HW4 [UNAMED AI MODEL] Optimization

Coen Petto

***Abstract*—This is a short and sweet application of my knowledge of OpenMP and code exploration to make [UNAMED AI MODEL] run faster.**

I. Introduction

THIS was performed on MAGGOT the CS machine with an Intel(R) Xeon(R) E-2278G CPU @ 3.40GHz and 8cores/16threads. My metrics were collected in the middle of the night, with no other users on the machine.

- This code was compiled with the POLYBENCH 4.2.1 harness, without RESTRICT, and with GCC 13.2 -O3.
- The three versions of [UNAMED AI MODEL] labeled "core", "pragma", and "super_pragma" are all implement a vareity of OpenMP pragmas.

Optimizations were made to only 3 different methods among all of the [UNAMED AI MODEL] code, and on a certain section for each one. The section varied between the three methods is displayed below. This is the GEMM loop inside of the Linear_layer_ds0, Linear_layer_ds1, and Linear_layer_ds2. To find out that these were important methods in terms of [UNAMED AI MODEL] inference you must first understand basic neural networks. Dense layers are always the most computationally demanding, and make sense of ALL attention, keys, and context collected by all data points. Secondly, you must understand that within these methods the complex reduction loop similar to GEMM cannot be trivially vectorized without the help of OpenMP. Lastly you can verify that these methods account for over 50 percent of the execution by commenting function calls manually and tracking the time differences.

```
l_gemm_i15:
  for (int i15 = 0; i15 < cst_0; i15 = (i15 +
      1))
    {
l_j14:
     for (int j14 = 0; j14 < cst_3; j14 = (j14
         + 1))
       {
l_S_k_0_k4:
        for (int k4 = 0; k4 < cst_1; k4 = (k4
            + 1))
          {
            v154[i15][j14] = (v154[i15][j14] +
                (v151[i15][k4] *
                v152[j14][k4]));
          }
       }
    }
```

Listing 1. core

```
#pragma scop
l_gemm_i9:
  #pragma omp parallel for
  for (int i9 = 0; i9 < cst_0; i9 = (i9 + 1))
    {
l_j9:
     for (int j9 = 0; j9 < cst_1; j9 = (j9 +
         1))
       {
l_S_k_0_k3:
        for (int k3 = 0; k3 < cst_1; k3 = (k3
            + 1))
          {
            v87[i9][j9] = (v87[i9][j9] +
                (v84[i9][k3] * v85[j9][k3]));
          }
       }
    }
#pragma endscop
```

Listing 2. pragma

```
#pragma scop
l_gemm_i9:
  #pragma omp parallel for
  for (int i9 = 0; i9 < cst_0; i9 = (i9 + 1))
    {
l_j9:
     #pragma omp parallel for
     for (int j9 = 0; j9 < cst_1; j9 = (j9 +
         1))
       {
l_S_k_0_k3:
        for (int k3 = 0; k3 < cst_1; k3 = (k3
            + 1))
          {
            v87[i9][j9] = (v87[i9][j9] +
                (v84[i9][k3] * v85[j9][k3]));
          }
       }
    }
#pragma endscop
```

II. Method

All experiments are driven by my `tuning.py` script.

The tuning script displays the times from the 3 different c files.

III. Command Reference and Sample Output

Command:

```
(base) maggot:~/CS553/[CS553] HW4 Coen
    Petto/[UNAMED AI MODEL]_package$ python3
    tuning.py
```

Example Output:

```
Output for [UNAMED AI MODEL]_core.c:
0.072928
Output for [UNAMED AI MODEL]_core_pragma.c:
```

```
0.024460
Output for [UNAMED AI
    MODEL]_core_super_pragma.c:
0.024521
```

IV. Coen's Analysis

Applying OpenMP to this specific GEMM loop inside of the dense layers created a very significant difference in execution time. The speedup in question is roughly 3.5–4×, with the baseline [UNAMED AI MODEL]_core.c running in 0.072928 seconds, and the optimized versions [UNAMED AI MODEL]_core_pragma.c and [UNAMED AI MODEL]_core_super_pragma.c running in 0.024460 and 0.024521 seconds respectively. Most surprisingly, there didn't seem to be a benefit to adding more parallelization beyond just the I-loop. The super_pragma version, which attempted to parallelize the J-loop inside each I-thread, performed basically the same as only I-loop optimizations. This suggests that for the given problem size and hardware, further parallelization may not scale effectively due to thread overhead or memory bandwidth limits.

V. Conclusion

Applying OpenMP to the key GEMM loops in [UNAMED AI MODEL]'s dense layers can yield substantial performance improvements. Parallelizing the outer loop alone was sufficient to reduce execution time significantly, while further nested parallelism provided minimal additional benefit.