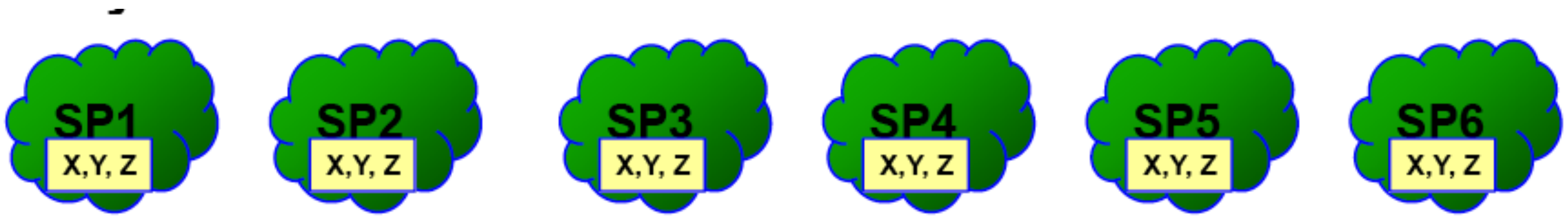
A SOA Event Model

High Performance Computing for High Energy Physics

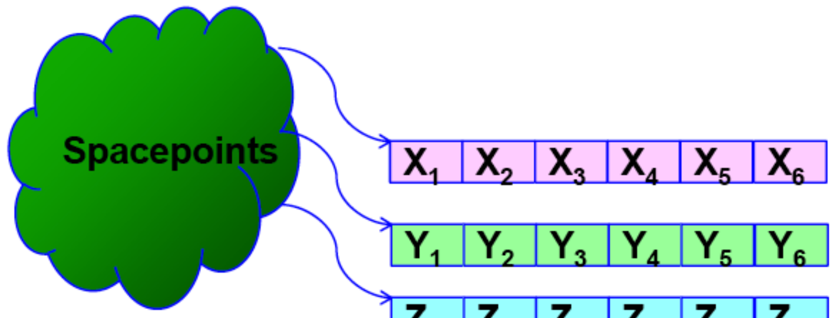
Vincenzo Innocente
CERN/EP/SFT
CMS Experiment

Data Organization: AoS vs SoA

- Traditional Object organization is an Array of Structures
  - Abstraction often used to hide implementation details at object level

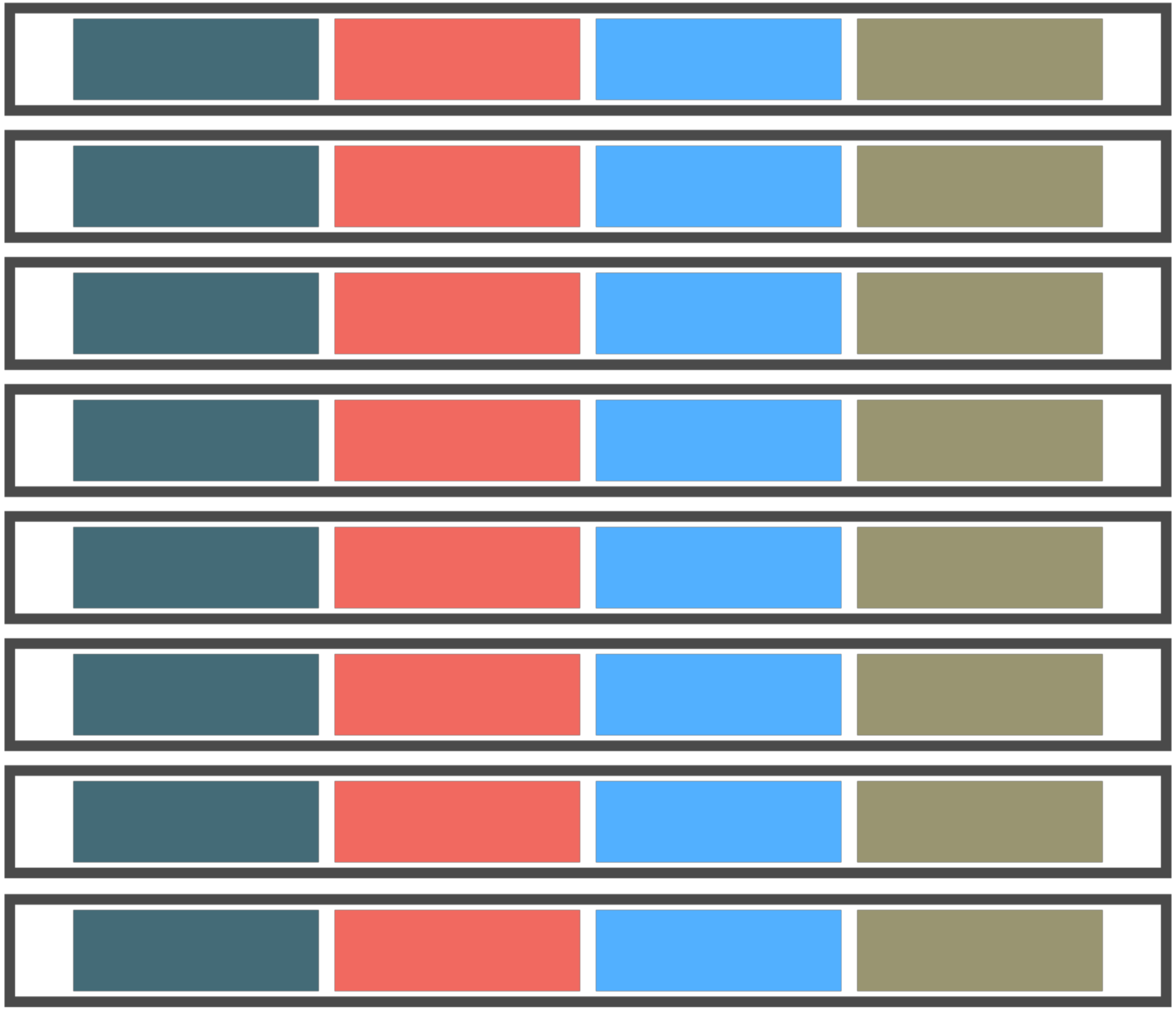


- Difficult to fit stream computing
- Better to use a Structure of Arrays
  - (column-wise storage)
- OO can wrap SoA as the AoS
  - Move abstraction higher
  - Expose data layout to the compiler
- Explicit copy in many cases more efficient
  - (notebooks vs whiteboard)



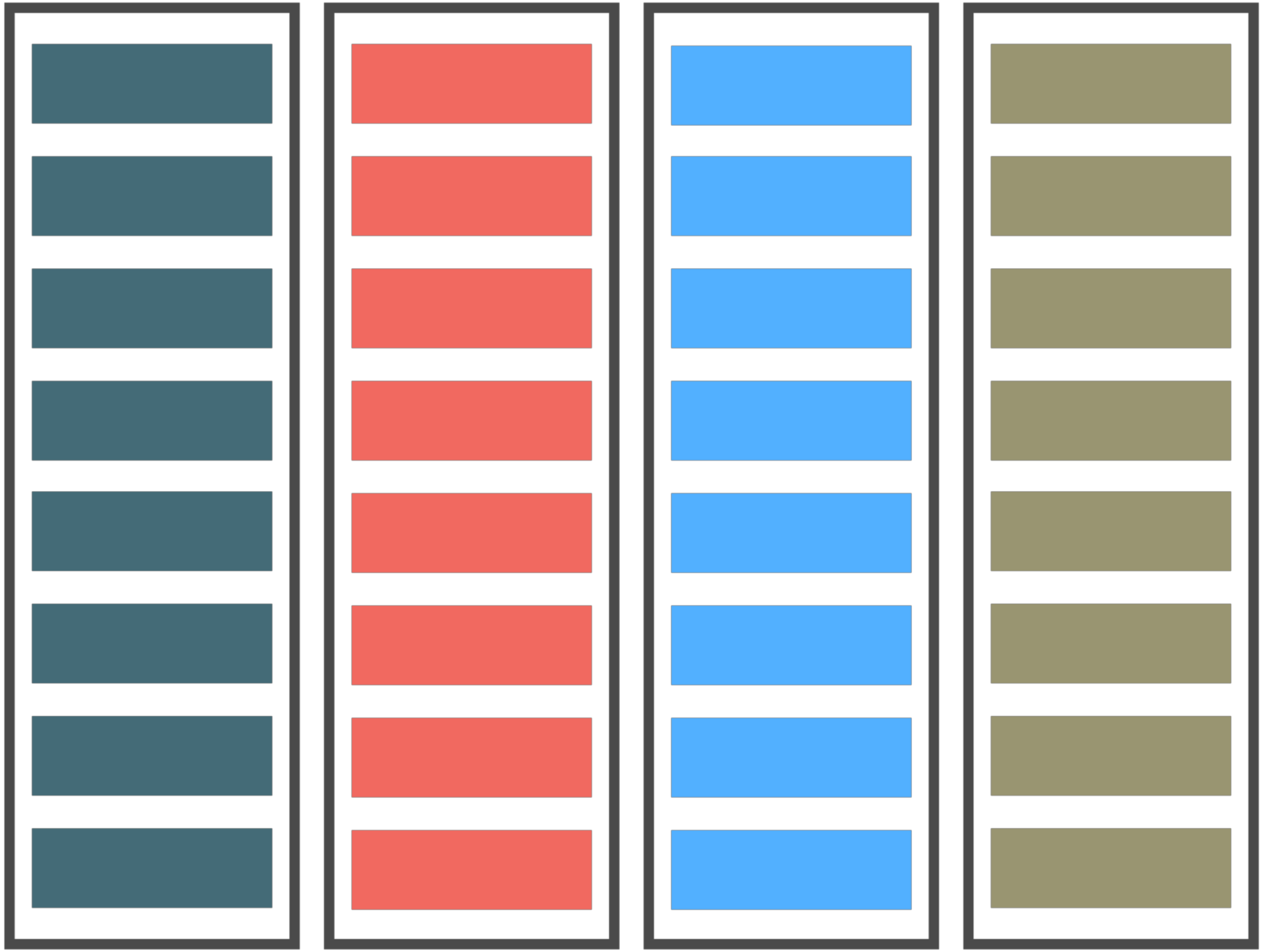
Array of Structures

T matrix[N][4];



std::vector<  >

Structure of Array
T matrix[4][N];

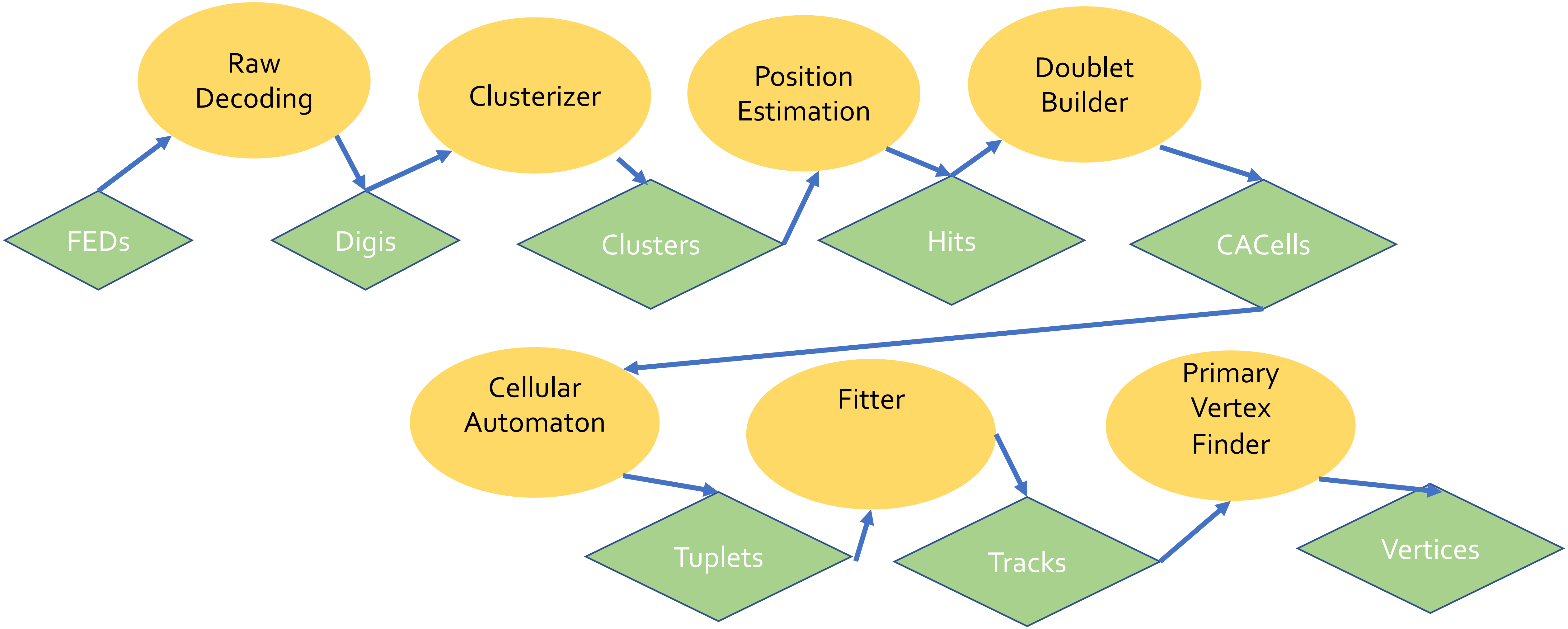


struct {std::vector< >; std::vector< >; std::vector< >; std::vector< >;}

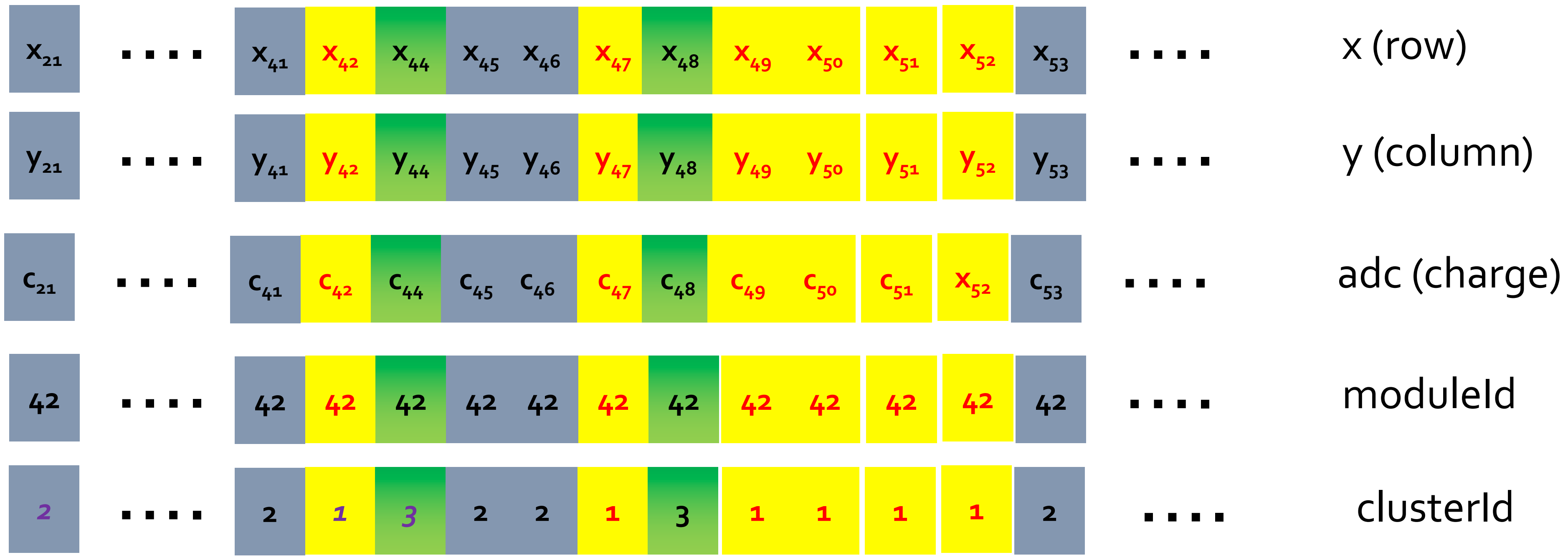
Why SOA?

- SOA naturally fits SIMD/SIMT architectures (stream computing)
- No padding or alignment issue: one can choose optimized types
- Its cache friendly in particular if only few elements are accessed
- Very Large SOA (in both dimensions) may affect TLB though

Pixel Tracks/Seed Workflow



Digis 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)

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)

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

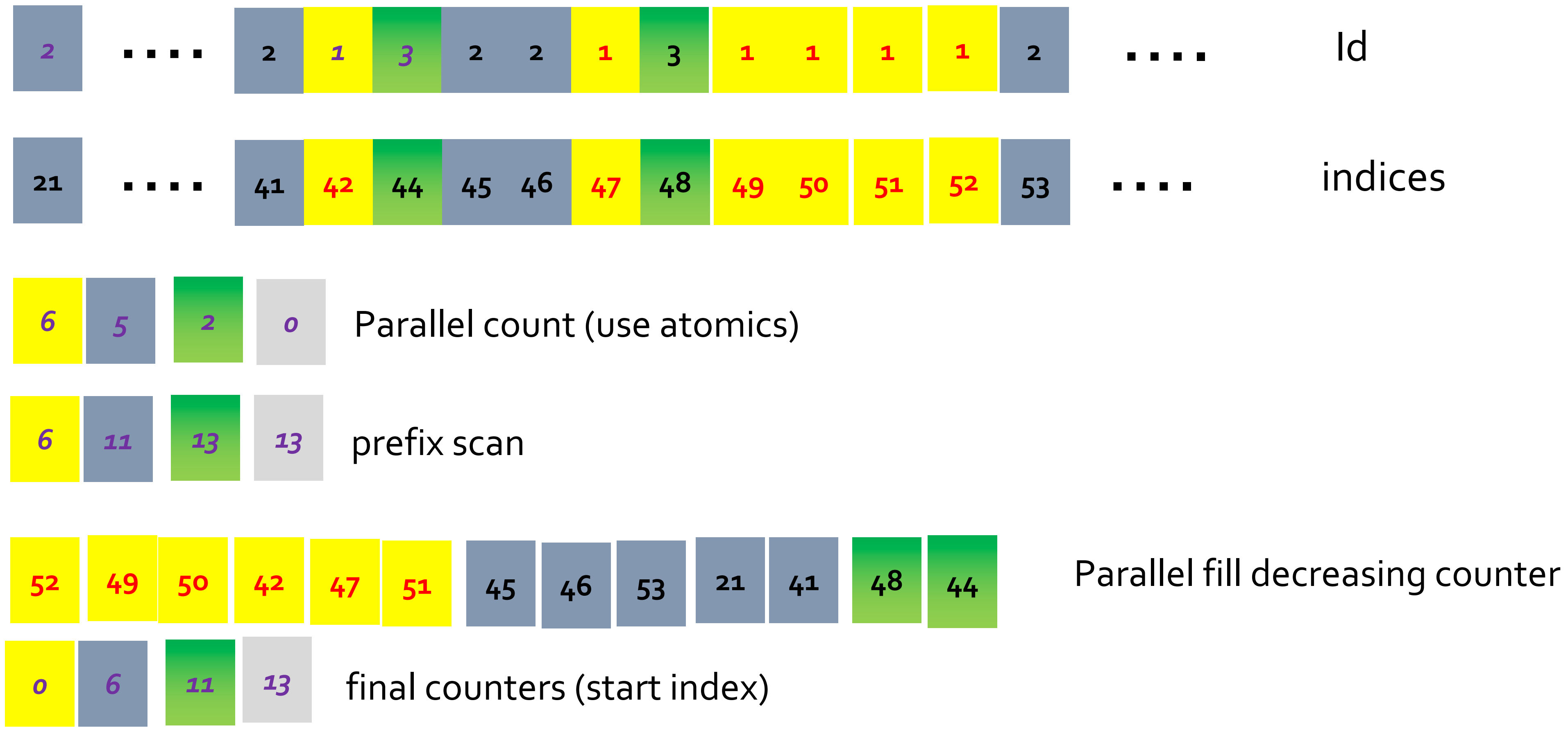
Accumulate in Cluster SOA

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

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!

Filling One to Many Association (aka vector of vector)



AoS, SoA, Eigen...

SoA

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

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

AoS

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

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);
```

More complex

```
struct Hit {
   Point p; // x,y,z
   CovXYZ err; // 6 elements
};

struct Track {
  Hit hits[4];
};

std::vector<Track> tracks;
```

```
constexpr uint32_t maxNumberOfTracks() { return 10*1024; }
constexpr uint32_t stride() { return maxNumberOfTracks();}

using Matrix3x4f = Eigen::Matrix<float,3,4>; // each hit in a column
using Map3x4f = Eigen::Map<Matrix3x4f,0,Eigen::Stride<3*stride(),stride()> >;

using Matrix6x4f = Eigen::Matrix<float,6,4>;
using Map6x4f = Eigen::Map<Matrix6x4f,0,Eigen::Stride<6*stride(),stride()> >;
```

More general

```
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??

```
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);//????
};
```

“Managing” Dynamic SOAs

https://github.com/cms-patatrack/cmssw/commit/8eaa29bd1066458obec5d0229da549c4cffad64d

What about my cute OO design?

```
struct Box {
  Eigen::AffineCompact3f transform;
  V3 halfWidth;

  template<typename P3>
  inline bool inside(P3 const & p) const {
    return ((transform*p).array().abs() <  halfWidth.array()).all();
  }
};
```

https://godbolt.org/z/N-1D22

```
void doAOS(std::vector<TSOS> &trajs, Box const & b,
                           std::vector<bool> & res) {
std::transform(trajs.begin(),trajs.end(),res.begin(),
                               [&](auto const& t){ return b.inside(t.position);});
}
```

```
void doSOA(TSOSsoa & trajSoa, Box const & b,
                              bool * res) {
 #pragma GCC ivdep
 for (auto i=0U; i<nTracks; ++i)
   res[i] = b.inside(trajSoa.position(i));
}
```

More Flexible: ASoA ("bucketized SoA")

```
template<uint32_t S>
struct alignas(128) SoA {

  static constexpr uint32_t stride() { return S; }
  static constexpr uint32_t mask() { return S-1;}
  static constexpr uint32_t shift() { return ilog2(S); }

  float a[S];
  float b[S];

  static_assert(isPowerOf2(S),
                     "stride not a power of 2");
  static_assert(sizeof(a)%128 == 0,
                    "size not a multiple of 128");

};
template< uint32_t S>
using ASoA = std::vector<SoA<S>>;
// std::vector<SoA<S>*>; std::list<SoA<S>>;????
```

```
using V = SoA<myStride>;
void sum(V * psoa, uint32_t n) {
  for (uint32_t i=0; i<n; i++) {
    auto j = i/V::stride(); // i>>V::shift();
    auto k = i%V::stride(); // i&V::mask();
    auto & soa = psoa[j];
    soa.b[k] += soa.a[k];
  }
}
```

```
void sum(V * psoa , uint32_t n) {
  auto nb = (n+V::stride()-1)/ V::stride();
  for (uint32_t j=0; j<nb; j++) {
    auto & soa = psoa[j];
    auto kmax = std::min(V::stride(),n - j*V::stride());
    for(uint32_t k=0; k<kmax; k++) soa.b[k] += soa.a[k];
  }
}
```

More Flexible: ASoA ("bucketized SoA") cuda

```
template<uint32_t S>
struct alignas(128) SoA {

  static constexpr uint32_t stride() { return S; }
  static constexpr uint32_t mask() { return S-1;}
  static constexpr uint32_t shift() { return ilog2(S); }

  float a[S];
  float b[S];

  static_assert(isPowerOf2(S),
                "stride not a power of 2");
  static_assert(sizeof(a)%128 == 0,
                "size not a multiple of 128");

};
```

```
__global__
void sum(V * psoa, int n) {
  auto first = threadIdx.x + blockIdx.x*blockDim.x;
  for (auto i=first; i<n; i+=blockDim.x*gridDim.x) {
    auto j = i/V::stride();
    auto k = i%V::stride();
    auto & soa = psoa[j];
    soa.b[k] += soa.a[k];
  }
}
```

```
__global__  // maps buckets to blocks
void sum(V * psoa, int n) {
  auto nb = (n+V::stride()-1)/V::stride();
  for (auto j=blockIdx.x; j<nb; j+=gridDim.x) {
    auto & soa = psoa[j];
    auto kmax = std::min(V::stride(),n - j*V::stride());
    for(uint32_t k=threadIdx.x; k<kmax; k+=blockDim.x) {
     soa.b[k] += soa.a[k];
    }
  }
```

ASoA: a possible implementation

https://github.com/VinInn/cmssw/blob/EigenSOA/HeterogeneousCore/CUDAUtilities/interface/ASoA.h

https://github.com/VinInn/cmssw/blob/EigenSOA/DataFormats/SiPixelDigi/interface/SiPixelDigisSoA.h

Advantaged of fixed size SoA

- Easy to manage and compose
  - Only one (de-)allocation and eventual copy for the whole SoA
- Correct alignment and sizing imposed/checked at compile time
- Compiler can take full advantage of knowing size and alignment
- Runtime sizing can be achieved using Arrays of SoAs
  - Bucketized SoA can help to mitigate pressure on TLB
  - Difficult to use with external utilities (memset!)