Towards ACTS KalmanFitter on GPUs

Xiaocong Ai

April 17, 2020

The observations so far

- What are working:
  - Template
  - Virtual function (works with CUDA, not with SYCL?)
  - Eigen works
    - Potential issue with dynamic-size matrix (coefficients are stored as a pointer to a dynamically-allocated array)
- What are NOT working:
  - std::function
  - Ordinary containers (e.g. std::vector, std::array) and algorithms (e.g. std::sort) in C++ STL are not usable
    - Use C array or Eigen::array/Eigen::matrix (static memory allocation)
    - Use explicit heap memory allocation where needed

ATLAS B field on GPUs

- Reminder of the object ownership:
  Acts::Grid →
  Acts::InterpolatedBFieldMapper →
  Acts::InterpolatedBFieldMap →
  Acts::EigenStepper →
  Acts::Propagator

- B field values stored at Acts::Grid requires large memory allocation (can only be done on heap)
  - std::vector is replaced by a pointer to a dynamically allocated memory

    But cudaMemCpy of propagator from host to device won't copy the field values!

```
/// @brief default constructor
///
/// @param [in] axes actual axis objects spanning the grid
ACTS_DEVICE_FUNC Grid(std::tuple<Axes...> axes) : m_axes(std::move(axes)) {
  m_values = new T[size()];
}

/// Copy constructor
///
/// @param rhs is the source Grid
ACTS_DEVICE_FUNC Grid(const Grid &rhs) : m_axes(rhs.m_axes) {
  m_values = new T[rhs.size()];
  memcpy(m_values, rhs.m_values, sizeof(T) * rhs.size());
}

/// Assignment constructor
///
/// @param rhs is the source Grid
ACTS_DEVICE_FUNC Grid &operator=(const Grid &rhs) {
  m_axes = rhs.m_axes;
  m_values = new T[rhs.size()];
  memcpy(m_values, rhs.m_values, sizeof(T) * rhs.size());
  return (*this);
}

/// @brief default destructor
///
ACTS_DEVICE_FUNC ~Grid() { delete[] m_values; }
```

```
private:
 /// set of axis defining the multi-dimensional grid
 std::tuple<Axes...> m_axes;
 /// linear value store for each bin
 std::vector<T> m_values;
```

→

```
private:
  /// set of axis defining the multi-dimensional grid
  std::tuple<Axes...> m_axes;
  /// pointer to linear value store for each bin
  T *m_values;
```

ATLAS B field on GPUs

- The awkward way out:
  - Explicit allocation of grid values are done on device
  - Grid values are then copied from host to device
  - Inside CUDA kernel, explicitly make the member pointer of Grid object explicitly point to the grid values on device

```
GPUERRCHK(cudaMalloc(&d_propagator, sizeof(PropagatorType)));
GPUERRCHK(cudaMalloc(&d_opt, sizeof(PropagatorOptions)));
GPUERRCHK(cudaMalloc(&d_pars, nTracks * sizeof(TrackParameters)));
GPUERRCHK(cudaMalloc(&d_ress, nTracks * sizeof(PropResultType)));
GPUERRCHK(cudaMalloc(&d_gridValPtr, gridSize * sizeof(GridValueType)));

// Copy from host to device
GPUERRCHK(cudaMemcpy(d_propagator, &propagator, sizeof(propagator),
                     cudaMemcpyHostToDevice));
GPUERRCHK(cudaMemcpy(d_opt, &propOptions, sizeof(PropagatorOptions),
                     cudaMemcpyHostToDevice));
GPUERRCHK(cudaMemcpy(d_pars, pars.data(), nTracks * sizeof(TrackParameters),
                     cudaMemcpyHostToDevice));
GPUERRCHK(cudaMemcpy(d_ress, ress.data(), nTracks * sizeof(PropResultType),
                     cudaMemcpyHostToDevice));
GPUERRCHK(cudaMemcpy(d_gridValPtr, gridValPtr, gridSize * sizeof(GridValueType),
                     cudaMemcpyHostToDevice));

// Run on device
int threadsPerBlock = 256;
int blocksPerGrid = (nTracks + threadsPerBlock - 1) / threadsPerBlock;
propKernel<<<blocksPerGrid, threadsPerBlock>>>(
    d_propagator, d_pars, d_opt, d_ress, d_gridValPtr, nTracks);
```

```
// Device code
__global__ void propKernel(PropagatorType *propagator, TrackParameters *tpars,
                           PropagatorOptions *propOptions,
                           PropResultType *propResult, Vector3D *gridValPtr,
                           int N) {
  // Awkwardly make the grid values pointer to point to memeory on device
  // explicitly
  propagator->refStepper().refField().refMapper().refGrid().refValues() =
      gridValPtr;

  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < N) {
    propagator->propagate(tpars[i], *propOptions, propResult[i]);
   }
 }
```

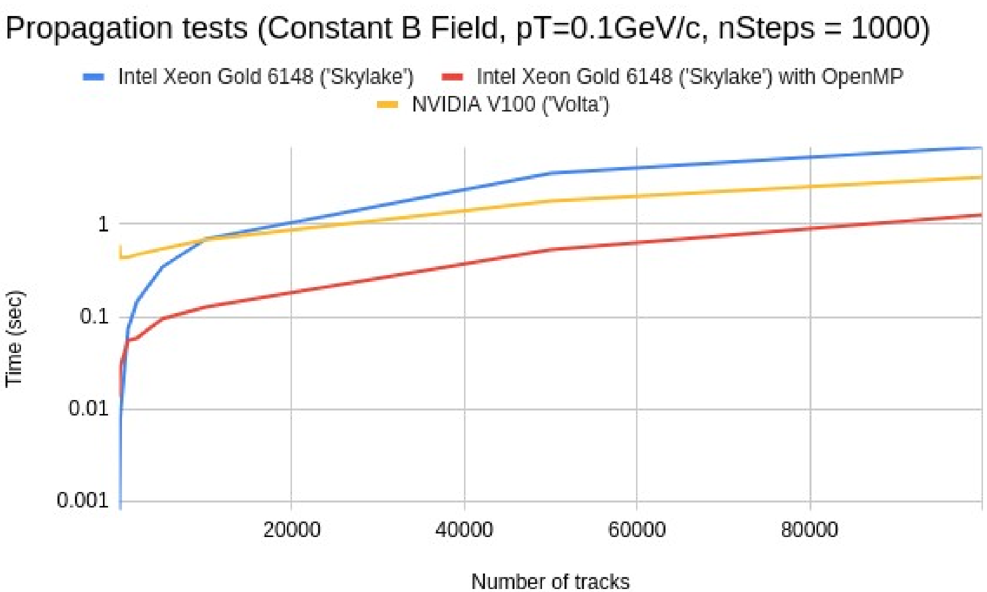
The results

- Results on GPUs are validated
- GPUs win over CPU@many threads execution when N_{tracks} > 30k at non-constant B field

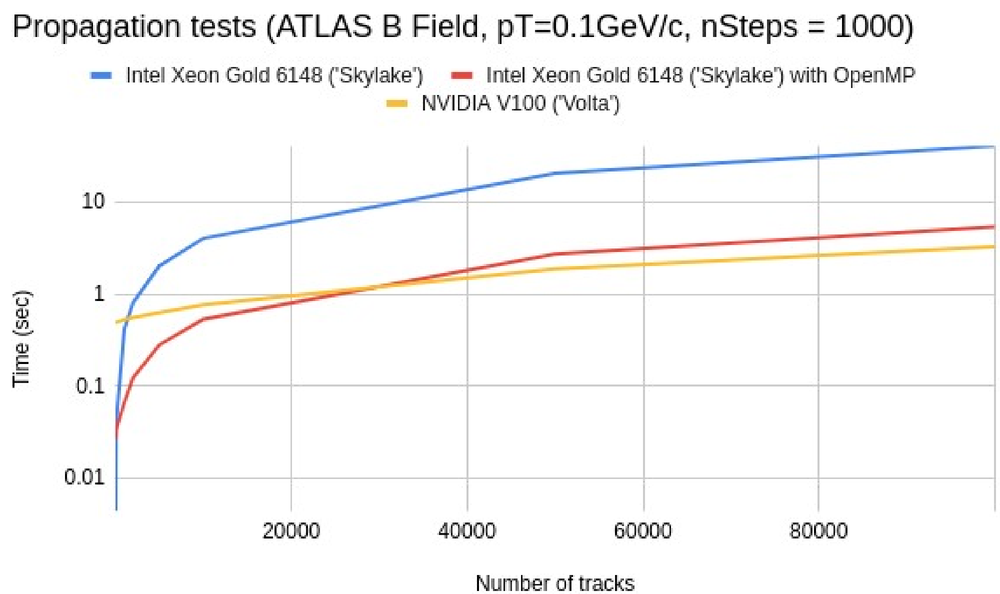
Validation of results with starting Pos = (0,0,0) and Mom = (1, 0, 1) GeV/c at ATLAS BField

prpagation positons on CPU	Propagation positions on GPU
v 0.383073 -2.92701e-05 0.383073	v 0.383073 -2.92701e-05 0.383073
v 0.766147 -0.000131716 0.766147	v 0.766147 -0.000131716 0.766147
v 1.14922 -0.000307337 1.14922	v 1.14922 -0.000307337 1.14922
v 1.53229 -0.000556133 1.53229	v 1.53229 -0.000556133 1.53229
v 1.91537 -0.000878105 1.91537	v 1.91537 -0.000878105 1.91537
v 2.29844 -0.00127325 2.29844	v 2.29844 -0.00127325 2.29844
v 2.68151 -0.00174158 2.68151	v 2.68151 -0.00174158 2.68151
v 3.06459 -0.00228308 3.06459	v 3.06459 -0.00228308 3.06459
v 3.44766 -0.00289775 3.44766	v 3.44766 -0.00289775 3.44766
v 3.83073 -0.00358561 3.83073	v 3.83073 -0.00358561 3.83073

Constant B Field



ATLAS B Field



Discussion

- Can we survive without STL containers and algorithms?
  - CUDA: thrust library
  - SYCL: SyclParallelSTL
- How to handle dynamic memory allocation?
  - Seems each relevant class needs to have a user-defined constructor/destructor-like cudaMalloc/cudaMemCpy method that calls those methods of its data members recursively
- Attempted to-do-list
  - Construct a simple geometry with a list of (planar) surfaces which could barely do local<=>global transforms and boundary check
  - Simplify the concept of measurement, track parameter and track state
  - Finally, work out a KalmanFilter with barely updater (and smoother)

In general, GPUs implementation of the whole ACTS seems not realistic and worthwhile