Portable Implementation of the p2z Benchmark using Alpaka

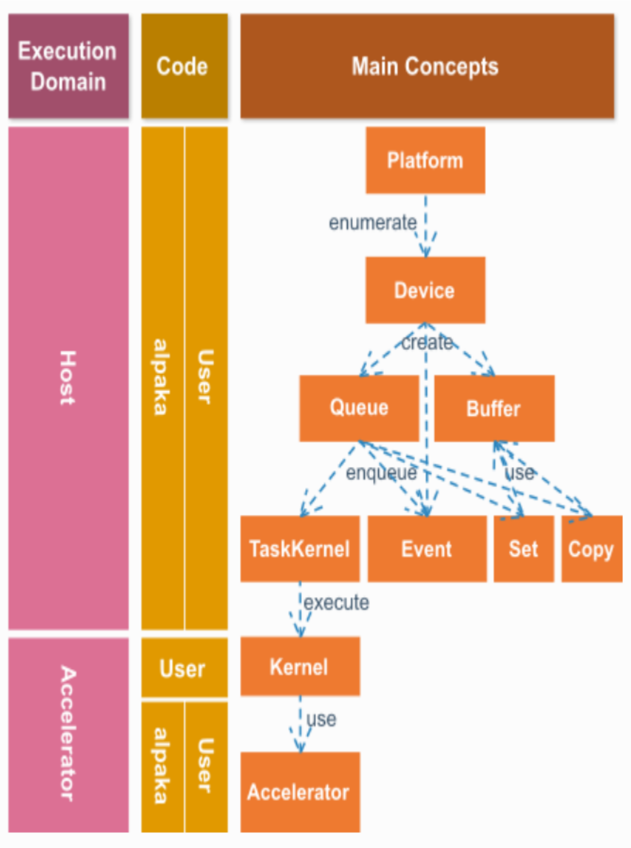
Cong Wang

BACKGROUND

- Today's computing in HEP:
  - code is written for the x86 platform is pretty much guaranteed to run everywhere in the world, from computing centers using batch systems to our own laptops.
  - GPUs and other accelerators provide more processing power for the same energy consumption as with x86-based supercomputers. It can lead to designing algorithms and implementing them for specific hardware platforms and combinations, and making it very difficult to use not only one but several of these platforms.
- Motivation:
  - investigate solutions for portability techniques that will allow the coding of an algorithm once, and the ability to execute it on a variety of hardware products from many vendors, especially including accelerators.

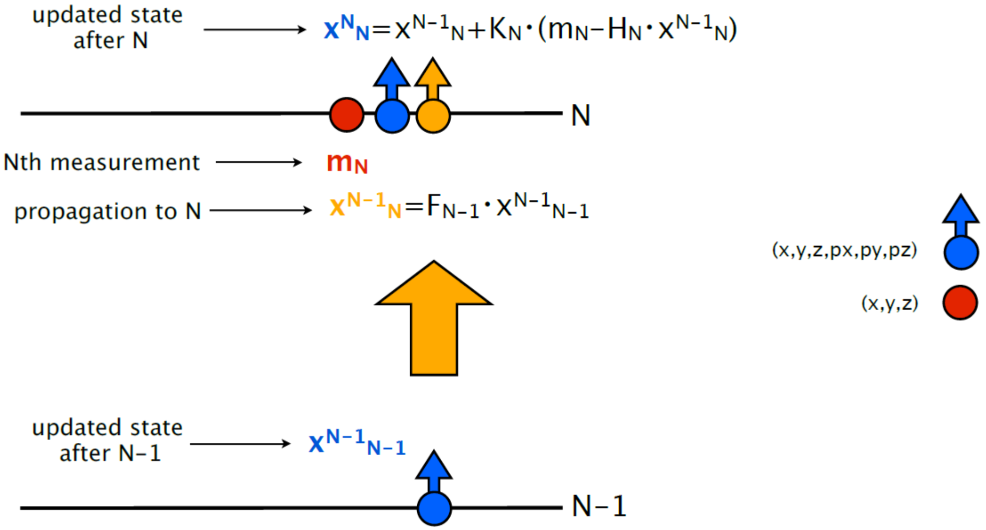
ALPAKA

- **A**bstraction **L**ibrary for **Pa**rallel **K**ernel **A**cceleration:
  - defines and implements an abstract interface for the *hierarchical redundant parallelism* model. The model exploits task- and data-parallelism as well as memory hierarchies at all levels of current multi-core architectures.
  - provides back-ends for *CUDA*, *OpenMP*, *TBB* and other methods. The policy-based C++ template interface provided allows for straightforward user-defined extension of the library to support other accelerators.
- Sustainable, heterogeneous, maintainable, testable, optimizable, extensible, data structure agnostic



PROPAGATE TO Z

- Propagate to z (p2z):
  - the layers at a fixed z positions (endcap disks). The particles are propagated in a homogeneous magnetic field with direction parallel to the z axis.
  - each track is propagated from layer N-1 to N and updated the track parameters based on the hit (measurement) located on layer N (mN) by using the Kalman Filter.
  - collision events inside a particle detector are independent of each other.
  - the various tracks created within an event can be built and fitted independently.
  - tracks are grouped together in batches to propagate from one layer of the detector to the next.



updated state after N → $x^N_N = x^{N-1}_N + K_N \cdot (m_N - H_N \cdot x^{N-1}_N)$

N

Nth measurement → m_N

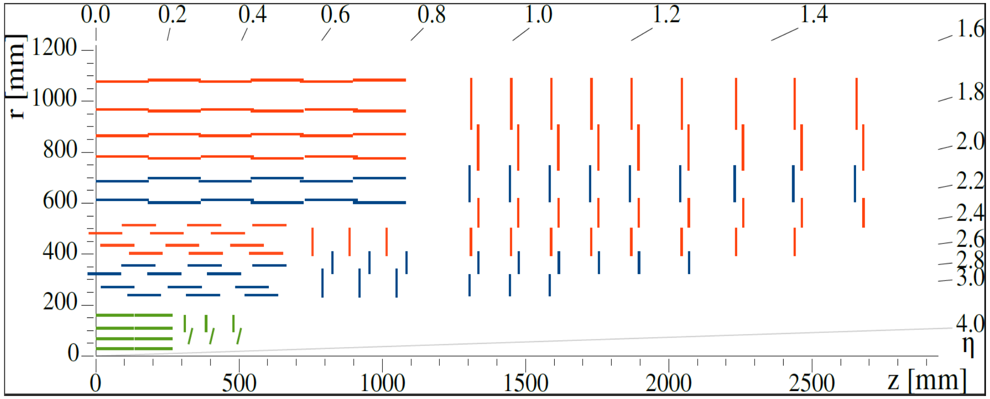
propagation to N → $x^{N-1}_N = F_{N-1} \cdot x^{N-1}_{N-1}$

(x,y,z,px,py,pz)

(x,y,z)

updated state after N−1 → x^{N-1}_{N-1}

N−1



- Parameters:
  - batch size (bsize), number of layers (nlayer), events (nevts), tracks (ntrks), and batches (nb=ntrks/bsize)

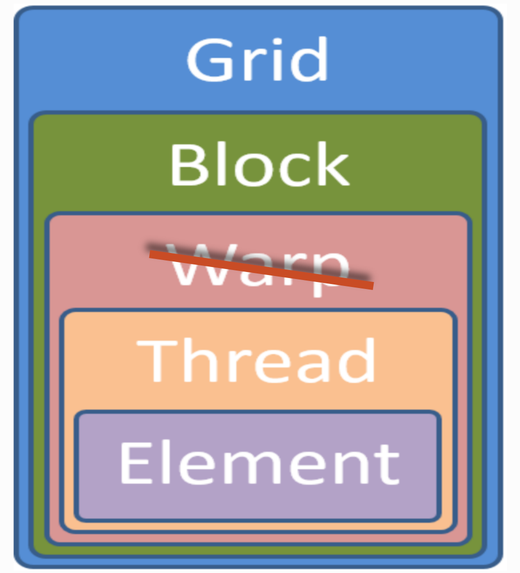
IMPLEMENTATION

- **Propagate-toz-test_alpaka_CPU:**
  - a universal grid-block and block-thread definition for **CpuThreads, Omp2threads**, **Omp2Blocks, and TbbBlocks** as portable accelerators.
  - each event & batch of tracks is distributed to one thread or block.
  - SIMD inside computing functions. **Each thread processes # of batch size elements** which could be treated at once due to the same instructions and no data dependency.
  - based on the accelerator, explicitly define the number of resources, loop over till the last event & batch of tracks.

threadIdx.x, threadIdx.y
= defined parallel threads

blockIdx.x * blockIdx.y
= defined parallel blocks

elementsperthread = batch size

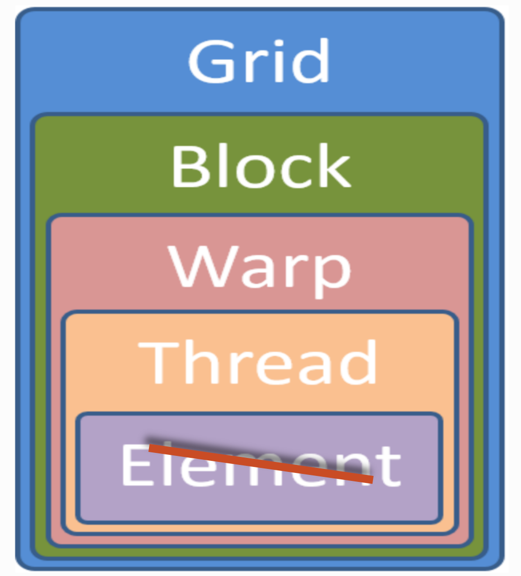


IMPLEMENTATION

- **Propagate-toz-test_alpaka_GPU:**
    - same code as CPU, same universal grid-block and block-thread definition as CPU for CudaRt accelerator.
    - each event & batch of tracks is distributed to one block, one thread are responsible for one track.
    - based on the accelerator, explicitly define the number of resources, loop over till the last event & batch of tracks.
    - one to multiple streams, asynchronous memory transfer.

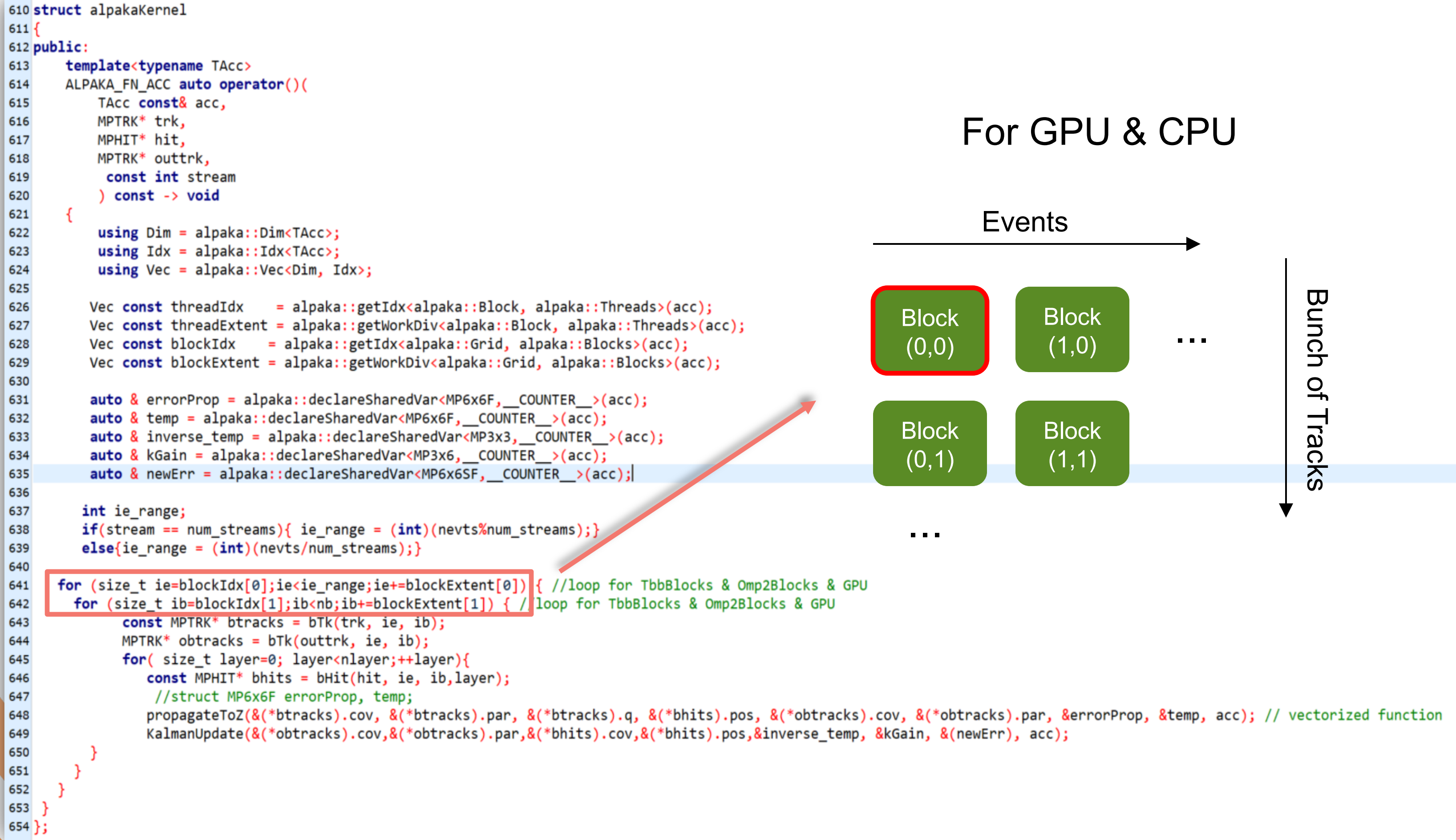
threadIdx.x * threadIdx.y
= defined parallel threads

blockIdx.x * blockIdx.y
= defined parallel blocks



CODE EXAMPLE

- Kernel:

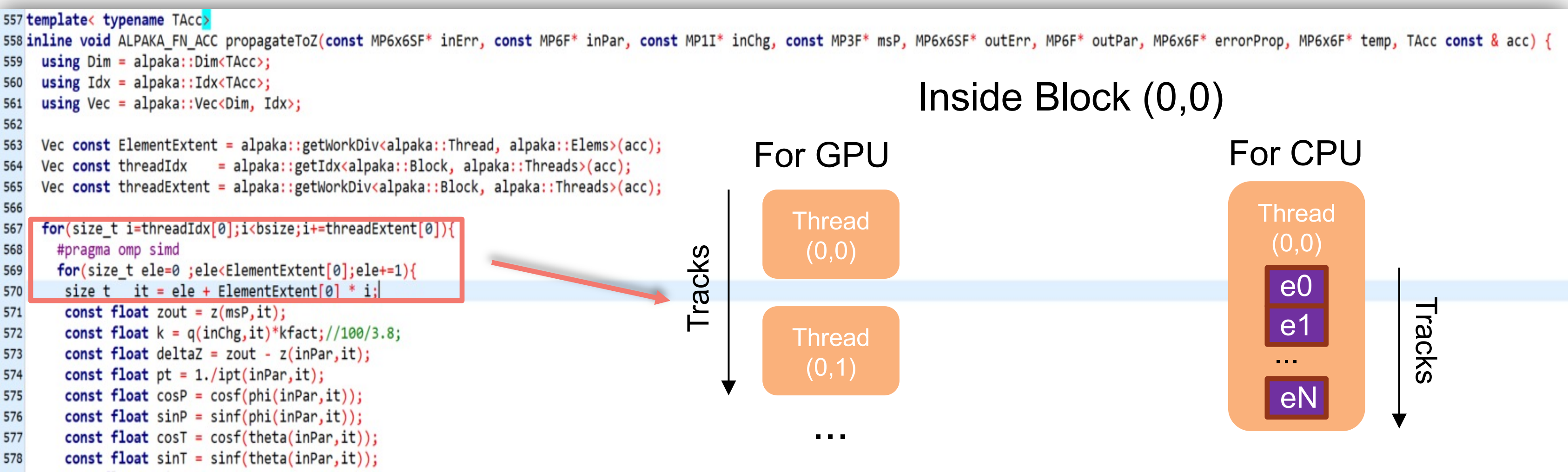


```
struct alpakaKernel
{
public:
    template<typename TAcc>
    ALPAKA_FN_ACC auto operator()(
        TAcc const& acc,
        MPTRK* trk,
        MPHIT* hit,
        MPTRK* outtrk,
         const int stream
        ) const -> void
    {
        using Dim = alpaka::Dim<TAcc>;
        using Idx = alpaka::Idx<TAcc>;
        using Vec = alpaka::Vec<Dim, Idx>;

       Vec const threadIdx    = alpaka::getIdx<alpaka::Block, alpaka::Threads>(acc);
       Vec const threadExtent = alpaka::getWorkDiv<alpaka::Block, alpaka::Threads>(acc);
       Vec const blockIdx     = alpaka::getIdx<alpaka::Grid, alpaka::Blocks>(acc);
       Vec const blockExtent = alpaka::getWorkDiv<alpaka::Grid, alpaka::Blocks>(acc);

       auto & errorProp = alpaka::declareSharedVar<MP6x6F,__COUNTER__>(acc);
       auto & temp = alpaka::declareSharedVar<MP6x6F,__COUNTER__>(acc);
       auto & inverse_temp = alpaka::declareSharedVar<MP3x3,__COUNTER__>(acc);
       auto & kGain = alpaka::declareSharedVar<MP3x6,__COUNTER__>(acc);
       auto & newErr = alpaka::declareSharedVar<MP6x6SF,__COUNTER__>(acc);

      int ie_range;
      if(stream == num_streams){ ie_range = (int)(nevts%num_streams);}
      else{ie_range = (int)(nevts/num_streams);}

   for (size_t ie=blockIdx[0];ie<ie_range;ie+=blockExtent[0]) { //loop for TbbBlocks & Omp2Blocks & GPU
     for (size_t ib=blockIdx[1];ib<nb;ib+=blockExtent[1]) { //loop for TbbBlocks & Omp2Blocks & GPU
         const MPTRK* btracks = bTk(trk, ie, ib);
         MPTRK* obtracks = bTk(outtrk, ie, ib);
         for( size_t layer=0; layer<nlayer;++layer){
            const MPHIT* bhits = bHit(hit, ie, ib,layer);
             //struct MP6x6F errorProp, temp;
            propagateToZ(&(*btracks).cov, &(*btracks).par, &(*btracks).q, &(*bhits).pos, &(*obtracks).cov, &(*obtracks).par, &errorProp, &temp, acc); // vectorized function
            KalmanUpdate(&(*obtracks).cov,&(*obtracks).par,&(*bhits).cov,&(*bhits).pos,&inverse_temp, &kGain, &(newErr), acc);
        }
      }
    }
  }
};
```

For GPU & CPU

CODE EXAMPLE

- Device (accelerator) functions:

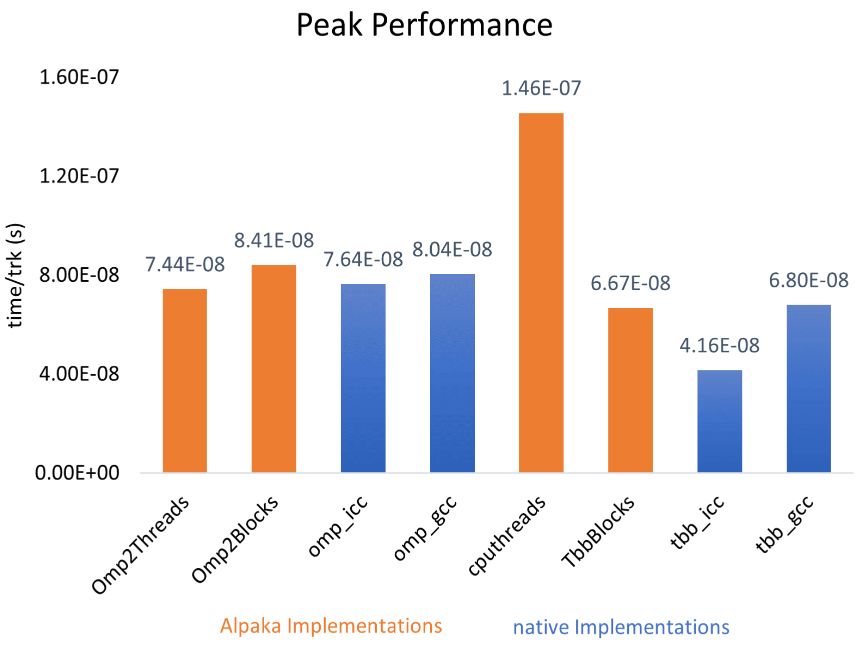


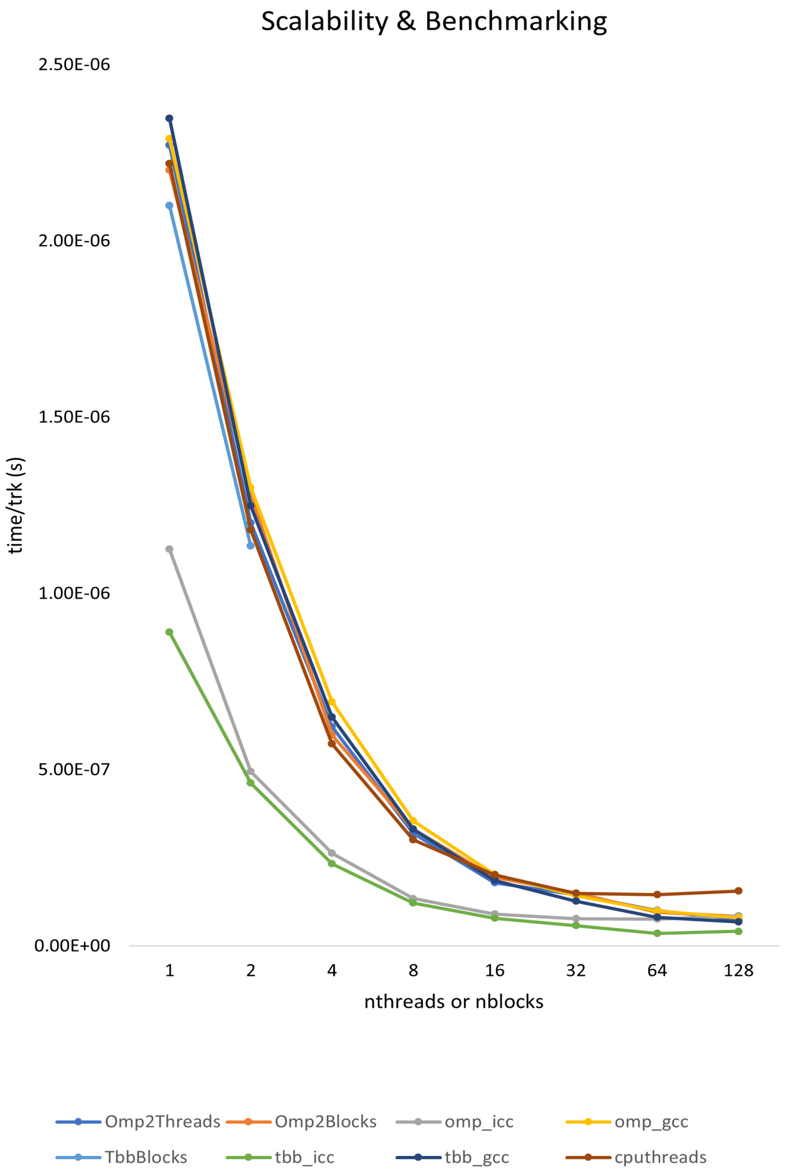
```
template< typename TAcc>
inline void ALPAKA_FN_ACC propagateToZ(const MP6x6SF* inErr, const MP6F* inPar, const MP1I* inChg, const MP3F* msP, MP6x6SF* outErr, MP6F* outPar, MP6x6F* errorProp, MP6x6F* temp, TAcc const & acc) {
  using Dim = alpaka::Dim<TAcc>;
  using Idx = alpaka::Idx<TAcc>;
  using Vec = alpaka::Vec<Dim, Idx>;

  Vec const ElementExtent = alpaka::getWorkDiv<alpaka::Thread, alpaka::Elems>(acc);
  Vec const threadIdx     = alpaka::getIdx<alpaka::Block, alpaka::Threads>(acc);
  Vec const threadExtent = alpaka::getWorkDiv<alpaka::Block, alpaka::Threads>(acc);

  for(size_t i=threadIdx[0];i<bsize;i+=threadExtent[0]){
    #pragma omp simd
    for(size_t ele=0 ;ele<ElementExtent[0];ele+=1){
     size_t   it = ele + ElementExtent[0] * i;
     const float zout = z(msP,it);
     const float k = q(inChg,it)*kfact;//100/3.8;
     const float deltaZ = zout - z(inPar,it);
     const float pt = 1./ipt(inPar,it);
     const float cosP = cosf(phi(inPar,it));
     const float sinP = sinf(phi(inPar,it));
     const float cosT = cosf(theta(inPar,it));
     const float sinT = sinf(theta(inPar,it));
```

COMPILATION

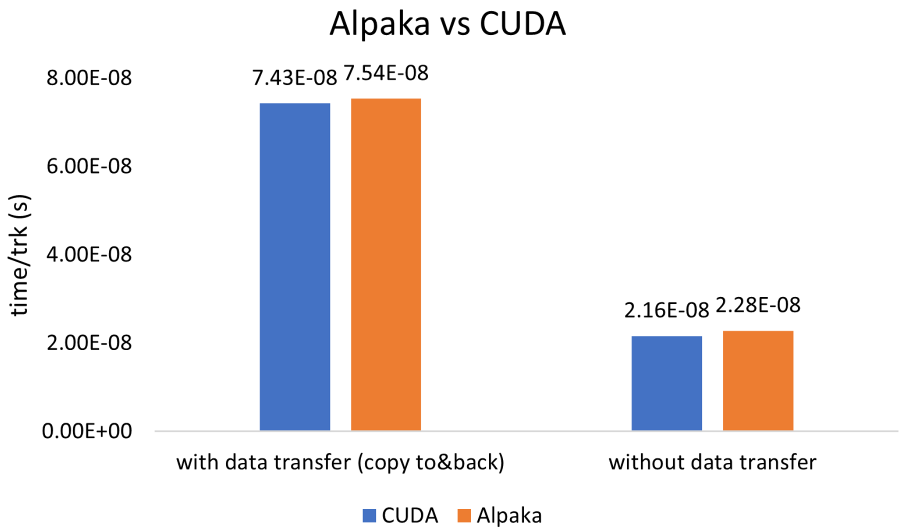
- Parameters:
  - bsize: 32, ntrks: 9600, nevts: 100, nb: 300, smear: 0.1, nlayer: 20, NITER: 5, Stream: 5
- Computing Architecture:
  - Intel Xeon Gold 6148@2.40 GHz for CPU, 80 cores in total.
  - V100 16 GB GPU on Wilson Cluster for GPU implementation testing.
- Compilers:
  - GCC, NVCC, no ICC
- Cmake option:
  - DCMAKE_CXX_COMPILER=g++
    DCMAKE_C_COMPILER=gcc
    DCMAKE_CUDA_COMPILER=nvcc
  - DALPAKA_ACC_CPU_B_SEQ_T_SEQ_ENABLED -
    DALPAKA_ACC_CPU_B_OMP2_T_SEQ_ENABLED -
    DALPAKA_ACC_CPU_B_SEQ_T_OMP2_ENABLED -
    DALPAKA_ACC_CPU_B_SEQ_T_THREADS_ENABLED -
    DALPAKA_ACC_CPU_B_TBB_T_SEQ_ENABLED -
    DALPAKA_ACC_GPU_CUDA_ENABLE

RESULTS CPU


Peak Performance
time/trk (s)
1.60E-07
1.20E-07
8.00E-08
4.00E-08
0.00E+00
7.44E-08
8.41E-08
7.64E-08
8.04E-08
1.46E-07
6.67E-08
4.16E-08
6.80E-08
Omp2Threads
Omp2Blocks
omp_icc
omp_gcc
cputhreads
TbbBlocks
tbb_icc
tbb_gcc
Alpaka Implementations
native Implementations


Scalability & Benchmarking
time/trk (s)
2.50E-06
2.00E-06
1.50E-06
1.00E-06
5.00E-07
0.00E+00
1
2
4
8
16
32
64
128
nthreads or nblocks
Omp2Threads
Omp2Blocks
omp_icc
omp_gcc
TbbBlocks
tbb_icc
tbb_gcc
cputhreads

RESULTS GPU


Alpaka vs CUDA
time/trk (s)
8.00E-08
6.00E-08
4.00E-08
2.00E-08
0.00E+00
7.43E-08
7.54E-08
2.16E-08
2.28E-08
with data transfer (copy to&back)
without data transfer
CUDA
Alpaka

CONCLUSION

- Alpaka Experience and Portability Matrix

Learning	**Code Conversion**	**Building level**	**Hardware map**	**Feature**
Easy to learn Good docs Lack of examples	Convertible with little more efforts	No major changes CMake provided	CPU, Nvidia GPU, AMD GPU No other supported	Reduction, Atomic Kernel Concurrency
Debugging	**User Support**	**Sustainability**	**Interoperability**	**Performance**
Easy to debug	Discussing thread Small community	Life cycle unpredictable	Mix with compiler directives and CUDA API	Minor loss or equivalent

REFERENCE

- E. Zenker, R. Widera, G. Juckeland et al., *Porting the Plasma Simulation PIConGPU to Heterogeneous Architectures with Alpaka*, video link (39 min), slides (PDF), DOI:10.5281/zenodo.6336086

- Alpaka: https://alpaka.readthedocs.io/en/latest/index.html

- Lantz, Steven, et al. "Speeding up particle track reconstruction using a parallel Kalman filter algorithm." *Journal of Instrumentation* 15.09 (2020): P09030.

- Bhattacharya, Meghna, et al. "Portability: A Necessary Approach for Future Scientific Software." *arXiv preprint arXiv:2203.09945* (2022).

QUESTIONS