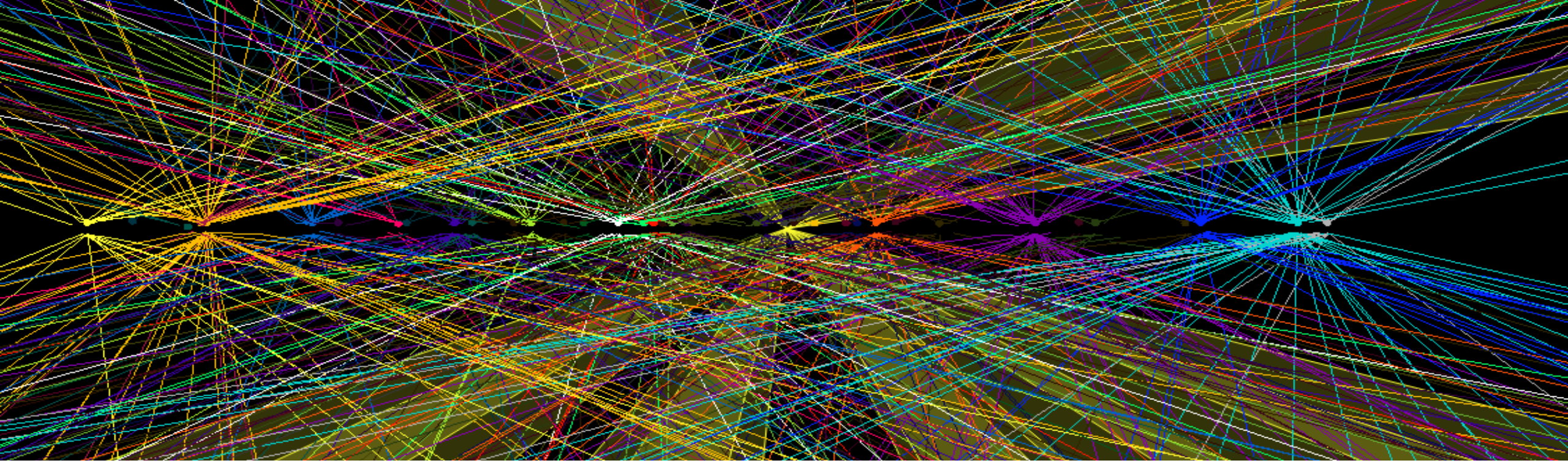


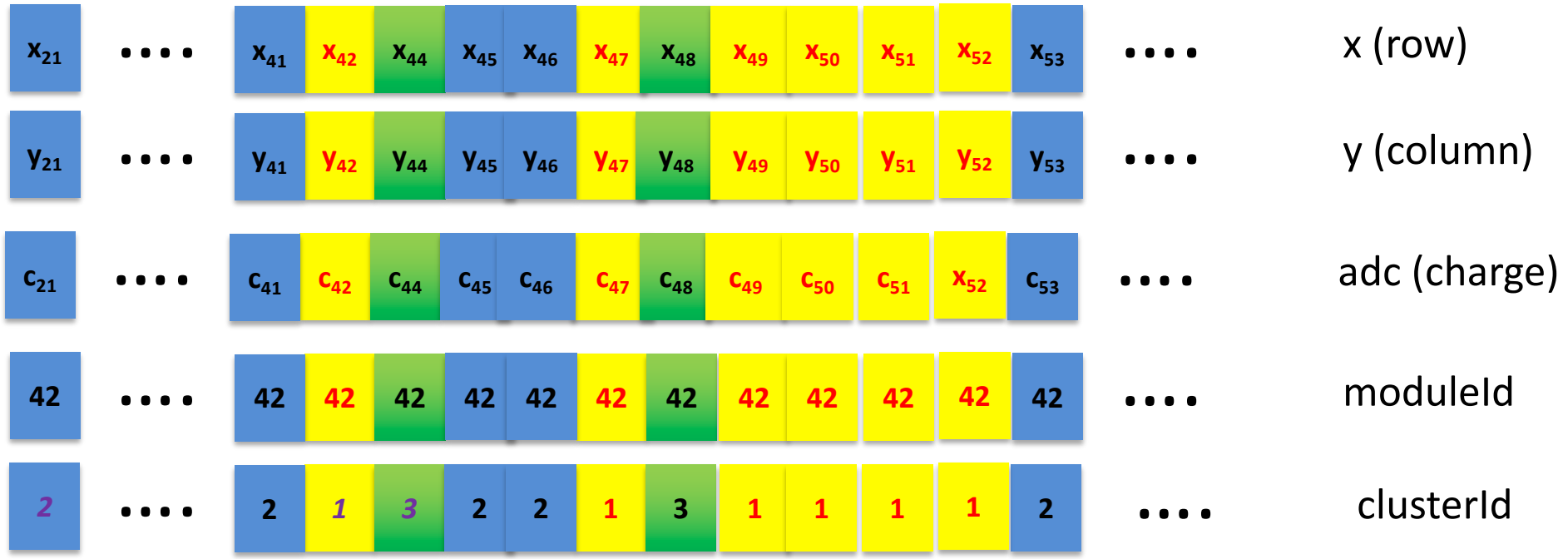
SoA model for Pixel Reconstruction (and bejond?)

Few Parallel Axioms on filling data structures

- Never Resize:
  - either size is known in advance
  - or fixed to some Max
  - or: first count, then fill
- Never delete: prefer masking
- Prefer *many-to-one* to *one-to-many* references
- Avoid sorting: is highly parallel unfriendly
  - Try to get it right from start!

- Atomic increment at SIMD/SIMT level is cheap:
  - Once per vector/warp (cuda: implemented in compiler, X86: do it yourself)

Digis/Cluster SOA



The result of the clusterizer is just the assignment of a clusterId to each digi (and the number of clusters per module as necessary side-effect)

Example of use of SOA

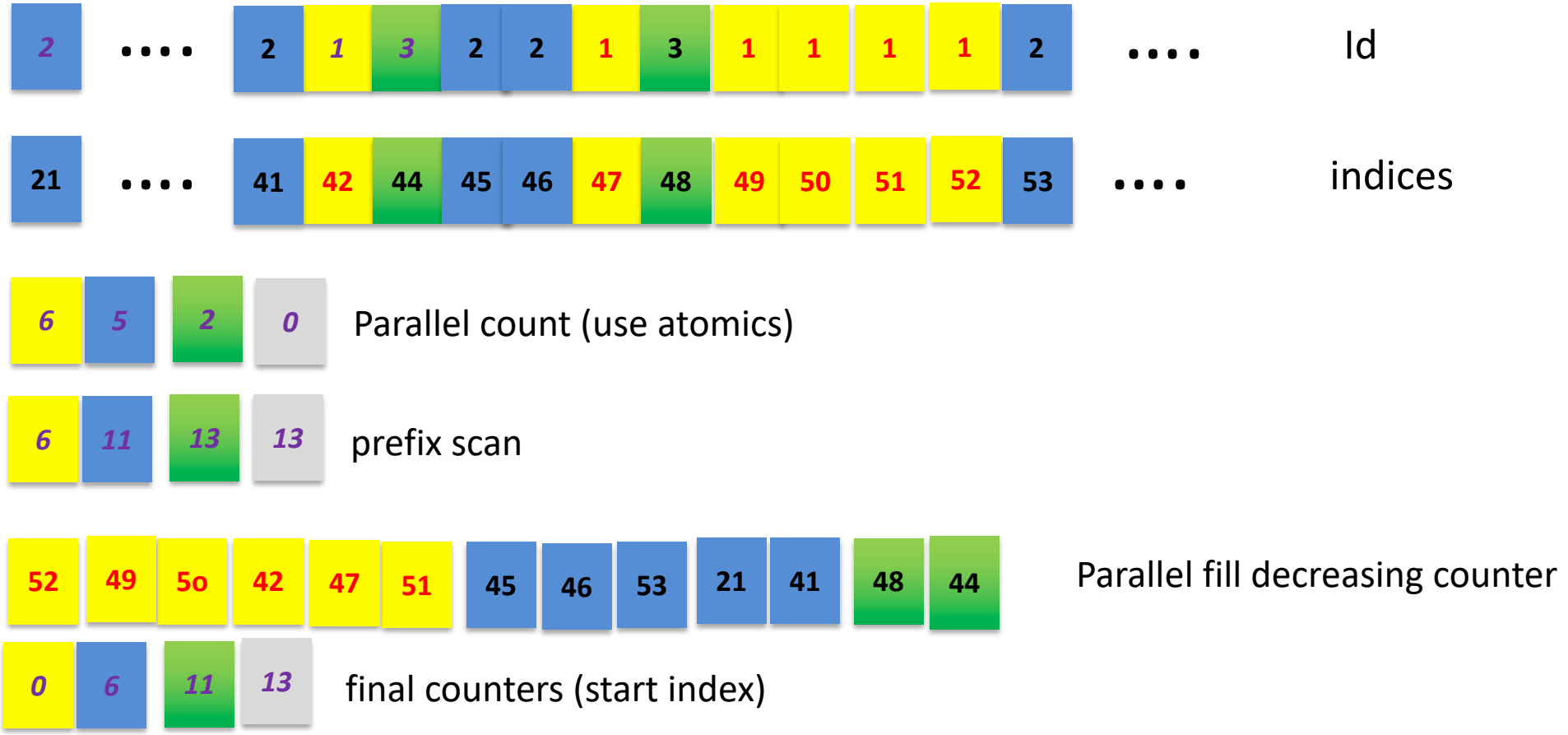
```
// compute cluster charge
__global__
void clusterCharge(int16_t const * clusterid, uint8_t const * adc,
                   int * charge, int n) {
  int first = blockDim.x * blockIdx.x + threadIdx.x;
  for (int i = first; i < n; i += gridDim.x*blockDim.x)
    if (clusterId(i) >=0)
      atomicAdd(&charge[clusterId(i)], adc[i]);
}
```

“loop” on digis

Accumulate in Cluster SOA

```
int threadsPerBlock = 256;  // a magic number
int blocksPerGrid =
  (maxNumOfDigis+ threadsPerBlock - 1) / threadsPerBlock;
clusterCharge<<<blocksPerGrid, threadsPerBlock>>>(clId,adc, charge,nDigis);
```

Filling One to Many Association (aka vector of vector)



Reminder: TrackingRecHits Legacy

- The CMSSW (ORCA) model for TrackingRecHits says that for any precision usage hits must be recomputed from the cluster using the most accurate track parameters
  - Defacto just a proxy to the Cluster + a transient cache for the local coordinates
  - Seeding uses its own caches (already a SoA)
    - Global coords, cluster shape,
  - Pattern recognition recomputes local coords each time
    - Because of KF, no use of global coords.

TrackingRecHit SoA for pixel reco

- Modelled to serve CA and fitting
  - Includes supporting structure (phi-histogram, etc)
- One to One to clusters (same indexing, no need of remapping)

Position (and Error) estimation

- Loop on Digis (not cluster) (blocked by Module as clusterizer)
  - Loop in cuda means one per thread...
- Fills standard object per cluster in shared memory with “CPE” info
  - Fully concurrent, use atomics
- Synchronize!
- Loop on “clusters” (CPE objects) and fill hit-SOA
  - local x,y,xerr,yerr
  - global xg,yg,zg,rg,phi (phi is an int16)
    - Beamspot subtracted!
  - Cluster shape info

SoA implementation

- Algorithms (Kernels) consume a “View”
  - Common to all “backends”: just a bunch of “encapsulated” pointers
- DataFormat is templated with a Traits (three different types…)
  - owns storage (i.e. its “deleter” aka pointer to the “free” function)
  - builds and provides the View (on Host!)
    - Filled by the “coder by hand”
    - Non trivial maintenance!
    - (Max)Size must be known at construction/allocation time (on Host)
  - encapsulates data transfer
- Traits
  - Native CPU
  - Native GPU
  - HOST: copy from GPU
  - Main difference is the version of malloc and free

```
class TrackingRecHit2DSOAView {
public:
  static constexpr uint32_t maxHits() { return gpuClustering::MaxNumClusters;
  using hindex_type = uint16_t;  // if above is <=2^16
  using Hist = HistoContainer<int16_t, 128, gpuClustering::MaxNumClusters, 8
sizeof(int16_t), uint16_t, 10>;

template<typename> friend class TrackingRecHit2DHeterogeneous;

  __device__ __forceinline__ uint32_t nHits() const { return m_nHits; }

  __device__ __forceinline__ float& xLocal(int i) { return m_xl[i]; }
  __device__ __forceinline__ float xLocal(int i) const { return __ldg(m_xl + i); }
….
private:
  // local coord
  float *m_xl, *m_yl;
  float *m_xerr, *m_yerr;
  // global coord
  float *m_xg, *m_yg, *m_zg, *m_rg;
  int16_t* m_iphi;
  // cluster properties
  int32_t* m_charge;
  int16_t* m_xsize;
  int16_t* m_ysize;
  uint16_t* m_detInd;
  // supporting objects
  ….
};
```

```
template<typename Traits>
class TrackingRecHit2DHeterogeneous {
public:

  template<typename T>
  using unique_ptr = typename Traits:: template unique_ptr<T>;

  explicit TrackingRecHit2DHeterogeneous(uint32_t nHits,
                                         pixelCPEforGPU::ParamsOnGPU const* cpeParams,
                                         uint32_t const* hitsModuleStart,
                                         cuda::stream_t<>& stream);
  TrackingRecHit2DSOAView* view() { return m_view.get(); }
  TrackingRecHit2DSOAView const* view() const { return m_view.get(); }

  auto nHits() const { return m_nHits; }

 // only the local coord and detector index
  cudautils::host::unique_ptr<float[]> localCoordToHostAsync(cuda::stream_t<>&
stream) const;  cudautils::host::unique_ptr<uint16_t[]>
detIndexToHostAsync(cuda::stream_t<>& stream) const;
  cudautils::host::unique_ptr<uint32_t[]>
hitsModuleStartToHostAsync(cuda::stream_t<>& stream) const;
private:
  static constexpr uint32_t n16 = 4;
  static constexpr uint32_t n32 = 9;
  unique_ptr<uint16_t[]> m_store16;  //!
  unique_ptr<float[]> m_store32;  //!
  unique_ptr<TrackingRecHit2DSOAView> m_view; //!
…
};
```

```
template<typename Traits>
TrackingRecHit2DHeterogeneous<Traits>::TrackingRecHit2DHeterogeneous(uint32_t nHits,
                                    pixelCPEforGPU::ParamsOnGPU const *cpeParams,
                                    uint32_t const *hitsModuleStart,
                                    cuda::stream_t<> &stream)
   : m_nHits(nHits), m_hitsModuleStart(hitsModuleStart) {
  edm::Service<CUDAService> cs;

  auto view = Traits:: template make_host_unique<TrackingRecHit2DSOAView>(cs,stream);
  view->m_nHits = nHits;
  m_view = Traits:: template make_device_unique<TrackingRecHit2DSOAView>(cs,stream);
  view->m_cpeParams = cpeParams;
  view->m_hitsModuleStart = hitsModuleStart;

  m_store16 = Traits:: template make_device_unique<uint16_t[]>(cs, nHits * n16, stream);
  m_store32 = Traits:: template make_device_unique<float[]>(cs, nHits * n32 + 11, stream);
  m_HistStore = Traits:: template make_device_unique<TrackingRecHit2DSOAView::Hist>(cs, stream);
  auto get16 = [&](int i) { return m_store16.get() + i * nHits; };
  auto get32 = [&](int i) { return m_store32.get() + i * nHits; };

  // copy all the pointers
  m_hist = view->m_hist = m_HistStore.get();
  view->m_xl = get32(0);
   …
  view->m_rg = get32(7);
  m_iphi = view->m_iphi = reinterpret_cast<int16_t *>(get16(0));
  view->m_charge = reinterpret_cast<int32_t *>(get32(8));
  …
  // transfer view
    if
#ifndef __CUDACC__
    constexpr
#endif
      (std::is_same<Traits,cudaCompat::GPUTraits>::value) {
      cudautils::copyAsync(m_view, view, stream);
    } else { m_view.reset(view.release());}
}
```

Owner + View model summary

- This model works because
  - There is not need for a common DataFormat-type between CPU and GPU as they are consumed in a very different way
    - The View is common and can be consumed by common algorithms
  - There is no use case for RecHits from CUDA besides Legacy
    - RecHitsSoA consumers better run all on the same device (no mixed wf)
    - This may change when DQM or Visualization will migrate to SoA
    - TrackingRecHitSoA may be spit to better serve multiple use cases

AoS, SoA, Eigen…

AoS

```
struct Point {
  float x,y,z;
};
std::vector<Point> points;
```

SoA

```
struct Points {
  std::vector<float> x,y,z;
};
```

```
struct Points {
  float x[N],y[N],z[N];
  uint32_t size;
};
```

SoA-View

```
struct Points {
  float * x,y,z;
  uint32_t size;
};
```

Unified-View

```
struct Points {
  float * p;
  uint32_t size;
};
```

EigenAoS

```
Map<Vector3f> point(p+3*i);
```

EigenSoA

```
Map<Vector3f,al, InnerStride<maxPoints>> point(p+i);
```

Static Size EigenSoA

- Efficient on CPU and GPU
- Indices computed at compile time
- Very compact assembly
- Storage included
- Trivial composition pattern
- memcpy compliant

```
template<typename M, int S>
struct alignas(128) MatrixSoA {
  using Scalar = typename M::Scalar;
  using Map = Eigen::Map<M, 0,
                  Eigen::Stride<M::RowsAtCompileTime*S,S> >;
  using CMap = Eigen::Map<const M, 0,
                    Eigen::Stride<M::RowsAtCompileTime*S,S> >;

  constexpr Map operator()(uint32_t i)  { return Map(data+i);}
  constexpr CMap operator()(uint32_t i) const { return CMap(data+i);}

  Scalar data[S*M::RowsAtCompileTime*M::ColsAtCompileTime];
  static_assert(isPowerOf2(S),"stride not a power of 2");
  static_assert(sizeof(data)%128==0, "size not a multiple of 128");
};
```

```
using V3 = Eigen::Vector3f;
using V15 = Eigen::Matrix<float,15,1>;
struct TSOS {
  V3 position;
  V3 momentum;
  V15 covariance;
};
```

```
template<int S>
struct TSOSsoa {
  static constexpr int stride() { return S;}
  MatrixSoA<V3,S> position;
  MatrixSoA<V3,S> momentum;
  MatrixSoA<V15,S> covariance;
};
```

Dynamic Size EigenSoA

- Not very efficient
- Indices computed at run time
- Quite verbose assembly
- Need external storage (it's a view)
- Cumbersome composition
- (max)size to be assigned at construction (allocation) time on Host

```
using DynStride = Eigen::Stride<Eigen::Dynamic,Eigen::Dynamic>;
template<typename M>
struct MatrixDynSoA {
  using Scalar = typename M::Scalar;
  using Map = Eigen::Map<M, 0, DynStride>;
  using CMap = Eigen::Map<const M, 0, DynStride>;
  constexp auto eStride() const {
        return DynStride(M::RowsAtCompileTime*stride,stride);
  }

  constexpr Map operator()(uint32_t i)  { return Map(data+i, eStride());}
  constexpr CMap operator()(uint32_t i) const { return CMap(data+i, eStride());}

  Scalar * data;
  int stride;
};
```

```
using V3 = Eigen::Vector3f;
using V15 = Eigen::Matrix<float,15,1>;
struct TSOS {
  V3 position;
  V3 momentum;
  V15 covariance;
};
```

```
template<int S>
struct TSOSsoa {
  MatrixDynSoA<V3> position;
  MatrixDynSoA<V3> momentum;
  MatrixDynSoA<V15> covariance;
 TSOSsoa(void *, int);//????
};
```

Track SoA

```
template <int32_t S>
struct TrajectoryStateSoA {
 using Vector5f = Eigen::Matrix<float, 5, 1>;
 using Vector15f = Eigen::Matrix<float, 15, 1>;

 static constexpr int32_t stride() { return S; }

 eigenSoA::MatrixSoA<Vector5f,S> state;
 eigenSoA::MatrixSoA<Vector15f,S> covariance;

 template<typename V3, typename M3, typename V2, typename M2>
 __host__ __device__ inline
 void copyFromCircle(V3 const & cp, M3 const & ccov, V2 const & lp,
M2 const & lcov, float b, int32_t i) {..}

 template<typename V5, typename M5>
 __host__ __device__ inline
 void copyFromDense(V5 const & v, M5 const & cov, int32_t i) {…}

 template<typename V5, typename M5>
 __host__ __device__ inline
 void copyToDense(V5 & v, M5 & cov, int32_t i) const {…}
};
```

```
template <int32_t S>  class TrackSoAT {
public:
 eigenSoA::ScalarSoA<uint8_t, S> m_quality;
  constexpr Quality quality(int32_t i) const { return (Quality)(m_quality(i));}

 eigenSoA::ScalarSoA<float, S> chi2;
  constexpr int nHits(int i) const { return detIndices.size(i);}

  // State at the Beam spot
  // phi,tip,1/pt,cotan(theta),zip
  TrajectoryStateSoA<S> stateAtBS;
  eigenSoA::ScalarSoA<float, S> eta;
  eigenSoA::ScalarSoA<float, S> pt;
  constexpr float charge(int32_t i) const { return
std::copysign(1.f,stateAtBS.state(i)(2)); }
  constexpr float phi(int32_t i) const { return stateAtBS.state(i)(0); }
  HitContainer hitIndices;
  HitContainer detIndices;

  // total number of tracks (including those not fitted)
  uint32_t m_nTracks;

};
using PixelTrackHeterogeneous =
HeterogeneousSoA<pixelTrack::TrackSoA>;
```

A sort of variant…

Same type for all targets (CPU,GPU,HOST)

Only one pointer valid

Easy to generalized to a single unique_ptr (for instance with a “switch” deleter + an enum)

Leave the implementation to the ingenuity of the volunteer

SoAFromCUDA producers are all identical

If edm is modified it can also hold ALL pointers and serve the correct one to the chosen target

```
// a heterogeneous unique pointer...
template<typename T>
class HeterogeneousSoA {
public:
  using Product = T;
explicit HeterogeneousSoA(cudautils::device::unique_ptr<T> && p) :
dm_ptr(std::move(p)) {}
  explicit HeterogeneousSoA(cudautils::host::unique_ptr<T> && p) :
hm_ptr(std::move(p)) {}
  explicit HeterogeneousSoA(std::unique_ptr<T> && p) :
std_ptr(std::move(p)) {}


  auto const * get() const {
    return dm_ptr ? dm_ptr.get() : (hm_ptr ? hm_ptr.get() : std_ptr.get());
private:
  // a union wan't do it, a variant will not be more efficient
  cudautils::device::unique_ptr<T> dm_ptr; //!
  cudautils::host::unique_ptr<T> hm_ptr; //!
  std::unique_ptr<T> std_ptr; //!  To be changed in malloc/free

};
```

ZVertexSoA w/o Eigen

```
struct ZVertexSoA {
    static constexpr uint32_t MAXTRACKS = 32*1024;
    static constexpr uint32_t MAXVTX = 1024;

    int16_t idv[MAXTRACKS];    // vertex index for each associated (original) track  (-1 == not associate)
    float zv[MAXVTX];          // output z-position of found vertices
    float wv[MAXVTX];          // output weight (1/error^2) on the above
    float chi2[MAXVTX];        // vertices chi2
    float ptv2[MAXVTX];        // vertices pt^2
    int32_t ndof[MAXVTX];      // vertices number of dof (reused as workspace for the number of nearest neighbors)
    uint16_t sortInd[MAXVTX];  // sorted index (by pt2)  ascending
    uint32_t nvFinal;          // the number of vertices

    __host__ __device__ void init() { nvFinal = 0; }

};
using ZVertexHeterogeneous = HeterogeneousSoA<ZVertexSoA>;
using ZVertexCUDAProduct =  CUDAProduct<ZVertexHeterogeneous>;
```

StaticSizeSoA + Heterogenous unique_ptr

- Common Object and common DataProduct
  - In principle all code can be shared
  - Algorithms (Kernels) consume SoA pointers (dataProduct.get())
  - SoAFromCUDA is trival
- Easy to compose using standard OO practices
  - Trivial maintenance

Refs

- Associations
- Ref, Handle
- Dandling pointer, swizzling, marshalling, relocation table
- Garbage collection, reference counting, scope-wiping

- EDM support is essential (actually enough the host-allocator API)
  - Implementation left to volunteer
  - Some solutions may require pointers/Refs to be marshalled in transfer
  - My prototype is: double pointer + ifdef + relocation table (no marshalling)