Bringing up EPI RISC-V Vector architecture Software

First steps towards a made-in-Europe high-performance microprocessor

Roger Ferrer*, Filippo Mantovani*, Jesús Labarta
Barcelona Supercomputing Center

Bologna, 22 Jan 2020

Motivation & Goals

The EPI project brings a chip with an accelerator (EPI Accelerator, EPAC) that can speed different workloads up but still needs to be programmed.

In this session we will focus only on the vector accelerator.

Our goal is to answer the following questions:

- What is our suggestion programming the EPI vector accelerator
  - There may be other valid approaches, today we will see one
- How to gain insights of an application running in the EPI vector accelerator
  - Does my application map well to the accelerator?

What is in the menu today?

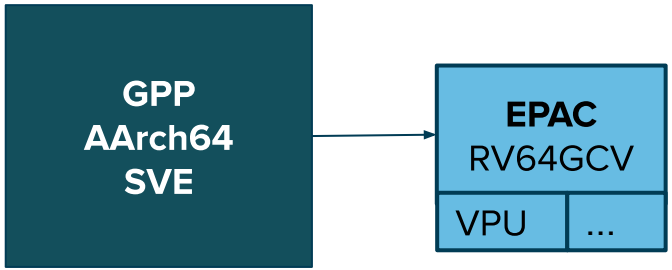
- Tools to compile and emulate vector code
  - LLVM-based Compiler
  - Vehave Functional Emulator

- Tools to gain insights
  - Compiler Explorer-based tool
  - Paraver Trace Visualization and Analysis

Context

The big picture of the EPI accelerator

A simplified view of the EPI architecture for the sake of this tutorial is as follows



A program starts running in the GPP. Parts of the program that are suitable for acceleration are offloaded to the accelerator. This will be done using OpenMP 4.5 **`#pragma omp target`**

Both the GPP and the accelerator run Linux. Memory is **not** shared between the GPP and EPAC but the goal is to minimize explicit copies thanks to the underlying memory architecture of EPI.

The vector accelerator

The vector accelerator is a RISC-V 64-bit core suitable for 64-bit Linux (RV64GC, i.e. mul/div, float32, float64, atomics) extended with the V-extension.

The V-extension is a standard vector extension of the RISC-V ISA still being drafted by the RISC-V community.

The vector accelerator is suitable for offloading HPC kernels that can be accelerated using vector instructions.

RISC-V V-extension (1)

RISC-V as an ISA is made up by a base ISA and a number of standard extensions. The V-extension provides vector processing to RISC-V.

Current stable draft (https://github.com/riscv/riscv-v-spec) is version 0.8. We will be using version 0.7.1 in this tutorial and the differences are not important today.

Main highlights of the V-extension are

- An implementation-defined length of the vector called VLEN
  - 32 vector registers of `VLEN` bits
  - Removes the “we need a whole new ISA if we want longer vectors” pain point (e.g. AVX2 → AVX512)

- An implementation-defined maximum element-width called ELEN
  - The program can operate with vectors of 8-bit elements up to `ELEN` bits elements
  - Alleviates the “my market segment does not care about vectors of 64-bit elements”

RISC-V V-extension (2)

- A “set vector length” instruction that sets both the number of elements being operated (VL, vector length) and the width of the elements being operated (SEW, single-element width)
  - VL is reminiscent of classical vector architectures
  - Alleviates vector ISAs taking a lot of encoding space because of the plethora of data-types

- A length multiplier LMUL that allows grouping vector registers
  - Allows maximizing register utilization without impacting code size at expense of a smaller number of registers
  - Valid values for LMUL are 1 (no grouping), 2 (16 registers of 2 × VLEN bits), 4 (8 registers of 4 × VLEN bits) and 8 (4 of 8 × VLEN bits)

- Vector masking support
  - Useful when vectorizing codes that have non-uniform control flow

RISC-V V-extension (3)

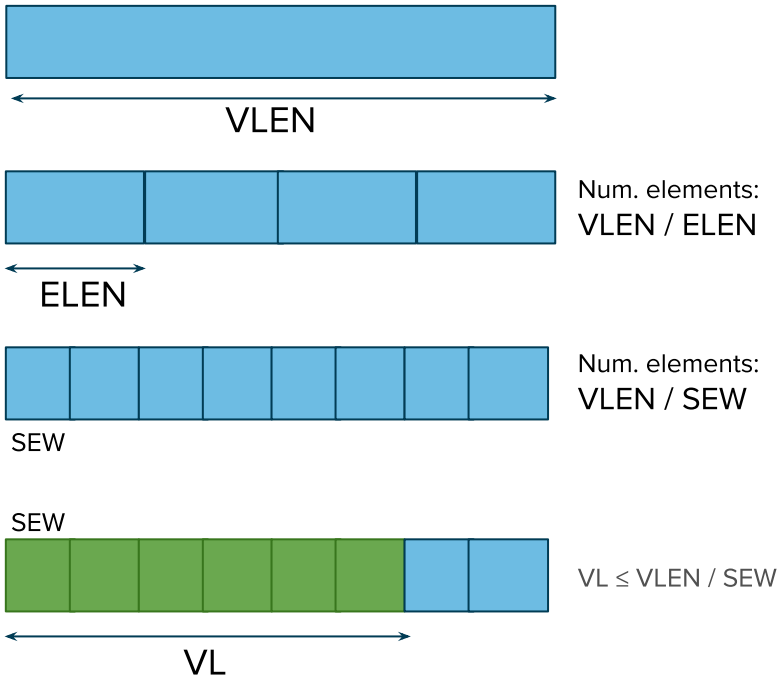
Quick recap before we continue.

The implementer chooses VLEN and ELEN

- A vector register has VLEN bits
- The maximum element width (in bits) we can divide a vector register is called ELEN

The programmer at runtime can define

- SEW, width in bits of the elements we are going to operate, 8 ≤ SEW ≤ ELEN
- VL, the number of **elements** (not bits!) we are going to operate 0 ≤ VL ≤ VLEN / SEW



RISC-V V-extension (4)

Examples

- Add two vectors of 32-bit integers

```
vsetvli x1, x2, e32, m1  // VL←min(VLEN / 32, x2); SEW←32; LMUL←1; x1←VL
vadd.vv v3, v1, v2       // v3[e] ← v1[e] + v2[e], forall 0 ≤ e < VL
```

- Add two vectors of doubles (float64)

```
vsetvli x1, x2, e64, m1  // VL←min(VLEN / 64, x2); SEW←64; LMUL←1; x1←VL
vfadd.vv v3, v1, v2      // v3[e] ← v1[e] + v2[e], forall 0 ≤ e < VL
```

- Add two vectors of doubles (float64) but using LMUL=2
  (can only use even-numbered registers)

```
vsetvli x1, x2, e64, m2  // VL←min((2*VLEN) / 64, x2); SEW←64; LMUL←1; x1←VL
vfadd.vv v4, v0, v2      // v{4,5}[e] ← v{0,1}[e] + v{2,3}[e],
                         //    forall 0 ≤ e < VL
```

Mapping “e” to v{2*i} or v{2*i+1} is implementation defined by SLEN

This is all very nice but how does it compare to other state-of-the-art vector ISAs?

	Intel AVX-512	Arm SVE	NEC SX-Aurora TSUBASA	RISC-V V-extension
Is the vector register size defined by the architecture?	**Yes**. 512 bit	**No**. From 128 bit to 2048 bit (multiples of 128)	**Yes**. Current generation is 16384 bits	**No**. As long as ELEN ≤ VLEN (both power of two)
Masking?	**Yes**. 8 mask registers	**Yes**. 16 predicate registers	**Yes**. 16 vector mask registers	**Yes**. Only `v0` as an implicit operand if the instruction is masked
Vector length?	**No**	**No**	**Yes**	**Yes**

This table is by no means meant to be exhaustive, there are other important differences like the set of data types supported by the ISA (e.g. fixed floating types, complex, polynomials, saturated arithmetic, etc).

The EPI vector accelerator

Because our context is HPC, our EPI vector accelerator is currently aiming at

- ELEN = 64 bit (i.e. up to vectors of int64 or double)
- VLEN = 16384 (i.e. a vector can hold 256 elements of int64 / doubles or 512 elements of int32 / floats, etc)

How to use the V-extension

How to use the V-extension in our programs

Ok so, how do we use the V-extension?

- Assembler.
  - Always a valid option but not the most pleasant 😁
- C/C++ builtins
  - Low-level mapping to the instructions but allows embedding it into an existing C/C++ codebase
  - Allows relatively quick experimentation
- `#pragma omp simd` (aka “Semi automated vectorization”)
  - Relies on vectorization capabilities of the compiler
    - Usually works but gets complicated if the code calls functions
  - Also useable in Fortran
- Autovectorization
  - All bets are off 😉

Work we did in EPI to enable the V-extension in our compiler

In EPI we want to ultimately be able to offload (via #pragma omp target) code that uses the V-extension (ideally using #pragma omp simd).

We took LLVM/clang and we had to do a few things

- Add support for assembly/disassembly the new V-extension instructions
- Devise a code generation mechanism
  - A vector operation of a specific data-type can bee seen as an (optional) “set VL, SEW” followed by the actual instruction
- Provide a mechanism to target that code generation mechanism from C/C++
  - Target specific builtins
- Adapted LLVM’s Loop Vectorizer to generate “something” that uses V-extension instructions
  - More on this later

Kudos to Arm folks for setting the foundation stone in LLVM on which this work was possible.

The first example: a vector addition

Let's now walk-through an extremely simple example of how we can use the V-extension.

Consider a vector addition:

```
void add_ref(long N, double *restrict c, double *restrict a,
             double *restrict b) {
  for (long i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

Ideally the compiler should be able to autovectorize this. But let's see how we can do that manually using builtins.

Vector addition with builtins (1)

```
void add_vec(long N, double *restrict c, double *restrict a,
             double *restrict b) {

  long gvl; // “Granted” vector length
  for (long i = 0; i < N;) {
      gvl = __builtin_epi_vsetvl(N - i, __epi_e64, __epi_m1);
      __epi_1xf64 va = __builtin_epi_vload_1xf64(&a[i], gvl);
      __epi_1xf64 vb = __builtin_epi_vload_1xf64(&b[i], gvl);
      __epi_1xf64 vc = __builtin_epi_vfadd_1xf64(va, vb, gvl);
      __builtin_epi_vstore_1xf64(&c[i], vc, gvl);
      i += gvl;
  }
}
```

“Requested” vector length

Single-element Width (SEW)

Length multiplier (LMUL)

Did you notice?
No epilog/tail loop for the remaining elements!

Vector addition with builtins (2)

What we get from the compiler

```
add_vec:                                # @add_vec
      addi  a4, zero, 1
      blt   a0, a4, .LBB1_3
      mv    a4, zero
.LBB1_2:                                # %for.body
      sub   a5, a0, a4
      vsetvli a6, a5, e64, m1
      slli  a7, a4, 3
      add   a5, a2, a7
      vle.v   v0, (a5)
      add   a5, a3, a7
      vle.v   v1, (a5)
      vfadd.vv    v0, v0, v1
      add   a5, a1, a7
      add   a4, a4, a6
      vse.v   v0, (a5)
      blt   a4, a0, .LBB1_2
.LBB1_3:                                # %for.end
      ret
```

Note: This code is not necessarily the best and there may be room for improvement here (WIP)

Beyond builtins: where are we now (1)

Builtins are useful for exploration and porting codes but they may be laborious to use in a codebase.

We would want the compiler to help us!

We took LLVM and made the necessary changes so the Loop Vectorizer could vectorize loops using the V-extension.

But we faced a few issues that are preventing us at the moment to fully exploit the V-extension.

- LLVM intermediate representation (IR) currently lacks the ability to represent masking other than in loads and stores
- LLVM IR cannot represent at all the concept of vector length

There is a proposal called Vector Predication now in discussion in LLVM to be able to express such concepts in the IR. The goal of the proposal is to benefit both ISAs that rely mainly on masking (AVX-512, SVE) and ISAs that rely on the vector length (Aurora SX, RISC-V V-extension).

Beyond builtins: where are we now (2)

Because of the current limitations in LLVM we can only do

- Emit two loops: a first one vectorized but working with the whole register (i.e. VL set to VLEN/SEW) and a tail/loop that is scalar ①
- Emit a single loop that relies on masking. Masking however is only used for load/store operations. ②

①
```
    vsetvli t0, zero, e64, m1 # VLMAX
    slli  t1, t0, 3
    ...
.LBB0_6: # vector loop (whole register)
    add   a5, a2, a4
    vle.v   v0, (a5)
    add   a5, a3, a4
    vle.v   v1, (a5)
    vfadd.vv    v0, v0, v1
    add   a5, a1, a4
    vse.v   v0, (a5)
    add   t2, t2, t0
    add   a4, a4, t1
    bne   t2, a7, .LBB0_6
    beqz  a6, .function-end
    # fall-through on to .LBB0_8
```

```
.LBB0_8: # scalar tail
    sub   a0, a0, a7
    slli  a4, a7, 3
    add   a1, a1, a4
    add   a3, a3, a4
    add   a2, a2, a4
.LBB0_9:
    fld   ft0, 0(a2)
    fld   ft1, 0(a3)
    fadd.d  ft0, ft0, ft1
    fsd   ft0, 0(a1)
    addi  a0, a0, -1
    addi  a1, a1, 8
    addi  a3, a3, 8
    addi  a2, a2, 8
    bnez  a0, .LBB0_9
```

②
```
.LBB0_2: # Single loop, VLMAX uses masks
    vmv.v.x     v0, t0
    vadd.vv     v0, v0, v2
    vmsleu.vv     v0, v0, v1
    add     a4, a1, a5
    vle.v     v3, (a4), v0.t
    add     a4, a2, a5
    vle.v     v4, (a4), v0.t
    vfadd.vv     v3, v3, v4 # not great
    add     a4, a0, a5
    vse.v     v3, (a4), v0.t
    add     t0, t0, a6
    add     a5, a5, a3
    bne     t0, a7, .LBB0_2
```

Other areas where the V-extension can be used

Exploration of these is still work in progress

- (Whole) Function Vectorization
  - This is useful for functions that are called from vectorized code (e.g. `#pragma omp simd`)
- Porting vector math libraries (like SLEEF) to V-extension using builtins
- Superword-level parallelism (SLP)

```
void saxpy4_ref(float * __restrict sy,
                float * __restrict sx,
                float sa) {
  sx[0] += sa*sy[0];
  sx[1] += sa*sy[1];
  sx[2] += sa*sy[2];
  sx[3] += sa*sy[3];
}
```

SLP

```
void saxpy4_vec(float * __restrict sy,
                float * __restrict sx,
                float sa) {
  __epi_2xf32 vx = __builtin_epi_vload_2xf32(sx, 4);
  __epi_2xf32 vy = __builtin_epi_vload_2xf32(sx, 4);
  vy = __builtin_epi_vfmul_2xf32(vy,
         __builtin_epi_vbroadcast_2xf32(sa, 4), 4);
  vx = __builtin_epi_vfadd_2xf32(vx, vy, 4);
  __builtin_epi_vstore_2xf32(sx, vx, 4);
}
```

Note: We need a way to ensure that 4 float elements will always be granted.

EPI tools for leveraging the V-extension

No silicon yet: what can we do?

Ok so we can use V-extension features via the builtins (and somehow via the vectorizer). But we do not have any hardware yet.

How can we assess whether the program is correct and whether we are making the most of the accelerator?

Unsurprisingly the answer is emulation.

Functional emulator: vehave

Vehave is a functional emulator, similar in spirit to the Arm Instruction Emulator for SVE, which traps on an illegal instruction, decodes it (using LLVM libraries) and emulates it.

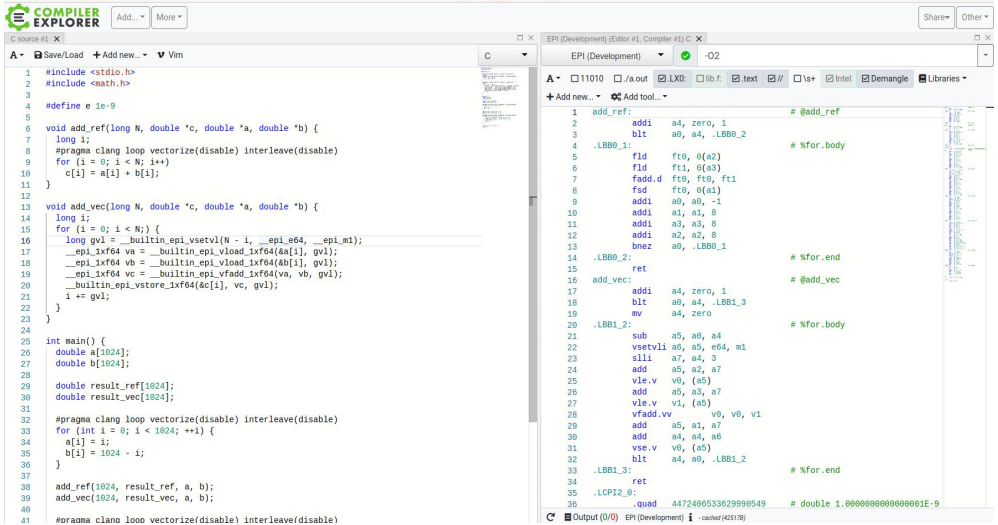
This way it allows us to run applications using V-extension instructions in a hardware running RISC-V Linux (like the HiFive Unleashed) or `qemu-system` (and even `qemu-user`).

It is not meant to be fast but allows us to validate that the compiler generates code that makes sense.

We also added tracing features to the emulator so the vector code can be analyzed, more on this later.

Compiler explorer

- We adapted Compiler Explorer (www.godbolt.org) to use our compiler and execute programs under vehave.
- This is mostly useful to showcase the builtins and evaluate the code generation of small snippets.



FPGA co-processor

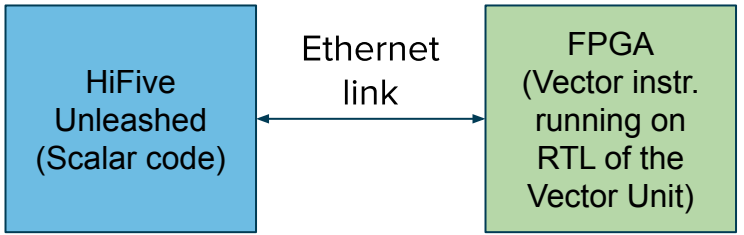
- Goal of EPI is producing an accelerator ➔ RTL development
- RTL needs to be verified
  - Functional verification ✗
  - Logical verification ✓

Goal:

- Test the software infrastructure (e.g., run binaries including the ISA vector extension)

Methodology:

- Map into FPGA the same RTL of the EPI Vector Unit as the one of the tapeout
- Run scalar code on the HiFive Unleashed board
- Offload vector instructions to the FPGA



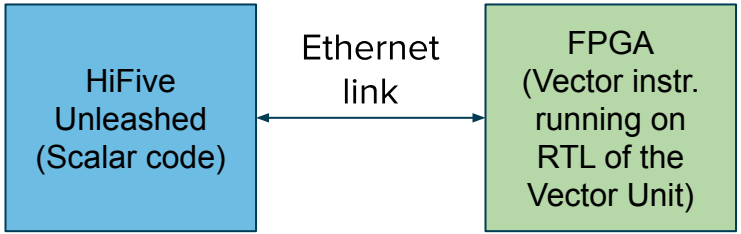
FPGA co-processor

Advantages

- Fast prototyping ➔ We can easily test new versions of the RTL in “little” time
- Local verification ➔ We can check that the Vector Unit is doing what is supposed to do
- Resource monitoring ➔ We can control FPGA resource utilization and timing
- Software enablement ➔ We can experiment with compiler and emulator
- Performance estimation ➔ We can assess the performance of HPC micro-kernels

Disadvantages

- Slow execution time
  (but always faster than pure software emulation)
- It needs vector instructions to be actually
  Supported by the RTL of the Vector Unit
- Tricky handling of memory accesses



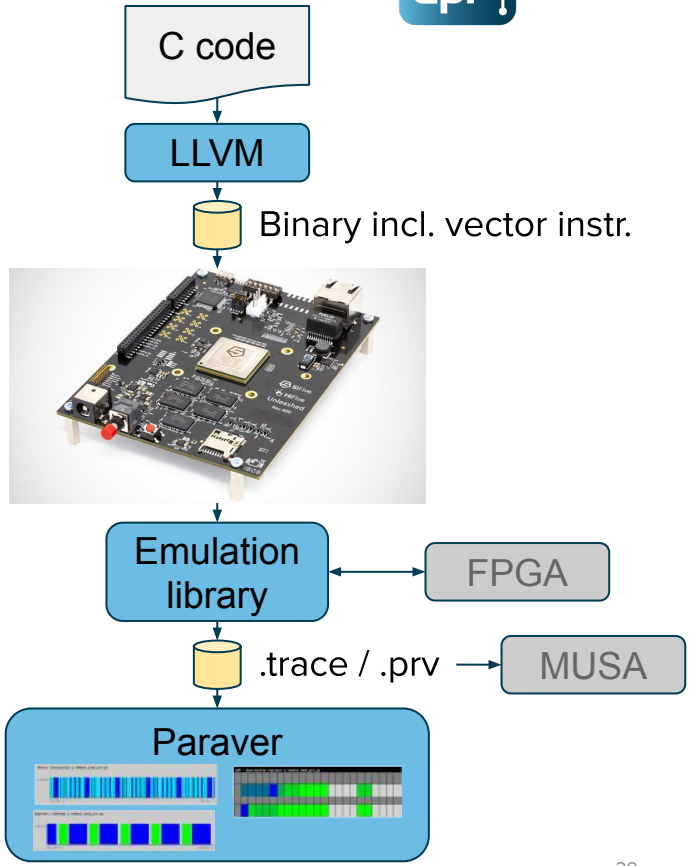
Tracing of vector code emulation

While emulating vector code
we collect several metrics into .trace files:

- Type of instruction
- Program counter value
- Registers addresses (source, destination)
- Memory addresses

.trace files are converted into .prv files
and explored with Paraver which allows to easily detect:

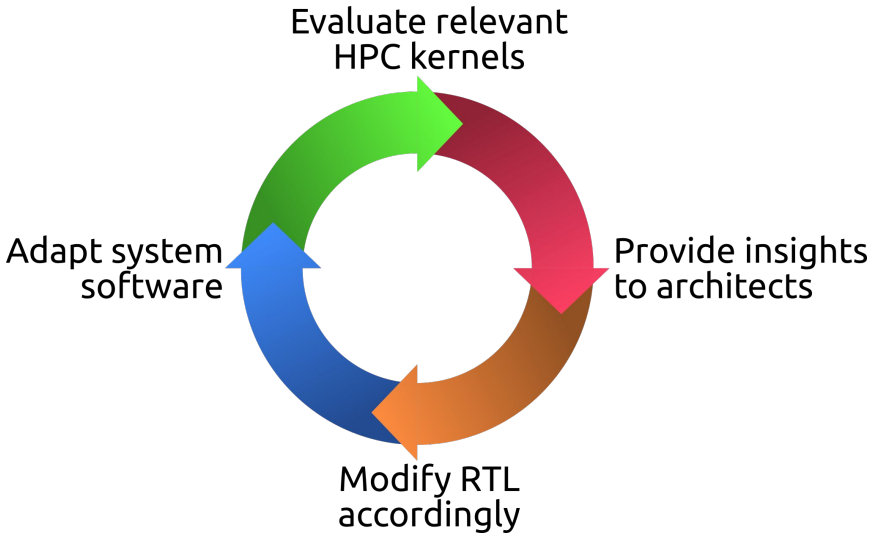
- Aggregated metrics
- Global patterns
- Microscopic behaviour



DEMO

Conclusions

- EPI develops the first RISC-V based accelerator targeting HPC leveraging the V-extension
  - RTL design of a Vector Unit
  - LLVM compiler support for the V-extension
- While RTL is becoming actual hardware, EPI develops tools for boosting the co-design cycle
  - Compiler explorer
  - Emulator of V-instructions (Vehave)
- Contact us
  - Roger Ferrer roger.ferrer@bsc.es
  - Filippo Mantovani filippo.mantovani@bsc.es



FRAMEWORK PARTNERSHIP AGREEMENT IN EUROPEAN LOW-POWER MICROPROCESSOR TECHNOLOGIES

THIS PROJECT HAS RECEIVED FUNDING FROM THE EUROPEAN UNION'S HORIZON 2020 RESEARCH AND INNOVATION PROGRAMME UNDER GRANT AGREEMENT NO 826647

EPI PARTNERS