Powering Plasma-Physics with RISC-V vector extension: the case of Vlasiator

Barcelona Supercomputing Center
Centro Nacional de Supercomputación


Gerard Oliva (corresponding author)
gerard.oliva@bsc.es


Pablo Vizcaino
pablo.vizcaino@bsc.es


Urs Ganse
urs.ganse@helsinki.fi


Filippo Mantovani
filippo.mantovani@bsc.es


Marta garcia-Gasulla
marta.garcia@bsc.es

European Processor Initiative to deliver a RISC-V based vector accelerator.
We provide a HW and SW ecosystem to enable co-design before the chip arrives.

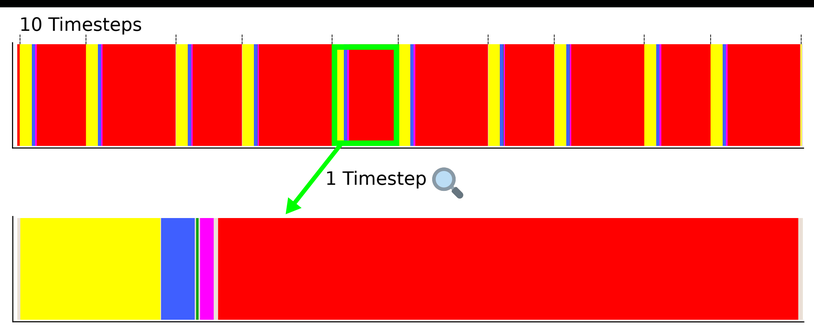
Software Development Vehicles to Enable Extended and Early Co-design: A RISC-V and HPC Case of Study DOI: 10.1007/978-3-031-40843-4_39

We also develop an evaluation methodology to study potential optimizations.
Our optimization techniques are CPU-agnostic and provide benefits in other architectures.

Short reasons for long vectors in HPC CPUs: a study based on RISC-V
DOI: 10.1145/3624062.3624231

Vlasiator is a novel, computationally intensive simulation that models ions differently for more accurate, noiseless space weather predictions. Space weather describes the Sun's variable effects on near-Earth space via solar wind, impacting technology like satellites and power grids.

Code structure



Phases
- compute_filtered_face_values_derivates
- filter_pqm_monotonicity_Init
- filter_pqm_monotonicity_Conditional
- compute_pqm_coeff_finalization
- propagate_whileInit_and_whileLoop

Mini-app of Vlasiator (subset of full app)
Iterative structure divided into timesteps
All timesteps have the same structure
Dominant phase is

Vector class

Templated C++ class
Defines common vector operations
Vector length is always known at compile time
Vlasiator code heavily depends on Vec

Vec4Simple

VecX<8>

VecX<16>

Vector length is no longer limited by the implementation

Original: Written for fixed-length vectors

```
template <class T>
static inline Vec4Simple<T> operator +
  (const Vec4Simple<T> &l, const Vec4Simple<T> &r){
return Vec4Simple<T>(
  l.val[0]+r.val[0],
  l.val[1]+r.val[1],
  l.val[2]+r.val[2],
  l.val[3]+r.val[3]);
 }
```

Extended version: Vector-length agnostic

```
template <size_t N, class T>
static inline VecX<N, T> operator +
  (const VecX<N, T> &l, const VecX<N, T> &r){
  VecX<N, T> vec;
  PRAGMAS
  for(unsigned long int i=0; i<l.get_size(); ++i){
    vec.val[i] = l.val[i] + r.val[i];
  }
  return vec;
}
```

2 Explicit loops

What the programmer codes

```
Vec foo(Vec a, Vec b, Vec c){
  Vec res1,res,res3;
  res1 = a+b;
  res2 = res1*0.5;
  res3 = res2-c;
  return res3;
}
```

What the compiler sees (overloaded operators)

```
Vec foo(Vec a, Vec b, Vec c){
  Vec res1,res,res3;
  for(int i=0;i<VECL;++i) res1[i]=a[i]+b[i];
  for(int i=0;i<VECL;++i) res2[i]=res1[i]*0.5;
  for(int i=0;i<VECL;++i) res3[i]=res2[i]-c[i];
  return res3;
}
```

```
vector load a
vector load b
vector add
vector store res1
```
```
vector load res1
vector mul.f
vector store res2
```
```
vector load res2
vector load c
vector sub
vector store res3
```

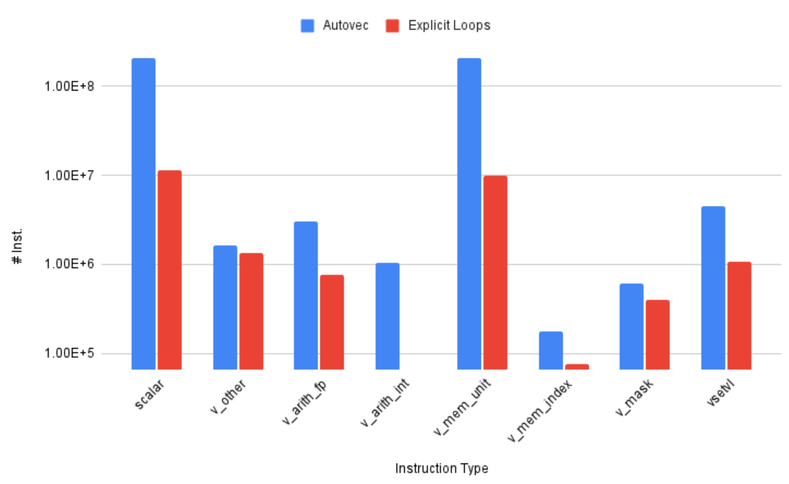
8 memory instr
3 arithmetic instr

Better approach: Explicit loops

```
Vec foo(Vec a, Vec b, Vec c){
  Vec res1,res,res3;
  for(int i=0;i<VECL;++i){
    res1[i] = a[i]+b[i];
    res2[i] = res1[i]*0.5;
    res3[i] = res2[i]-c;
  return res3;
}
```

```
vector load a
vector load b
vector add
vector mul.f
vector load c
vector sub
vector store res3
```

4 memory instr
3 arithmetic instr



1 Increase Average Vector Length

Metric definition

The average vector length (AVL) is a metric representing the mean number of data elements operated on per vector instruction during execution.

Compile time parameter VECL
Affects: Problem size
VECL <= 64

} Proposal: Increase VECL to increment problem size and leverage long vectors

VECL=8

	vadd	vfadd	vfmul	vle	vse	vfsub	vmflt
Avg. VL	8	8	8	8	8	8	8
#Instr.	1	2	4	21	14	2	3

VECL=16

	vadd	vfadd	vfmul	vle	vse	vfsub	vmflt
Avg. VL	16	16	16	16	16	16	16
#Instr.	1	2	4	21	14	2	3

VECL=32

	vadd	vfadd	vfmul	vle	vse	vfsub	vmflt
Avg. VL	32	32	32	32	32	32	32
#Instr.	1	2	4	21	14	2	3

VECL=64

	vadd	vfadd	vfmul	vle	vse	vfsub	vmflt
Avg. VL	64	64	64	64	64	64	64
#Instr.	1	2	4	21	14	2	3

3 Increase in Vector Instruction Mix

Metric definition

Calculated as the ratio of vector instructions to total instructions, the vector instruction mix is a value ranging from 0 to 1 representing the extent to which code has been vectorized.

Code analysis

```
inline Vec [add/mul/sub/...](Vec op1, Vec op2)

Vec R1 = add_vector(A,C);
Vec R2 = mul_vector(A,B);
Vec R3 = sub_vector(C,R2);
```

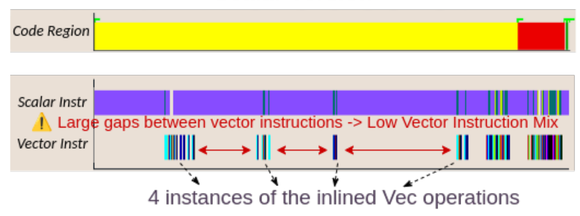
Arguments passed by value
Functions are called back-to-back

} Each function call requires copying arguments

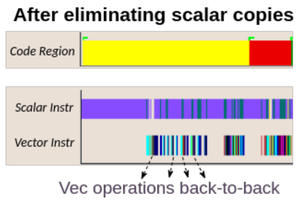
Proposal: Pass arguments by reference

```
inline Vec [add/mul/sub/...](Vec & op1, Vec & op2)
```

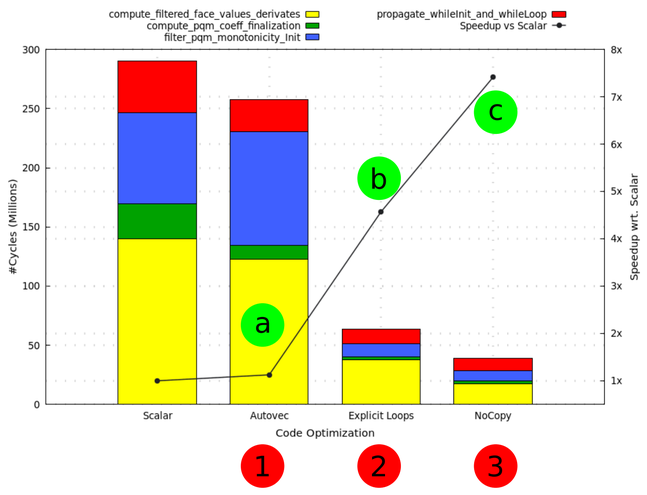
Scalar Version

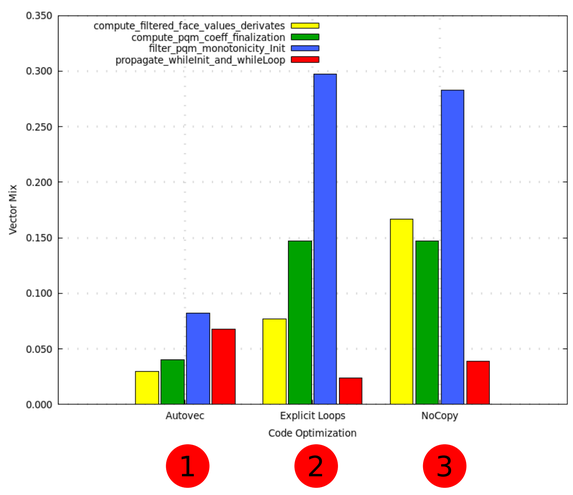


After eliminating scalar copies



Conclusions and next steps





a Auto-vectorization alone does not provide significant speedup

b Adding explicit loops achieves a speedup of slightly more than 4x

c Eliminating scalar copies (in addition to explicit loops) increases the speedup to slightly more than 7x

These speedups were achieved using a vector length of 64 (VECL = 64)
Limiting factor: allow VECL greater than 64
Possible benefit: Avg. VL up to 256

Further analysis of Vector Mix:
- Eliminate non-computing instructions like mask creation and data shuffling
- Compile and run/emulate with rvv1.0 vector specification

Run the full Vlasiator application to study the effect of optimizations derived from the mini-app

Evaluate portability of code changes to other CPU architectures