Breadth-first Search in CUDA

Matthias Springer *
Tokyo Institute of Technology
matthias.springer@acm.org

Abstract

This work presents a number of algorithms found in literature for breath-first search (BFS) on GPUs in CUDA, along with a small performance evaluation section.

1. Breadth-first Search

Breadth-first search (BFS) is a well-studied algorithm in parallel computing with many practical applications. It can be used to decide vertex reachability and is the foundation for other algorithms such as single-source shortest path (SSSP) or the Ford-Fulkerson algorithm for finding the minimum cut/maximum flow of a graph. In this work, we focus on BFS and how to parallelize it for GPUs in CUDA. We present a number of performance optimizations along with a performance study at the end.

Given a directed graph $G = (V, E)$, BFS assigns every vertex v the distance from a given start node s, i.e., the minimum number of edges to reach v from s. In contrast to depth-first search (DFS), BFS is a good candidate for parallelization, because the neighborhood of a vertex can be explored in parallel.

2. Sequential Algorithm

Algorithm 1 shows a sequential version of BFS. It maintains a queue of vertices which should be processed. When a vertex is processed, the algorithm updates the distance of all unprocessed vertices and adds them to the queue. There is no dedicated flag to indicate whether a vertex was visited. Instead, an infinite distance value means that the vertex was not processed yet.

Algorithm 1 Sequential BFS

1: **for all** v $\in V$ **do**
2: v.distance $\leftarrow \infty$
3: start_vertex.distance $\leftarrow 0$
4: q $\leftarrow$ **new** queue
5: q.push(start_vertex)
6: **while** !q.empty() **do**
7: v $\leftarrow$ q.pop()
8: **for all** n $\in$ v.neighbors **do**
9: **if** n.distance = ∞ **then**
10: n.distance $\leftarrow$ v.distance + 1
11: q.push(n)

3. Vertex-centric CUDA Implementation

Algorithm 2 shows pseudo code for an unoptimized GPU implementation. It is a vertex-centric implementation, i.e., parallelism is expressed in terms of vertices. For example, there may be one thread per vertex. If there are less threads than vertices, then work should be divided evenly among all vertices. This implementation does not maintain a queue, but every vertex is processed in every iteration, making the algorithm similar to the Bellman-Ford algorithm for SSSP.

* Student ID: 15D54036

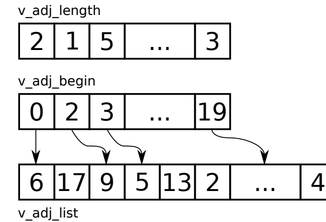


Figure 1: Example: Adjacency List Data Structure

Algorithm 2 Unoptimized CUDA BFS

1: **for all** v $\in V$ **do**
2: v.distance $\leftarrow \infty$
3: start_vertex.distance $\leftarrow 0$
4: still_running $\leftarrow$ *true*
5: **while** still_running **do**
6: still_running $\leftarrow$ *false*
7: **for all** v $\in V$ **in parallel do**
8: **for all** n.distance $>$ v.neighbors + 1 **do**
9: n.distance $\leftarrow$ v.distance + 1
10: still_running $\leftarrow$ *true*

The CUDA implementation represents the graph with three integer arrays (Figure 1) [1].

- `v_adj_list`: A contatenation of all adjacency lists of all vertices (size $|E|$).
- `v_adj_begin`: An array of size $|V|$, storing offsets into the previous array. The ith offset markes the position where the adjacency list for ith vertex begins.
- `v_adj_length`: An array of size $|V|$, storing the length of every adjacency list. This information is held redundantly. It could also be computed using only the previous array (except for the last value).

Figure 9 shows the source code of a basic vertex-centric implementation. Every thread in this implementation is assigned $\frac{|V|}{n}$ vertices, where n is the number of threads. The first outer loop ensures that every vertex is processed, even if less threads than vertices are spawned. For every vertex, the program iterates over all neighbors. If the destination vertex is reachable from the current vertex with a shorter distance, the distance in the `result` array is updated. The boolean value `still_running` is set to true, if at least one distance value is updated. This value is used to decide whether another iteration of the algorithm should be run.

This algorithm suffers from two shortcomings. First, the inner `for` loop induces branch divergence if vertices within a warp

(group of 32 threads) have a different out-degree, resulting in a different number of loop iterations. Second, this implementation processes vertices even the distance of their incoming vertices has not changed. Such vertices do not have to be processed in an iteration. In the following, we present optimizations that attempt to solve these shortcomings.

3.1 Vertex Frontier

This version maintains a boolean `frontier` array of size $|V|$. In every iteration, a vertex i is processed only if `frontier[i]` is set to *true*. A vertex is added to the frontier if its distance was is updated. The frontier corresponds to the queue in a sequential implementation. It ensures that only those vertices are processed whose distance from the source vertex is one unit larger than the vertices from the previous iteration. To make sure that a distance value is not updated with a larger one, vertices are only updated in a single iteration but not afterwards[1]. This is indicated by a boolean value in the `visited` array. Alternatively, this implementation could also compare distance values, similar to the previous implementation.

Figure 10 shows the source code of this implementation. In addition to the `frontier` and `visited` arrays, this implementation has an additional array `updated`. Instead of setting frontier flags directly, the first kernel only sets `updated` flags. A second kernel sets the frontier flag for a given vertex only if its `updated` flag was set before. This is necessary to ensure that a thread does not start processing vertices that were added to the frontier earlier in the same iteration. If the number of threads running in parallel equals the number of vertices, this indirection is not required.

3.2 Vertex Frontier with Iteration Counter

The previous implementation maintains four arrays and launches two kernels per iteration. It can be simplified based on the observation that every vertex is updated with the same distance value in an iteration [3]. The kernel of the optimized implementation (Figure 11) uses only a single array for the distance values and receives an additional *iteration number* parameter. A vertex is part of the frontier if its distance value equals the current iteration number. All following implementations in this section are based on this implementation.

3.3 Deferring Outliers

The next optimization will attempt to reduce the problem of warp divergence due to different out-degrees of vertices within a warp. Before a vertex is processed, its out-degree is analyzed. If it exceeds a certain threshold value, it is added to a queue and processed at the very end of the iteration (Figure 2) [3]. Therefore, it does not stall a warp with mostly vertices of lower out-degree. This optimization does not reduce the total amount of work. Instead, it processes vertices of low out-degree together and vertices of high out-degree together, in order to reduce waiting time.

The queue of deferred vertices is stored in shared memory, i.e., there is one queue per block. New vertices are inserted using an atomic add operation. Such operations are slow in general, but acceptable if performed on shared memory. If the shared memory queue is full, vertices are no longer deferred but processed immediately. To avoid this problem, programmers can increase the size of the queue up to 48 KB (depending on the configuration of L1 cache and shared memory). However, this limits the number of blocks that can run in parallel. Alternatively, blocks can be made smaller, reducing the probability of a queue running full.

Figure 12 shows the source code of this optimization (kernel only). The out-degree threshold is defined by `DEFER_MAX` and the

[1] Multiple threads might update the same distance value at the same time. This is data race, but it is harmless: All threads will write the same value.

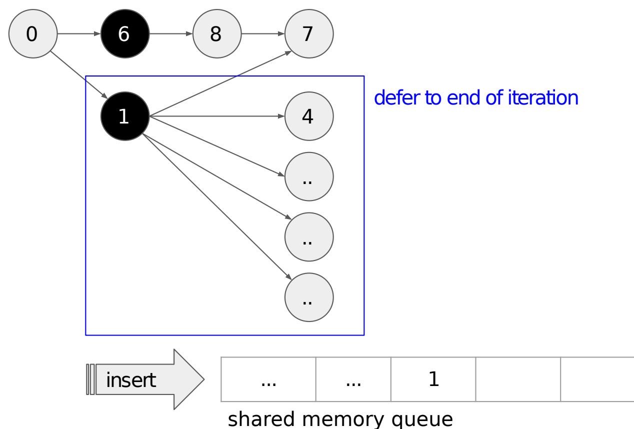


Figure 2: Deferring Outliers in Shared Memory Queue

size of the queue can be controlled with `D_BLOCK_QUEUE_SIZE`. The check in Line 25 determines whether a vertex should be deferred or processed. It is processed directly if its out-degree is below the threshold or the queue is full. Since `blockDim.x` many threads may insert a vertex into the queue at a time, the queue is considered full if it has less than `blockDim.x` free slots. This is required because the capacity check and the inserting process are not atomic: There could be more threads in the `else` branch than free slots are available, because the atomic add is performed after the decision is made to insert a vertex into the queue.

3.4 Data Reordering / Redirection

Another approach to reduce the problem of warp divergence is to sort all vertices by out-degree, such that all threads within a warp process vertices with a similar out-degree at the same time. This can be done with a job reordering array or by physically changing the order of data [5]. The first option involves reading an additional redirection array, so it is potentially slower than reodering data physically. The benchmarked implementations do not change the structure of the outermost loop of the kernel; the kernel is identical to the kernel in Figure 11. Therefore, if less threads than vertices are used, a thread might first process a vertex with low out-degree and then a vertex with high out-degree. The same is the case for all other threads within a the warp, so all threads within a warp are likely to process vertices of similar out-degree at the same time.

3.5 Virtual Warp-centric Programming

The next implementation presents another technique for reducing warp divergence known as *Virtual Warp-centric Programming* [3]. This technique divides a program into multiple SISD (single instruction, single data) and SIMD (single instruction, multiple data) phases. It introduces the notion of a *virtual warp*, which is a fraction of a warp, i.e., every warp consists of multiple virtual warps (Figure 3). Within a SISD phase, only one thread in a virtual warp is executing. This is done via replicated computation, i.e., every thread processes the same data and instructions. Within a SIMD phase, every thread in a virtual warp processes different data.

In BFS, every virtual warp is assigned one vertex. The SISD phase consists of calculating the next vertex ID and checking if a vertex belongs to the current iteration. The SIMD phase consists of iterating over the neighborhood. All neighborhood vertices are distributed evenly among all threads within a virtual warp. If the size of a virtual warp is `W_SZ` and a vertex has n neighbors, then every thread in that virtual warp performs $\frac{n}{\texttt{W_SZ}}$ many updates.

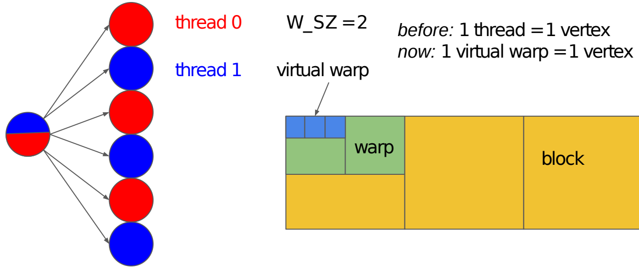


Figure 3: Virtual Warp-centric Programming for BFS

From a technical perspective, this technique is a mixture of a vertex-centric and an edge-centric implementation. The virtual warp size `W_SZ` controls how many threads are assigned the same vertex and how many edges share one thread.

3.6 Hierarchical Frontier Queue

The next optimization uses a different technique for maintaining a frontier of vertices. Instead of the boolean `frontier` array, vertices are inserted into a hierarchical queue [4]. The queue contains of two levels: First, every block has a queue in shared memory, into which vertices are inserted. Second, at the end of every iteration, all shared memory queues are merged into a single global memory queue (Figure 4). Atomic operations are required to calculate the next offset at which a vertex is inserted in a queue. Atomic operations are reasonably fast in shared memory but slow in global memory. Since the size of every shared memory queue is known at the end of an iteration, only `gridDim.x` (number of blocks) many atomic operations are required in global memory.

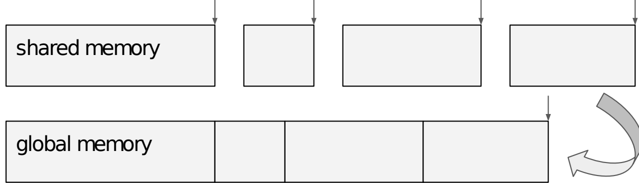


Figure 4: Frontier Queue in Shared and Global Memory

Figure 14 shows the source code for this implementation. Its main limitation is the size of the shared memory queue. Large graphs cannot be processed if a shared memory queue overflows. To keep the size of queues small, the implementation writes queues to the shared memory after every iteration of the outermost loop. This increases the number of atomic operations in global memory but allows us to process larger graphs. The constant `BLOCK_QUEUE_SIZE` can be adapted to allow for even larger graphs. However, this will increase the shared memory usage and put a lower limit on the maximum number of blocks that can run in parallel. Alternatively, the size of blocks can be reduced to make queue overflows less probable. Another problem of this implementation is that there may be duplicate vertices in a queue. If two vertices update the same vertex i, then i will be added to the queue twice.

3.7 Frontier Queue with Prefix Sum

The previous implementation suffers from performance and scaling issues due to atomic operations and the limited size of shared memory. The next optimization uses a different approach for generating a vertex frontier queue: First, the kernel performing BFS generates a boolean frontier array. Then, that boolean frontier array is converted into an array of vertex IDs. The second step is the challenging one. It can be done in parallel with a prefix sum array.

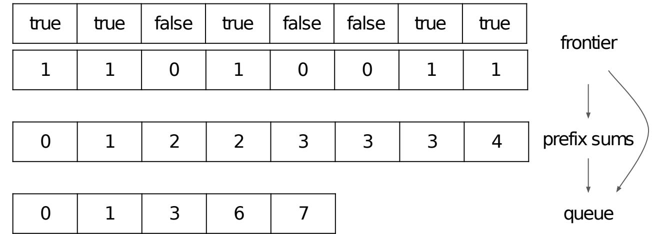


Figure 5: Converting Frontier to Queue

A prefix sum array P of an array A is an array of same size, where $P[i]$ contains the sum of all previous values in A, i.e., $\sum_{j=0}^{i} A[j]$. Let us assume that every thread is assigned one vertex when generating the queue Q from the frontier array F, and P is the prefix sum array of F (false = 0, true = 1). The thread with ID i writes the value i into $Q[P[i]]$ if $F[i]$ is true. An example of this process is shown in Figure 5.

There is a well-studied two-phase prefix sum algorithm [2] for GPUs consisting of an up-sweep phase and a down-sweep phase. The first phase corresponds to a parallel reduce. The second phase can be seen as a reverse operation. We do not explain the details here but refer to previous work instead. An efficient CUDA implementation is challenging for three reasons: First, the implementation must be able to handle special cases if the size of the frontier array is not a power of two. This is similar to the special cases that arise in a parallel reduction. Second, computations should be done in shared memory to avoid reading from/writing to slow global memory. However, the frontier arrays of large graphs might not fit in shared memory entirely. In that case, partial results can be calculated in shared memory, similar to an efficient implementation of parallel reduction. Third, a straight forward implementation in shared memory suffers from bank conflicts, but there are techniques to avoid them (e.g., adding padding to data in shared memory).

The implementation used in this work (simplified version in Figure 15) is taken from the Nvidia website[2] and fully optimized. It utilizes shared memory, has only few bank conflicts, and supports frontiers of arbitrary size.

4. Edge-centric CUDA Implementation

The implementations in this section are edge-centric, i.e., parallelism is expressed in terms of edges. To that end, a different data structure is used. The graph is encoded in two arrays `v_adj_from` and `v_adj_to` of size $|E|$, where the first array contains start vertex IDs of edges and the second array contains target vertex IDs.

4.1 Unoptimized Implementation

This implementation does not maintain a vertex frontier and is similar to the algorithm shown in Figure 9. It attempts to update vertices using every edge in the graph (Figure 16).

4.2 Vertex Frontier with Iteration Counter

This implementation maintains a vertex frontier using an iteration counter and is similar to the algorithm shown in Figure 11. Its source code is shown in Figure 17.

[2] `https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html`

Dataset	$\|V\|$	$\|E\|$	avg. degree	degree stddev	med. degree	diameter
Slashdot	77360	905468	11.705	36.844	3	10
WikiTalk	2394385	5021410	2.097	99.915	0	9
California Roads	1965206	5533214	2.816	0.995	3	849

Figure 6: Dataset Characteristics

5. Benchmarks

This section contains an evaluation of all previously described BFS variants. All benchmarks were run five times (minimum is reported) on the TSUBAME 2.5 supercomputer at Tokyo Institute of Technology with three real datasets (Figure 6) taken from the Stanford Network Analytics Project (SNAP): *Slashdot* is a friends graph of the Slashdot news website from November 2008. *WikiTalk* is based on talk pages of Wikipedia from January 2008. Vertices represent registered users and there is a link $i \rightarrow j$ if user i edited user j's talk page at least once. *CA Roads* is a graph of the California road network.

Figure 7 gives an overview of all BFS variants presented in this work. Benchmarks were run for a variety of block size/grid size combinations (see x/y axis labels of Figure 8) and every data point shows the best configuration.

For Slashdot, the unoptimized first vertex-centric implementation is slowest, as expected. Having a frontier of vertices can cut the runtime in half. Most notably, this dataset performs very well with virtual warp-centric programming at a virtual warp size of 16. This is because the average out-degree of vertices is close to that number, allowing for good parallelism when exploring neighbors, while avoiding warp divergence. Choosing a slightly smaller virtual warp size could possibly increase performance even more, probably close to the performance of the edge-centric version. Maintaining a vertex queue does not pay off in this dataset, because the number of vertices is small, resulting in minimal overhead for checking the frontier field of all vertices.

For the California Road Network, the version utilizing prefix sums (*frontier_scan*) is slowest. The graph is in fact cut off: The best configuration with prefix sums has a runtime of 1375288 microseconds. This BFS variant does not perform well in any of our experiments, probably due to its complexity and the relatively low overhead of checking all vertices (instead of a frontier) that could be saved. However, in this dataset it performs especially bad, because the number of vertices is high, making the prefix sum computation expensive. The version with a frontier queue performs best, because the average out-degree is low and only few of the many vertices are active in an iteration. Virtual warp-centric programming does not pay off, because the average out-degree is too low for additional parallelism.

For WikiTalk, using a frontier queue is best. This is because the graph is sparse, as can be seen from the out-degree median. Reordering vertices by out-degree also gives a little bit of performance improvement, indicating that there is some degree imbalance, as can also be seen from the out-degree variance in this dataset. Similar to the previous dataset, the version with prefix sums is slowest.

References

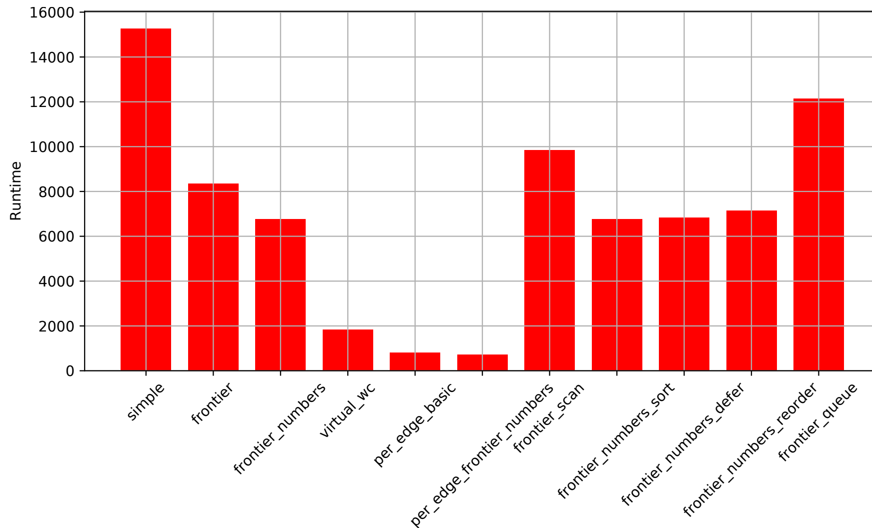
[1] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

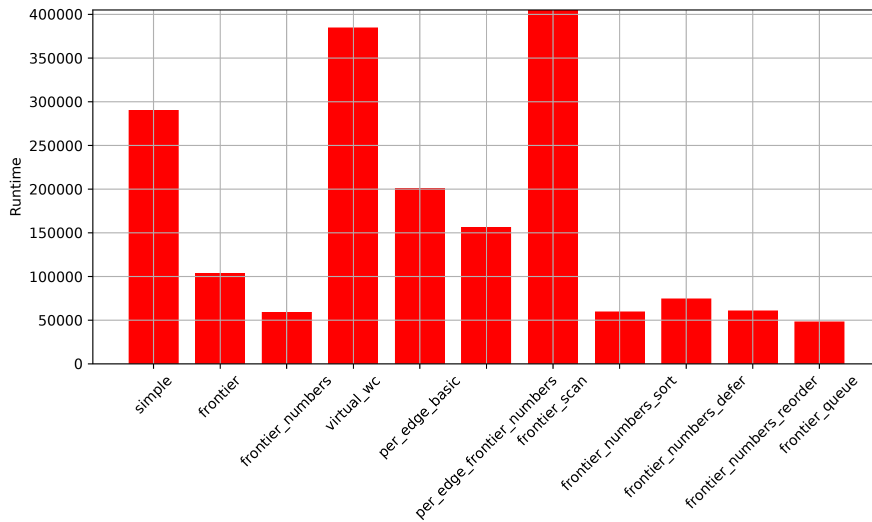
[3] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[4] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

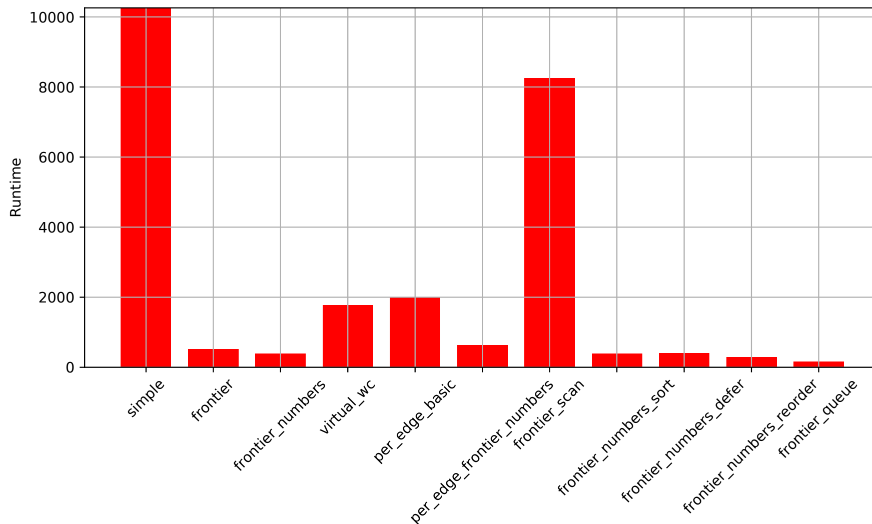
[5] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *SIGPLAN Not.*, 46(3):369–380, March 2011.



(a) Slashdot

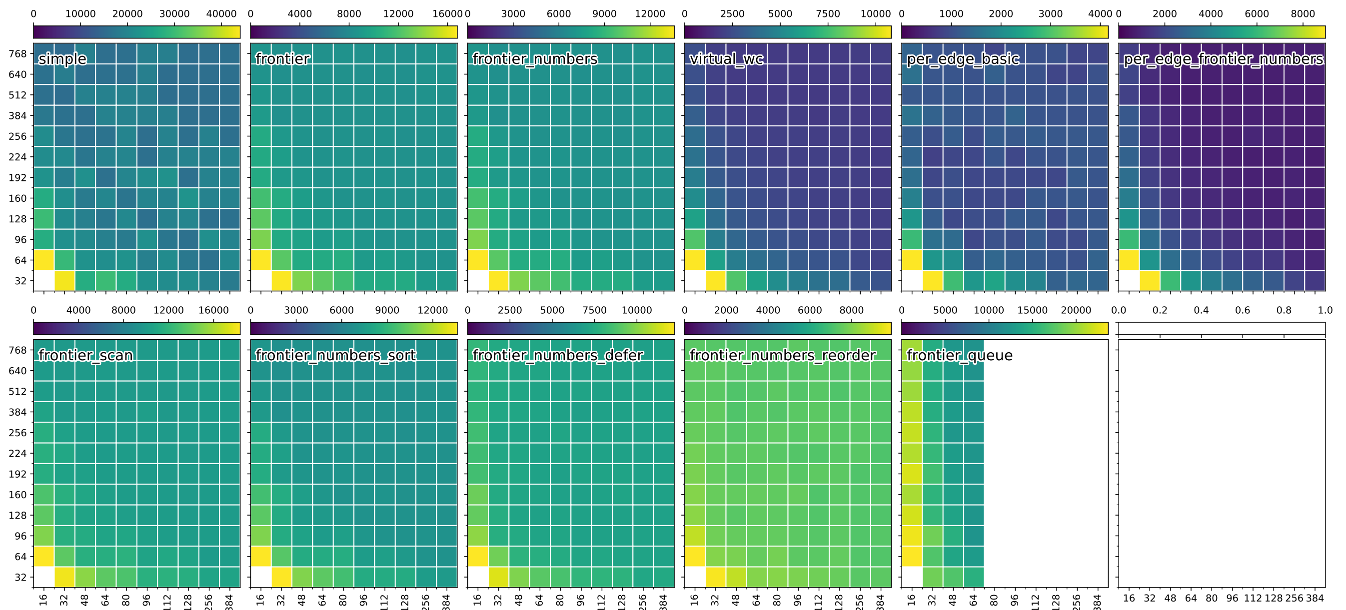


(b) California Road Network

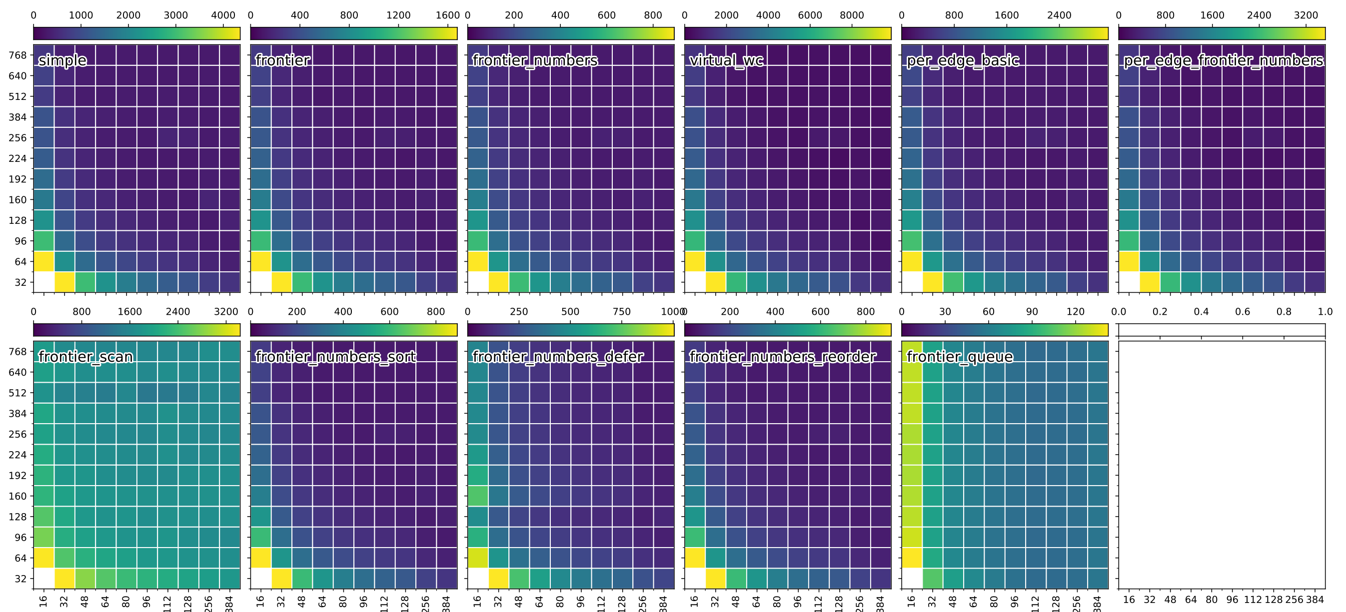


(c) WikiTalk

Figure 7: Performance Study: Overview of runtime of various BFS implementations



(a) Slashdot



(b) California Road Network

Figure 8: Performance Study: Runtime at different grid sizes (x axis) and block sizes (y axis)

```
__global__ void kernel_cuda_simple(...)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int num_threads = blockDim.x * gridDim.x;

  for (int v = 0; v < num_vertices; v += num_threads)
  {
    int vertex = v + tid;

    if (vertex < num_vertices)
    {
      for (int n = 0; n < v_adj_length[vertex]; n++)
      {
        int neighbor = v_adj_list[v_adj_begin[vertex] + n];

        if (result[neighbor] > result[vertex] + 1)
        {
          result[neighbor] = result[vertex] + 1;
          *still_running = true;
        }
      }
    }
  }
}

void run()
{  /* Setup and data transfer code omitted */
  while (*still_running)
  {
    cudaMemcpy(k_still_running, &false_value, sizeof(bool) * 1, cudaMemcpyHostToDevice);

    kernel_cuda_simple<<<BLOCKS, THREADS>>>(...);
    cudaMemcpy(still_running, k_still_running, sizeof(bool) * 1, cudaMemcpyDeviceToHost);
  }

  cudaThreadSynchronize();
}
```

Figure 9: Unoptimized Vertex-centric Implementation

```
__global__ void kernel_cuda_frontier(...)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    for (int v = 0; v < num_vertices; v += num_threads)
    {
        int vertex = v + tid;

        if (vertex < num_vertices && frontier[vertex])
        {
            frontier[vertex] = false;

            for (int n = 0; n < v_adj_length[vertex]; n++)
            {
                int neighbor = v_adj_list[v_adj_begin[vertex] + n];

                if (!visited[neighbor])
                {
                    result[neighbor] = result[vertex] + 1;
                    updated[neighbor] = true;
                }
            }
        }
    }
}

__global__ void kernel_cuda_frontier_update_flags(...)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    for (int v = 0; v < num_vertices; v += num_threads)
    {
        int vertex = v + tid;

        if (vertex < num_vertices && updated[vertex])
        {
            frontier[vertex] = visited[vertex] = *still_running = true;
            updated[vertex] = false;
        }
    }
}

void run()
{
    do
    {
        *still_running = false;
        cudaMemcpy(k_still_running, still_running, sizeof(bool) * 1, cudaMemcpyHostToDevice);

        kernel_cuda_frontier<<<BLOCKS, THREADS>>>(...);
        kernel_cuda_frontier_update_flags<<<BLOCKS, THREADS>>>(...);

        cudaMemcpy(still_running, k_still_running, sizeof(bool) * 1, cudaMemcpyDeviceToHost);
    } while(*still_running);

    cudaThreadSynchronize();
}
```

Figure 10: Vertex-centric Implementation with Frontier

```
__global__ void kernel_cuda_frontier_numbers(int iteration, ...)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    for (int v = 0; v < num_vertices; v += num_threads)
    {
        int vertex = v + tid;

        if (vertex < num_vertices && result[vertex] == iteration)
        {
            for (int n = 0; n < v_adj_length[vertex]; n++)
            {
                int neighbor = v_adj_list[v_adj_begin[vertex] + n];

                if (result[neighbor] == MAX_DIST)
                {
                    result[neighbor] = iteration + 1;
                    *still_running = true;
                }
            }
        }
    }
}

void run()
{
    int iteration = 0;

    do
    {
        *still_running = false;
        cudaMemcpy(k_still_running, still_running, sizeof(bool) * 1, cudaMemcpyHostToDevice);

        kernel_cuda_frontier_numbers<<<BLOCKS, THREADS>>>(iteration++, ...);

        cudaMemcpy(still_running, k_still_running, sizeof(bool) * 1, cudaMemcpyDeviceToHost);
    } while(*still_running);

    cudaThreadSynchronize();
}
```

Figure 11: Vertex-centric Implementation with Frontier and Explicit Iteration Counter

```
#define DEFER_MAX 16
#define D_BLOCK_QUEUE_SIZE 2048

__global__ void kernel_cuda_frontier_numbers_defer(int iteration, ...)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    __shared__ int queue_size;
    __shared__ int next_queue[D_BLOCK_QUEUE_SIZE];

    if (threadIdx.x == 0)
    {
        queue_size = 0;
    }

    __syncthreads();

    for (int v = 0; v < num_vertices; v += num_threads)
    {
        int vertex = v + tid;

        if (vertex < num_vertices && result[vertex] == iteration)
        {
            if (v_adj_length[vertex] < DEFER_MAX || queue_size >= D_BLOCK_QUEUE_SIZE - blockDim.x)
            {
                for (int n = 0; n < v_adj_length[vertex]; n++)
                {
                    int neighbor = v_adj_list[v_adj_begin[vertex] + n];

                    if (result[neighbor] == MAX_DIST)
                    {
                        result[neighbor] = iteration + 1;
                        *still_running = true;
                    }
                }
            }
            else
            {
                // Add to queue (atomicAdd returns original value)
                int position = atomicAdd(&queue_size, 1);
                next_queue[position] = vertex;
            }
        }

        __syncthreads();
    }

    // Process outliers
    for (int v = 0; v < queue_size; v += blockDim.x)
    {
        if (v + threadIdx.x < queue_size)
        {
            int vertex = next_queue[v + threadIdx.x];

            for (int n = 0; n < v_adj_length[vertex]; n++)
            {
                int neighbor = v_adj_list[v_adj_begin[vertex] + n];

                if (result[neighbor] == MAX_DIST)
                {
                    result[neighbor] = iteration + 1;
                    *still_running = true;
                }
            }
        }
    }
}
```


Figure 12: Vertex-centric Implementation with Frontier, Explicit Iteration Counter and Deferring Outliers

```
#define W_SZ 16

__global__ void kernel_cuda_virtual_wc(int iteration, ...)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    for (int v2 = 0; v2 < num_vertices; v2 += num_threads)
    {
        int vertex2 = v2 + tid;

        // W_SZ many threads are processing vertex2
        int warp_id = vertex2 / W_SZ;
        int warp_offset = vertex2 % W_SZ;

        for (int v = 0; v < W_SZ; v++)
        {
            int vertex = warp_id * W_SZ + v;

            if (vertex < num_vertices && result[vertex] == iteration)
            {
                for (int n = 0; n < v_adj_length[vertex]; n += W_SZ)
                {
                    int neighbor_index = n + warp_offset;

                    if (neighbor_index < v_adj_length[vertex])
                    {
                        int neighbor = v_adj_list[v_adj_begin[vertex] + neighbor_index];

                        if (result[neighbor] == MAX_DIST)
                        {
                            result[neighbor] = result[vertex] + 1;
                            *still_running = true;
                        }
                    }
                }
            }
        }
    }
}
```

Figure 13: Vertex-centric Implementation with Frontier, Explicit Iteration Counter and Virtual Warp-centric Programming

```
#define BLOCK_QUEUE_SIZE 8192

__global__ void kernel_cuda_frontier_queue(int iteration, ...)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    __shared__ int input_queue_size;

    if (threadIdx.x == 0)
    {
        input_queue_size = *input_queue;
    }
    __syncthreads();

    __shared__ int queue_size;
    __shared__ int next_queue[BLOCK_QUEUE_SIZE];
    queue_size = 0;
    __syncthreads();

    for (int v = 0; v < input_queue_size; v += num_threads)
    {
        if (v + tid < input_queue_size)
        {
            int vertex = input_queue[v + tid + 1];

            for (int n = 0; n < v_adj_length[vertex]; n++)
            {
                int neighbor = v_adj_list[v_adj_begin[vertex] + n];

                if (result[neighbor] == MAX_DIST)
                {
                    result[neighbor] = iteration + 1;

                    // Add to queue (atomicAdd returns original value)
                    int position = atomicAdd(&queue_size, 1);
                    next_queue[position] = neighbor;
                }
            }
        }
        __syncthreads();

        __shared__ int global_offset;
        if (threadIdx.x == 0)
        {
            // First value is size of queue
            global_offset = atomicAdd(output_queue, queue_size);
        }
        __syncthreads();

        // Copy queue to global memory
        for (int i = 0; i < queue_size; i += blockDim.x)
        {
            if (i + threadIdx.x < queue_size)
            {
                output_queue[global_offset + i + threadIdx.x + 1] = next_queue[i + threadIdx.x];
            }
        }
        __syncthreads();

        queue_size = 0;
        __syncthreads();
    }
}
```

Figure 14: Vertex-centric Implementation with Frontier and Hierarchical Queue

```
__global__ void kernel_cuda_frontier(int *queue, ...) {
    /* Process elements in queue */
}

__global__ void kernel_cuda_frontier_scan(int *prefix_sum, ...) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;
    for (int v = 0; v < ceil_num_vertices; v += num_threads) {
        int vertex = v + tid;
        if (vertex < num_vertices) {
            if (updated[vertex]) visited[vertex] = true;
            prefix_sum[vertex] = frontier[vertex] = updated[vertex];
            updated[vertex] = false;
        }
        else if (vertex < ceil_num_vertices) {
            frontier[vertex] = false;
            prefix_sum[vertex] = 0;
        }
    }
}

__global__ void kernel_cuda_combined_sweeps(int *g_odata, int *g_idata, int n) {
    __shared__ int temp[1024]; // Determines max. size of frontier (#vertices)
    int thid = threadIdx.x;
    int offset = 1;
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n >> 1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }

    if (thid == 0) temp[n - 1] = 0; // clear the last element

    for (int d = 1; d < n; d *= 2) { // traverse down tree & build scan
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            int t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1];
}

__global__ void kernel_cuda_generate_queue(
        int *prefix_sum, bool *frontier, int *queue, int num_vertices) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < num_vertices && frontier[tid]) queue[prefix_sum[tid] + 1] = tid;

    // Set size of queue
    if (tid == num_vertices - 1) queue[0] = prefix_sum[tid] + (int) frontier[tid];
}
```

Figure 15: Vertex-centric Implementation with Frontier and Prefix Sum (only shared memory)

```
__global__ void kernel_cuda_per_edge_basic(
    int *v_adj_from, int *v_adj_to, int num_edges, int *result, bool *still_running)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    for (int e = 0; e < num_edges; e += num_threads)
    {
        int edge = e + tid;

        if (edge < num_edges)
        {
            int to_vertex = v_adj_to[edge];
            int new_len = result[v_adj_from[edge]] + 1;

            if (new_len < result[to_vertex])
            {
                result[to_vertex] = new_len;
                *still_running = true;
            }
        }

    }
}
```

Figure 16: Unoptimized Edge-centric Implementation

```
__global__ void kernel_cuda_per_edge_frontier_numbers(int iteration, ...)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;

    for (int e = 0; e < num_edges; e += num_threads)
    {
        int edge = e + tid;

        if (edge < num_edges)
        {
            int from_vertex = v_adj_from[edge];
            int to_vertex = v_adj_to[edge];

            if (result[from_vertex] == iteration && result[to_vertex] == MAX_DIST)
            {
                result[to_vertex] = iteration + 1;
                *still_running = true;
            }
        }
    }
}
```

Figure 17: Edge-centric Implementation with Frontier and Iteration Counter