GSoC 2018 Project Report

Project: Optimize and Integrate Standalone Tracking Library (SixTrackLib).

Introduction

SixTrackLib is a standalone particle tracking library. In particle physics, the process of fitting particle tracks through observed interaction points is called "particle tracking". The particle accelerator is modelled as a sequence of beam elements. There are different types of beam elements in the scope of this project – Drift, Multipole, Cavity, etc. Each type of beam element has a set of properties that describe it – length, order of multipole, etc. By storing the properties of the beam elements sequentially to a chunk of memory, the description of the machine can be serialized. In other words, beam elements model the properties of the different regions of the particle accelerator. When a particle passes through a region of the accelerator, its physical properties, such as momentum, position etc. are changed depending on the region. The mapping or function that models the change in the particles' properties due to a beam element is a *tracking function* associated with that beam element.

Description

Note: The code for GSoC 2018 is hosted on https://github.com/ssomesh/sixtracklib_gsoc18/tree/master. Please read the README.md.

During GSoC 2018, I developed a standalone minimal parallel implementation of SixTrackLib. The implementation consists of four tracking functions: `track_drift_particle`, `track_drift_exact_particle`, `track_cavity_particle`, `track_align_particle`.

The code requires OpenCL 1.2 and uses C for implementation for the Device side and the C++ wrappers for the Host side.

The pseudo-code for the minimal implementation of the SixTrackLib code is as below:

```
for( int t = 0; t < NUM_TURNS ; ++t ) {
  for( int particle_index = 0; particle_index < NUM_PARTICLES;
     ++particle_index ) {
    for( int beam_elem_index = 0; beam_elem_index < NUM_BEAM_ELEMENTS;
       ++beam_elem_index ) {
      beam_element = beam_elements[ii];
      be_type = get_type( beam_element );
      switch( be_type ) {
        case DRIFT: // call to 'track_drift_particle'
        case DRIFT_EXACT: // call to 'track_drift_exact_particle'
        case CAVITY: // call to 'track_cavity_particle'
        case ALIGN: // call to 'track_align_particle'
      };
    }
```

```
  }
}
```

Listing 1: SixTrackLib – pseudo code

From Listing 1, we observe that each particle can be processed (i.e. tracked) in parallel with the other particles. Hence, in the parallel implementation we assign one work-item to each particle.

We studied the performance of the parallel implementation in four scenarios:

1. The switch-case is inside the kernel
   (a) All tracking functions are inside one kernel and the kernel is invoked once from the host.
   (b) There is one kernel for each of the tracking functions. The tracking function corresponding to each of the beam elements is called (applied to) in a sequence for all the particles from the CPU.
2. The switch case is moved out of the kernels to the host. There is one kernel for each of the tracking functions. All type of beam elements are considered, so all kernels are called appropriately.
3. The switch case is removed from the host as well as from the kernels. The beam elements of each type are put serially and the appropriate tracking kernels are invoked for the beam elements for each of the particles.

Experimental Setup

We wanted to write performance portable code that runs reasonably well on different platforms.

We tested the code on various multicore and manycore platforms, viz :

- AMD Redeon 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 Ryzen [1] 7 1700X (8 core) CPU

With the recent proliferation of GPUs and multicore CPUs, testing the performance on these platforms is relevant.

[1]The CPU series from AMD is **Ryzen** not Rayzen

Results and Discussion

`Note`: The execution times presented hereafter are measured over 10 runs of the concerned block of code. The execution time reported is the *arithmetic mean* of the execution times of *last* 5 runs of the code-block. This is done to reduce the effect of caching and other hardware specific issues so that the execution times we measure are stable.

We time the execution times of the OpenCL kernels using Event based profiling APIs provided by OpenCL.

The host side code snippets are timed using the *gettimeofday* function provided in C and C++.

The results for each of the aforementioned scenarios is discussed below.

Scenario 1

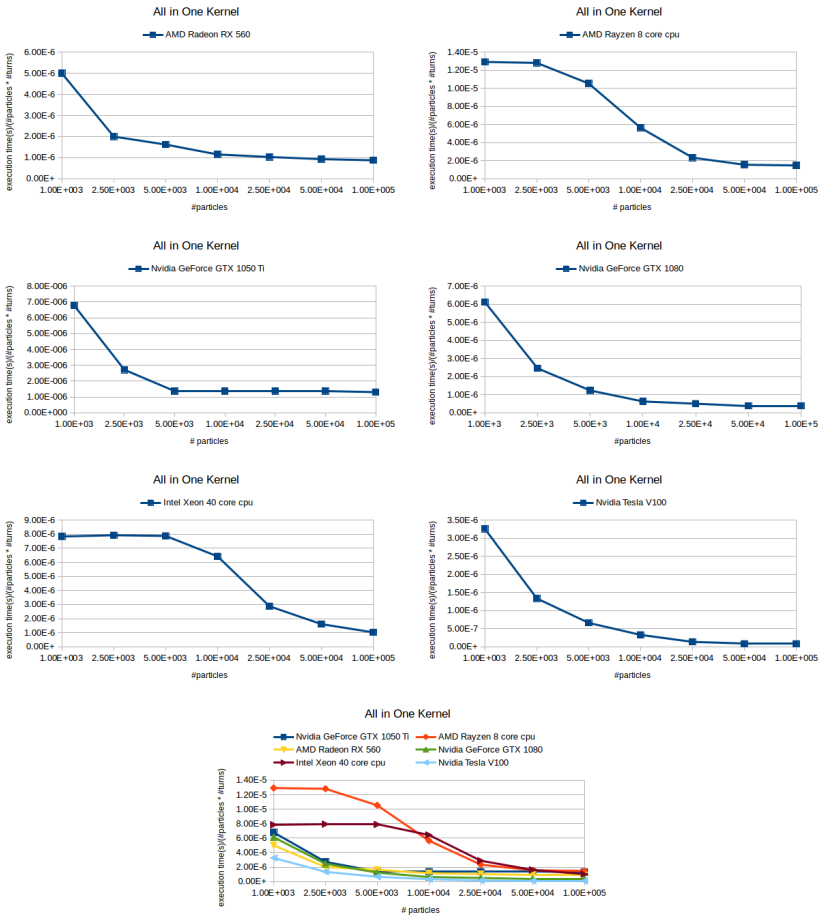

Figure 1: Scenario 1(a): number of particles in the range 10^3 – 10^5

Figure 1 and Figure 2 show the average execution time of the regular kernel, containing all

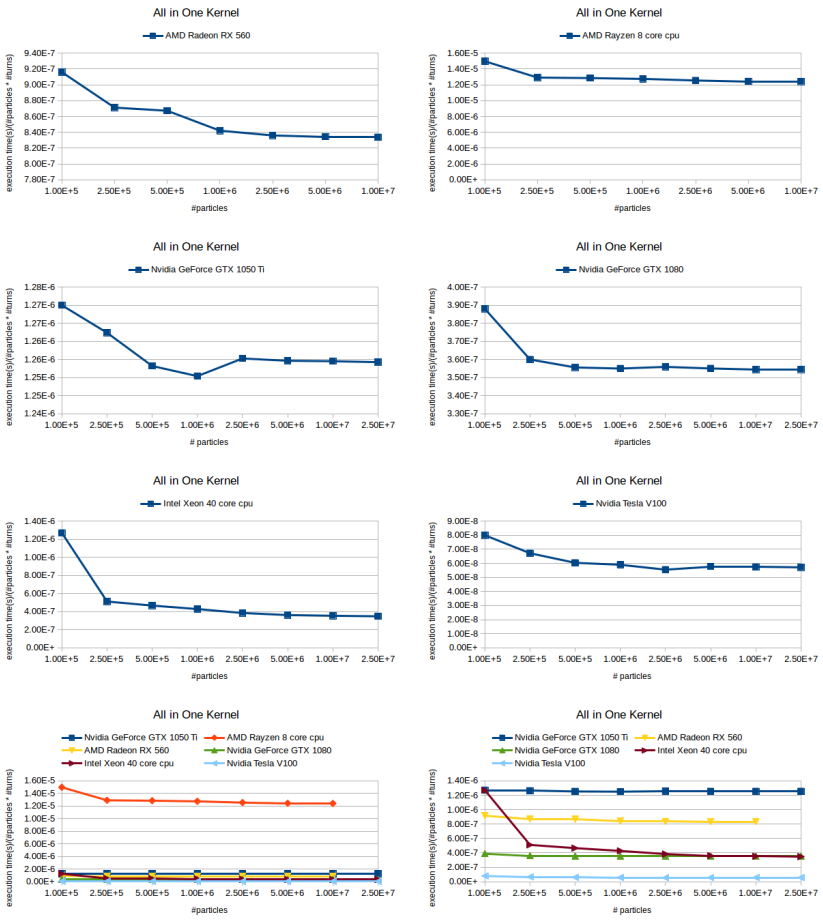

Figure 2: Scenario 1(a): number of particles in the range $10^5 - 2.5 * 10^7$

tracking functions, on various platforms. The y-axis of the plots is the normalized execution time expressed as $\frac{\text{execution time}}{\#\text{particles} \times \#\text{turns}}$.

Please note that in scenario 1(a), the workgroup-size on the various platforms is as follows:

- AMD Redeon RX 560 GPU: 256
- Nvidia GeForce GTX 1050 Ti: 896
- Nvidia GeForce GTX 1080: 896
- Nvidia Tesla V100: 896
- Intel Xeon CPU E5-2640 v4 (40 core): 40
- AMD Ryzen 7 1700X (8 core): 16

`Warm-up effect`:

We observe that when kernel(s) is(are) called multiple times, there is a noticeable fluctuation in their execution times in the first few invocations. There after, the execution times stabilize. We term this as the *warm-up* effect. For example, consider the execution times of the tracking kernels when these are called in a sequence multiple times.

A representative example from scenario 1(b) on Nvidia GeForce GTX 1080:

`track_align_particle`: 5.539s, 5.542s, 5.548s, 5.554s, 5.555s, 5.558s, 5.556s, 5.555s, 5.560s, 5.558s

Here, we observe that the first few (5) execution times for the `track_align_particle` vary a bit, but these stabilize as we call the set of tracking kernels multiple times.

This is primarily because of the caching effects and other hardware issues/properties.

We compensate for the `warm-up` effect by timing the code-block over 10 runs. We discard the time values obtained from the first 5 runs and take an average of the execution times in the last 5 runs. We consider the average so computed as the average execution time of the code-block or kernel.

One caveat is that the warm up effect is seen for the first runs of each kernel independent of the other kernels. For some kernels, the execution time tends to settle earlier than others. However, in our case we found the values to stabilize after 5 calls. This can be considered as a tunable parameter that can be determined empirically.

Scenario 2:

Figure 3 and Figure 4 show the average execution time of the code block when the switch-case construct has been moved to the CPU. There is a separate kernel defined for each of the tracking functions. The y-axis of the plots is the normalized execution time expressed as $\frac{\text{execution time}}{\#\text{particles} \times \#\text{turns}}$.

Please note that in scenario 2, the workgroup-size on the various platforms is as follows:

- Nvidia GeForce GTX 1050 Ti: 1024
- Nvidia GeForce GTX 1080: 1024
- Nvidia Tesla V100: 1024

In Scenario 2, we split the monolithic kernel in Scenario 1(a) into multiple kernels. We do this in an attempt to reduce the private memory used by each work-item and in turn reduce the register pressure. The second objective was to move the switch-case construct from the GPU to the CPU to reduce thread divergence on the GPU. The results show an average slow-down (a negative speedup) in the range ($1.6\times - 2\times$) when we move from scenario 1(a) to scenario 2.

Our claim of reduced thread-private memory for smaller kernels is supported by the work-group size in the scenario 1(a) and scenario 2. The smaller maximum work-group size (896) on the Nvidia GPUs in scenario 1 in comparison to the maximum work-group size (1024) in scenario 2 hints at the fact that the larger kernel uses more registers than the smaller kernels.

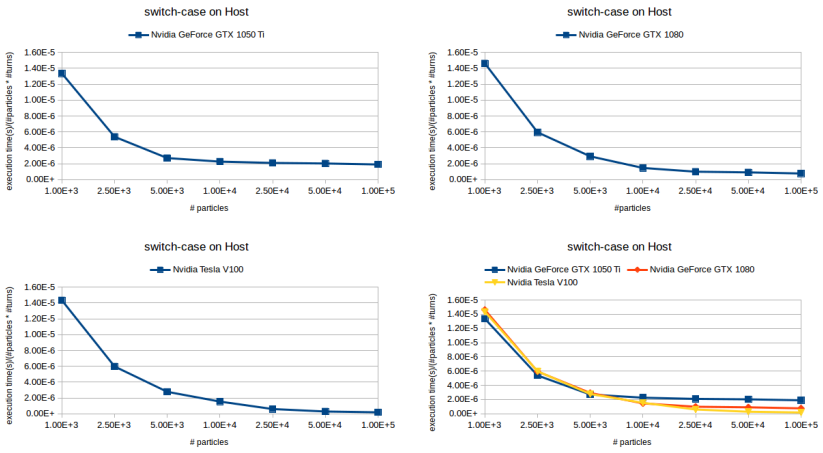

Figure 3: Scenario 2: number of particles in the range 10^3 – 10^5

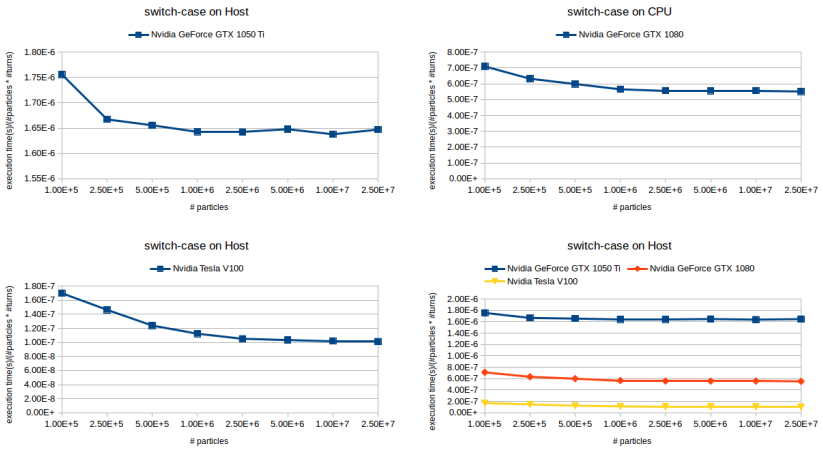

Figure 4: Scenario 2: number of particles in the range 10^5 – $2.5 * 10^7$

The observed slow down may step from the fact that the tracking functions considered here are quite simple and so the benefits from launching the kernels are overshadowed by the overheads of launching the kernel and the branching (through switch-case constructs) on the CPU.

Although in the specific case of the kernel at hand, we do not observe any gains in speedup upon splitting the kernel, this strategy would be helpful when we deal with large kernels having *enough* to do.

In addition, by considering and implementing scenario 2, we provide a framework of sorts for applying the optimization of splitting the kernel into multiple smaller ones. One reason for doing so is potential performance, the other is *interoperability* with codes that need to do expensive operations at each step.

Scenario 3

```
for(int t=0; t < NUM_TURNS; ++t) { // on the cpu
  for(int n=0; n < 250; ++n)
    track_drift_particle // tracking kernel for drift beam element
  for(int n=250; n < 500; ++n)
    track_drift_exact_particle // tracking kernel for drift_exact beam
       element
  ...
}
```

Listing 2: Scenario 3 – pseudo code

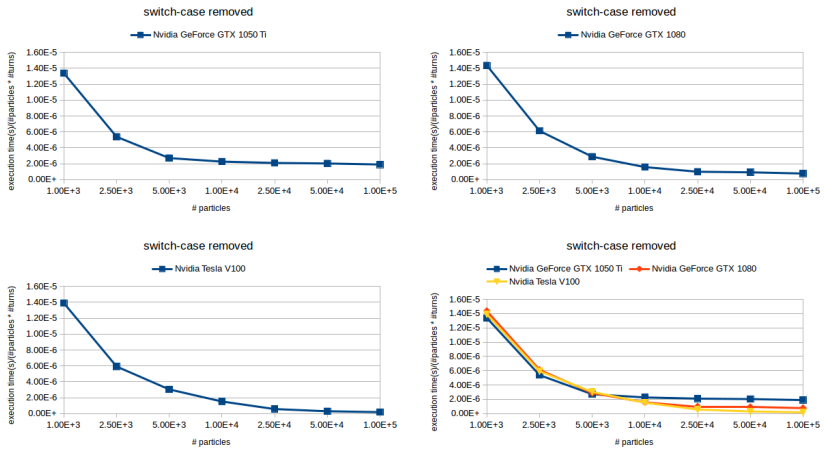

Figure 5: Scenario 3: number of particles in the range 10^3 – 10^5

Figure 5 and Figure 6 show the average execution time of the code block when the switch-case construct has been removed from the CPU. There is a separate kernel defined for each of the tracking functions. The beam elements are arranged in a serial fashion. The various

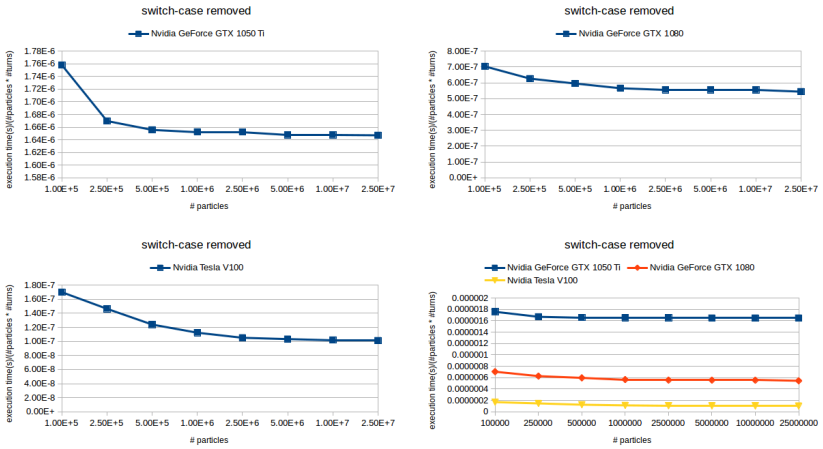

Figure 6: Scenario 3: number of particles in the range $10^5 - 2.5 * 10^7$

tracking kernels are invoked corresponding to the beam elements. The y-axis of the plots is the normalized execution time expressed as $\frac{\text{execution time}}{\#\text{particles} * \#\text{turns}}$.

The pseudo code for scenario 3 is shown in Listing 2.

Please note that in scenario 3, the workgroup-size on the various platforms is as follows:

- Nvidia GeForce GTX 1050 Ti: 1024
- Nvidia GeForce GTX 1080: 1024
- Nvidia Tesla V100: 1024

An important point is that the difference in the execution times for scenario 2 and scenario 3 gives a pessimistic upperbound on the overheads incurred in the two cases. The plots in Figure 7 and Figure 8 show a comparative study of the execution times in the two scenarios.

Conclusion

We study the effects on execution times of the parallel implementation of SixTrackLib under various scenarios. We tried out an optimization strategy of breaking a monolithic kernel into many simpler kernels and provide a framework for applying such optimizations in other cases as well.

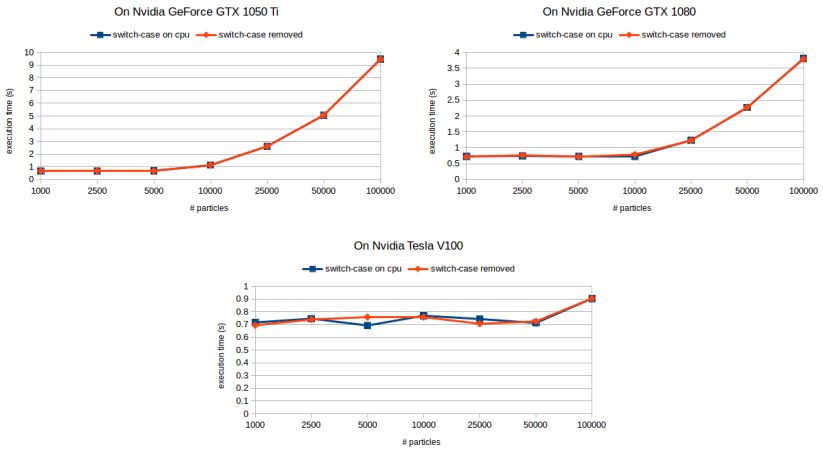

Figure 7: Comparison of execution times for studying overheads: number of particles in the range 10^3 – 10^5

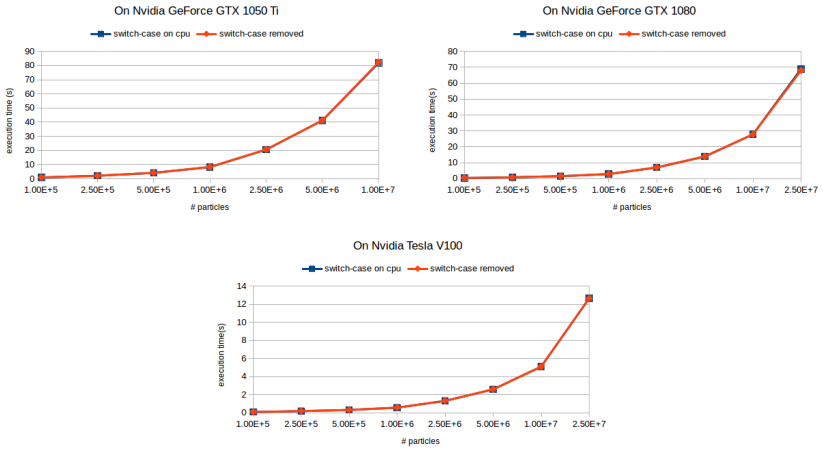

Figure 8: Comparison of execution times for studying overheads: number of particles in the range 10^5 – $2.5 * 10^7$