DSPLang

Source Code at
https://github.com/CaptainIRS/dsplang

Goals

- Develop a vector-processing DSL for architectures (especially NPUs) with high bit-width vectors.
  - Should be able to be lowered to multiple architectures with ease
  - Should leverage MLIR to progressively lower the DSL
- Focus on the Qualcomm Hexagon DSP/NPU as the target.
  - Specifically, the Hexagon Vector eXtensions (HVX) co-processor
  - Currently, there is no way to lower MLIR to HVX instructions as LLVM codegen doesn’t transform code to HVX
  - Lasting contribution: first project to expose HVX instructions to the wider MLIR ecosystem

Motivation

- NPU usage is on the rise for use-cases like ML, image processing, vector processing workloads due to power-efficiency.
- NPUs generally use ultra-wide registers for processing, but LLVM doesn’t utilize the max bandwidth even with maximum optimizations enabled, unless code is written in backend-specific intrinsics.
- Writing code with intrinsics is not a good developer experience.

Motivation (w.r.t Hexagon NPU)

LLVM with -O3 -fvectorize does not use ultra-wide vectors at all

```
434: {      r14 = memuh(r0+r7<<#0x0)
438:        r15 = memuh(r6+r7<<#0x0) }
43c: {      r10 = mpyi(r15,r14)
440:        r14 = add(r15,r14) }
444: {      r14 = mpyi(r10,r14)
448:        r7 = or(r28,#0x6)
44c:        memh(r1+r7<<#0x0) = r14.new }
```

Intended optimal code that uses the 1024-bit wide vectors

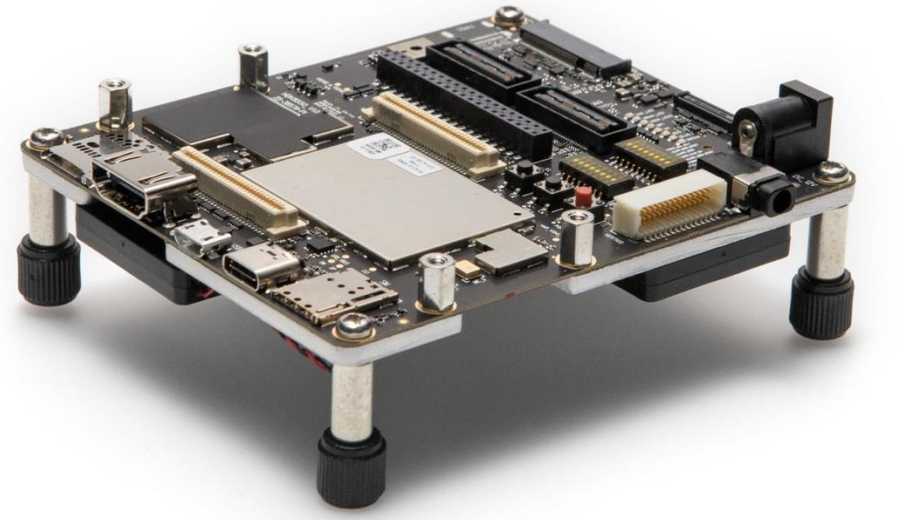
```
4f0: {      r12 = add(r0,r14)
4f4:        v28 = vmemu(r12+#0x0) }
4f8: {      v0 = vmemu(r28+#0x0) }
4fc: {      v29.h = vadd(v0.h,v28.h) }
500: {      v0.h = vmpyi(v0.h,v28.h) }
504: {      v0.h = vmpyi(v0.h,v29.h) }
508: {      r12 = add(r2,r3)
50c:        r3 = add(r0,r3)
510:        vmemu(r12+#0x0) = v0 }
```

Hardware Target

Background information about the specific hardware target – the Qualcomm Hexagon NPU

Hardware Target

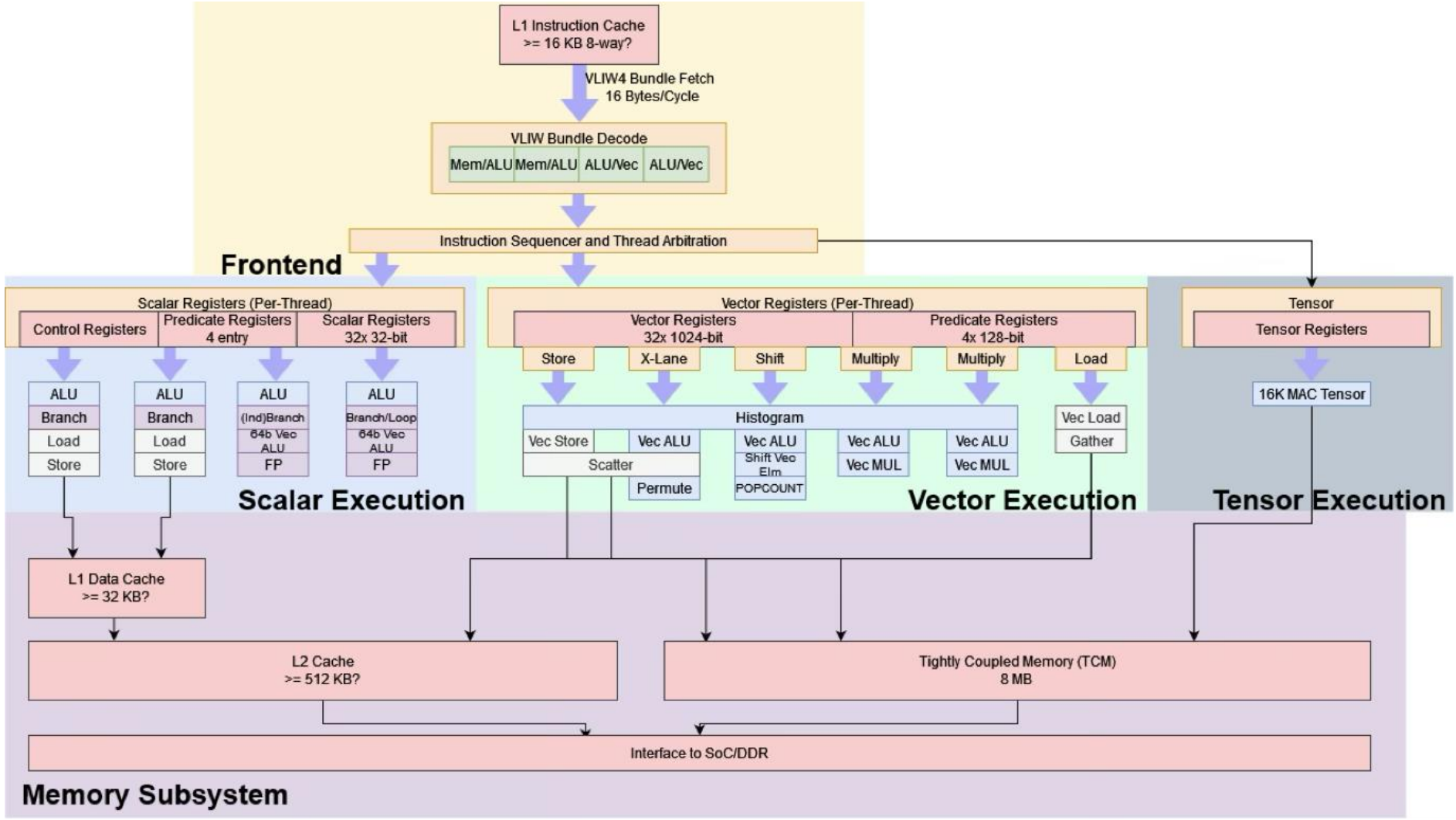
- To keep the project focused, the chosen target is Qualcomm's Hexagon DSP/NPU
- Specifically, the QCS8550 SoC with the Hexagon v73 DSP/NPU (Snapdragon Gen 2)
- The NPU has a HVX (Hexagon Vector eXtensions) and a HMX (Hexagon Matrix eXtensions) co-processor



Key aspects of the Target – Quick Intro

- VLIW instruction set
  - Depends on compiler optimizations instead of hardware optimizations like superscalar execution for performance
- Runs at a much lower clock-frequency than the application processor of the SoC (typically an ARM processor)
- Power-efficient
  - Since there is no power-consuming hardware optimizations like out-of-order execution, aggressive branch prediction, etc.

Key aspects of the Target – Architecture



Source: https://chipsandcheese.com/p/qualcomms-hexagon-dsp-and-now-npu

Revisiting the Motivation

LLVM with -O3 -fvectorize only produces code that runs on the *Scalar Execution Unit (DSP)*

```
434: {      r14 = memuh(r0+r7<<#0x0)
438:        r15 = memuh(r6+r7<<#0x0) }
43c: {      r10 = mpyi(r15,r14)
440:        r14 = add(r15,r14) }
444: {      r14 = mpyi(r10,r14)
448:        r7 = or(r28,#0x6)
44c:        memh(r1+r7<<#0x0) = r14.new }
```

We need to use the *Vector Execution Unit (HVX co-processor)*

```
4f0: {      r12 = add(r0,r14)
4f4:        v28 = vmemu(r12+#0x0) }
4f8: {      v0 = vmemu(r28+#0x0) }
4fc: {      v29.h = vadd(v0.h,v28.h) }
500: {      v0.h = vmpyi(v0.h,v28.h) }
504: {      v0.h = vmpyi(v0.h,v29.h) }
508: {      r12 = add(r2,r3)
50c:        r3 = add(r0,r3)
510:        vmemu(r12+#0x0) = v0 }
```

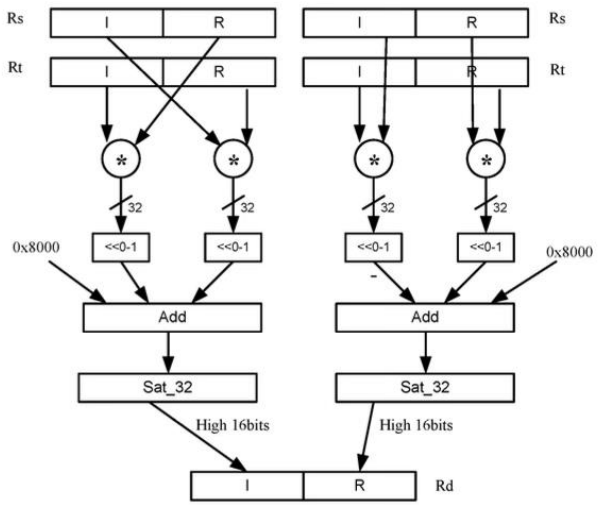
Key aspects of the Target – VLIW “Packet”

64-bit Load and

64-bit Store with post-update addressing

```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

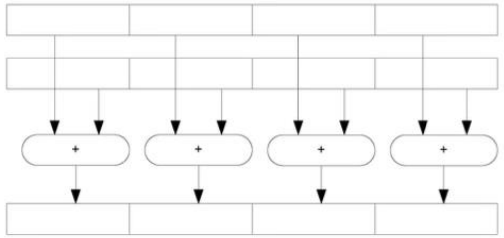
Complex multiply with round and saturation



Zero-overhead loops

- Dec count
- Compare
- Jump top

Vector 4x16-bit Add



Key aspects of the Target – Software

- The ISA docs for the DSP and HVX unit are public, whereas the HMX (Tensor) unit is private
- LLVM has a Hexagon backend and intrinsics that wrap most of the instructions of the DSP and HVX unit (even some undocumented HMX instructions?)
- LLVM codegen has a very basic “AutoHVX” pass, which doesn’t do any complex transformations (so far)
- The linker was recently made open-source by Qualcomm
- QEMU fork of Qualcomm has emulation support for the DSP+HVX

Key aspects of the Target – Software Ecosystem

- The Hexagon SDK is partly closed-source and partly source-available. Currently SDK is needed for pre-built libc/libc++ (which can be built on our own), libhexagon (only for instrumentation).
- Note: Hexagon SDK can only access the DSP and HVX units. The HMX unit is accessible only through high-level access using QNN/SNPE SDKs.
- LLVM bundled with the SDK is proprietary and does have intrinsics for the HMX unit and some extra optimization passes.
- Projects like Halide and XNNPACK have native support for Hexagon.

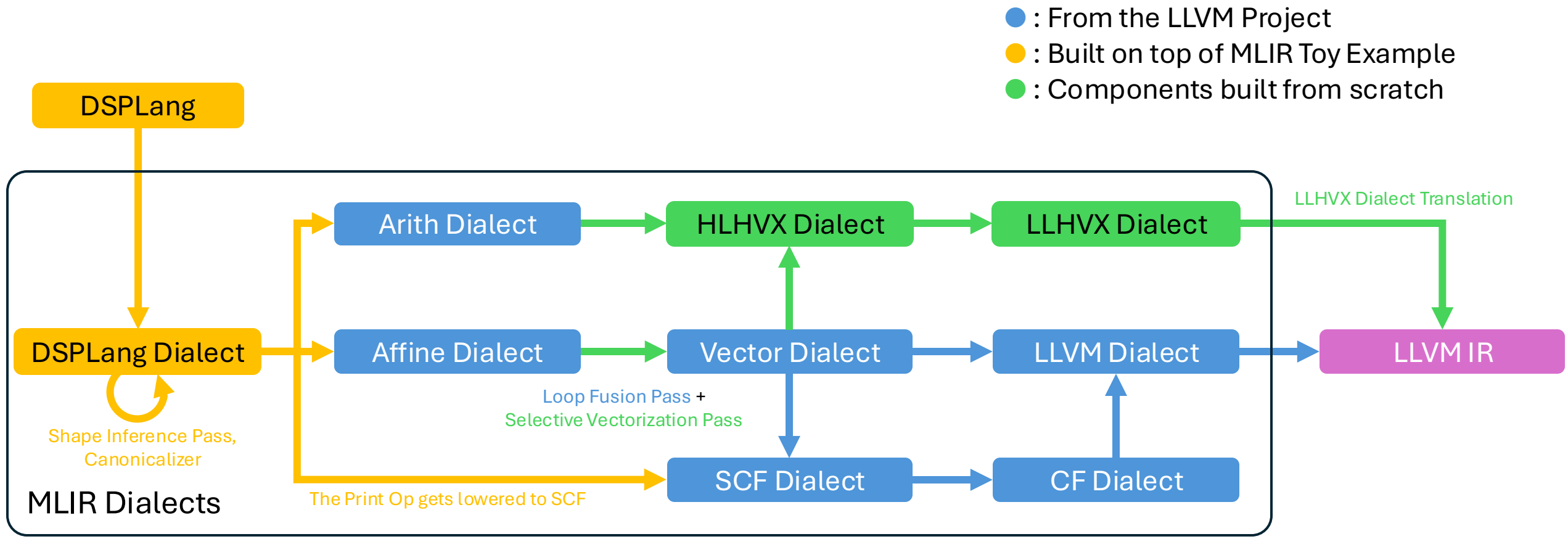
Design

Design/Architecture of the whole project

High Level Overview (Top-Down)

- DSPLang frontend has the AST structure, Lexer, Parser and MLIR Generator components
- DSPLang is lowered to the DSPLang Dialect that operates on Tensors
- The DSPLang Dialect is lowered to Affine + Arith + SCF
- Affine Vectorization is applied, which lowers loads/stores to Vector Dialect
- The Arith Dialect is lowered to the HLHVX (High-Level HVX) Dialect after a vectorization pass
- The HLHVX Dialect is lowered to the LLHVX (Low-Level HVX) Dialect
- Every dialect except LLHVX is lowered to the LLVM Dialect
- LLVM Dialect and LLHVX Dialect are translated to LLVM IR
- Object file is built using the IR and linked to get the executable

Architecture Diagram



Introducing 2 new Dialects

LLHVX (Low-Level HVX) Dialect

- Basic interface between MLIR and LLVM IR
- Every bit-width is treated as the same type (e.g. 1024 bits is always vector<32xi32>)
- Translated directly to LLVM IR
- Translation rules specified in TableGen format
- All rules are automatically generated

HLHVX (High-Level HVX) Dialect

- Interface between other dialects and LLHVX
- Has the correct type information (e.g. vector<64xi16> for a vector of half-words)
- Converted to LLHVX Dialect
- Conversion patterns specified in PDLL format instead
- Most patterns are automatically generated, few are hand-written

DSPLang Dialect

- The interface between DSPLang and other dialects of MLIR
- Operates on Tensor and Memref types
- Shape Inference pass is applied before further lowering

```
dsplang.func @dsplanglib(%arg0: memref<10000xi16>, %arg1: memref<10000xi16>, %arg2: memref<10000xi16>) {
  %0 = dsplang.bufferize(%arg0 : memref<10000xi16>) to tensor<10000xsi16>
  %1 = dsplang.bufferize(%arg1 : memref<10000xi16>) to tensor<10000xsi16>
  %2 = dsplang.mul %0, %1 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
  %3 = dsplang.reshape(%2 : tensor<*xsi16>) to tensor<10000xsi16>
  %4 = dsplang.add %0, %1 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
  %5 = dsplang.reshape(%4 : tensor<*xsi16>) to tensor<10000xsi16>
  %6 = dsplang.mul %3, %5 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
  dsplang.assign %arg2 = %6 : tensor<*xsi16> to memref<10000xi16>
  dsplang.return
}
```

Selective Vectorization Pass

- Uses the underlying mechanism of the SuperVectorizer pass from the Affine Dialect
- SuperVectorizer is not suitable for our purposes as it assumes every loop need to be vectorized by the same factor
- In our case the factor depends on the bit-width of the elements in the vector. Bit-width × vectorization factor should be 1024 (register bit-width)
- Applied adaptively for every loop based on the datatypes operated on in the loop

Frontend – High-level changes to the MLIR Toy example

- Currently restricted to 1D tensors
- Added support for datatypes (only float64 used to be supported)
- Support for function arguments
- Compile to object file instead of running in OrcJIT (not supported by the Hexagon backend)
- Added support for memory operations like loading the vectors from a Memref and storing to a Memref (mainly for interfacing with C code)

Journey through the Passes

Progressive Lowering in Action

DSPLang Source

- Function takes 3 arguments, one is a result pointer
- Variable declarations and function arguments must have types (with sign)
- This function will be called from the C wrapper with the main function. If function name is main, it can be executed as is.

```
def dsplanglib(c<10000, si16>, d<10000, si16>, e*<10000, si16>) {
  var f<10000, si16> = c * d;
  var g<10000, si16> = c + d;
  e = f * g;
}
```

DSPLang Dialect

```
dsplang.func @dsplanglib(%arg0: memref<10000xi16>, %arg1: memref<10000xi16>, %arg2: memref<10000xi16>) {
  %0 = dsplang.bufferize(%arg0 : memref<10000xi16>) to tensor<10000xsi16>
  %1 = dsplang.bufferize(%arg1 : memref<10000xi16>) to tensor<10000xsi16>
  %2 = dsplang.mul %0, %1 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
  %3 = dsplang.reshape(%2 : tensor<*xsi16>) to tensor<10000xsi16>
  %4 = dsplang.add %0, %1 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
  %5 = dsplang.reshape(%4 : tensor<*xsi16>) to tensor<10000xsi16>
  %6 = dsplang.mul %3, %5 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
  dsplang.assign %arg2 = %6 : tensor<*xsi16> to memref<10000xi16>
  dsplang.return
}
```

Inliner Pass is Applied

- All functions except “dsplanglib” or “main” are inlined and wrapped in a module. This makes further transformations easier.

```
// -----// IR Dump After Inliner (inline) //----- //
module {
  dsplang.func @dsplanglib(%arg0: memref<10000xi16>, %arg1: memref<10000xi16>, %arg2: memref<10000xi16>)
    %0 = dsplang.bufferize(%arg0 : memref<10000xi16>) to tensor<10000xsi16>
    %1 = dsplang.bufferize(%arg1 : memref<10000xi16>) to tensor<10000xsi16>
    %2 = dsplang.mul %0, %1 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
    %3 = dsplang.reshape(%2 : tensor<*xsi16>) to tensor<10000xsi16>
    %4 = dsplang.add %0, %1 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
    %5 = dsplang.reshape(%4 : tensor<*xsi16>) to tensor<10000xsi16>
    %6 = dsplang.mul %3, %5 : (tensor<10000xsi16>, tensor<10000xsi16>) -> tensor<*xsi16>
    dsplang.assign %arg2 = %6 : tensor<*xsi16> to memref<10000xi16>
    dsplang.return
  }
}
```

Shape Inference Pass is Applied

```
// -----// IR Dump After (anonymous namespace)::ShapeInferencePass (dsplang-shape-inference) //----- //
dsplang.func @dsplanglib(%arg0: memref<10000xi16>, %arg1: memref<10000xi16>, %arg2: memref<10000xi16>) {
  %0 = dsplang.bufferize(%arg0 : memref<10000xi16>) to tensor<10000xsi16>
  %1 = dsplang.bufferize(%arg1 : memref<10000xi16>) to tensor<10000xsi16>
  %2 = dsplang.mul %0, %1 : tensor<10000xsi16>
  %3 = dsplang.reshape(%2 : tensor<10000xsi16>) to tensor<10000xsi16>
  %4 = dsplang.add %0, %1 : tensor<10000xsi16>
  %5 = dsplang.reshape(%4 : tensor<10000xsi16>) to tensor<10000xsi16>
  %6 = dsplang.mul %3, %5 : tensor<10000xsi16>
  dsplang.assign %arg2 = %6 : tensor<10000xsi16> to memref<10000xi16>
  dsplang.return
}
```

Canonicalizer Pass is Applied

- Redundant ReshapeOps are eliminated

```
// -----// IR Dump After Canonicalizer (canonicalize) //----- //
dsplang.func @dsplanglib(%arg0: memref<10000xi16>, %arg1: memref<10000xi16>, %arg2: memref<10000xi16>) {
  %0 = dsplang.bufferize(%arg0 : memref<10000xi16>) to tensor<10000xsi16>
  %1 = dsplang.bufferize(%arg1 : memref<10000xi16>) to tensor<10000xsi16>
  %2 = dsplang.mul %0, %1 : tensor<10000xsi16>
  %3 = dsplang.add %0, %1 : tensor<10000xsi16>
  %4 = dsplang.mul %2, %3 : tensor<10000xsi16>
  dsplang.assign %arg2 = %4 : tensor<10000xsi16> to memref<10000xi16>
  dsplang.return
}
```

Affine Passes

DSPLang to Affine Pass

```
%alloc = memref.alloc() : memref<10000xi16>
%alloc_0 = memref.alloc() : memref<10000xi16>
%alloc_1 = memref.alloc() : memref<10000xi16>
affine.for %arg3 = 0 to 10000 {
  %0 = affine.load %arg0[%arg3] : memref<10000xi16>
  %1 = affine.load %arg1[%arg3] : memref<10000xi16>
  %2 = arith.muli %0, %1 : i16
  affine.store %2, %alloc_1[%arg3] : memref<10000xi16>
}
affine.for %arg3 = 0 to 10000 {
  %0 = affine.load %arg0[%arg3] : memref<10000xi16>
  %1 = affine.load %arg1[%arg3] : memref<10000xi16>
  %2 = arith.addi %0, %1 : i16
  affine.store %2, %alloc_0[%arg3] : memref<10000xi16>
}
affine.for %arg3 = 0 to 10000 {
  %0 = affine.load %alloc_1[%arg3] : memref<10000xi16>
  %1 = affine.load %alloc_0[%arg3] : memref<10000xi16>
  %2 = arith.muli %0, %1 : i16
  affine.store %2, %alloc[%arg3] : memref<10000xi16>
}
affine.for %arg3 = 0 to 10000 {
  %0 = affine.load %alloc[%arg3] : memref<10000xi16>
  affine.store %0, %arg2[%arg3] : memref<10000xi16>
}
memref.dealloc %alloc_1 : memref<10000xi16>
memref.dealloc %alloc_0 : memref<10000xi16>
memref.dealloc %alloc : memref<10000xi16>
```

Loop Fusion Pass

```
%alloc = memref.alloc() : memref<10000xi16>
%alloc_0 = memref.alloc() : memref<10000xi16>
%alloc_1 = memref.alloc() : memref<10000xi16>
affine.for %arg3 = 0 to 10000 {
  %0 = affine.load %arg0[%arg3] : memref<10000xi16>
  %1 = affine.load %arg1[%arg3] : memref<10000xi16>
  %2 = arith.muli %0, %1 : i16
  affine.store %2, %alloc_1[%arg3] : memref<10000xi16>
  %3 = affine.load %arg0[%arg3] : memref<10000xi16>
  %4 = affine.load %arg1[%arg3] : memref<10000xi16>
  %5 = arith.addi %3, %4 : i16
  affine.store %5, %alloc_0[%arg3] : memref<10000xi16>
  %6 = affine.load %alloc_1[%arg3] : memref<10000xi16>
  %7 = affine.load %alloc_0[%arg3] : memref<10000xi16>
  %8 = arith.muli %6, %7 : i16
  affine.store %8, %alloc[%arg3] : memref<10000xi16>
  %9 = affine.load %alloc[%arg3] : memref<10000xi16>
  affine.store %9, %arg2[%arg3] : memref<10000xi16>
}
memref.dealloc %alloc_1 : memref<10000xi16>
memref.dealloc %alloc_0 : memref<10000xi16>
memref.dealloc %alloc : memref<10000xi16>
```

Affine Selective Vectorize Pass

```
%alloc = memref.alloc() : memref<10000xi16>
%alloc_0 = memref.alloc() : memref<10000xi16>
%alloc_1 = memref.alloc() : memref<10000xi16>
affine.for %arg3 = 0 to 10000 step 64 {
  %c0_i16 = arith.constant 0 : i16
  %0 = vector.transfer_read %arg0[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %c0_i16_2 = arith.constant 0 : i16
  %1 = vector.transfer_read %arg1[%arg3], %c0_i16_2 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %2 = arith.muli %0, %1 : vector<64xi16>
  vector.transfer_write %2, %alloc_1[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %c0_i16_3 = arith.constant 0 : i16
  %3 = vector.transfer_read %arg0[%arg3], %c0_i16_3 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %c0_i16_4 = arith.constant 0 : i16
  %4 = vector.transfer_read %arg1[%arg3], %c0_i16_4 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %5 = arith.addi %3, %4 : vector<64xi16>
  vector.transfer_write %5, %alloc_0[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %c0_i16_5 = arith.constant 0 : i16
  %6 = vector.transfer_read %alloc_1[%arg3], %c0_i16_5 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %c0_i16_6 = arith.constant 0 : i16
  %7 = vector.transfer_read %alloc_0[%arg3], %c0_i16_6 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %8 = arith.muli %6, %7 : vector<64xi16>
  vector.transfer_write %8, %alloc[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %c0_i16_7 = arith.constant 0 : i16
  %9 = vector.transfer_read %alloc[%arg3], %c0_i16_7 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  vector.transfer_write %9, %arg2[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
}
memref.dealloc %alloc_1 : memref<10000xi16>
memref.dealloc %alloc_0 : memref<10000xi16>
memref.dealloc %alloc : memref<10000xi16>
```

Arith to HLHVX Pass

```
%alloc = memref.alloc() : memref<10000xi16>
%alloc_0 = memref.alloc() : memref<10000xi16>
%alloc_1 = memref.alloc() : memref<10000xi16>
affine.for %arg3 = 0 to 10000 step 64 {
  %0 = vector.transfer_read %arg0[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %1 = vector.transfer_read %arg1[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %2 = vector.bitcast %0 : vector<64xi16> to vector<64xsi16>
  %3 = vector.bitcast %1 : vector<64xi16> to vector<64xsi16>
  %4 = "hlhvx.vmpyi.Vh.VhVh"(%2, %3) : (vector<64xsi16>, vector<64xsi16>) -> vector<64xsi16>
  %5 = vector.bitcast %4 : vector<64xsi16> to vector<64xi16>
  vector.transfer_write %5, %alloc_1[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %6 = vector.transfer_read %arg0[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %7 = vector.transfer_read %arg1[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %8 = vector.bitcast %6 : vector<64xi16> to vector<64xsi16>
  %9 = vector.bitcast %7 : vector<64xi16> to vector<64xsi16>
  %10 = "hlhvx.vadd.Vh.VhVh"(%8, %9) : (vector<64xsi16>, vector<64xsi16>) -> vector<64xsi16>
  %11 = vector.bitcast %10 : vector<64xsi16> to vector<64xi16>
  vector.transfer_write %11, %alloc_0[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %12 = vector.transfer_read %alloc_1[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %13 = vector.transfer_read %alloc_0[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %14 = vector.bitcast %12 : vector<64xi16> to vector<64xsi16>
  %15 = vector.bitcast %13 : vector<64xi16> to vector<64xsi16>
  %16 = "hlhvx.vmpyi.Vh.VhVh"(%14, %15) : (vector<64xsi16>, vector<64xsi16>) -> vector<64xsi16>
  %17 = vector.bitcast %16 : vector<64xsi16> to vector<64xi16>
  vector.transfer_write %17, %alloc[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %18 = vector.transfer_read %alloc[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  vector.transfer_write %18, %arg2[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
}
memref.dealloc %alloc_1 : memref<10000xi16>
memref.dealloc %alloc_0 : memref<10000xi16>
memref.dealloc %alloc : memref<10000xi16>
```

HLHVX to LLHVX Pass

```
%alloc = memref.alloc() : memref<10000xi16>
%alloc_0 = memref.alloc() : memref<10000xi16>
%alloc_1 = memref.alloc() : memref<10000xi16>
affine.for %arg3 = 0 to 10000 step 64 {
  %0 = vector.transfer_read %arg0[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %1 = vector.transfer_read %arg1[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %2 = vector.bitcast %0 : vector<64xi16> to vector<32xi32>
  %3 = vector.bitcast %1 : vector<64xi16> to vector<32xi32>
  %4 = "llhvx.intr.vmpyih_128B"(%2, %3) : (vector<32xi32>, vector<32xi32>) -> vector<32xi32>
  %5 = vector.bitcast %4 : vector<32xi32> to vector<64xi16>
  vector.transfer_write %5, %alloc_1[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %6 = vector.transfer_read %arg0[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %7 = vector.transfer_read %arg1[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %8 = vector.bitcast %6 : vector<64xi16> to vector<32xi32>
  %9 = vector.bitcast %7 : vector<64xi16> to vector<32xi32>
  %10 = "llhvx.intr.vaddh_128B"(%8, %9) : (vector<32xi32>, vector<32xi32>) -> vector<32xi32>
  %11 = vector.bitcast %10 : vector<32xi32> to vector<64xi16>
  vector.transfer_write %11, %alloc_0[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %12 = vector.transfer_read %alloc_1[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %13 = vector.transfer_read %alloc_0[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  %14 = vector.bitcast %12 : vector<64xi16> to vector<32xi32>
  %15 = vector.bitcast %13 : vector<64xi16> to vector<32xi32>
  %16 = "llhvx.intr.vmpyih_128B"(%14, %15) : (vector<32xi32>, vector<32xi32>) -> vector<32xi32>
  %17 = vector.bitcast %16 : vector<32xi32> to vector<64xi16>
  vector.transfer_write %17, %alloc[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
  %18 = vector.transfer_read %alloc[%arg3], %c0_i16 {in_bounds = [true]} : memref<10000xi16>, vector<64xi16>
  vector.transfer_write %18, %arg2[%arg3] {in_bounds = [true]} : vector<64xi16>, memref<10000xi16>
}
memref.dealloc %alloc_1 : memref<10000xi16>
memref.dealloc %alloc_0 : memref<10000xi16>
memref.dealloc %alloc : memref<10000xi16>
```

Lowering to LLVM Dialect

```
^bb2:  // pred: ^bb1
  %63 = llvm.extractvalue %17[1] : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %64 = llvm.getelementptr %63[%61] : (!llvm.ptr, i64) -> !llvm.ptr, i16
  %65 = llvm.load %64 {alignment = 2 : i64} : !llvm.ptr -> vector<64xi16>
  %66 = llvm.extractvalue %11[1] : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %67 = llvm.getelementptr %66[%61] : (!llvm.ptr, i64) -> !llvm.ptr, i16
  %68 = llvm.load %67 {alignment = 2 : i64} : !llvm.ptr -> vector<64xi16>
  %69 = llvm.bitcast %65 : vector<64xi16> to vector<32xi32>
  %70 = llvm.bitcast %68 : vector<64xi16> to vector<32xi32>
  %71 = "llhvx.intr.vmpyih_128B"(%69, %70) : (vector<32xi32>, vector<32xi32>) -> vector<32xi32>
  %72 = llvm.bitcast %71 : vector<32xi32> to vector<64xi16>
  %73 = llvm.extractvalue %57[1] : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %74 = llvm.getelementptr %73[%61] : (!llvm.ptr, i64) -> !llvm.ptr, i16
  llvm.store %72, %74 {alignment = 2 : i64} : vector<64xi16>, !llvm.ptr
```

LLVM Dialect to LLVM IR

```
52:                                               ; preds = %49
  %53 = extractvalue { ptr, ptr, i64, [1 x i64], [1 x i64] } %30, 1, !dbg !10
  %54 = getelementptr i16, ptr %53, i64 %50, !dbg !10
  %55 = load <64 x i16>, ptr %54, align 2, !dbg !10
  %56 = extractvalue { ptr, ptr, i64, [1 x i64], [1 x i64] } %25, 1, !dbg !10
  %57 = getelementptr i16, ptr %56, i64 %50, !dbg !10
  %58 = load <64 x i16>, ptr %57, align 2, !dbg !10
  %59 = bitcast <64 x i16> %55 to <32 x i32>, !dbg !10
  %60 = bitcast <64 x i16> %58 to <32 x i32>, !dbg !10
  %61 = call <32 x i32> @llvm.hexagon.V6.vmpyih.128B(<32 x i32> %59, <32 x i32> %60), !dbg !10
  %62 = bitcast <32 x i32> %61 to <64 x i16>, !dbg !10
  %63 = extractvalue { ptr, ptr, i64, [1 x i64], [1 x i64] } %48, 1, !dbg !10
  %64 = getelementptr i16, ptr %63, i64 %50, !dbg !10
  store <64 x i16> %62, ptr %64, align 2, !dbg !10
```

Running the LLVM Optimizer @ O3

- The optimizer gets rid of redundant loads and stores between every operation within the loop

```
16:                                               ; preds = %63, %15
  %17 = phi i64 [ 0, %15 ], [ %91, %63 ]
  %18 = getelementptr i16, ptr %1, i64 %17, !dbg !8
  %19 = load <32 x i32>, ptr %18, align 2, !dbg !8
  %20 = getelementptr i16, ptr %6, i64 %17, !dbg !8
  %21 = load <32 x i32>, ptr %20, align 2, !dbg !8
  %22 = tail call <32 x i32> @llvm.hexagon.V6.vmpyih.128B(<32 x i32> %19, <32 x i32> %21), !dbg !8
  %23 = tail call <32 x i32> @llvm.hexagon.V6.vaddh.128B(<32 x i32> %19, <32 x i32> %21), !dbg !9
  %24 = tail call <32 x i32> @llvm.hexagon.V6.vmpyih.128B(<32 x i32> %22, <32 x i32> %23), !dbg !10
  %25 = getelementptr i16, ptr %11, i64 %17, !dbg !7
  store <32 x i32> %24, ptr %25, align 2, !dbg !7
```

Final Hexagon Assembly

```
408: {      r12 = setbit(r3,#0x8)
40c:        r5 = add(r0,r12)
410:        v24 = vmemu(r5+#0x0) }
414: {      r28 = add(r1,r12)
418:        v0 = vmemu(r14+#0x0) }
41c: {      v25.h = vadd(v0.h,v24.h) }
420: {      v0.h = vmpyi(v0.h,v24.h) }
424: {      v0.h = vmpyi(v0.h,v25.h) }
428: {      r5 = add(r2,r12)
42c:        vmemu(r5+#0x0) = v0 }
```

Tour of the Source Code

Brief Overview of Critical Parts

HLHVX and LLHVX Ops are Automatically Generated

LLHVX TableGen Definitions

```
// GENERATED BY scripts/gen_hvxops.py - DO NOT EDIT MANUALLY

def VrmpybubRtt128B :
    LLHVX_OneResultIntrOp<
        "vrmpybub_rtt_128B"
    >,
    Arguments<(ins
        VectorOfLengthAndType<[32], [I32]>:$arg0,
        I64:$arg1
    )>;

def VrmpybubRttAcc128B :
    LLHVX_OneResultIntrOp<
        "vrmpybub_rtt_acc_128B"
    >,
    Arguments<(ins
        VectorOfLengthAndType<[64], [I32]>:$arg0,
        VectorOfLengthAndType<[32], [I32]>:$arg1,
        I64:$arg2
    )>;
```

HLHVX TableGen Definitions

```
// GENERATED BY scripts/gen_hvxops.py - DO NOT EDIT MANUALLY

def HVX_vextract_R_VR :
    HLHVX_Op<
        "vextract.R.VR"
    >,
    Arguments<(ins
        VectorOfLengthAndType<[1024], [I1]>:$arg0,
        AnyI32:$arg1
    )>,
    Results<(outs
        AnyI32:$result
    )>;

def HVX_hi_V_W :
    HLHVX_Op<
        "hi.V.W"
    >,
    Arguments<(ins
        VectorOfLengthAndType<[2048], [I1]>:$arg0
    )>,
    Results<(outs
        VectorOfLengthAndType<[1024], [I1]>:$result
    )>;
```

LLHVX to LLVMIR Translation Test Suite is Automatically Generated

```
// RUN: llhvx-translate %s -mlir-to-llvmir -split-input-file | FileCheck %s

// CHECK-LABEL: define <64 x i32> @vrmpybub_rtt_128B
func.func @vrmpybub_rtt_128B(%A : vector<32 x i32>, %B : i64) -> vector<64 x i32> {
    // CHECK: call <64 x i32> @llvm.hexagon.V6.vrmpybub.rtt.128B(
    // CHECK-SAME: <32 x i32> %{{[0-9]+}},
    // CHECK-SAME: i64 %{{[0-9]+}})
    %0 = "llhvx.intr.vrmpybub_rtt_128B"(%A, %B) : (vector<32 x i32>, i64) -> vector<64 x i32>
    return %0 : vector<64 x i32>
}

// CHECK-LABEL: define <64 x i32> @vrmpybub_rtt_acc_128B
func.func @vrmpybub_rtt_acc_128B(%A : vector<64 x i32>, %B : vector<32 x i32>, %C : i64) -> vector<64 x i32> {
    // CHECK: call <64 x i32> @llvm.hexagon.V6.vrmpybub.rtt.acc.128B(
    // CHECK-SAME: <64 x i32> %{{[0-9]+}},
    // CHECK-SAME: <32 x i32> %{{[0-9]+}},
    // CHECK-SAME: i64 %{{[0-9]+}})
    %0 = "llhvx.intr.vrmpybub_rtt_acc_128B"(%A, %B, %C) : (vector<64 x i32>, vector<32 x i32>, i64) -> vector<64 x i32>
    return %0 : vector<64 x i32>
}
```

HLHVX to LLHVX Conversion is Automatically Generated – Leverages PDLL instead of TableGen

```
// GENERATED BY scripts/gen_hvxops.py - DO NOT EDIT MANUALLY

Pattern Rewrite_vextract_R_VR {
    let root = op<hlhvx.vextract.R.VR>(arg0: Value, arg1: Value);

    rewrite root with {
        let arg2 = op<hlhvx.CastVectorToGeneric>(arg0) { length = attr<"32 : ui32"> };
        let arg3 = op<llhvx.intr.extractw_128B>(arg2, arg1) -> (type<"i32">);
        let arg4 = op<hlhvx.CastIntegerFromGeneric>(arg3);
        replace root with arg4;
    };
}

Pattern Rewrite_hi_V_W {
    let root = op<hlhvx.hi.V.W>(arg0: Value);

    rewrite root with {
        let arg1 = op<hlhvx.CastVectorToGeneric>(arg0) { length = attr<"64 : ui32"> };
        let arg2 = op<llhvx.intr.hi_128B>(arg1) -> (type<"vector<32 x i32>">);
        let arg3 = op<hlhvx.CastVectorFromGeneric>(arg2) { length = attr<"1024 : ui32">, type = attr<"\"i1\""> } -> (type
        replace root with arg3;
    };
}
```

Arith to HLHVX Lowered using PDLL

```
Pattern Rewrite_arith_addi_128xsi16 {
    let root = op<arith.addi>(arg0: Value, arg1: Value) -> (type<"vector<128 x i16>">);

    rewrite root with {
        let carg0 = op<vector.bitcast>(arg0) -> (type<"vector<128 x si16>">);
        let carg1 = op<vector.bitcast>(arg1) -> (type<"vector<128 x si16>">);
        let result = op<hlhvx.vadd.Wh.WhWh>(carg0, carg1) -> (type<"vector<128 x si16>">);
        let cresult = op<vector.bitcast>(result) -> (type<"vector<128 x i16>">);
        replace root with cresult;
    };
}

Pattern Rewrite_arith_xori {
    let root = op<arith.xori>(arg0: Value, arg1: Value) -> (resultType: Type);

    rewrite root with {
        let carg0 = op<vector.bitcast>(arg0) -> (type<"vector<1024 x i1>">);
        let carg1 = op<vector.bitcast>(arg1) -> (type<"vector<1024 x i1>">);
        let result = op<hlhvx.vxor.V.VV>(carg0, carg1) -> (type<"vector<1024 x i1>">);
        let cresult = op<vector.bitcast>(result) -> (resultType);
        replace root with cresult;
    };
}
```

Conclusion

Outcomes and Future Work

Outcomes

- First-of-its-kind dialects introduced to allow the rest of the MLIR ecosystem to interface with HVX operations.
- A DSL for vector processing with basic functionality – binary operations, functions, memory operations and debug print was implemented.
- End-to-end DSL to device workflow is demonstrated on the Hexagon platform.
- MLIR base allows for easily targeting other architectures.
- End-to-end fully open-source and reproducible compiler stack.

Future Work

- Currently only binary operations and expressions are supported. Reduction based workflows will be done (supported by Affine Dialect’s vectorization passes).
- Support for sliding-window kind of operations, which are extensively used in image processing.
- Automatic L2 pre-fetching could be done in the loop preamble.