APPAGATO:

an APproximate PArallel and stochastic GrAph querying TOol for biological networks

Vincenzo Bonnici[1], Federico Busato[1], Giovanni Micale[2], Nicola Bombieri[1], Alfredo Pulvirenti[3], Rosalba Giugno[3]

[1]Department of Computer Science, University of Verona, Strada le Grazie 15 - 37134 Verona and
[2]Department of Math and Computer Science, University of Catania, Viale A. Doria 6 - 95125 Catania and
[3]Department of Clinical and Experimental Medicine, University of Catania, via Palermo, 636 - 95122 Catania.

Abstract

This document details the characteristics, input parameters and suggested usage of the APPAGATO tool included in package.

Requirements

- A recent version of C++ compiler
- A stable release of CMake version ≥ v3.4 with support for FindCUDA module
  - CUDA Toolkit version ≥ 7
  - NVIDIA Kepler or Maxwell GPU Architectures (with compute capability ≥ v3.0)

NOTES:

- APPAGATO is a standalone software and does not require any additional C/C++ libraries (such as Boost library).
- APPAGATO does not contain any direct system calls to ensure the portability of the code.
- The sequential version of APPAGATO has been tested on the following compilers: gcc/g++ v4.8.4, gcc/g++ v4.9.2, gcc/g++ v4.9.3, clang v3.6, clang v3.7, clang v3.8
- The parallel version of APPAGATO has been tested on the following versions of CUDA toolkit: v7, v7.5
- APPAGATO has been tested on linux operating system (Debian v7, Windows with CygWINx64, MacOS Yosemite).

CygWIN: https://cygwin.com/install.html
Required package: Devel Category

Getting Starting

Follow these steps to compile APPAGATO:

HIT: Make sure your system meets the Requirements.

HIT: The “Documentation” directory contains a Quick Start Guide to install and configure the CUDA toolkit.

```
$ cd build
$ cmake [ -DARCH=<your_GPU_compute_cabability> ] ..
$ make
```

HIT: The sequential version does not require any GPU device.
HIT: The compute capability of a GPU determines its general specifications and available features.
HIT: If you do not know the compute capability of your GPU, you can find it on
https://developer.nvidia.com/cuda-gpus
Terminal example:

```
fbusato@hpcb-srvcuda01:~/APPAGATO$ cd build/
fbusato@hpcb-srvcuda01:~/APPAGATO/build$ cmake -DARCH=35 ..
-- The CXX compiler identification is GNU 4.9.2
-- Check for working CXX compiler: /storage/fbusato/bin/gcc-4.9.2/build/bin/g++
-- Check for working CXX compiler: /storage/fbusato/bin/gcc-4.9.2/build/bin/g++ --
works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found CUDA: /usr/local/cuda-7.5 (found version "7.5")

Test on cmake v3.2.2  System: Linux-3.2.0-4-amd64

-- Performing Test COMPILER_SUPPORTS_CXX11
-- Performing Test COMPILER_SUPPORTS_CXX11 - Success

GPU architecture: 3.5

Selected CMAKE_BUILD_TYPE: Release

-- Configuring done
-- Generating done
-- Build files have been written to: /storage/fbusato/APPAGATO/build
fbusato@hpcb-srvcuda01:~/APPAGATO/build$ make -j
Scanning dependencies of target SeqAppagato_info
[  1%] [  3%] [  5%] Building NVCC (Device) object
CMakeFiles/ParallelAppagato_rel.dir/__/lib/XLib/src/Cuda/ParallelAppagato_rel_generated
_Timer.cu.o
Building NVCC (Device) object
CMakeFiles/ParallelAppagato_rel.dir/src/Device/ParallelAppagato_rel_generated_approx_su
bgraph_iso.cu.o
Scanning dependencies of target Converter
Building NVCC (Device) object
CMakeFiles/ParallelAppagato_rel.dir/src/Device/ParallelAppagato_rel_generated_similarit
y.cu.o
Scanning dependencies of target SeqAppagato_rel
[  7%] [  9%] [ 10%] Building NVCC (Device) object
CMakeFiles/ParallelAppagato_info.dir/__/lib/XLib/src/Cuda/ParallelAppagato_info_generat
ed_Timer.cu.o
Building NVCC (Device) object
CMakeFiles/ParallelAppagato_info.dir/src/Device/ParallelAppagato_info_generated_approx_
subgraph_iso.cu.o
...
```

The build chain generates four different executables:

- **SeqAppato_info**: *Sequential version of the algorithm*. It prints information like number of valid solutions, number of unique solutions, memory used, single step timing, size of input, etc.
- **SeqAppato_rel**: *Fast Sequential version of the algorithm*. It prints only the results and it should be used to timing the whole algorithm.
- **ParallelAppato_info**: *Parallel version of the algorithm*. It prints information like number of valid solutions, number of unique solutions, memory used, single step timing, size of input, etc.
- **ParallelAppato_rel**: *Fast Parallel version of the algorithm*. It prints only the results and it should be used to timing the whole algorithm.

NOTE: The build chain generates **ParallelAppato_info** and **ParallelAppato_rel** only if the CUDA toolkit is found on the system (see: cmake **find_cuda** package https://cmake.org/cmake/help/v3.4/module/FindCUDA.html).

Usage

NOTE: All versions have the same parameters.

```
$ ./ParallelAppagato_info <TARGET_GRAPH> <QUERY_GRAPH> <NUMBER_OF_SOLUTIONS>
                          [ <SIMILARITY_MATRIX> ]
```

<TARGET_GRAPH>: Input Target graph (should has .gdu extension)

<QUERY_GRAPH >: Input Query graph (should has .gdu extension)

<NUMBER_OF_SOLUTIONS>: Number of solutions requested by the user

[<SIMILARITY_MATRIX>]: optional parameter that indicate the file containing the similarity matrix

Example:

```
$ ./SeqAppagato_info ../Example_PPI/Homo_sapiens_43.gfu
                     ../Example_PPI/Homo_sapiens_43_32V.gfu 100
```

ADVANCED CONFIGURATION:

config.h file contains advanced configuration parameters of APPAGATO, such as grid configuration of each kernel and the maximum number of registers to use. (It should not be modified).

Input Format

Graphs are stored in text files containing one or more items.

The current input format (*gdu* format) allows the description of UNDIRECTED graphs with labels on nodes.

NOTE: each edge is reported only one time.

The general format of graph files is:

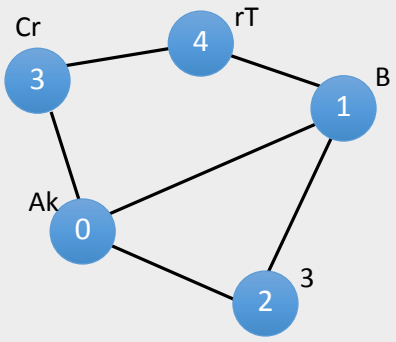
```
#[graph_name]
[number_of_nodes]
[label_of_first_node]
[label_of_second_node]
...
[label_of_last_node]
[number_of_edges]
[node_id] [node_id]
[node_id] [node_id]
...
```

WARNING: [graph_name] and [labels] items must not contain blank lines or spaces.

NOTE: Labels are case **sensitive**.

An example of input file is the following:

```
#my_graph
5
Ak
B
3
Cr
rT
6
0 1
2 1
3 4
0 3
0 2
4 1
```



Query Graph Constraints:

- The number of vertices must be less than 254 and the number of edges must be less than 65536
- The graph must be (weakly) connected

Target Graph Constraint:

- The number of vertices must be less than 65536 and the number of edges must be less than 2^{32}

Similarity matrix Constraint:

- The matrix must has a number of rows equal to the number of vertices of query graph and a number of columns equals to the number of vertices of target graph

Soft Constraint:

- The number of solutions must be less than "MAX_SOLUTIONS" value set in the config.h file. The value can be increased without any problems. Over a threshold, the tool informs that the stack size is not sufficient. To increase the application stack size you can type on the terminal:
  `$ ulimit -s unlimited`

Output

The output shows the matches between query and target graphs.

- Each line represents a graph match between query graph and target graph. The number of matches are always equal to the number of query vertices. Syntax:
  `{ Match }`
  `{ Match }`
  …
- Each group (*A, B*) represents a match between a query vertex and a target vertex, where *A* and *B* are the respective vertex IDs specified in the input file. Syntax:
- `{ (Query_vertex`$_1$`,Target_vertex`$_1$`), (Query_vertex`$_2$`,Target_verte`$_2$`), … }`

Output example:

```
0  ./ParallelAppagato_rel ../Example_PPI/Homo_sapiens_43.gfu
   ../Example_PPI/Homo_sapiens_43_4V.gfu 4

1  {(2,8214)(1,1072)(3,5817)(0,4982}
2  {(1,1986)(0,1719)(3,1982)(2,1988)}
3  {(3,276)(2,236)(0,1554)(1,3972)}
4  {(0,3575)(2,3571)(3,3576)(1,345)}
5
```

The results generates 4 solutions (query-target matches)
The line **1** specifies that the query vertex **2** has been matched with the vertex **8214** in the target graph, the query vertex **1** has been matched with the vertex **1072** in the target graph, etc.

Dataset

For the dataset and other graphs refer to RI dataset:

http://ferrolab.dmi.unict.it/ri/datasets.html PPI

Reports

APPAGATO has been tested with several tools to ensure the correctness and to get code statics.

Valgrind: VALGRIND is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.

Flags: `--track-fds=yes --leak-check=full --track-origins=yes --show-reachable=yes -v --max-stackframe=2818064`

Cuda-memcheck: CUDA-MEMCHECK is a functional correctness checking suite included in the CUDA toolkit. This suite contains multiple tools that can perform different types of checks. The *memcheck* tool is capable of precisely detecting and attributing out of bounds and misaligned memory access errors in CUDA applications. The tool also reports hardware exceptions encountered by the GPU. The *racecheck* tool can report shared memory data access hazards that can cause data races. This document describes the usage of these tools. The *initcheck* tool can report cases where the GPU performs uninitialized accesses to global memory. The *synccheck* tool can report cases where the application is attempting to use CTA synchronization in divergent code.

Flags: `--tool initcheck --tool synccheck --report-api-errors all --leak-check full`

CPPCheck: CPPCHECK is a static analysis tool for C/C++ code. CPPCHECK primarily detects the types of bugs that the compilers normally do not detect. The goal is to detect only real errors in the code (i.e. have zero false positives).

Flags: `--enable=all`

Clang Static Analyzer: The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C program. The goal of the Clang Static Analyzer is to provide a industrial-quality static analysis framework for analyzing C, C++, and Objective-C programs that is freely available, extensible, and has a high quality of implementation.

Lyzard: LYZARD calculates how complex the code 'looks' rather than how complex the code real 'is'. People will need this tool because it's often very hard to get all the included folders and files right when they are complicated.

Cloc: CLOC counts blank lines, comment lines, and physical lines of source code in many programming languages

The directory “Documentation/Reports” contains all reports of the tools used to test APPAGATO.