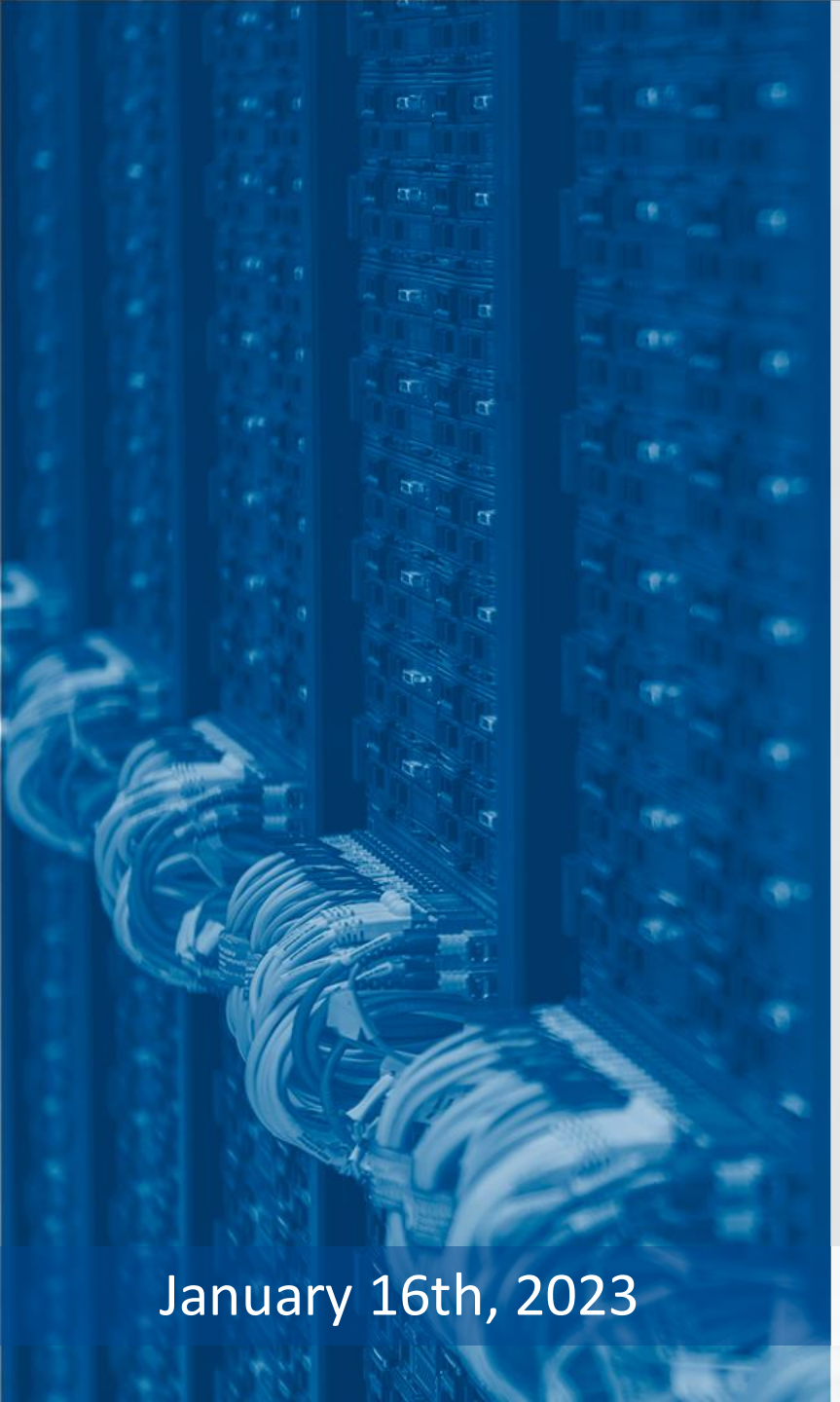


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

LLVM, the RISC-V Vector Extension and its vector length

Roger Ferrer Ibáñez

January 16th, 2023

HiPEAC 2023 - European RISC-V HPC Roadmap

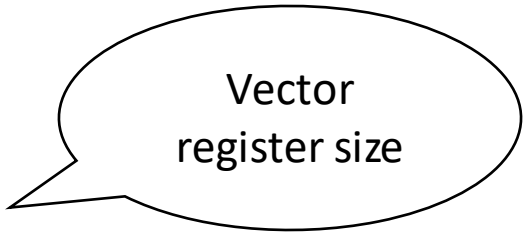
Vectors and RISC-V

- Vector/SIMD instructions can accelerate a broad range of computationally intensive applications commonly found in the HPC domain
  - Vectorization
- The RISC-V Vector Extension (RVV) provides vector computation capabilities to the RISC-V ecosystem
  - The hardware implementor can choose the vector register size (VLEN bits)
- The design is flexible enough to be useable in many scenarios
  - Compact set of instructions with wide applicability
- **Compilers should be able to make the most of this flexible design!**

Vector register size

- SIMD ISAs have linked the width of the datapath to the size of the vector
  - Typically vector instructions have operated on all the elements of the vector
- As the number of elements in a vector gets larger, it gets more challenging for the hardware
  - It may not be possible to have a datapath as wide as the vector register
- The more elements a vector can fit, the more challenging it becomes to ensure efficient utilisation in software
  - e.g., VLEN=512-bit vector registers means we can do 16 FP32 or 32 FP16/bfloat

Vectors for everyone



Name	RVV version	VLEN (bits)
Vitruvius+ (this work)	0.7.1	16384
[23] Ara	0.5	16384
[5] Xuantie910 VPU	0.7.1	256
[2] Andes NX27V	1.0	512
[40] SiFive P270	1.0rc	256
[39] SiFive X280	1.0	512

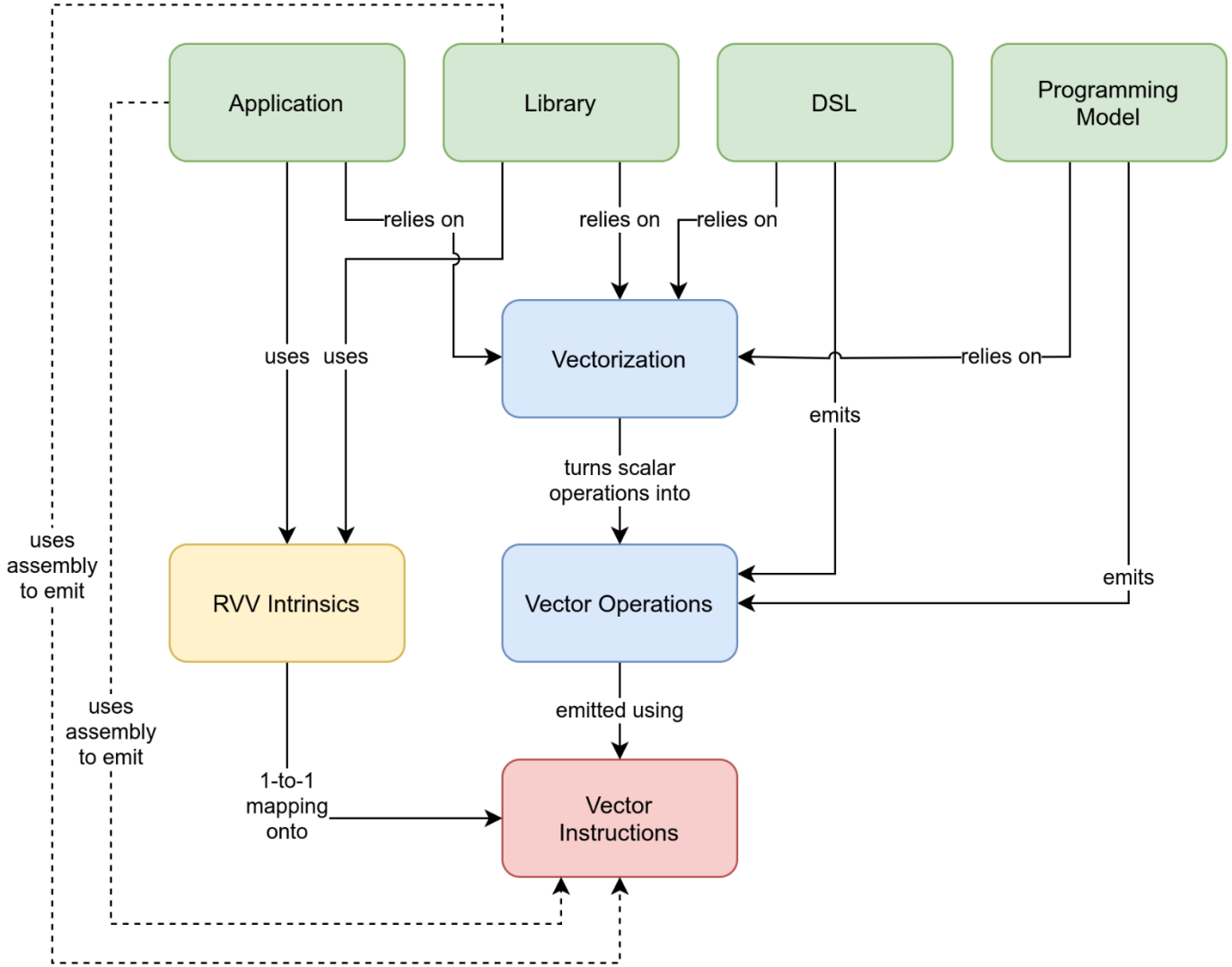
Minervini et al. “Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications” https://dl.acm.org/doi/abs/10.1145/3575861

How to use the RISC-V Vector Extension

- Compilers can help accessing the features of RVV in several levels that trade productivity with control

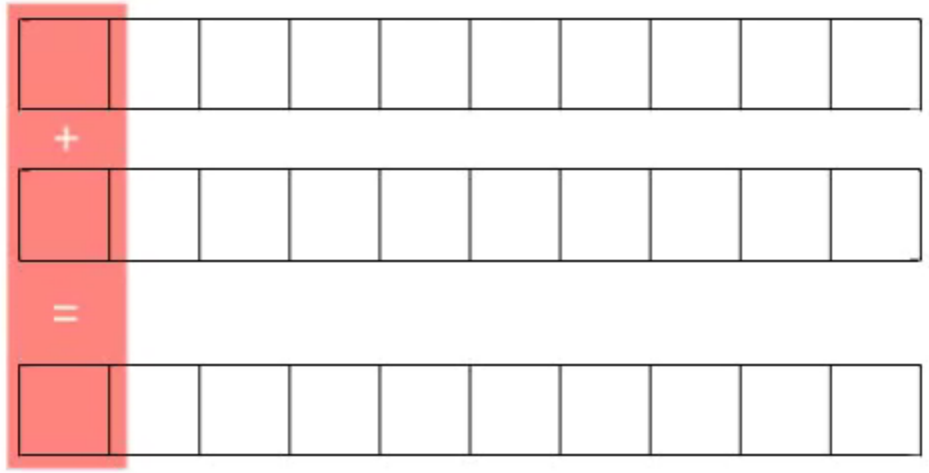
- High level abstractions (e.g., DSLs), programming models, libraries
  - Including JIT approaches
- Automatic vectorization
  - Compilers are often very conservative and will vectorize only if it is an obvious win
- Semi-automatic vectorization (e.g., #pragma omp simd)
  - “Nudge” the compiler into vectorizing
- Language intrinsics (e.g., C/C++ Builtins)
  - Last resort beyond jumping onto assembly language

The way to vector instructions



Vectorization

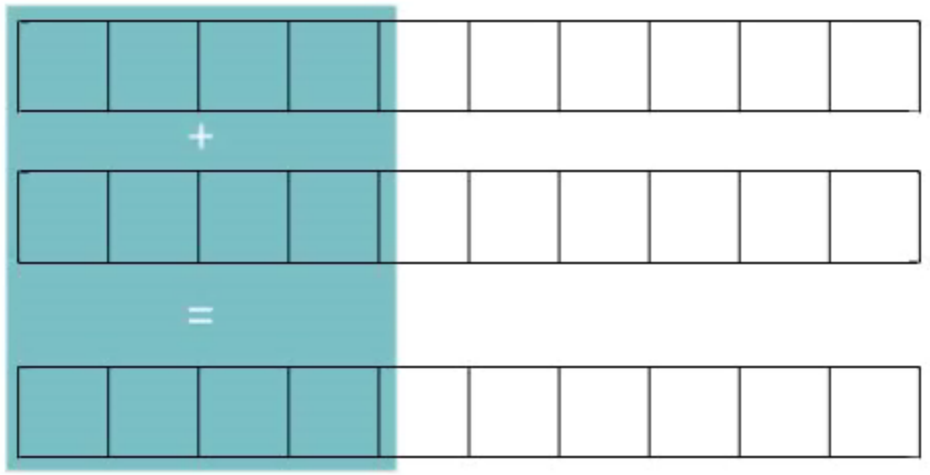
- The process where the compiler decides to use vector operations where the original code did not explicitly use them
- Most of the time vectorization is applied to loops



Typical vectorisation for SIMD (1)

- The vectorised loop is only executed if we can fill the whole vector register
  - Each original iteration corresponds to a vector element
- Otherwise, a scalar loop is used instead
  - Known as the *epilogue* as it is commonly run in the last iterations
- If we can fit many elements, say, 32 FP16 in VLEN=512 bit, it means we are giving up to 31 iterations
  - May not be an issue for loops with a large iteration count
- We can use this approach on RISC-V, the model is flexible enough
  - This is available on upstream LLVM now

Typical vectorisation for SIMD (2)



How we can vectorize a loop in RISC-V (1)

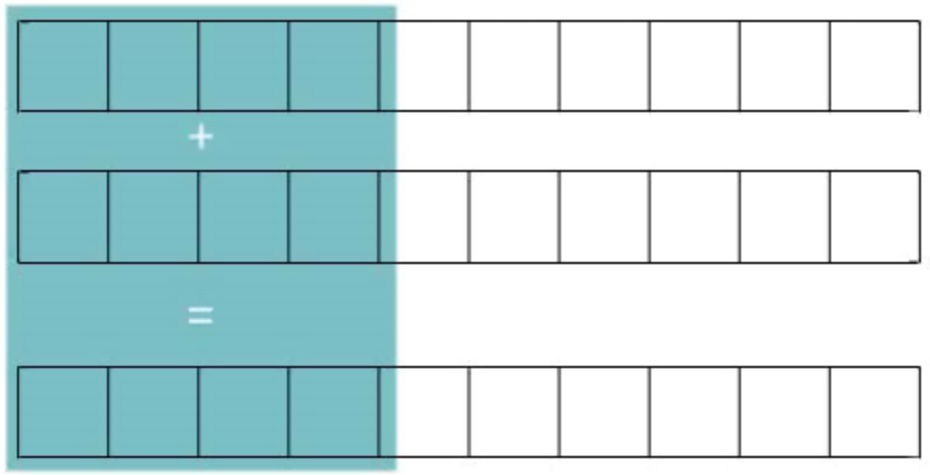
- RISC-V has the concept of vector length (`vl`), that allows us to **operate** with a part of the vector register, not necessarily all the elements
  - This includes memory accesses

- The epilogue loop now can be replaced by straight-line (i.e., not a loop anymore) code that sets `vl` and executes the remaining vector iteration

- Or we can choose to fold the epilogue into a single loop that sets the `vl` in every iteration

- At BSC, we have been implementing vector length-based vectorization in LLVM

How we can vectorize a loop on RISC-V (2)



Some more details about `vl`

- The vector length (`vl`) is part of the global state of the hardware thread
  - It is an implicit operand of the vector instructions
- Global state is difficult to handle by compilers
- For efficient code generation we had to promote it in the compiler as a regular value
  - We did the same with intrinsics
- A user of RVV intrinsics can manage the vector length like a regular (immutable) integer variable initialized with a call to a “set vector length” intrinsic
  - The program can use more than one vector length if needed

Function vectorization

- We also extended the vector-length vectorization to functions

- We build on top of `#pragma omp declare simd`
  - OpenMP SIMD support that allows declaring that vector versions of a function exists
  - The compiler can vectorize loops that calls a function with vector versions
  - The compiler can also generate the vectorized version

- The vector version of the function receives the vector length
  - This enables the creation of vector libraries that can be **vector register size agnostic**

Vector “length” agnostic

- Typically, SIMD ISAs have prescribed a vector register size
- Arm introduced the concept of “vector length agnostic” (VLA) with their Scalable Vector Extension
  - (In the context of this talk VLA means “**vector register size** agnostic”)
  - The opposite has been dubbed “vector length specific” (VLS) i.e., the software assumes a size of the physical vector register size
- RVV supports vector register size agnostic programming
  - Same vectorized code can run correctly in different implementations of RVV unchanged (regardless of VLEN, the vector register size)

Non-loop vectorization

- Super-level Word Parallelism (SLP) is a mechanism that allows to vectorise code that has been unrolled/replicated

- Thanks to vector length, we are not limited to specific groups or sizes, specially if the compiler knows the vector register size
  - If not known (for vector register size agnostic code generation) a runtime check can be used plus a fallback to the original code

Vector length and BSC projects

- At BSC, we chose to use vector register size agnostic code generation
  - It comes at a price because we are explicitly hiding information to the compiler
- We have several RVV projects with different VLENs at BSC
  - The code is portable among them
- It is easy to create vectors that are even too long for LLVM itself
  - The code generation (backend) type system of LLVM structure has a number of limits
- The possibility of using vector register size fixed code generation is not lost
  - LLVM has this capability and it can be used for RISC-V

In summary

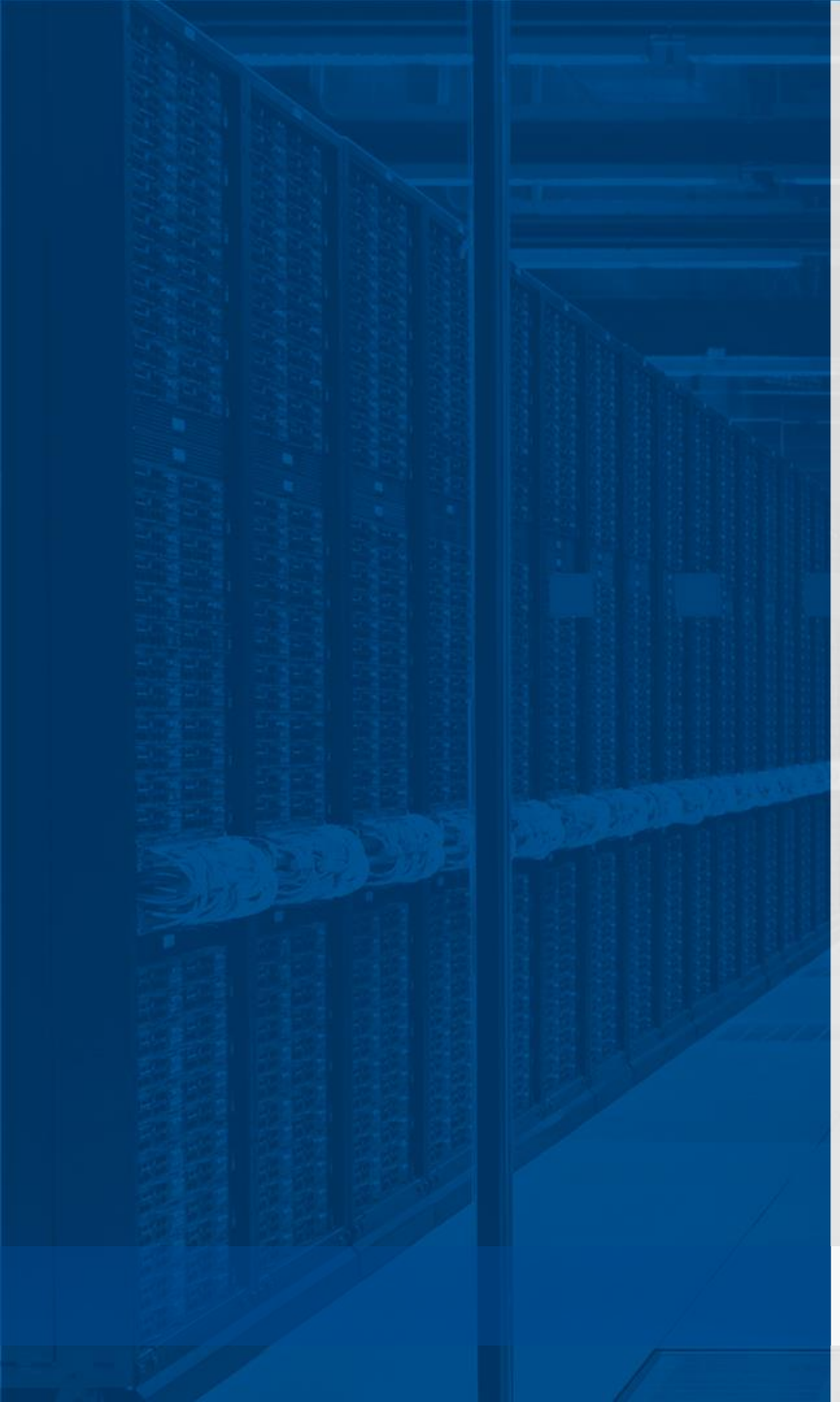
- RISC-V Vector Extension is flexible thanks to its vector length feature

- It was a bit of a challenge for the compiler to be able to use this feature
  - Even if maximum throughput is going to be achieved with the largest vector length possible by the hardware, **no code that is worth vectorizing should be left behind!**

- Enables vector register size agnostic vectorization which is used by different groups at BSC
  - Different vector hardware implementations
  - Algorithms and application development

…one last thing

- It may look like we have solved all our problems in vectorization on RISC-V when using the vector length
  - Not really!
- For instance
  - Can instruction scheduling be done efficiently in the context of the BSC VPU, which features long vectors and a decoupled design
  - Can register allocation be made vector length aware? (i.e., copies and spill code of vector registers)
  - Can we “revectorize” (using a JIT) VLA code so it becomes VLS and then optimise it again?
- If you like compilers and you are up for a challenge then, talk to me! ☺



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Questions?

The European Processor Initiative (EPI) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement EPI-SGA1: 826647 and under EPI-SGA2: 101036168. Please see http://www.european-processor-initiative.eu for more information.

The European PILOT project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No.101034126. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Italy, Switzerland, Germany, France, Greece, Sweden, Croatia and Turkey.

The MEEP project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 946002. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Croatia, Turkey

roger.ferrer@bsc.es