Vector Operation Description in VeGen

1 Vector Operation

What would be a good abstraction for vector operations? To be honest, I cannot identify any linear algebra-related properties in those SIMD instructions. They are simply hardware implementations of frequently used operations in regular code.

If we consider LLVM IR to be an abstraction over the scalar parts of various ISAs, then it follows that if we do not thoroughly consider certain types of new instructions like SIMD, they will only be utilized in highly specialized libraries. Consequently, they will not be incorporated into the IR or the compiler optimization process.

2 Abstraction in Vegen

In the paper *VeGen: A Vectorizer Generator for SIMD and Beyond*, an instruction is represented as a list of lanes, each of which is characterized by a BoundOperation. The author has kindly open-sourced the code for this paper, allowing us to examine its abstraction over vector instructions.

For example, the `_mm256_addsub_pd` is like this:

```
InstBinding(
    "_mm256_addsub_pd", {"avx"}, InstSignature{{256, 256}, {256}, false},
    {BoundOperation(&Operation3, {InputSlice{0, 0, 64}, InputSlice{1, 0, 64}}),
     BoundOperation(&Operation4,
                    {InputSlice{0, 64, 128}, InputSlice{1, 64, 128}}),
     BoundOperation(&Operation3,
                    {InputSlice{0, 128, 192}, InputSlice{1, 128, 192}}),
     BoundOperation(&Operation4,
                    {InputSlice{0, 192, 256}, InputSlice{1, 192, 256}})},
    1.0)
```

2.1 InstSignature

```
struct InstSignature {
  std::vector<unsigned> InputBitwidths;
  std::vector<unsigned> OutputBitwidths;
  bool HasImm8;
  unsigned numInputs() const { return InputBitwidths.size(); }
  unsigned numOutputs() const { return OutputBitwidths.size(); }
};
```

This structure records the number and bitwidths of the input and output parameters of a vector instruction.

```
InstSignature{{256, 256}, {256}, false}
```

The only unusual aspect is the mysterious `HasImm8` member. This is because some vector instructions receive an immediate operand (typically an 8-bit integer) that cannot be passed as a variable (i.e., a register) but only as an immediate value. This distinction necessitates different code generation, hence the need to record this information.

2.2 InputSlice

```
struct InputSlice {
  unsigned InputId;
  unsigned Lo, Hi;
  unsigned size() const { return Hi - Lo; }
  bool operator<(const InputSlice &Other) const {
    return std::tie(InputId, Lo, Hi) <
           std::tie(Other.InputId, Other.Lo, Other.Hi);
  }
};
```

The input slice structure further divides the input long vector operands. The `InputId` member indicates which operand this slice comes from. The `Lo` and `Hi` members describe the slice range.

Overall, an input slice is a division of some inputs, in order to feed parts of the inputs to a operation.

2.3 Operation

```
// Interface that abstractly defines an operation
struct Operation {
  virtual ~Operation() {}
  struct Match {
    bool Commutative; // applies when the operation is binary
    std::vector<llvm::Value *> Inputs;
    // FIXME: make this an Instruction instead
    llvm::Value *Output;
    bool operator<(const Match &Other) const {
      return std::tie(Output, Inputs) < std::tie(Other.Output, Other.Inputs);
    }
    bool operator==(const Match &Other) const {
      return Output == Other.Output && Inputs == Other.Inputs;
    }
  };
  // 'match' returns true if 'V' is matched.
  // If a match is found, additionally return the matched liveins
  virtual bool match(llvm::Value *V, llvm::SmallVectorImpl<Match> &Matches) const
= 0;
```

```
};
```

The `Operation` class has a virtual and abstract `match` method. This means that different operations will have different patterns to match. The concrete `match` method is required to fill in a vector of `match` objects, which capture the input and output values in the original code (the code to be vectorized).

For example, Operation17's match method is shown below. If the value has a 64-bit width and is a zero extension of another 32-bit value tmp0, then the noncommutative operation from tmp0 to value is pushed as a match to the match list.

```
class : public Operation {
  bool match(Value *V, SmallVectorImpl<Match> &Matches) const override {
    Value *tmp0;
    bool Matched = hasBitWidth(V, 64) &&
                   PatternMatch::match(V, m_ZExt(m_Value(tmp0))) &&
                   hasBitWidth(tmp0, 32);
    if (Matched)
      Matches.push_back({false,
                         // matched live ins
                         {tmp0},
                         // the matched value itself
                         V});
    return Matched;
  }
} Operation17;
```

2.4 BoundOperation

```
// An operation explicitly bound to an instruction and its input(s)
class BoundOperation {
  const Operation *Op;
  std::vector<InputSlice> BoundSlices;
public:
  // A bound operation
  BoundOperation(const Operation *Op, std::vector<InputSlice> BoundSlices)
      : Op(Op), BoundSlices(BoundSlices) {}
  const Operation *getOperation() const { return Op; }
  llvm::ArrayRef<InputSlice> getBoundSlices() const { return BoundSlices; }
};
```

Each BoundOperation produces one element in the output vector. Note that the InputSlice is simply a recorded relationship between the output vector's element and the input vector's elements. The matched Operation holds references to the Value object in an LLVM IR module.

2.5 InstBinding

Putting it all together,

```
// An instruction is a list of lanes,
// each of which characterized by a BoundOperation
class InstBinding {
  InstSignature Sig;
  std::string Name;
  std::vector<std::string> TargetFeatures;
  std::vector<BoundOperation> LaneOps;
  llvm::Optional<float> Cost;

public:
  InstBinding(std::string Name, std::vector<std::string> TargetFeatures,
              InstSignature Sig, std::vector<BoundOperation> LaneOps,
              llvm::Optional<float> Cost = llvm::None)
      : Sig(Sig), Name(Name), TargetFeatures(TargetFeatures), LaneOps(LaneOps),
        Cost(Cost) {}
  virtual ~InstBinding() {}
  virtual float getCost(llvm::TargetTransformInfo *,
                        llvm::LLVMContext &) const {
    assert(Cost.hasValue());
    return Cost.getValue();
  }
  llvm::ArrayRef<std::string> getTargetFeatures() const {
    return TargetFeatures;
  }
  const InstSignature &getSignature() const { return Sig; }
  llvm::ArrayRef<BoundOperation> getLaneOps() const { return LaneOps; }
  virtual llvm::Value *emit(llvm::ArrayRef<llvm::Value *> Operands,
                            IntrinsicBuilder &Builder) const {
    return Builder.Create(Name, Operands);
  }
  llvm::StringRef getName() const { return Name; }
};
```