OMR Architecture: Vector IL Opcodes

November 25th 2021

Gita Koblents

Outline

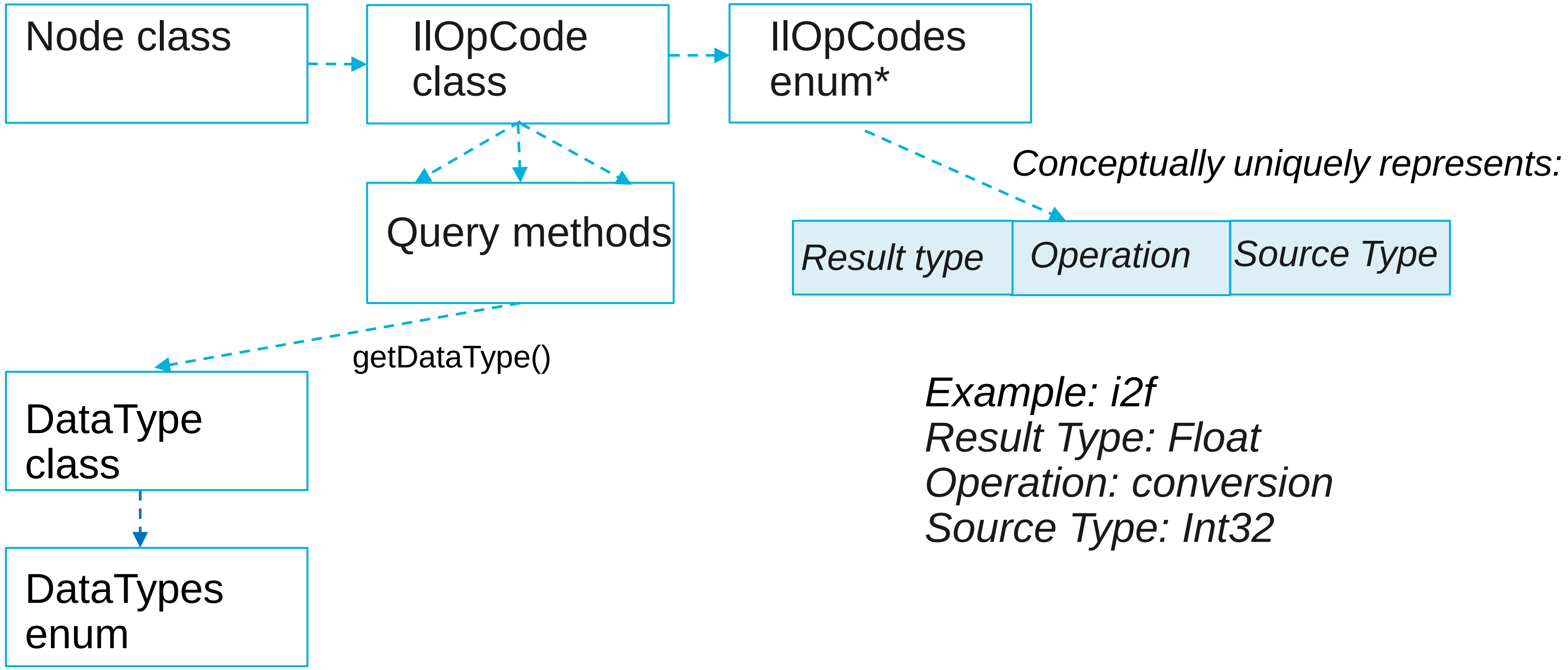
Motivation

- **We would like to support vector operations for various vector lengths – 64, 128, 256, 384, 512, 640**, … **2048** (in 128 increments on ARM SVE; 17 in total)
  - To exploit current and future vector register sizes on Intel, Power, z, ARM
  - Can be used by auto-SIMD
  - Are necessary for the Panama(Vector API) project
    - In Vector API, SPECIES_PREFERRED can have any length supported on a platform
  - We should be able to use different sizes in one compilation

- **Current approach**
  - 128-bit length is implied
  - TR::DataTypes describe vector layout precisely, e.g. **VectorInt32** is 4 32-bit integers
  - Opcodes are ‘typelesss’, e.g. **vadd**, **vconst,** etc. but data type can be derived
    - From children
    - From symbol reference
    - Cached on the node, e.g. for **vconst**

- **Limitations of the current approach**
  - Only one length is supported
  - Adding new data type enum for every possible combination of vector length and element type
  - Current typeless opcode implementation reduces the number of opcodes but is error prone, e.g. incorrect type can be derived while a node is being constructed or modified

Current Data Structures



Example: i2f
Result Type: Float
Operation: conversion
Source Type: Int32

*Is used to dereference Opcode Properties and other tables

High Level Problem Statement

- **Definitions**

  E: a set of vector element types (e.g. Int8, Int16, Int32, … Double, etc.)
  S: a set of vector lengths (64, 128, 256, etc)
  O: a set of operations (load, add, sub, reduce, conv, …)
  D: a set of all possible data types, **D = E x S**
  OP: a set of all possible opcodes, **OP = D x D x O** (result type, source type*, operation)

  Constraint 1: Optimizer should be agnostic to the number of vector elements.
  Constraint 2: Codegen should be structured in such a way that asking for a specific member of **D** or **OP** is not necessary and more general queries are used.

*a modifier for an operation, e.g. **((Int32, 128),(Int32, 128), load)** means 4 integers need to be loaded and the result will be 4 integers as well. A typical example of different source and result type is a conversion operation.

High Level Problem Statement cont’d

There are 3 ways to represent sets **D** and **OP** in a program in general and in OMR specifically:

(A) As classes. For example, set **D** can be a class DataType with two instance variables: _e and _s. Set **OP** can be represented as class ILOpCode with 3 instance variables: _d1, _d2, _o

1. Considerable code changes to OMR and down-stream projects

(B) As integral value **I** that uniquely represents each member of the set. For example, a mapping function can be provided that returns a unique integer given (d1, d2, o) and similar function for (e, s). There also should be a function that returns each member of the tuple(or triple) given **I**. These functions can be easily created by encoding each member into a certain group of bits inside **I**.

1. Not easy to see the meaning of the opcode or data type in the debugger
2. Compatibility with the existing OMR enums has to be ensured

(C) As a list of all set members in the form of constants(enums) that can be directly referred to in the source.

1. violates Constraints 1 and 2
2. although the size of the enum can be possibly smaller than the size of **I** (if only allowed combinations are listed) creating and maintaining this list manually will be error-prone. A side table describing each enum’s properties will be needed.
3. automatically generating enums with encoded values would be similar to (B)

Implementation Details for Proposal B

The following slides propose (B)

DataType

- **SVE considerations**
  - SVE architecture introduces scalable vectors. There are various rules which vector lengths are allowed but most notably vector lengths that are multiple of 128 bits, up to 2048 bits. This implies at least 16 lengths
  - Only one vector length can be configured at a time. It can be queried at runtime and therefore known to JIT
  - It’s not completely clear if vector length can be reconfigured while a process is running but we can consider simply not allowing it(at least in the short term)

- **Which vector lengths should we support?**
  - **Proposal:** we should support a union of the following:
    - Known vector lengths on supported architectures:

      128: on Power, Z

      128, 256, 512 on Intel

      N: variable length on ARM SVE but only one value per process
    - lengths required by known languages and libraries

      64, 128, 256, 512: Vector API
    - non-vector length: to indicate scalar values, we will use 0 to indicate non-vectors
    - Total: 0, 64, 128, 256, 512, N = 6 lengths (3 bits)

DataType cont’d

- **DataTypes can be encoded into 1 byte as:**
  - upper 4 bits: vector length (1 more bit than needed, reserved for future)
  - lower 4 bits: existing scalar types (up to existing NumTypes enum)
  - use static_assert (NumTypes < 16)
  - if more space is needed in the future we will make **DataTypes** enum bigger

- **getDataType()** can be used as before, but values >= NumTypes are vector types and cannot be used as index to any tables

- **If unsupported vector length/type is specified:**
  - There will be limited number of places that will create original vectors: VectorAPIExpansion and autoSIMD. Those will query codegen which length/types are supported: similar to the way it already happens in autoSIMD.
  - other places will create new types based on previously created types. Type consistency will be ensured similar to way it is done for other types

ILOpCode

- Encode vector opcode’s result type, source type, and operation into **TRILOpCode::_opcode** (currently 4 bytes)
  - **DataType**: 1 byte (see previous slide)
  - **Operation**: 2 bytes
  - **Total**: 1 byte result type + 1 byte source type + 2 bytes operation = 4 bytes
  - If at some point, we run out of bits we can change **ILOpCodes enum** to fit long.
  - use static_assert (LastOMROp <= xffff)
  - ensure **ILOpCodes enum** is of size int32_t
- We can continue getting the value of a vector opcode and use it similar to other opcodes. We just won’t refer to them by name in the source.
- Vector opcodes can be uniquely identified by query methods, such as **isVector()**, **getVectorResultElementType()**, **getVectorResultSize()**, **getVectorOperation()**. Note that in this case, vector operation(vadd, vreduce, vpermute) will be a separate enum that can be used to encode and group vector operations
- Dispatch code that uses ILOpCodes enum to index static tables needs to be amended for vector type
- Portable AOT
  - Invalidate method if compiled and actual vector length don’t match

Code Changes

class Node

~~DataTypes _datatype;~~
// was used for typeless

 ILOpCode _opCode;
opcodes

public:
  DataType getDataType();
// will always ask opcode

class DataType

DataTypes _type;
DataTypes getDataType();

isVector(){return _type> NumTypes;}
createVectorType();
getVectorSize();
getVectorElementType();

enum DataTypes
  Int8,
  Int16,
  Int32,
  Int64,
  Float,
  Double,
  Address,
  Aggregate,
#include "il/DataTypesEnum.hpp"
  NumTypes

"invisible DataTypes enums"

class ILOpCode
private:
ILOpcodes _opcode;

public:
ILOpCodes getOpCodeValue();
DataType getDataType()
 isVector() {return _opcode > LastOMROp;}
createVectorOpcode(DataType, DataType, vectorOperations);
 getVectorResultElementType();
 getVectorResultSize();
 getVectorSourceElementType();
 getVectorSourceSize();
 getVectorOperation();

enum ILOpCodes : uint32_t

  iadd
  fadd
  …
  LastOMROp

"invisible ILOpCodes enums"

enum VectorOperations
  vload,
  vstore
  vadd
  vreduce
  …

Black: current
Red: new

- DataTypes greater than NumTypes and ILOpCodes greater than LastOMROp cannot be used to index any of the static tables that we have
- Vector DataTypes and ILOpCodes will be greater those values because their upper halves will be non-zero
- Current Vector types and opcodes will be removed

DataType Examples

```
if (node1->getDataType() == node2->getDataType() ) {
   …  // will work as before
}
```

```
if (node->getDataType() == TR::Int32) {   // will not match vector data type
   …
}
else if (node->getDataType().isVector() &&
            node->getDataType().getVectorResultElementType() == TR::Int32) {
}
```

```
switch (node->getDataType()) {
      case TR::Int32:
      …
      default:
         if (node->getDataType().isVector() &&
            node->getDataType().getVectorResultElementType() == TR::Int32) {

         }
}
```

Node Creation Examples

```
TR::ILOpCodes loadOpCode = TR::ILOpCode::indirectLoadOpCode(TR::Int8);

TR::ILOpCodes loadOpCode = TR::ILOpCode::indirectLoadOpCode(createVectorType(Int32,4)); // 4 integers
```

```
static TR::ILOpCodes indirectLoadOpCode(TR::DataTypes type)
    {
    switch(type)
      {
      case TR::Int8:     return TR::bloadi;
      case TR::Int16:    return TR::sloadi;
      …
      case TR::Float:    return TR::floadi;
      default:
          if (type.isVectorType())  {
               // result and source type are the same for vloadi       ⎫
               return type<<24 | type << 16 | TR::vloadi; }            ⎭  // inside  createVectorOpCode()
        else
          TR_ASSERT(0, "no load opcode for this datatype");
      }
    return TR::BadILOp;
    }
```

Using Opcodes Examples

```
if (node1->getOpCodeValue() == node2->getOpCodeValue() ) {
   …  // will work as before
}
```

```
if (node->getOpCodeValue() == TR::iadd) {   // will not match vector opcode
   …
}
else if (node->getOpCode().isVectorAdd() &&
            node->getOpCode().getVectorResultElementType() == TR::Int32) {
}
```

```
switch (node->getOpCodeValue()) {
      case TR::iadd:
      …
      default:
         if (node->getOpCode().isVectorAdd() &&
            node->getOpCode().getVectorResultElementType() == TR::Int32) {

         }
}
```

Trace File Examples

```
vadd_ix4
   vload_ix4
   vload_ix4

vconv_ix4_fx4
   vload_ix4
```

```
getName(ILOpCode opcode) {

   If (opcode.isVector()) {
       resTypeString = “”;
       if (opcode.getVectorResultType() != opcode.getVectorSourceType())
            resTypeString = “_” . getName(opcode.getVectorResultType());
       srcTypeString = getName(opcode.getVectorSourceType());
       operationString = getName(opcode.getVectorOperation());

       name = operationString . “_” . srcTypeString  . “resTypeString”;
   }
}
```

Summary

- Full encapsulation of the opcode representation
- Conceptually close to the current(non-vector opcodes) approach of keeping all the info in the opcode
- No size increase: only vector opcodes that are used in the method will be instantiated
- Isolated: no changes to the existing non-vector code
- No references to specific vector enums in the code
- Can be extended and optimized in the future
  - for example, utilize the fact that for most of the opcodes result and source types are the same

Backup Slides