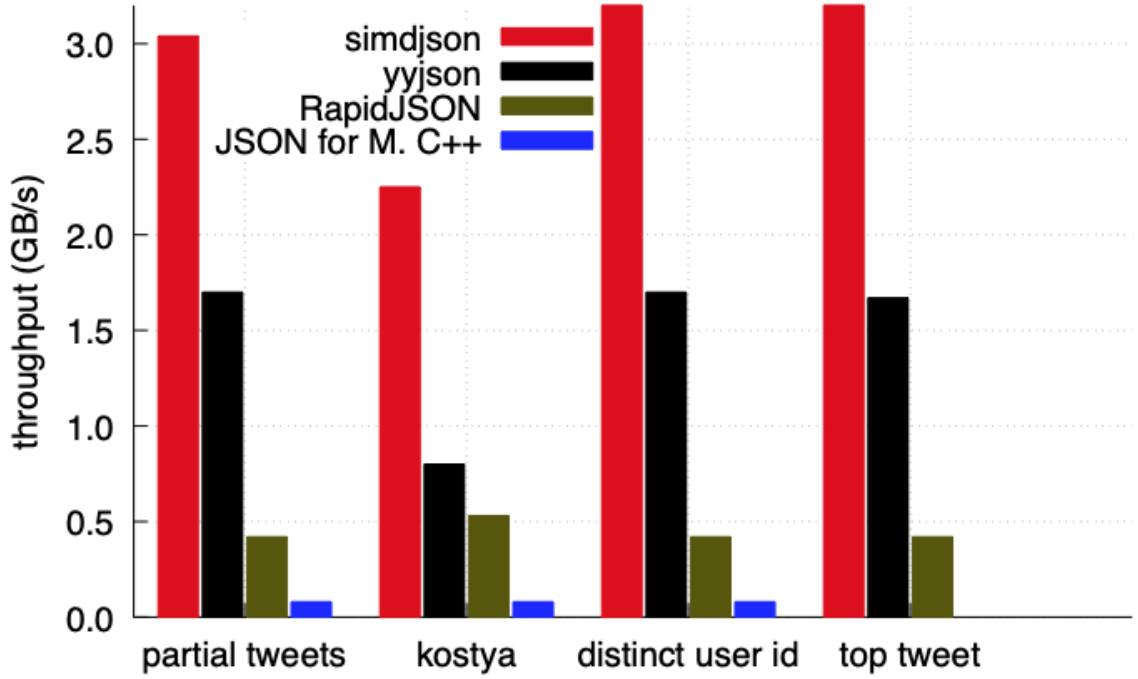
Portable and Performant Explicit Vector Programming

Jade Abraham (@jabraham17), Advanced Programming Team, HPE

PNW PLSE 2026
May 5, 2026

Why Vector Programming

- Compilers are great at optimizing code, but not perfect
  - Especially when it comes to vectorization
- Turning scalar code into vectorized often requires more imagination and knowledge of the domain space than compilers have



https://github.com/simdjson/simdjson#performance-results

Portable Vector Programming

- Better vectorization performance is often locked behind portability issues
  - What vector architecture? SSE or AVX or Neon or SVE or ...
  - What vector lane sizes are available?

- “I just want to write portable, fast, high-level code!”

```
1 void saxpy(float a, const float* __restrict__ x, const float* __restrict__ y,
2              float* __restrict__ z, std::size_t n) {
3     for (std::size_t i = 0; i < n; i++) {
4         z[i] = a * x[i] + y[i];
5     }
6 }
```

Sugar for Portable Operations: Highway and C++ 26

```cpp
#include <hwy/highway.h>
namespace hn = hwy::HWY_NAMESPACE;

void saxpy(float a, const float* HWY_RESTRICT x, const float* HWY_RESTRICT y,
           float* HWY_RESTRICT z, std::size_t n) {
  const hn::FixedTag<float, 8> d;
  const auto as = hn::Set(d, a);
  for (std::size_t i = 0; i < n; i += hn::Lanes(d)) {
    const auto xs = hn::Load(d, x + i);
    const auto ys = hn::Load(d, y + i);
    const auto zs = as * xs + ys;
    hn::Store(zs, d, z + i);
  }
}
```

```cpp
#include <simd>
void saxpy(float a, const float* __restrict__ x, const float* __restrict__ y,
           float* __restrict__ z, std::size_t n) {
  for (std::size_t i = 0; i < n; i += std::simd::vec<float, 8>::size()) {
    const auto xs = std::simd::unchecked_load(x + i, std::simd::vec<float, 8>::size());
    const auto ys = std::simd::unchecked_load(y + i, std::simd::vec<float, 8>::size());
    const auto zs = a * xs + ys;
    std::simd::unchecked_store(zs, z + i, std::simd::vec<float, 8>::size());
  }
}
```

Many frameworks, one common pattern

```
for each index, offset by lane width
    load x
    load y
    z = a * x + y
    store z
```

My solution for Chapel: chpl Vector Library

```
1  use CVL;
2  proc saxpy(a: real(32), x: [?D] real(32), y: [D] real(32), ref z: [D] real(32)) {
3    type vec = vector(real(32), 8);
4    forall (xs, ys, zs) in zip(vec.vectors(x), vec.vectors(y), vec.vectorsRef(z)) {
5      zs = a * xs + ys;
6    }
7  }
```

```
1  proc saxpy(a: real(32), x: [?D] real(32), y: [D] real(32), ref z: [D] real(32)) {
2    forall (xs, ys, zs) in zip(x, y, z) {
3      zs = a * xs + ys;
4    }
5  }
```

Let the computation shine through!

Design of CVL

I want to write explicit vector code in Chapel...

- ...without calling C/assembly
- ...that is portable across architectures
- ...that works orthogonally with existing Chapel features
- ...that is fast

I would like my implementation...

- ...to not be a maintenance nightmare
- ...to look nice

Design of CVL: Iteration

In Chapel, it's common to see two kinds of loops with arrays

- Iterating over the indices of an array

```
forall idx in myDomain {
  const myElt = myArray[idx];
  …
}
```

```
forall idx in vectorType.indices(myDomain) {
  const myElt = vectorType.load(myArray, idx);
  …
}
```

- Iterating over the array itself

```
forall myElt in myArray {
  …
}
```

```
forall myElt in vectorType.vectors(myArray) {
  …
}
```

Design of CVL: Portability

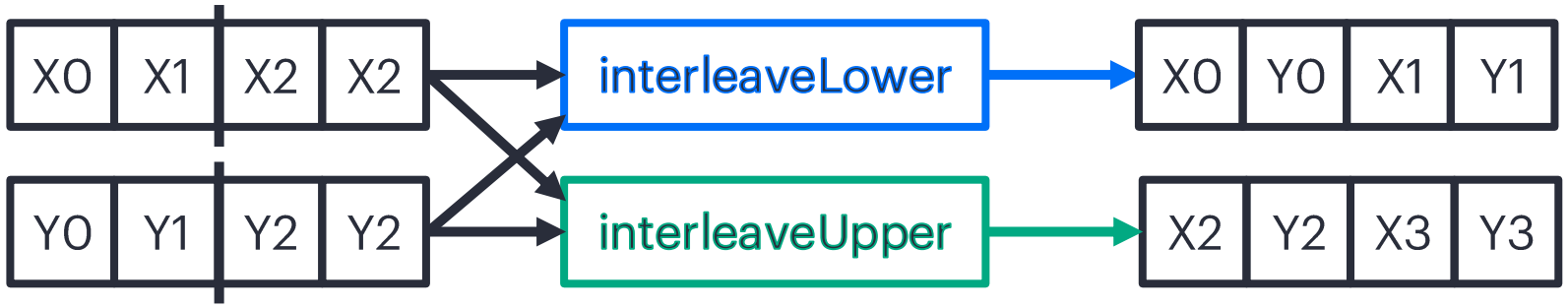
Vector types map to real hardware types where possible

```
type vec = vector(real(32), 8);
```

- On x86, maps to a 256-bit register
- On Arm-Neon, maps to two 128-bit registers

CVL provides a common set of shuffles instead of arbitrary shuffles

- swapPairs, blendLowHigh, interleaveLower, deinterleaveLower, etc



Design of CVL: Parallel and Distributed

Chapel's biggest strength is its native support for writing parallel and distributed code

- CVL has built-in serial, parallel, and distributed iterators to support this

Serial (order-independent)

```
foreach myElt in vectorType.vectors(myArray) {
  ...
}
```

Parallel

```
forall myElt in vectorType.vectors(myArray) {
  ...
}
```

Parallel and Distributed

```
forall myElt in vectorType.vectors(myDistributedArray) {
  ...
}
```

One big example: k-means clustering

```
type VT = vector(points.T, 4);
type VT_IDX = vector(int, 4);
```

```
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);
}
```

Using vector indices for many different vectors

Branch-less update for the minimum distance

```
inline proc distance(type VT, const ref p1, i1: int, const ref p2X, const ref p2Y): VT {
  const p1X = VT.load(p1.x, i1);
  const p1Y = VT.load(p1.y, i1);

  const p1XDiff = p1X - p2X;
  const p1YDiff = p1Y - p2Y;

  return sqrt(p1XDiff * p1XDiff + p1YDiff * p1YDiff);
}
```

Reusable vector code

Fully vectorized math

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

Conclusion

CVL provides good abstractions over the hardware

- Maintains a good balance of portability and performance
- Computation is expressed directly, not hidden by bookkeeping

Available on GitHub: https://github.com/jabraham17/cvl

Future work

- Find a nice ergonomic story for tail loops
- Support more architectures and vector sizes
  - i.e., SVE and AVX-512