Documentation of 2018 ECE Summer Research Internship:

**GPU Implementation of Triangle Counting Algorithm without Matrix Multiplication**

Minji Kim

minjik4@andrew.cmu.edu

I. Introduction

Over the course of 6 weeks from July 2nd, 2018 to August 10th, 2018, I have completed a research on GPU Implementation of triangle counting algorithm. The triangle counting algorithm has been developed for a submission for 2017 Graph Challenge. Summary of the algorithm can be found in the paper “First look: Linear algebra-based triangle counting without matrix multiplication,” written by a team of Carnegie Mellon University researchers led by Professor Low. The core mission of this research was to discover and analyze the performance of the triangle counting algorithm when implemented in GPU.

II. CUDA Numba – Sparse Matrix

The first implementation was done using CUDA Numba. CUDA Numba supports easier GPU programming by compiling Python code into CUDA kernels and device functions used in CUDA execution model. Due to the lack of experience in C programming, I decided to familiarize myself with CUDA programming model and GPU architecture by writing kernels using CUDA Numba.

In order to execute files using CUDA Numba, a few installations are necessary on the GPU. All of these can be installed directly from the terminal window using the corresponding commands listed in installation guides linked below.

* Anaconda: <https://conda.io/docs/user-guide/install/index.html>
* Cuda toolkit: <https://anaconda.org/anaconda/cudatoolkit>
* Numba: <http://numba.pydata.org/numba-doc/0.13/install.html>

After installing all necessary components of CUDA Numba, one will be able to execute the following implementation of triangle counting algorithm with sparse adjacency matrices as the input of the kernel.

|  |  |
| --- | --- |
| File Name | count\_triangles.py |
| Execution Command | $ python count\_triangles.py |
| Profiling Command | $ /usr/local/cuda/bin/nvprof python count\_triangles.py |

III. CUDA Numba – CSR Matrix

The next implementation was also done using CUDA Numba. However, unlike the first implementation, the inputs of the CUDA kernels were now in the form of CSR matrices. CSR matrices consist of IA and JA matrices, which are used to convey information of a single adjacency matrix. The code for the implementation can be found in the following file.

|  |  |
| --- | --- |
| File Name | count\_triangles\_CSR.py |
| Execution Command | $ python count\_triangles\_CSR.py |
| Profiling Command | $ /usr/local/cuda/bin/nvprof python count\_triangles\_CSR.py |

IV. CUDA C – CSR Matrix

The final implementation of the algorithm was accomplished by writing the code in pure CUDA C. I decided to compare the results from CUDA Numba CSR implementation and CUDA C CSR implementation to see if the Numba compiling process had any significant effect on the execution time. The basic algorithm is identical to the one used for CUDA Numba implementation. However, some modifications were necessary in order to optimize its performance when implemented with CUDA C. One major change is all components of the algorithms were stored in the form of pointers, which is unavailable in Python language, which accelerates memory access time and array operations.

|  |  |
| --- | --- |
| File Name | count\_triangles\_C.cu |
| Execution Command | $ nvcc count\_triangles\_C.cu  $ ./a.out |
| Profiling Command | $ /usr/local/cuda/bin/nvprof ./a.out |

V. Results

After testing the functionality of all three implementations, I ran several experiments using large graph data sets provided by Stanford Large Network Dataset Collection. All profiling was done using NVIDIA’s built in profiler “nvprof”. The last column shows the speed up rate when the algorithm was implemented in CUDA C compared to that implemented in CUDA Numba.

Empty cells in the table indicate that the kernel execution was not successful due to unresolved errors. Most of the errors occurred due to the memory space in the GPU, since few of the largest datasets were not executable for both CUDA Numba and CUDA C implementation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Dataset Name | Nodes | Edges | Triangles | Numba\* | C\* | speed up |
| Amazon0302 | 262111 | 1234877 | 717719 | 773.46 | 16.528 | 46.8 |
| Amazon0312 | 400727 | 3200440 | 3686467 | 3836.37 | 778.09 | 4.93 |
| Amazon0505 | 410236 | 3356824 | 3951063 | 3655.88 | 287.24 | 12.73 |
| Amazon0601 | 403394 | 3387388 | 3986507 | 3600.71 | 186.91 | 19.26 |
| as-caida20071105 | 26475 | 106762 | 36365 | 331.97 | 250.67 | 1.32 |
| BrightKite | 58228 | 214078 | 494728 | 224.35 | 63.709 | 3.52 |
| ca-AstroPh | 18772 | 198110 | 1351441 | 797.75 | - | - |
| ca-CondMat | 23133 | 93497 | 173361 | 124.69 | 7.4746 | 16.68 |
| ca-GrQc | 5242 | 14496 | 48260 | 13.893 | - | - |
| cit-HepPh | 34546 | 421578 | 1276868 | 547.65 | 60.95 | 8.99 |
| cit-HepTh | 27770 | 352807 | 1478735 | 1248.58 | 609.33 | 2.05 |
| cit-Patents | 3774768 | 16518948 | 7515023 | 3682.99 | - | - |
| email-Enron | 36692 | 183831 | 727044 | 415.82 | 186.79 | 2.23 |
| email-EuAll | 265214 | 420045 | 267313 | 3582.6 | - | - |
| Facebook | 4039 | 88234 | 1612010 | 536.56 | 31.826 | 16.86 |
| Flickr | 105938 | 2316948 | 107987357 | - | - | - |
| LiveJournal | 4847571 | 68993773 | 285730264 | - | - | - |
| oregon1\_010331 | 10670 | 22002 | 17144 | 294.09 | 217.87 | 1.35 |
| oregon1\_010407 | 10729 | 21999 | 15834 | 290.85 | 203.8 | 1.43 |
| oregon1\_010414 | 10790 | 22469 | 18237 | 296.07 | 223.6 | 1.32 |
| oregon1\_010428 | 10886 | 22493 | 17645 | 296.45 | 207.91 | 1.43 |
| oregon1\_010505 | 10943 | 22607 | 17597 | 294.27 | 217.27 | 1.35 |
| oregon1\_010512 | 11011 | 22677 | 17598 | 303.2 | 211.74 | 1.43 |
| orkut | 3072441 | 117185083 | 627584181 | - | - | - |
| p2p-Gnutella04 | 10876 | 39994 | 934 | 1.7692 | 1.3075 | 1.35 |
| p2p-Gnutella05 | 8846 | 31839 | 1112 | 1.5437 | 1.0058 | 1.53 |
| p2p-Gnutella06 | 8717 | 31525 | 1142 | 1.8443 | 1.3724 | 1.34 |
| p2p-Gnutella08 | 6301 | 20777 | 2383 | 1.6965 | 1.2117 | 1.4 |
| p2p-Gnutella09 | 8114 | 26013 | 2354 | 1.8307 | 1.3019 | 1.41 |
| p2p-Gnutella24 | 26518 | 65369 | 986 | 12.209 | 8.657 | 1.41 |
| p2p-Gnutella25 | 22687 | 54705 | 806 | 0.64484 | 0.443 | 1.46 |
| p2p-Gnutella30 | 36682 | 88328 | 1590 | 0.67136 | 0.361 | 1.86 |
| p2p-Gnutella31 | 62586 | 147892 | 2024 | 966.15 | 0.689 | 1402.25 |
| roadNet-CA | 1965206 | 2766607 | 120676 | 58.671 | 4.5959 | 12.77 |
| roadNet-PA | 1088092 | 1541898 | 67150 | 31.963 | 2.5307 | 12.63 |
| roadNet-TX | 1379917 | 1921660 | 82869 | 39.648 | 3.1548 | 12.57 |
| soc-Epinions1 | 75879 | 508837 | 1624481 | 1536.91 | 696.42 | 2.21 |
| soc-Slashdot0811 | 77360 | 905468 | 551724 | 957.98 | 654.82 | 1.46 |
| soc-Slashdot0902 | 82168 | 948464 | 602592 | 970.36 | 652.91 | 1.49 |

\*Kernel Execution time (32 threads) in miliseconds

VI. Conclusion

The performance results suggest that for all sizes of datasets, GPU implementation using CUDA C is much more efficient than that using CUDA Numba. The kernel execution time of C implementation speeds up by as much as 1402 times the Numba implementation, which is surprisingly high considering that fact that both implementations used identical algorithms. The major factor contributing to such disparity is the fact that Numba library has to go through the extra process of compiling Python code into CUDA C code. However, it is possible that there are other factors that contribute to the difference, such as the structure of C code that allows direct access and manipulation using pointer variables, unlike the structure Python code where pointer operations are infeasible.

It is also significant to point out the fact that the performance for both implementations are often worse than the performance of CPU parallel implementation of the same algorithm. The CPU performance can be found in the original paper of the triangle counting algorithm. It is an interesting finding, since my initial assumption was that GPU implementation will be much faster than the CPU parallel implementation since GPU’s are optimized for executing parallel programming codes. The reason behind GPU’s poor performance for this particular algorithm is left for further study in the future.

VII. Future Work

In order to continue the study in this research, it is important to first debug the implementations so they are able to process very large datasets such as Flickr and LiveJournal. With the kernel execution time for largest datasets, the performance will be more measureable.

Moreover, more study on why CUDA C execution speed up rates vary so much depending on the dataset can be done to further analyze the performance results from two different implementations. This will also help gain insight in how CUDA kernels are executed for this specific algorithm and why GPU implementations show worse performance than CPU implementation.

Lastly, more research should be done on the reason why this particular algorithm is more optimized for CPUs instead of GPUs, or if it is possible for GPU implementations to show better performance through other optimizations in the code. Further understanding on these problems will allow for one to find the most efficient algorithm for triangle counting in GPUs if not the algorithm used in this research.