# Graph Processing Optimization

8 November, 2024

## Introduction

In this lab you will study and optimize a graph processing kernel that converts a graph in its edge list format into a graph in a Compressed Sparse Row (CSR) format. The purpose of this lab is to help you understand the concept of data sparsity, its effect on the optimization of software, and its consequences for hardware design. The lab provides starter code that converts from an edge list to a CSR. You should read and understand this code. Your primary implementation task is to re-write this code to be more cache-friendly. You should optimize the code using propagation blocking.

## Propagation Blocking

The propagation blocking optimization is an algorithmic transformation for algorithms that manipulate sparse data inputs and perform random-access writes to their output. The main idea behind the optimization is that the output space is divided into cache-friendly bins. All of the output elements in a bin should fit into the cache.

As the algorithm processes inputs, each of those input items corresponds to some update to one or more output elements. Propagation blocking files each of these updates into the bin corresponding to the output element being updated. After populating the bins, propagation blocking does a second pass over the data, streaming through a bin's elements, and performing their updates.

You will design a bin data structure and any routines that you need to populate your bins. You will then implement a propagation blocking CSR builder that processes edges. You will need a propagation blocking neighbor counting kernel and a propagation blocking neighbor population kernel. The basic (i.e., non-propagation-blocking) version of these two kernels are in `csr.c` and you should use these to guide your implementation. You should use the version of the propagation blocking algorithm that we learned about in class.

The basic structure of the algorithm in this assignment will work like the following pseudocode:

```
bin(...){
  for(e in edges){
    b = find_bin(bins, e.dst)
    b[next++] = (e.src,e.dst)
  }
}
```

```
binread(...){
  for(B in bins){
    for(e in B){
      update(neighbor_count[e.dst],...)
    }
  }
  ... /*compute OA from neigh_count, etc*/

  for(B in bins){
    for(e in B){
      update(neighbor_array[ ... ],...)
    }
  }

}
```

A few things to note about implementing this pseudocode as real code.

- You might notice that you have to run through the edge list multiple times to first populate the bins, then to build the output CSR's OA and NA arrays. Think about why running through the edge list sequentially multiple times is not as much of a performance penalty as traversing edges and updating vertex data multiple times.
- Your bin data structures must be compact and memory efficient. An array is a good choice (as opposed to a more complex structure like a C++ STL Vector or a map or something). Remember, you want to be streaming through the items filed into a bin and then accessing only the subset of the vertex property array (i.e., neighbor_count or neighbor_array) that fits into the cache.
- Even if you split the range of vertices evenly across the bin structures, the amount of edges in a bin may vary *wildly* due to differences in graph structure. Your 19th bin might have 2 edges in it and your 20th bin may have 2000000 if a very "popular" node in the graph happens to map into your 20th bin, instead of your 19th bin.

## Structure and Mechanism of the Lab

You should get comfortable with what is happening in `el2csr.c`. `el2csr` takes in an edge list and produces a csr file. This program is the main program that you will be working on, because this is the program that transforms an edge list into a CSR.

Your job will be to re-write `el2csr.c` (or to write a new module with similar functionality) so that it uses propagation blocking to make the CSR construction step more cache friendly. You will probably need to look at each of the steps that the original version of this file is doing and figure out how to either eliminate the step, or to change it to refer to edges in a bin structure instead of an edge list.

There is ample helper code to make your life (hopefully) easier. `csr2el.c` takes in a csr (as generated by `el2csr`) and produces an edge list from the csr. Note that if you run these consecutively, the CSR produced by `el2csr` may not match the edge list that you fed into `csr2el` because constructing a csr "canonicalizes" the order of edges in the graph; the new edge list will be sorted by src. If you want to diff two edgelists, you can run each through `el2csr` and then through `csr2el` to canonicalize them both. Having done that, a diff will either report differences, or that they are the same if they have the same set of edges.

The graph encoding is a binary format built for the assignment. A CSR starts with two unsigned longs — the number of vertices, then the number of edges — followed by the OA (a number of unsigned longs equal to the number of vertices) and then the NA (a number of unsigned longs equal to the number of edges). The hexdump program might come in hand when you are trying to read the contents of these files.

An edge list is similarly encoded as a binary data structure. Every even-numbered 8-byte word in the file is the source of an edge. Every odd-numbered 8-byte word in the file is the dst of an edge with the previous even-numbered 8-byte word as its src. In other words: the file is a list of edges encoded as pairs of unsigned longs, one for the src, one for the dst.

## Caveat

One bit of debatable software engineering that you will have to endure — `MAX_VTX` defines the number of vertices in the graph. For reasons that are inessential to the application, this is *statically* defined in graph.h. If you rebuild any tool (`rand_graph`, `csr2el`, `el2csr`, or your version that uses propagation blocking) and that re-built tool refers to `NUM_VTX`, you need to rebuild everything. Also, any graphs you generated using the old value of `NUM_VTX` probably won't work with version of the tools built using a new value of `NUM_VTX`.

## Evaluation

You will evaluate the performance of your implementation using your cache simulator from lab2. Your goal is to demonstrate that propagation blocking improves the performance of CSR building compared to directly processing edge list data. It might be simplest to model a single level of cache hierarchy, rather than a multi-level hierarchy. This simplifying assumption will make the modeled machine less realistic, but your data easier to understand. If you do not have a working cache simulator after lab 2, contact the course staff and we can provide you with a cache simulator that you can use for this lab.

- You are responsible for designing a study that demonstrates the performance improvement of propagation blocking quantitatively
- You must choose one or more appropriate metrics
- You must choose an appropriate baseline software and (modeled) hardware configuration
- You must include experiments that characterize your solution. These experiments should evaluate the sensitivity of your system to the range of vertices captured by each bin (bin size), which is equivalent to the number of bins. You may also want to study sensitivity to graph size and cache configuration.
- All claims of results in your study of propagation blocking must be justified in your write up and supported quantitatively by the data that your experiments produce. We expect to see plots included in your writeup that support your main conclusions.

Submit your code and writeup (as a pdf file) to Gradescope.

## A Tour of the Code

The core graph processing functions are decomposed into `el.c/el.h` and `csr.c/csr.h`.

- `el.c/el.h` contains a single function that will be useful for initializing an edge list from a file that contains an edge list.
- `csr.c/csr.h` contains a collection of functions that manipulate a csr.
- `void *CSR_EL_count_neigh(el_t*,csr_offset_t*);` This function populates the OA array with the count of neighbors, taken while traversing an edge list.
- `void CSR_cumul_neigh_count(vertex_t *,vertex_t *);` This function accumulates the neighbor counts, going from counts of neighbors to OA offsets. The input arguments are an array to be accumulated and an array that will be a copy of that array that was accumulated.
- `void CSR_print_neigh_counts();` This function prints neighbor counts in ascii and may be useful for debugging.
- `void *CSR_EL_neigh_pop(el_t*,csr_t*);` This function populates the NA of the CSR based on the OA and the contents of the edge list.
- `void CSR_out(char *,unsigned long,csr_t *);` This function writes out the binary CSR structure to a specified file.

- `void CSR_EL_out(csr_t *,char *,unsigned long, unsigned long, int fdebug);` This function writes out an edges list constructed by traversing a CSR, optionally printing the resulting edge list to stdout, if `fdebug == 1`.
- `csr_t * CSR_in(char *,unsigned long *, unsigned long *);` This function loads a csr in from a file containing a csr in the binary format of the assignment.

*REMEMBER*: `graph.h` specifies `MAX_VTX`, which is the number of vertices (even if some are not part of an edge) and if you change `MAX_VTX`, you must recompile and regenerate your graphs!

## Tips for Completing the Assignment

- `hexdump -C graph.g > graph.g.hexdump` This command will produce a hexdump of graph.g, allowing you to examine the contents of the graph that your program spits out. You will find it useful to use hexdump to read out the bytes of edge lists, CSRs, and, perhaps, edgelists generated from CSRs.
- It may be useful to use `diff` to compare edge lists. One useful strategy will be to convert from el to csr, then back to an el, and evaluate the contents of the graph.
- You should consider a range of graph sizes. To start with, you will probably want to understand how everything works using tiny graphs (like 10 nodes and 10 edges).
- Scaling might be interesting. We tested all of the tools here on graphs up to 100000000 vertices and 100000000 edges (producing a multi-GB file). It is not really possible to debug a system running on such a large graph, but you should be sure to test what you build with graphs that are very large, to be sure you do not have any scalability problems. One example of a scaling problem would be the inability to run graphs larger than 4 billion nodes (why might you run in to that bug?) Another example of a scaling problem would be the inability to continue to improve performance, even with propagation blocking, after a certain size of graph (or perhaps cache).
- You may see an improvement in performance of part of the program's execution even on your native machine (although many factors may make it difficult to see this effect without a lot of tuning). If you time the inner-loop of CSR generation (omitting the relatively slow and unoptimizable file I/O parts), you should see that propagation blocking helps even natively. The effects will be much clearer in simulation, where you can directly measure how cache performance changes.