CS 5220 Project 3 – Team 6 Mid Report

Taejoon Song, Junteng Jia, Joshua Cohen
{ts693, jj585, jbc264}@cornell.edu

November 12, 2015

1 INTRODUCTION

The Floyd–Warshall algorithm, in computer science, is an alogorithm for finding the minimum patahs in a weighted graph. (i.e. All-pairs shortest path problem)
Our goal in this assignment includes:

1. Profiling: To find the bottleneck of the code.
2. Paralleization : To make it paralle by using MPI.
3. Optimization (Tuning) : Finally, tune it aggressively to get highest performance.

The rest of the report is organized as follows. In Section 2, we introduce a baseline timing result from initial copy, and show our profiling result. Section 4 discusses our approach for parallelization-related work and Section 3 discusses vectorization. Our evaluation result will be shown in Section 5. Finally, Section 6 suggestes what should be done more after peer reviews.

2 TIMING

2.1 PROFILING

In order to find the bottlenecks of path, we first profiled our code using Intel's VTune Amplifier (It was broken in cluster, so we used it on local machie), as shown in Figure 1.

```
amplxe-cl -collect advanced-hotspots ./path
amplxe-cl -report hotspots -source-object function=<NAME>
```

Figure 1: VTune Amplifier Command

2.2 Initial Profile Result

Initial profile result can be found at Figure 2, and more detail in Figure 3.
We found that "square" function is the main bottleneck and do the most critical steps for this program, so this point is where we started to tune the code.

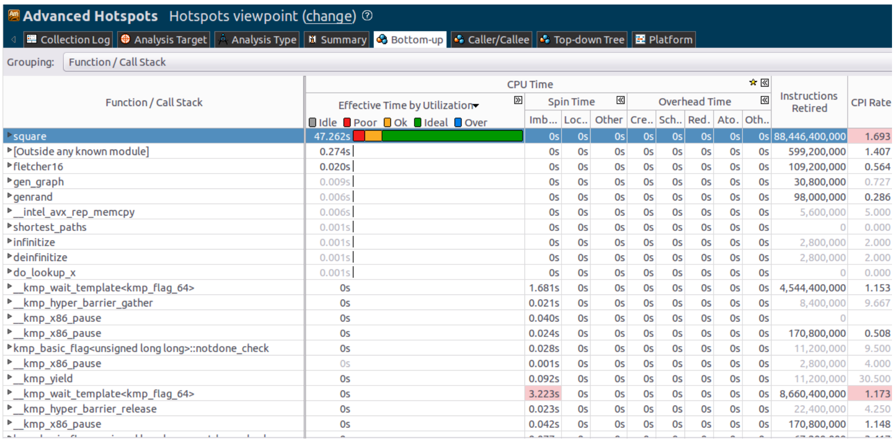

Figure 2: Initial Profile Analysis

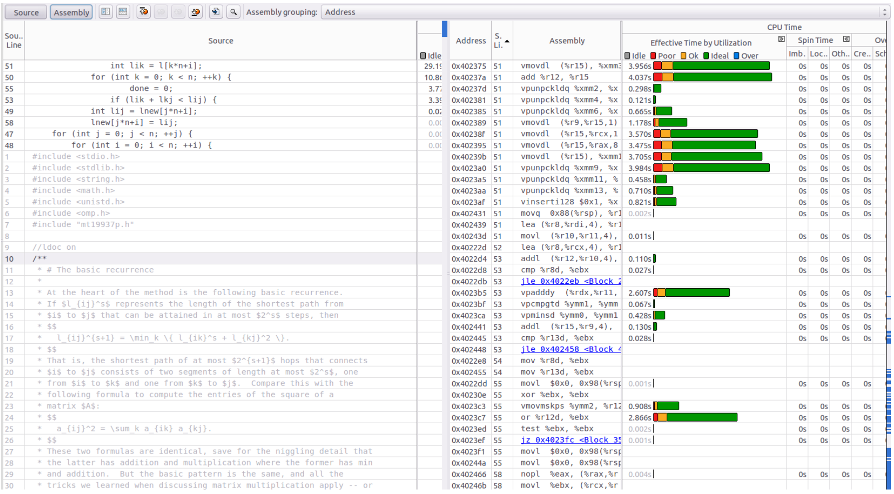

Figure 3: Initial Assembly Result

2.3 Initial Timing Result

Initial timing result is shown in Figure 4. As we can see the program is running with 8 threads using OpenMP and it takes 11.0818 seconds for 2000 vertices.

```
justin@Dell-XPS15:~/Documents/courses/parallel/path$ ./path.x -n 2000
== OpenMP with 8 threads
n:      2000
p:      0.05
Time:   11.0818
Check:  E16C
```

Figure 4: Initial Timing Result

3 Vectorization

In order to vectorize properly vectorize our code, we looked at the output of `ipo_out.optrpt` after compiling with flags `-qopt-report=5 -qopt-report-phase=vec`. We first were able to vectorize our call to `square` within `shortest_paths` within `path.c` by explicitly precomputing the transpose of `l` during each call to `square`, and then replacing the assignment of `lik` directly from `l`, as

```
int lik = l[k*n+i]
```

to an assignment instead from the transpose, as

```
int lik = l_T[i*n+k]
```

We also attempted to solve the issue of unaligned memory access from within `l` and `l_T` by replacing calls to `malloc` with `_mm_malloc` (and, correspondingly, calls to `free` with calls to `_mm_free`), using a byte alignment of 32 since we're compiling with AVX2 (using the flag `-xcore-avx2`). This solved some of the issues with unaligned access, according to the vectorization report, but there are still cases with unaligned access reported.

4 Parallelization

Parallel..

5 Evaluation

5.1 Timing After Vectorization

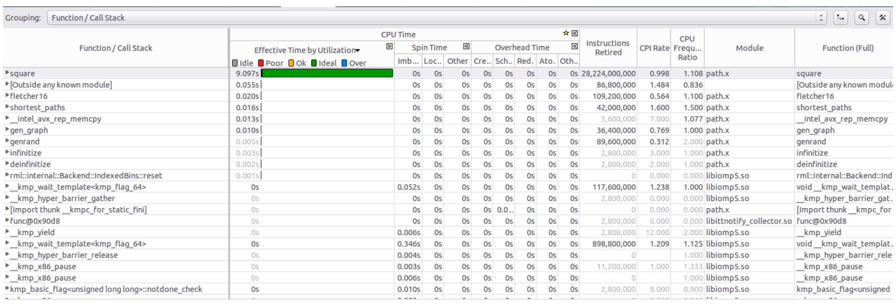

Figure 5: Profile Analysis After Vectorization

As show in the picture, for the same problem size, the function `square` is more than 5 times faster than it was before, when we are running it with in Vtune, where one core is used to trace other threads. In fact, running it with command line and time it with build in function `omp_get_time()` shows the saving gives us a factor of more than 9 times faster.

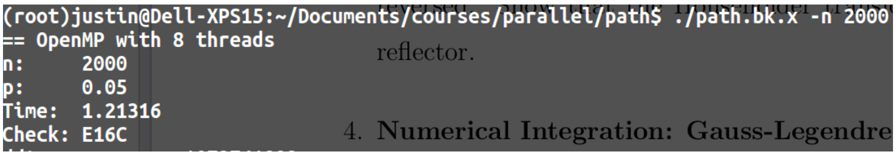

Figure 6: Timing Result After Vectorization

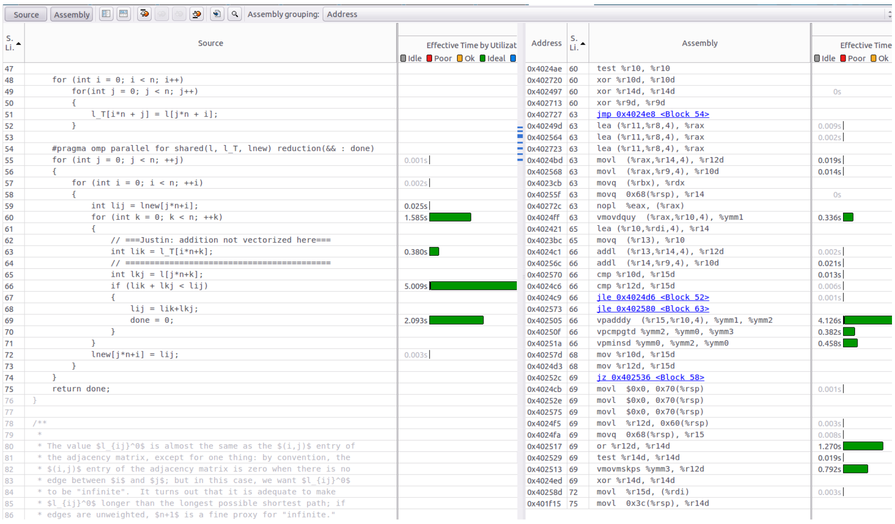

Figure 7: Assembly Analysis After Vectorization

The assembly analysis is more informative as it tells how well our functions have been vectorized as well as the timing. From the assembly analysis after vectorization, we can tell that the computations in the inner has been vectorized. Which is where the savings come from.

5.2 Data-Type Optimization

As we went through tests, we found that the maximum distance between two vertices is always below 20, even when we are calculating a 10000 by 10000 graph. Then it naturally follows that, we don't necessarily need a 4-byte int to store the distance information.

To fit more data into register so that we can carry out more operation per cycle, we used a prototype called ddt, which can be any data type such as long, short, and char. In fact, this gives us 30% saving when we are using char to store the distance between two vertices.

Notice in Figure 8, we printed out a variable called `ddt_upper_range` which tells us the largest distance we can have, in order to use one certain data-type. In the case with char, the largest distance is 127, which is proved to be more than enough by our tests.

5.3 Midterm Profile Result

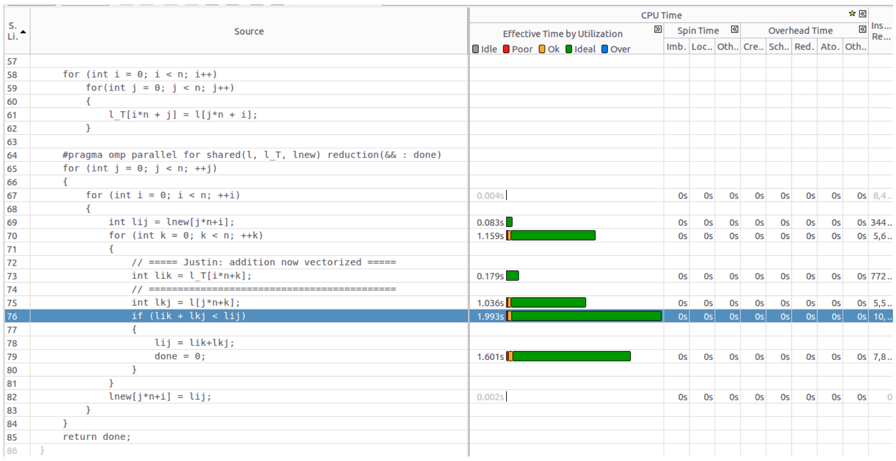

Figure 8: Initial Assembly Result

The assembly analysis after using data-type char is down below. Comparing with Figure 7, we can tell that the major saving comes from the addition between `lij` and `ljk`, which consist with what we would expect, since we can now fit more additions into vector register to be calculated in one cycle.

5.4 Initial Timing Result

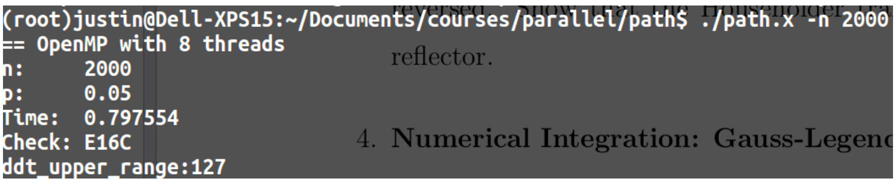

Figure 9: Midterm Timing Result

Midterm timing result is shown in Figure 9. As we can see the program is running with 8 threads using OpenMP and it takes 0.7976 seconds for 2000 vertices.

6 Future Work

- **Vectorization.** According to the vectorization report, we fixed most of the issues with loops not being properly vectorized. We still are having some issues with unaligned memory accesses, so we need to make sure that we're using the correct byte alignment and figure out how to properly make our indexing and memory accesses properly aligned. However, we are planning on focusing mainly on our MPI implementation.

- **Work Minimization.**
- **Compile Time Sizing.**
- **Blocking.**