moviCompile: An LLVM based compiler for heterogeneous SIMD code generation

Erkan Diken, Roel Jordans, *Martin J. O'Riordan
Eindhoven University of Technology, Eindhoven
(*) Movidius Ltd., Dublin

LLVM devroom FOSDEM'15
Brussels, Belgium

February 1, 2015

Content

Background
- SIMD
- Heterogeneous SIMD
- SHAVE Vector Processor

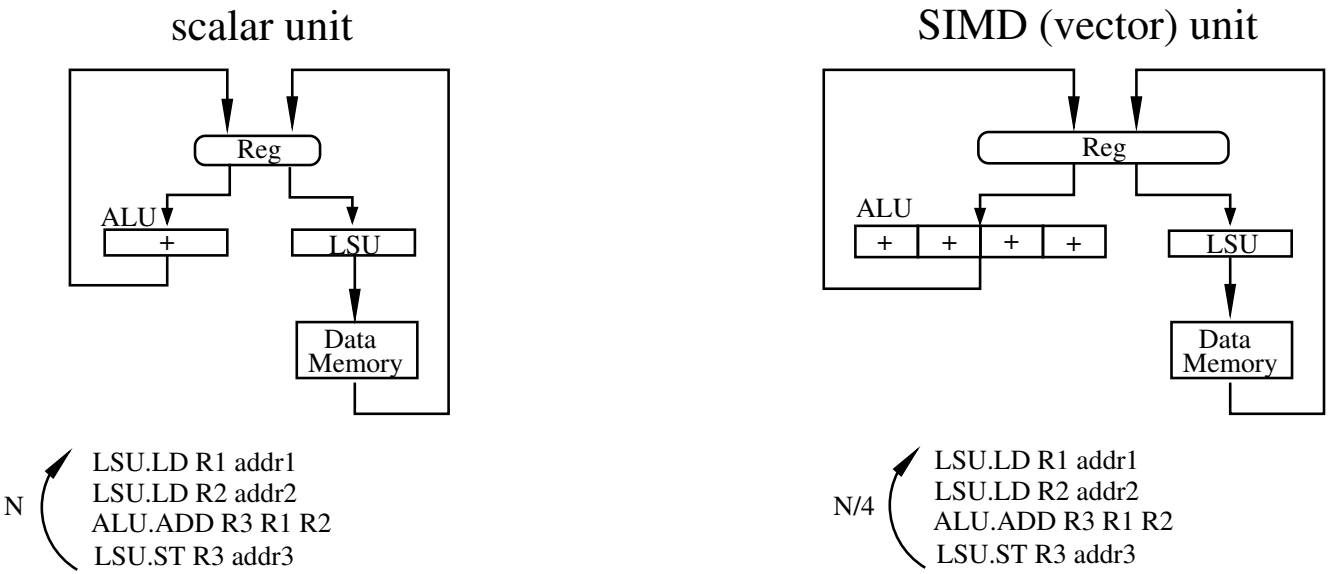
Code Generation
- SIMD Code generation for SHAVE
- Contribution
- Adding a new vector type
- Type Legalization
- Common Errors
- Instruction Selection and Lowering

Results

Conclusion

SIMD

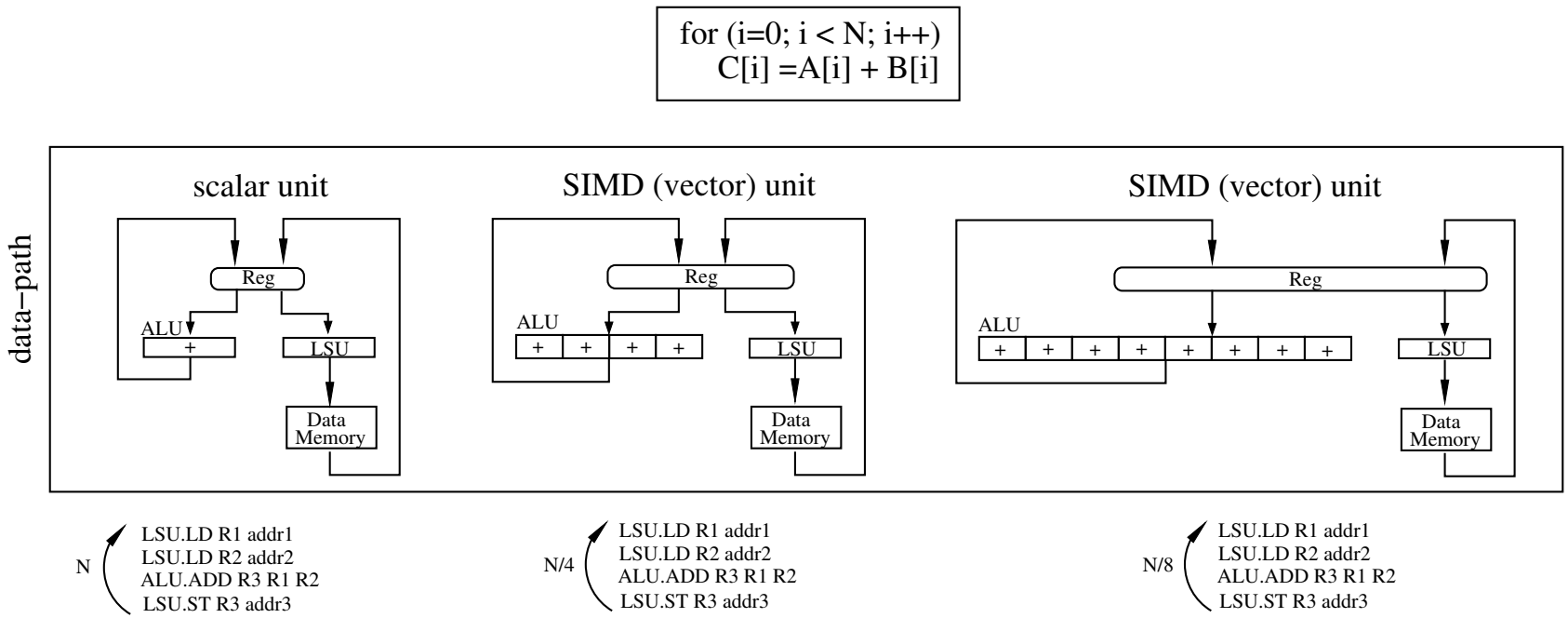
```
for (i=0; i < N; i++)
    C[i] =A[i] + B[i]
```



- Single-instruction multiple-data (SIMD) model of execution
- The same instruction applies to all processing elements
- Improves performance and energy efficiency

HETEROGENEOUS SIMD

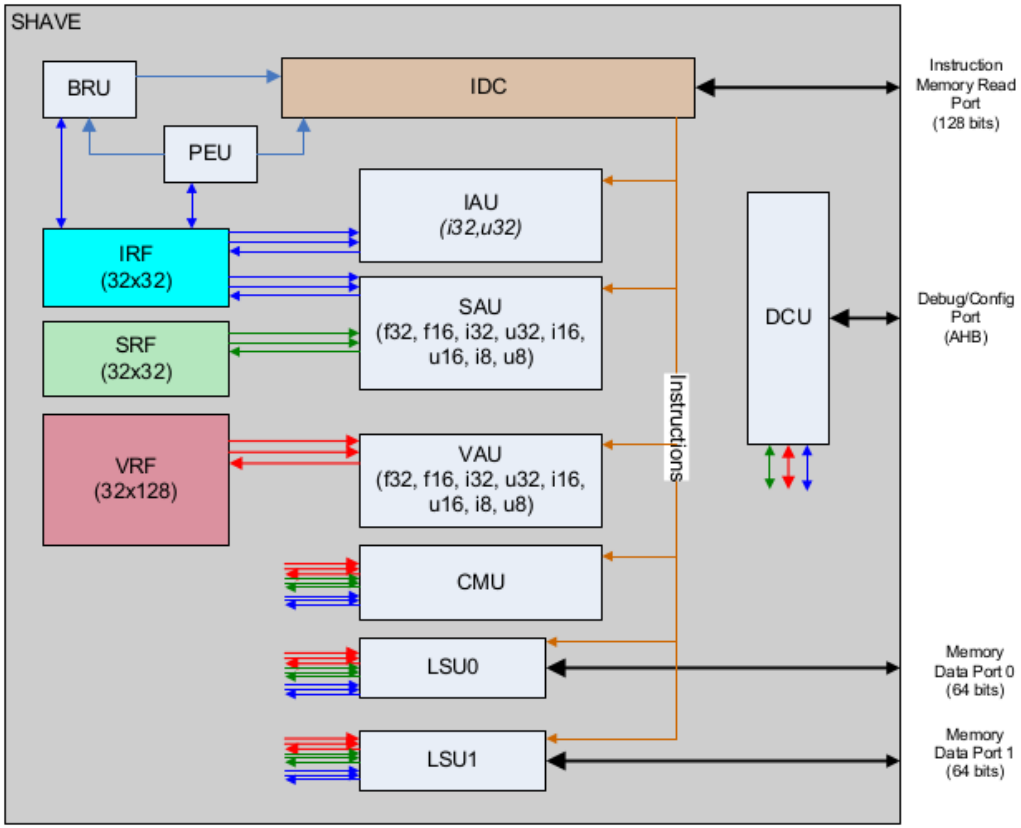
- Variable SIMD-width: Intel's SSE/AVX support 128/256/512-bit SIMD, 1024-bit in the future



- Our focus: VLIW data-path with multiple native SIMD-widths

SHAVE VECTOR PROCESSOR

The SHAVE (Streaming Hybrid Architecture Vector Engine) VLIW vector processor



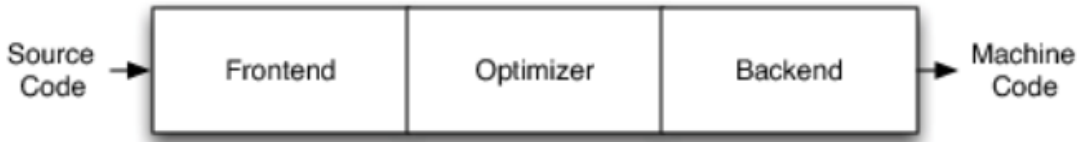
SIMD Code generation for SHAVE

- VAU is designed to support 128-bit vector arithmetic of 8/16/32-bit integer and 16/32-bit floating-point types.
- Instruction set (ISA) supports a range of precision
- Current compiler supports 128-bit and 64-bit SIMD code generation.
- 128-bit legal vector types: 16 x i8, 8 x i16, 4 x i32, 8 x f16, 4 x f32
- 64-bit legal vector types: 8 x i8, 4 x i16, 4 x f16
- What about 32-bit vector types: 4 x i8, 2 x i16, 2 x f16 (short vectors) ?

Contribution

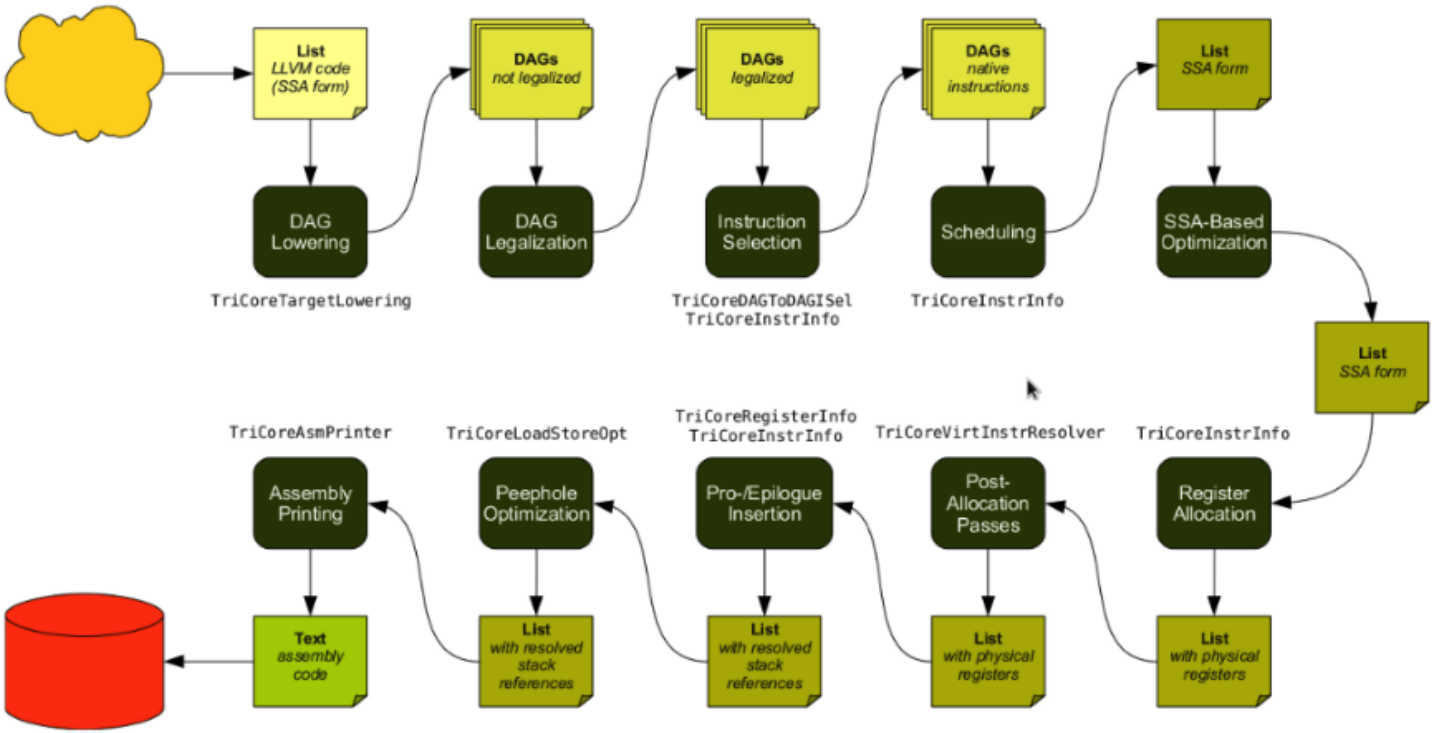
- Short vectors are promoted to longer types before vector computation on VAU
- SAU supports 32-bit vector arithmetic of 8/16-bit integer and 16-bit floating-point types.
- Contribution: Adding compiler support for 32-bit SIMD code generation.
- SIMD code for short vector types (e.g. 4 x i8, 2 x i16, 2 x f16) that can be executed on 32-bit SAU next to 128/64-bit VAU instruction

LLVM CODE GENERATION FLOW



(*) Tutorial: Creating an LLVM Backend for the Cpu0 Architecture (http://jonathan2251.github.io/lbd/llvmstructure.html)

LLVM CODE GENERATION FLOW



- Already in place: data-layout, triple, target registration, register set and classes, instruction set definitions
- Main focus on TableGen, type legalization and lowering for instruction selection

(*) Tutorial: Creating an LLVM Backend for the Cpu0 Architecture (http://jonathan2251.github.io/lbd/llvmstructure.html)

Listing 1: 4 x i8

```
define i32 @main() {
entry:

; memory allocation on run-time stack
%xptr = alloca <4 x i8>
%yptr = alloca <4 x i8>
%zptr = alloca <4 x i8>

; load the vectors
%x = load <4 x i8>* %xptr
%y = load <4 x i8>* %yptr

; add the vectors
%z = add <4 x i8> %x, %y

; store the result vector back to stack
store <4 x i8> %z, <4 x i8>* %zptr

ret i32 0
}
```

Listing 2: Assembly code with long vector operations

```
main:
 IAU.SUB i19 i19 16
 LSU1.LDO32 i9 i19 8
 LSU1.LDO32 i10 i19 12
 NOP 4
 CMU.CPIV.x32 v14.0 i9
 CMU.CPIV.x32 v15.0 i10
 CMU.CPVV.i8.i16 v14 v14
 CMU.CPVV.i8.i16 v15 v15
 VAU.ADD.i16 v15 v15 v14
 NOP
 BRU.JMP i30
  || CMU.VSZMBYTE v15 v15 [Z2Z0]
 CMU.CPVV.u16.u8s v15 v15
 CMU.CPVI.x32 i17 v15.0
 IAU.ADD i19 i19 16
  || LSU0.LDIL i18 0
  || LSU1.STO32 i17 i19 4
```

Before you start

- http://llvm.org/docs/WritingAnLLVMBackend.html
- *Building an LLVM Backend* by Fraser Cormack and Pierre-Andre Saulais
- LLVM build in debug mode
- ./llc -debug, -print-after-all, -debug-only=shave-lowering
- -view-dag-combine1-dags: displays the DAG after being built, before the first optimization pass.
- -view-legalize-dags: displays the DAG before legalization.
- -view-dag-combine2-dags: displays the DAG before the second optimization pass.
- -view-isel-dags: displays the DAG before the Select phase.
- -view-sched-dags: displays the DAG before Scheduling.
- Get ready with your favorite editor (emacs llvm mode)

ADDING A NEW TYPE OF V4I8

Type Legalization: Make v4i8 vector type legal for the target

```
unsigned supportedIntegerVectorTypes[] = {MVT::v16i8, MVT::v8i16, MVT::↩
    v4i32, MVT::v4i16, MVT::v8i8, MVT::v4i8};
```

Specify which types are supported:

Listing 3: SHAVERegisterInfo.td

```
def IRF32: RegisterClass<"SHAVE", [i32, v4i8], 32,
            (add,
            I10, I9 ... //register list
            )>;
```

Register class association: register class is available for the value type

Listing 4: SHAVELowering.cpp

```
addRegisterClass(MVT::v4i8, &SHAVE::IRF32RegClass)
```

First build, First Error

tblgen: error: Could not infer all types in pattern!

```
class IAU_RROpC<SDNode opc, RegisterClass regVT, string asmstr> :
   SHAVE_IAUInstr<(outs regVT:$dst), (ins regVT:$src),
                  !strconcat(asmstr, " $dst $src"),
                  [(set regVT:$dst, (opc regVT:$src))]>;
```

Well-typed class:

```
class IAU_RROpC<SDNode opc, RegisterClass regVT, string asmstr> :
   SHAVE_IAUInstr<(outs regVT:$dst), (ins regVT:$src),
                  !strconcat(asmstr, " $dst $src"),
                  [(set (i32 regVT:$dst), (opc (i32 regVT:$src)))]>;
```

First Test, Second Error: Cannot select

- v4i8 is legal type now (Type Legalization ✓)
- Pattern matching and instruction selection
- Which operations are supported for supported ValueTypes ?
  - Legal: The target natively supports this operation.
  - Promote: This operation should be executed in a larger type.
  - Expand: Try to expand this to other operations.
  - Custom: Use the LowerOperation hook to implement custom lowering.
- Start with adding patterns in .td files for legal operations

```
class SAU_RRROpC<SDNode opc, RegisterClass regVT, ValueType vt, string↩
      asmstr> :
     SHAVE_SAUInstr<(outs regVT:$dst), (ins regVT:$src1, regVT:$src2),
          !strconcat(asmstr, " $dst $src1 $src2"),
          [(set (vt regVT:$dst), (opc regVT:$src1, regVT:$src2))]>;
```

```
multiclass SAU_IRF_8_16_32_RRROp<SDNode opc, string asmstr>
{
  //scalar types
  def _i32 : SAU_RRROpC<opc, IRF32,   i32, !strconcat(asmstr, ".i32")>;

  // Vector types
  def _v4i8  : SAU_RRROpC<opc, IRF32, v4i8, !strconcat(asmstr, ".i8")>;
}
```

```
  defm SAU_ADD : SAU_IRF_8_16_32_RRROp<add, ".ADD">;
```

Assembly string:

```
SAU.ADD.i8 $dst $src1 $src2
```

CUSTOM LOWERING

Add callback for operations that are NOT supported by the target:

```
setOperationAction(ISD::EXTRACT_SUBVECTOR, MVT::v4i8, Custom);
```

```
SDValue LowerOperation(SDValue Op, SelectionDAG &DAG) const;
{
    switch(op.getOpcode())
    {
      ...
      case ISD::EXTRACT_SUBVECTOR :
           return SHAVELowerEXTRACT_SUBVECTOR(op, DAG);
      ...
    }
}
```

```
SDValue SHAVELowering::SHAVELowerEXTRACT_SUBVECTOR(SDValue op, ↩
     SelectionDAG &DAG) const
{
  SDNode *Node = op.getNode();
  SDLoc dl = SDLoc(op);
  SmallVector<SDValue, 8> Ops;

  SDValue SubOp = Node->getOperand(0);
  EVT VVT = SubOp.getNode()->getValueType(0);
  EVT EltVT = VVT.getVectorElementType();
  unsigned idx = Node->getConstantOperandVal(1);

  EVT VecVT = op.getValueType();
  unsigned NumExtElements = VecVT.getVectorNumElements();

  for (unsigned i=0; i < NumExtElements; i++) {
    Ops.push_back(DAG.getNode(ISD::EXTRACT_VECTOR_ELT, dl, EltVT, SubOp↩
         , DAG.getConstant(idx+i, MVT::i32, false)));
  }

  return DAG.getNode(ISD::BUILD_VECTOR, dl, op.getValueType(), Ops);
}
```

Listing 5: Assembly code with short vector operations

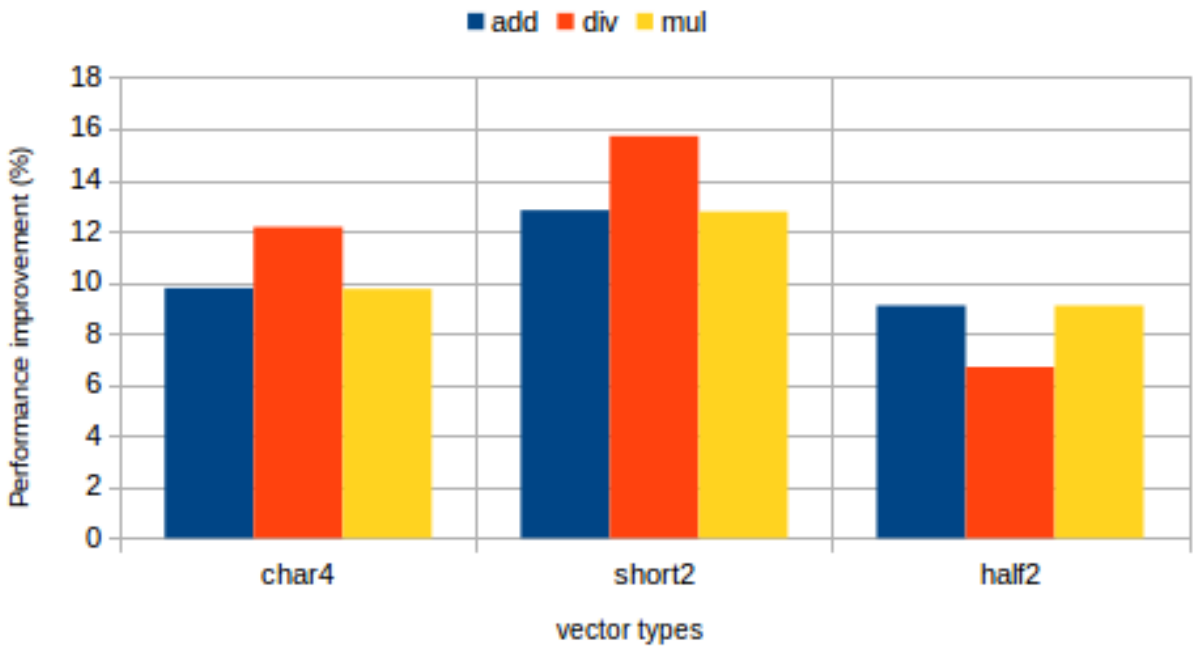
```
main:
 IAU.SUB i19 i19 16
 LSU1.LDO32 i10 i19 12
  || LSU0.LDO32 i9 i19 8
 NOP 2
 BRU.JMP i30
 NOP 2
 SAU.ADD.i8 i10 i10 i9
 NOP
 IAU.ADD i19 i19 16
  || LSU0.LDIL i18 0
  || LSU1.STO32 i10 i19 4
```

Listing 6: IR code with two different vector types

```
define <4 x i8> @main(<4 x i8> %a, <4 x i8> %b,
                      <8 x i8> %x, <8 x i8> %y,
                      <8 x i8>* %zptr){
entry:
        %c = add <4 x i8> %a, %b
        %z = add <8 x i8> %x, %y
        store <8 x i8> %z, <8 x i8>* %zptr
        ret <4 x i8> %c
}
```

Listing 7: Heterogeneous SIMD assembly code

```
main:
        BRU.JMP i30
        CMU.CPVI.x32 i9 v22.0
        CMU.CPVI.x32 i10 v23.0
        VAU.ADD.i8 v15 v21 v20
            || SAU.ADD.i8 i10 i10 i9
        NOP
        CMU.CPIV.x32 v23.0 i10
            || LSU1.ST64.l v15 i18
```


add
div
mul
Performance improvement (%)
18
16
14
12
10
8
6
4
2
0
char4
short2
half2
vector types

Thank you for your attention!

Questions ?

Contact: e.diken@tue.nl
LinkedIn: nl.linkedin.com/in/erkandiken/

TU/e Technische Universiteit Eindhoven University of Technology

Movidius

HiPEAC

LLVM COMPILER INFRASTRUCTURE