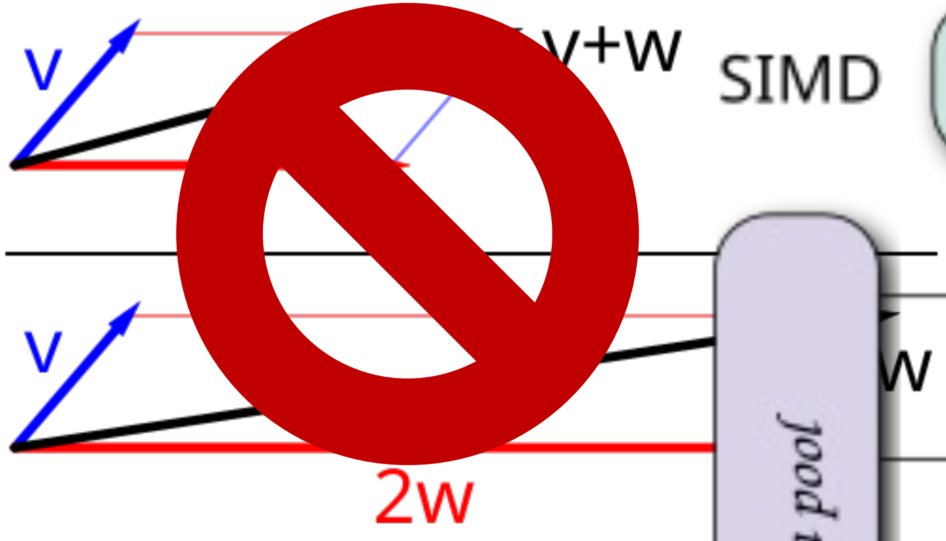
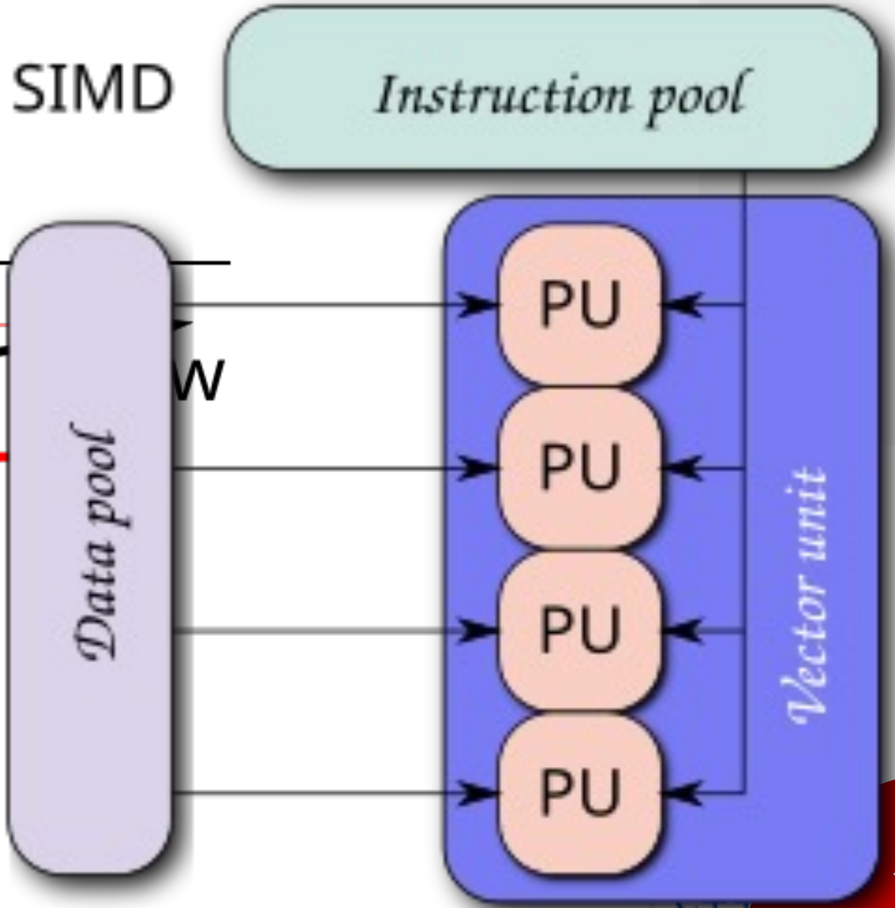
Unlocking Portable and Performant Vector Programming with chpl Vector Library

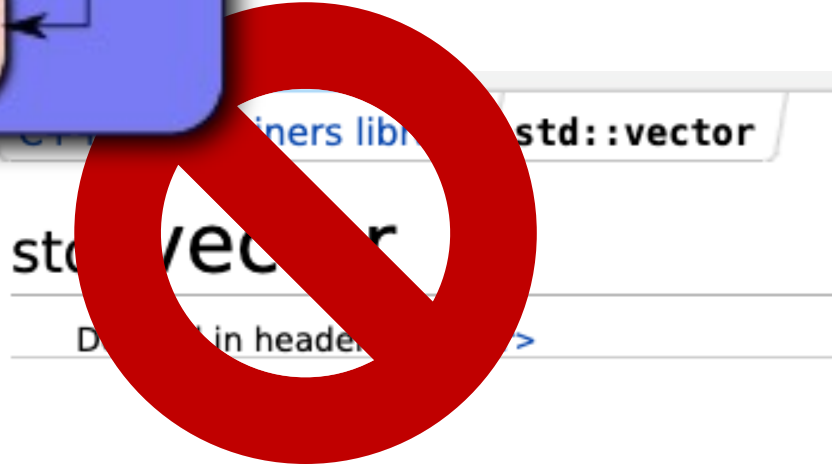
Jade Abraham

October 10, 2025

Vector?


v
v
w
2w


SIMD
Instruction pool
Data pool
PU
PU
PU
PU
Vector unit


std::vector

Vector Programming

- Vector programming is using SIMD execution units to process data in parallel within a single thread
  - This is instruction level parallelism
  - Why? More parallelism = more speed!

- For many applications, you don’t have to explicitly use it or even know about it
  - Compilers are awesome!
  - Yay free performance!

- So we can end the talk here?

Thank you!

Vector Programming

- Vector programming is making using SIMD execution units to process data in parallel within a single thread
  - This is instruction level parallelism
  - Why? More parallelism = more speed!

- For many applications, you don’t have to explicitly use it or even know about it
  - Compilers are awesome!
  - Yay free performance!

- ~~So we can end the talk here?~~

- What happens when the compiler can’t do it for us?
  - The compiler may not know its safe or know how to make it safe (floating point error is a pain)
  - Did we write our code in an easy-to-read way for humans, but bad for SIMD?
  - ...et cetera...

Vector Programming - Chapel’s missing piece

- Chapel’s *multiresolution philosophy*
  - Both high- and low-level features
  - The high-level features are implemented in terms of the low-level features
- This works great for multi-core/distributed parallelism

```
forall a in Arr {
  // ...something interesting
}
```

```
coforall l in Arr.targetLocales() do on l do
  coforall t in 0..#here.maxTaskPar {
    const mySlice: range(?) = ...
    for i in mySlice {
      ref a = Arr[i];
      // ...something interesting
    }
  }
```

- What about instruction-level parallelism?

```
foreach a in Arr {
  // ...something interesting
}
```

What’s my goal?

- I want to write explicit vector code…
  - …without calling C/assembly
  - …that is portable across architectures
  - …that works orthogonally with existing Chapel features
  - …that is fast

- I would like my code…
  - …to not be a maintenance nightmare
  - …to look nice

Introducing CVL

CVL – chpl Vector Library

- Provides a new portable 'vector' type which matches a hardware vector register
  - Supports 128-bit and 256-bit vectors with 'int(?w)' and 'real(?w)'
  - Currently supports x86 SSE/AVX and Arm Neon
- Supports many common vector operations
  - Basic math, bit manipulation, and comparisons
  - Memory operations (load/store, limited support for 'gather' and load/store masks)
  - Shuffles/permutations/blends
  - Trigonometry (via Sleef - https://github.com/shibatch/sleef)
- Integrates seamlessly with Chapel
  - Works with many Chapel container types (arrays, c_ptr, tuples, and bytes)
  - Works with parallel and distributed code
  - Everything is written in pure-ish Chapel

- Open source: https://github.com/jabraham17/cvl

Examples, please?

The “Hello World” of HPC/Vector programming

```
proc stream(a: real, x: [?D] real, y: x.type, ref z: x.type) {
  forall i in D {
    z[i] = a * x[i] + y[i];
  }
}

use CVL;

proc streamWithCVL(a: real, x: [?D] real, y: x.type, ref z: x.type) {
  type vec = vector(real, 4);

  const av = a: vec;
  forall i in D by vec.numElts {
    const xv = vec.load(x, i);
    const yv = vec.load(y, i);
    const zv = av * xv + yv;
    zv.store(z, i);
  }
}
```

Specify the size of the vector,
It must match a hardware type!

Create a vector from ‘a’

Adjust the iteration to be every 4th index

Load/store the memory

The "Hello World" of HPC/Vector programming

```
use CVL;

proc streamWithCVL(a: real, x: [?D] real, y: x.type, ref z: x.type) {
  type vec = vector(real, 4);

  const av = a: vec;
  forall i in D by vec.numElts {
    const xv = vec.load(x, i);
    const yv = vec.load(y, i);
    const zv = av * xv + yv;
    zv.store(z, i);
  }
}
```

Specify the size of the vector,
It must match a hardware type!

Create a vector from 'a'

Adjust the iteration to be every 4th index

Load/store the memory

- Explicit vector operations that are distributed and parallel!
- But it is overly verbose, hiding the actual computation
- We can do better

The “Hello World” of HPC/Vector programming

```
use CVL;

proc streamWithCVL(a: real, x: [?D] real, y: x.type, ref z: x.type) {
  type vec = vector(real, 4);

  forall (zv, xv, yv) in zip(vec.vectorsRef(z),
                             vec.vectors(x), vec.vectors(y)) {
    zv = a * xv + yv;
  }
}
```

The scalar is automatically made into a vector

The iterators handle all the load/store logic for us

```
proc stream(a: real, x: [?D] real, y: x.type, ref z: x.type) {
  forall i in D {
    z[i] = a * x[i] + y[i];
  }
}
```

The "Hello World" of HPC/Vector programming

- Is the CVL version faster/better than the plain Chapel version?
  - Default Rectangular arrays: identical performance
  - Block distributed: CVL is ~2x slower
  - Block Cyclic distributed: CVL is A LOT slower

- The gap is likely Chapel specific optimizations that explicit SIMD thwarts

- Just because you can, doesn't mean you should

Something harder?

Kmeans Clustering

```
for cIdx in centroids.D {
  const cX = centroids.x[cIdx], cY = centroids.y[cIdx];
  forall pIdx in points.D with (ref points) {
    const dist = distance(points, pIdx, centroids, cIdx);
    if dist < points.minDist[pIdx] {
      points.minDist[pIdx] = dist;
      points.clusterId[pIdx] = cIdx;
    }
  }
```

Compute the distance

Conditionally update the minimum

```
for cIdx in centroids.D {
  const cIdxVec = new VT_IDX(cIdx);
  const cVecX = new VT(centroids.x[cIdx]), cVecY = new VT(centroids.y[cIdx]);
  forall pIdx in VT.indices(points.D) with (ref points) {
    const dist = distance(VT, points, pIdx, cVecX, cVecY);
    const minDist = VT.load(points.minDist, pIdx);
    const oldClusterId = VT_IDX.load(points.clusterId, pIdx);

    const mask = dist < minDist;
    var newMinDist = bitSelect(mask, dist, minDist);
    var newClusterId = bitSelect(mask.transmute(VT_IDX), cIdxVec, oldClusterId);

    newMinDist.store(points.minDist, pIdx);
    newClusterId.store(points.clusterId, pIdx);
  }
```

Compute the distance

Determine which value to use

Always update the minimum

Kmeans Clustering

- Is the CVL version faster/better than the plain Chapel version?
  - At small problem sizes they are the same
  - At big problem sizes CVL beats plain Chapel

	1 million points	10 million points	100 million points
Chapel	0.413s	8.723s	78.106s
Chapel + CVL	0.346s	3.004s	64.306s

- What's the catch?

```
record pointsList {
  type T;
  const D: domain(1);
  var x: [D] T;
  var y: [D] T;
  var clusterId: [D] int;
  var minDist: [D] T;
}
```

```
record pointsList {
  type T;
  const D: domain(1);
  var xy: [D] point(T);
  var clusterId: [D] int;
  var minDist: [D] T;
}
record point {
  type T;
  var x: T;
  var y: T;
}
```

- If I use the wrong data structure
  - The plain Chapel code is slower
  - It is much harder to hand vectorize

How does it work?

A brief dive into the implementation

- The top-level 'vector' type is implemented by multiple layers of type abstractions
  - 'vector(eltType, numElts)' constructs an 'implType(eltType, numElts)'
  - 'implType' is implemented for each architecture/bit-width as a type-only type

- Each 'implType' has a set of operations and behaviors it must conform to
  - If the underlying hardware has a different behavior, shuffle the vector to match (e.g. pairwise adds)
  - Arbitrary shuffles/permutations/blends are not permitted

- At the lowest level, each operation on 'implType' is either
  - directly calling a compiler intrinsic
  - calling a C wrapper around a compiler intrinsic

- 'implType' is a fantastic example of Chapel metaprogramming
  - Compile-time dispatch greatly reduces boilerplate
  - Everything is done at compile-time, all you are left with in the generated code are the vector operations

How does it compare?

Who does vectorization the best?

- Nbody (50,000,000 iterations) from the Computer Language Benchmark Game

	M1 Arm (8 cores)	Intel Xeon E5-2690 v3 (24 cores)	AMD EPYC 7662 (128 cores)
Chapel	1.330s	3.490s	2.731s
Chapel + CVL	1.626s	2.621s	2.434s
Chapel + CVL (fma)	1.511s	2.437s	2.378s
C	2.730s	5.940s	4.150s
C (x86 Intrinsics)	N/A	1.911s	2.648s
Fortran	2.444s	4.025s	3.930s
Rust	1.449s	3.333s	3.268s

Handcoded C is fast, but not portably fast

- Chapel! (kinda)

Is vector code faster?

- RGB -> Grayscale using integers (problem size scaled per platform)

	M1 Arm (8 cores)	Intel Xeon E5-2690 v3 (24 cores)	AMD EPYC 7662 (128 cores)
Chapel	1.009	6.505	1.524
Chapel + CVL	0.247	0.847	0.349

4x-8x improvements!

- RGB -> Grayscale using floating point (problem size scaled per platform)

	M1 Arm (8 cores)	Intel Xeon E5-2690 v3 (24 cores)	AMD EPYC 7662 (128 cores)
Chapel	1.024	8.760	1.700
Chapel + CVL	0.242	0.845	0.337

4x-10x improvements!

- Yes!

Conclusion

- CVL lets programmers fill a missing gap in Chapel's parallel story
  - Portable, performant, and pretty vector code
- CVL is ready for use!
  - https://github.com/jabraham17/cvl
- CVL is not a silver bullet for performance in Chapel, but it is another tool in the toolbox

- What's next?
  - Expanded 'vectorsRef( )' support
  - Find a nice ergonomic story for tail loops
  - Leverage the Chapel compiler for more flexible shuffles
  - Even tighter integration with Chapel arrays
    - Close the distributed array performance gap
    - Support 2D arrays without 'reshape( )'

Thank you!