Clara's ATLAS Thesis

Clara Fleisig

June 20, 2024

1 Preventing Compiler Optimizations

1.1 Motivation

As mentioned in **??**, data stored in ROOT files were used to test potential solutions. These files are a proxy for a live incoming data stream which would exist if the solution were implemented.

A potential complication is that a C++ compiler may perform optimizations during compilation that rely on data at specific addresses already being known. For example, if identical solutions was tested twice with the same hits in a compiled program, the solution may perform better on the second test. This is because the values at specific addresses may already be known and stored on the stack from when the first test was executed, whereas the first test needed to retrieve those same values from the heap. It is convenient to be able to test the same solution multiple times within a single program for evaluating how the solution's performance scales with the size of data, Monte Carlo simulations, and testing multiple solutions in the same program. Thus, this chapter outlines a strategy for avoiding undesirable compiler optimizations. In particular, how to test a program that tests runtime scaling with the number of ATHENA hits, and performs Monte Carlo simulations without encountering unwanted compiler optimizations.

1.2 Setup

To test compiler behaviour I implemented a simple program intended to search for randomly generated hits. To prepare for tests the program performed the following:

1. Create a ROOT file which contains vectors of random target hits. These hits are samples of a uniform distribution of z, ρ and ϕ covering the range of the entire ItK detector.

2. Read target hits from ROOT file from generated in the previous step into a vector of Hit objects. Each hit object stores the x-coordinate, y-coordinate and z-coordinate of the target hit.

3. Read ATHENA hits from ROOT file into a vector of Hit objects like the ones used for the target hits.

The search algorithm's goal was to find the closest ATHENA hit for a given target hit, see Listing 1 for the implementation code.

```
1    Hit_org::Hit closest_hit(0, 0, 0);
2    float min_dis = std::numeric_limits<float>::max();
3
4    for(int i = n_hits; i>n_hits; i++){
5        auto cur_dis = calc_dis(hit_vec.at(i), targ_hit);
6        if(cur_dis < min_dis){
7            min_dis = cur_dis;
8            closest_hit = hit_vec.at(i);
9        }
10   }
```

Listing 1: Test Search Algorithm

This code calculates the euclidean distance between the target hit and the first *n_hits* stored in *hit_vec*, where *hit_vec* stores ATHENA hits. If the newly calculated euclidean distance is less than the current value in *min_dis*, the current value of *min_dis* is replaced by the newly calculated value. *closest_hit* is also correspondingly updated. Since this algorithm needs to read the value of

every ATHENA hit, it is predicted to have an of $O(n)$ runtime efficiency, where n is the number of ATHENA hits searched (n_hits).

First, the simple but naive approach to test the scaling of runtime efficiencies was investigated. This approach is to reuse the same vector of target hits and ATHENA hits every time the algorithm's runtime is evaluated. To test the scaling of the algorithms runtime, the number of hits searched (n_hits) was varied.

1.3 Results

1.3.1 Reusing the Target and ATHENA Hit Vectors