Optimize and Integrate Standalone Tracking Library (SixTrackLib)

Somesh Singh

Department of Computer Science and Engineering
Indian Institute of Technology Madras

September 04, 2018

Google Summer of Code 2018

Background

- `SixTrackLib` is a standalone particle tracking library.
- The particle accelerator is modelled as a sequence of *beam* elements.
- Store the properties of the beam elements sequentially to a chunk of memory --> description of the machine is serialized.
- *Tracking function* : models the change in the particles' properties due to a beam element.

Objective

- Implement a standalone minimal parallel version of `SixTrackLib`[1]
- Study the effect of various optimization strategies on the naïve parallel code.

- Used `C` and required `OpenCL 1.2` for the device side code.
- Used `C++` wrappers for the host side code.

[1] Source code on https://github.com/ssomesh/sixtracklib_gsoc18.git

Pseudo code for SixTrackLib

```
1  for( int t = 0; t < NUM_TURNS; ++t )
2  {
3    for( int particle_index = 0;
4              particle_index < NUM_PARTICLES;
5              ++particle_index )
6    {
7      for( int beam_elem_index = 0;
8                beam_elem_index < NUM_BEAM_ELEMENTS;
9                ++beam_elem_index )
10     {
11         beam_element = beam_elements[ii];
12         be_type = get_type( beam_element );
13         switch( be_type )
14         {
15           case DRIFT: // call to 'track_drift_particle'
16           case DRIFT_EXACT: // call to 'track_drift_exact_particle'
17           case CAVITY: // call to 'track_cavity_particle'
18           case ALIGN: // call to 'track_align_particle'
19         };
20     }
21   }
22 }
```

Parallelization strategy: Assign one work-item to each particle

Studies carried out

Studied the performance of the parallel implementation in the following scenarios:

1. The switch-case is inside the kernel; all tracking functions inside one kernel.
2. The switch-case is moved out of the kernels to the host.
3. The switch-case is removed, both, from the kernels and the host.

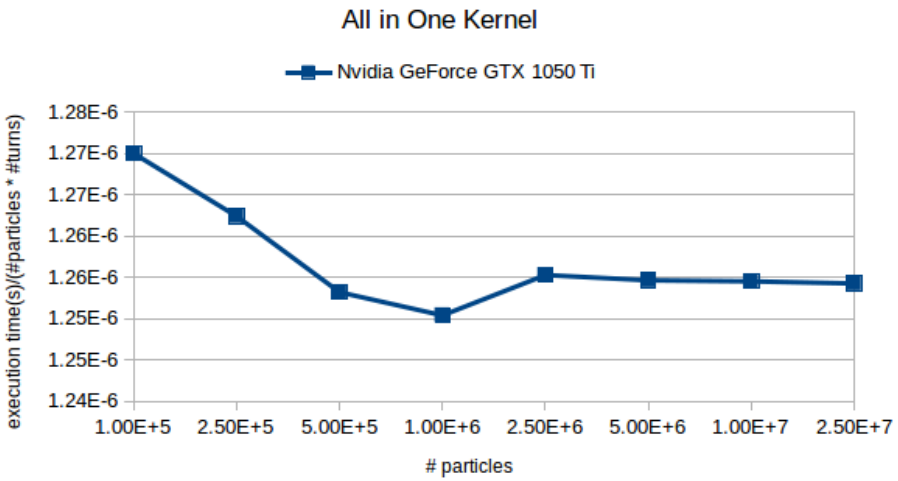
Experimental Setup

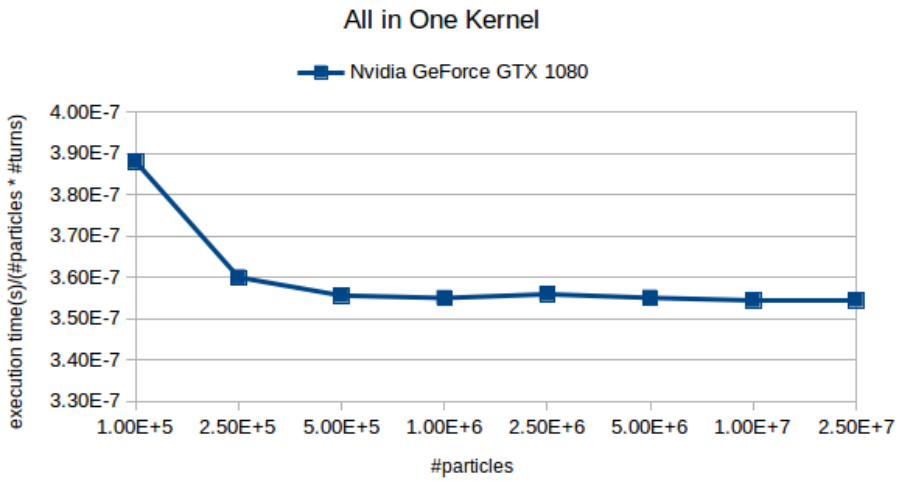
Hardware used for benchmarking our code:

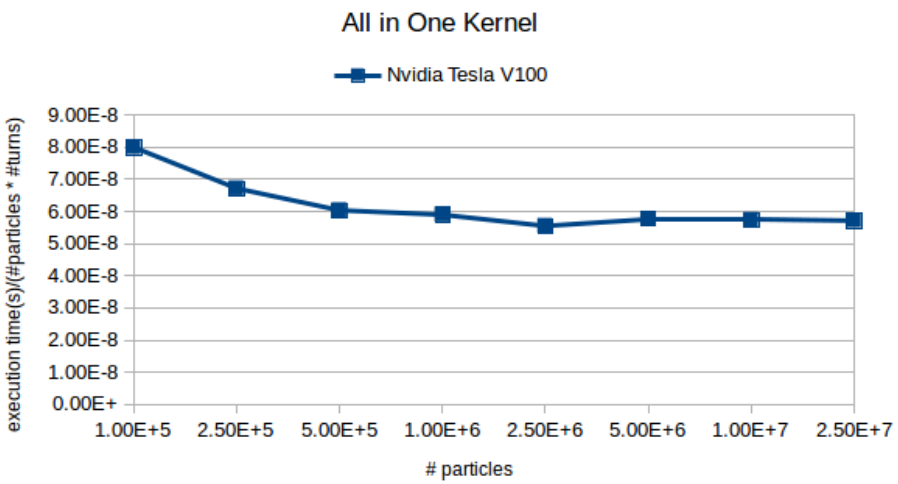
- `AMD Radeon RX 560 GPU`
- `Nvidia GeForce GTX 1050 Ti GPU`
- `Nvidia GeForce GTX 1080 GPU`
- `Nvidia Tesla V100 GPU`
- `Intel Xeon CPU E5-2640 v4 (40 core)`
- `AMD Ryzen7 1700X (8 core) CPU`

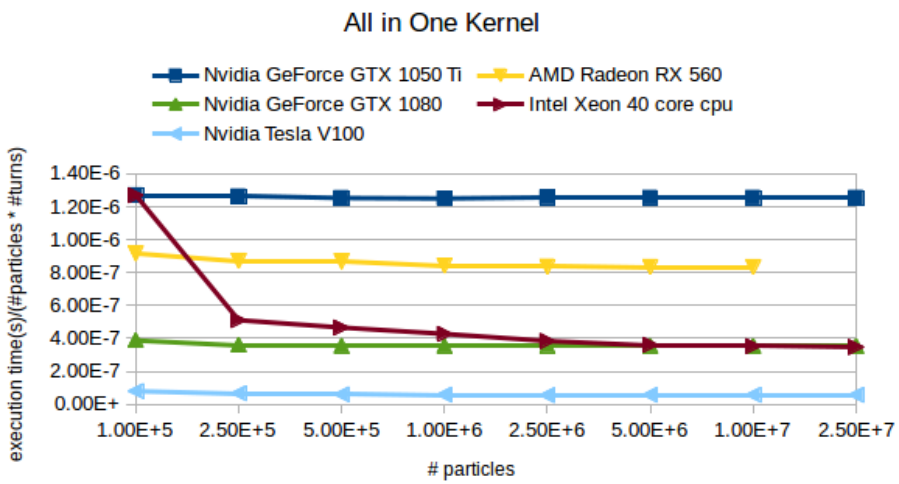
Results

Scenario 1: # particles in the range 10^5 – 2.5×10^7



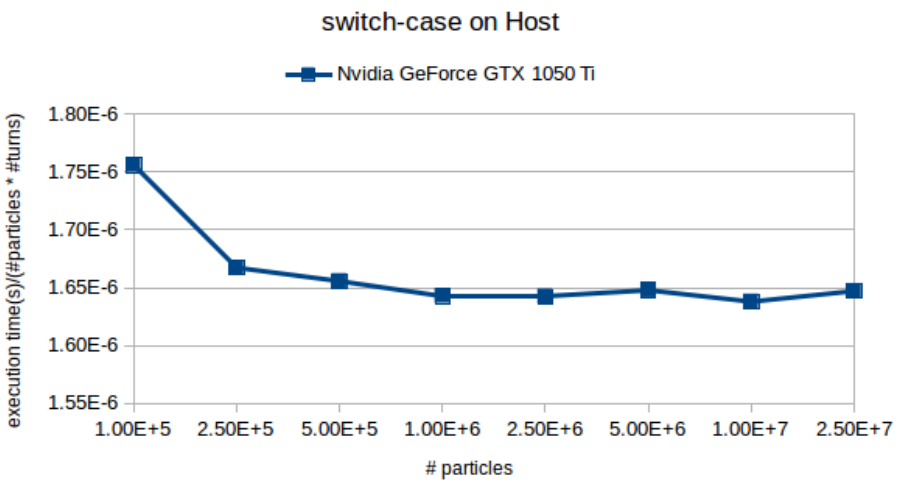


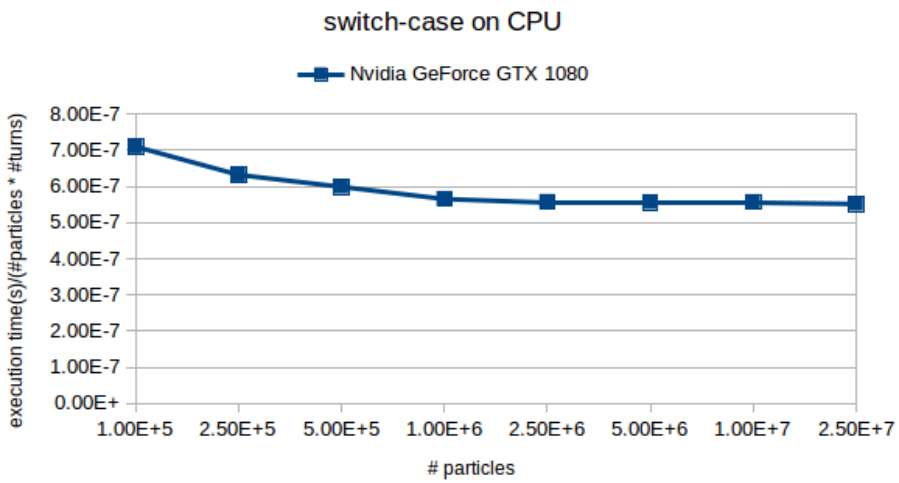


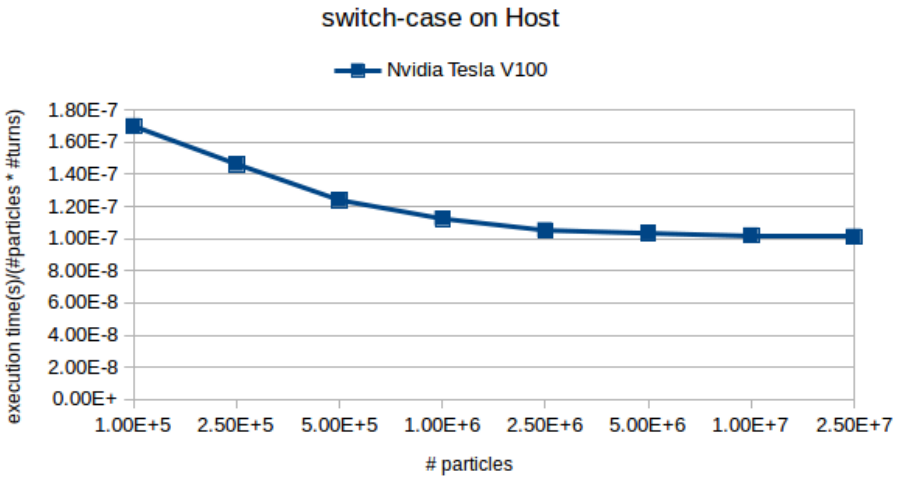


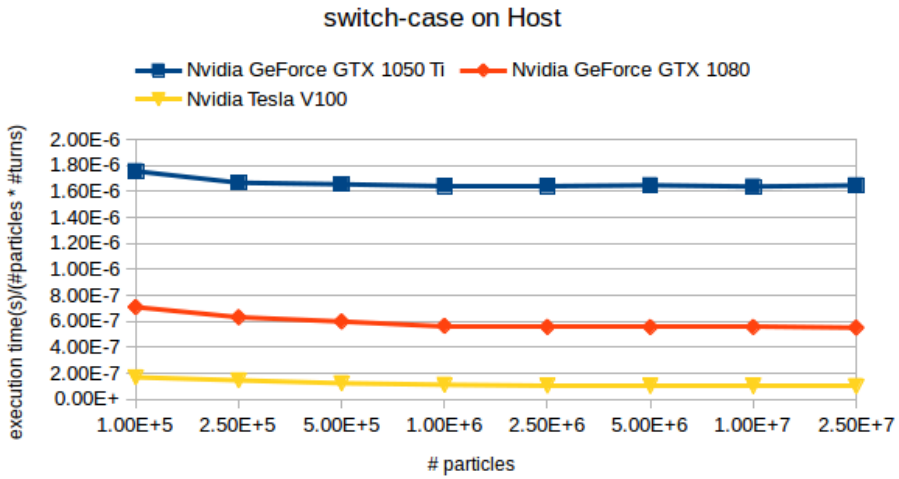
Results

Scenario 2 & Scenario 3 : # particles in the range 10^5 – 2.5×10^7









Observations

- In scenario 2, we split the monolithic kernel in scenario 1 into multiple kernels.
- Reduce the private memory used by each work-item and in turn reduce the register pressure.
- Move switch-case to CPU $\implies$ reduce thread divergence on the GPU.
- When kernels are called multiple times, there is a noticeable fluctuation in their execution times in the first few invocations, which we term as *warmup effect*.
- The average execution times in scenario 2 and scenario 3 are very similar.

Observations

- Observed slow down w.r.t scenario 1 $\implies$ benefits overshadowed by the overheads of launching the kernel and the branching on the CPU.
- Helpful when we deal with larger kernels having *enough* to do.
- The difference in the execution times for scenario 1 compared with scenario2 / scenario 3 gives a rough estimate of the overheads incurred from one case to the other.

Summary

1. Studied the effects on execution times of the parallel implementation of `SixTrackLib` under different scenarios.

2. Implemented an optimization strategy of breaking a monolithic kernel into many simpler kernels and provided a framework for applying such optimizations.

   `Advantages`:

   - Results in better *performance*.
   - *Interoperability* with codes that need to do expensive operations at different steps.

Summary

1. Studied the effects on execution times of the parallel implementation of `SixTrackLib` under different scenarios.

2. Implemented an optimization strategy of breaking a monolithic kernel into many simpler kernels and provided a framework for applying such optimizations.

   `Advantages`:

   - Results in better *performance*.
   - *Interoperability* with codes that need to do expensive operations at different steps.

Thank You