddressInfo, which stores a pointer to an Instruction and an unsigned integer Id. The Instruction pointer is called Leader and it represents the first instruction in a group of consecutive accesses. The Id is the linear number assigned to the instruction, starting from 0 for the Leader.

- The class has two private fields: Info and MemberCounts. Info is a DenseMap that maps an Instruction pointer to an AddressInfo struct. MemberCounts is a DenseMap that maps an Instruction pointer to an unsigned integer. Both maps use the Leader instruction as the key.

- The class has two constructors: a default constructor and a constructor that takes a reference to a ConsecutiveAccessDAG object. A ConsecutiveAccessDAG is another class that represents a directed acyclic graph of consecutive memory accesses. The second constructor initialzes the Info and MemberCounts fields based on the information in the AccessDAG.

- Two instructions are adjacent if they have the same Leader and their Ids differ by one.

6.3 Load & Store

Consecutive loads need to be packed into a load pack, and consecutive stores need to be packed into a store pack. VeGen defines AccessLayoutInfo to store the analysis results of consecutive memory accesses. It groups a bunch of consecutive accesses into a group, records their offsets from the lowest address access, and defines the lowest access instruction as the leader of the group.

6.4 Reverse Post Ordering

VeGen maintains the reverse post-order traversal of basic blocks within a function using a mapping from LLVM BasicBlock to an unsigned integer, denoted as BlockOrdering.

6.5 Methods

6.5.1 checkIndependence

This function determines whether an instruction is independent of the elements in a bit vector that depends on another bit vector. It assumes that every instruction has a scalar id in the vector pack context and that a set of instructions can be represented by a bit vector. A bit vector can map an integer to a boolean value. Therefore, the dependency question is equivalent to some set operations. Given the id of an instruction, we check whether the id belongs to the elements or their dependencies, and whether the dependencies of the instruction (according to the Global Dependence Analysis results) intersect with the elements. If all these conditions are true, then the computation of the elements cannot affect the instruction.

6.5.2 isCompatible

6.5.3 canSpeculateAt & findSpeculationCond

6.5.4 matchSecondaryInsts