Efficient Hybrid Breadth-First Search on GPUs

Takaaki Hiragushi[1] and Daisuke Takahashi[2]

[1] Graduate School of Systems and Information Engineering, University of Tsukuba
[2] Faculty of Engineering, Information and Systems, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
hiragushi@hpcs.cs.tsukuba.ac.jp, daisuke@cs.tsukuba.ac.jp

Abstract. Breadth-first search (BFS) is a basic algorithm for graph processing. It is a very important algorithm because a number of graph-processing algorithms use breadth-first search as a sub-routine. Recently, large-scale graphs have been used in various fields, and there is a growing need for an efficient approach by which to process large-scale graphs. In the present paper, we present a hybrid BFS implementation on a GPU for efficient traversal of a complex network, and we achieved a speedup of up to 29x, as compared to the previous GPU implementation. We also applied an implementation for GPUs on a distributed memory system. This implementation achieved a speed of 117.546 GigaTEPS on a 256-node HA-PACS cluster with 1,024 NVIDIA M2090 GPUs and was ranked 39th on the June 2013 Graph500 list.

Keywords: GPGPU, Breadth-first search, Graph500.

1 Introduction

Graph-based structures are useful for solving various problems, and processing real-world data (e.g., social network and web link network) as a graph is helpful for obtaining beneficial information. Some applications require processing of large-scale graphs. Therefore, efficient algorithms to process large-scale graphs are necessary. Graph processing is a typical data-intensive application. The Graph500 [1] benchmark has been in place since 2010 for the comparison of supercomputers based on the performance of data-intensive applications.

Breadth-first search (BFS) is an essential algorithm among the graph processing algorithms, and some complex graph processing algorithms use BFS as a sub-routine. The Graph500 benchmark uses BFS on a large-scale graph as a problem. The hybrid BFS algorithm [2] works efficiently to traverse graphs with a small-world property such as a complex network. The hybrid BFS achieved impressive speedups on CPU-based systems but the hybrid BFS implementation for GPU-based systems has not yet been evaluated. However, the concept of the hybrid BFS does not depend on target architecture, and the hybrid BFS should also work efficiently on GPU-based systems with a suitable implementation.

In the present paper, we present an implementation of a hybrid BFS that works efficiently on GPU-based systems and the result of an evaluation of the proposed implementation.

J. Kołodziej et al. (Eds.): ICA3PP 2013, Part II, LNCS 8286, pp. 40–50, 2013.

```
function breadth-first-search (root)
  frontier ← { root }
  parents ← [ -1, -1, ..., -1 ]
  while frontier ≠ ∅
    next ← top-down-step(frontier, parents)
    frontier ← next
  end while
  return parents
```

Fig. 1. Procedure of level-synchronized BFS [2]

2 Hybrid Breadth-First Search

Level-synchronized BFS is a widely used method for parallel BFS. This method performs BFS using the procedure described in Fig. 1. The level of the vertices in the frontier remains constant. The level is the hop distance from a root vertex. In order to parallelize BFS, the top-down-step function is performed in parallel.

The hybrid BFS is based on level-synchronized BFS and uses two BFS approaches: top-down BFS and bottom-up BFS. The hybrid-BFS switches the approach used to process each level for efficient traversal. This algorithm is effective on a graph having a small-world property. In this section, we describe an outline of the hybrid BFS.

In top-down BFS, the computation of each iteration is performed by checking the visitation status of all vertices that neighbor any vertex in the frontier. When unvisited vertices are found in this procedure, the vertices are added to the set of the next vertices. The algorithm of top-down BFS is given in Fig. 2(a).

Unlike top-down BFS, bottom-up BFS processes each level by checking all edges not visited by its endpoint. The algorithm of the bottom-up BFS is described in Fig. 2(b). Bottom-up BFS works efficiently when many vertices are contained in a set of results of each traversal step. The sets of results for each step often contain many vertices when a target graph has a small-world property. Therefore, using bottom-up BFS may improve traversal performance.

The hybrid BFS heuristically determines which approach is used to process the next level. Traversal begins from top-down BFS and switches to bottom-up BFS when $m_f > m_u/\alpha$ is satisfied and switches back to top-down BFS when $n_f > |V|/\beta$ is satisfied, while traversing by bottom-up BFS. In those equations, m_f is the sum of the out-degrees of vertices in the frontier, m_u is the sum of the out-degrees of unvisited vertices, n_f is the number of vertices in the frontier, and α and β are heuristic parameters.

3 Single-GPU Algorithm

3.1 Graph Data Representation

The proposed implementation uses an adjacency matrix represented by compressed sparse rows (CSR) to represent graph data. It consists of offsets for

```
function top-down-step
            (frontier, parents)
  next ← ∅
  for u ∈ frontier do
    for v ∈ neighbors(u) do
      if parents[v] = -1 then
        parents[v] ← u
        next ← next ∪ { v }
      end if
    end for
  end for
  return next
```

(a) The top-down BFS approach

```
function bottom-up-step
            (frontier, parents)
  next ← ∅
  for v ∈ vertices
    if parents[v] = -1
      for n ∈ neighbors[v]
        if n ∈ frontier
          parents[v] ← n
          next ← next ∪ { n }
        end if
      end for
    end if
  end for
  return next
```

(b) The bottom-up BFS approach

Fig. 2. Single step of the top-down BFS and the bottom-up BFS approaches [2]

each row and column indices of each non-zero element. In order to treat the directed graph in bottom-up BFS, the proposed implementation also prepares a transposed graph in CSR format.

3.2 Top-Down BFS

The proposed top-down BFS implementation is based on the two-phase method presented by Merrill et al. [5]. This method divides the traversal of each level into two kernels: Expand and Contract. Expand kernel enumerates all vertices that are neighbors of any vertex in the frontier of the current level. To construct an array of the frontier efficiently, this kernel uses different methods based on the degree of vertices. Contract kernel removes duplicated vertices and previously visited vertices from the results of the expand kernel. This kernel uses visitation status bitmap, label data and some heuristic techniques to perform this task efficiently.

The original implementation of the two-phase method does not correctly maintain the visitation status bitmap to avoid atomic operations, but the proposed top-down BFS implementation uses an atomic operation to update the visitation status bitmap because efficient bottom-up BFS implementation requires a precise visitation status bitmap. Therefore, the proposed top-down BFS implementation is inefficient compared to the original two-phase implementation. Counting the number of vertices and calculating the sum of out-degrees of vertices in the frontier are needed in order to determine an approach for processing the next level. We implemented an additional kernel for the purpose of calculation.

3.3 Bottom-Up BFS

We implemented a new kernel for bottom-up traversal on a GPU. We describe this algorithm in Fig. 3. The kernel consists of four procedures.

```
shared prefix_scan[block_size + 1], scratch[SCRATCH_SIZE]
global_id ← block_id × block_size + thread_id
unvisited ← not visited[global_id]
popcnt ← population_count(unvisited)
prefix_scan ← cta_prefix_scan(popcnt)
count ← prefix_scan[block_size]
modified_vis ← 0, p ← 0, rp ← 0
while p < count
  // Generate an Array of Unvisited Vertices
  s_offset ← prefix_scan[thread_id] + rp − p, s_begin ← s_offset
  while rp < popcnt and s_offset < SCRATCH_SIZE
    shift ← 31 − count_leading_zeros(unvisited)
    scratch[s_offset] ← (global_id << 5) + shift
    unvisited ← unvisited xor (1 << shift)
    rp ← rp + 1, s_offset ← s_offset + 1
  end while
  syncthreads()
  remainder ← min(count − p, SCRATCH_SIZE)
  // Check the Visitation Status of Neighbors of Unvisited Vertices
  i ← thread_id
  while i < remainder
    v ← scratch[i], u ← 0
    cur ← transposed_rows[v], end ← transposed_rows[v + 1]
    while cur < end
      u ← transposed_colind[cur]
      if texture_fetch(tex_visited, u >> 3) and (1 << (u and 7)) ≠ 0 then
        break
      end if
      cur ← cur + 1
    end while
    // Update Label and Visitation Status
    scratch[i] ← 0
    if cur < end then
      scratch[i] ← (1 << (v and 31))
      label[v] ← u
    end if
    i ← i + block_size
  end while
  p ← p + SCRATCH_SIZE
  syncthreads()
  // Gather the Updated Visitation Status
  for i ∈ [s_begin, s_offset)
    modified_bits ← modified_bits or scratch[i]
  end for
  synthreads()
end while
next_vis[global_id] ← modified_bits
```

Fig. 3. Bottom-up BFS on a GPU

Generate an Array of Unvisited Vertices: First, each thread reads 32 bits from the bitmap denoting the visitation status on the global memory. Next, each thread constructs an array of unvisited vertices on shared memory from the bitset that was read. If there are too many unvisited vertices to write to shared memory, this procedure writes as many unvisited vertices as possible and writes the remaining unvisited vertices after execution of the following procedure.

Check the Visitation Status of Neighbors of Unvisited Vertices: Each thread picks an unvisited vertex from the array that is constructed in the previous procedure, and the visitation status of the neighbors of the selected vertex is checked. The selected vertex will be contained in the next frontier if any of the neighbors has already been visited.

Update Label and Visitation Status: When a thread finds a visited vertex among the neighbors of a selected vertex, the label is updated immediately by this thread, but the visitation status is tentatively written to shared memory in order to avoid atomic operation to global memory. The temporary visitation status will be written into global memory by the next procedure.

Gather the Updated Visitation Status: Each thread collects the new visitation status corresponding to vertices that are assigned in the first procedure. The 32-bit-width partial visitation bitmap is computed from the collected data and written to global memory.

3.4 Optimizations for Bottom-Up BFS

Arranging the Adjacency Matrix: The loop that checks the visitation status of neighbors can be aborted immediately when an already visited vertex is found. Therefore, checking the visitation status of vertices that have a high probability of being visited in the early part of this loop improves the traversal performance. Since the level of vertices that have large in-degree is often small, the proposed implementation checks the visitation status in descending order of in-degree.

Remove Unreachable Vertices: With the exception of the root vertex, none of the vertices can be visited when the in-degree of the vertex is 0. It is wasteful to assign threads to check their neighbors. This reduces the number of wasteful tasks by eliminating vertices that are unreachable before generating the array of unvisited vertices.

Using the Texture Cache: Checking the visitation status of neighboring vertices results in significant random accessing of the global memory. The performance of this memory accessing is improved by using cache memory. The same bitmap is also accessed by generating an array of unvisited vertices. However, the present implementation does not use the texture cache for this task, because this may result in cache pollution and may decrease the random access performance. Moreover, using cache memory for this task provides little benefit because this accessing always coalesces. Therefore, the proposed implementation does not use a cached access path for this task to improve the overall performance.

Reducing the Use of Shared Memory: The procedures presented herein use shared memory to store the unvisited vertices array and the updated visitation status. If shared memory is allocated separately for each purpose, the shared memory usage becomes large, which causes a decrease in occupancy. The shared memory usage can be reduced by eliminating unnecessary information from an array of an unvisited vertex. Since the procedure for modifying the visitation status bitmap does not use the upper bits of the vertex index, the use of arrays on shared memory for these purposes can be reduced by merging the lower bits of the vertex index and the visitation status.

3.5 Switching Between Two Approaches

The proposed implementation selects an approach for processing each level heuristically. The detail of this method is described in Section 2. This implementation uses $\alpha = 48$ and $\beta = 20$ as tuning parameters.

Top-down BFS uses an array to represent a set of vertices, whereas bottom-up BFS uses a bitmap to represent a set of vertices. Since these two approaches use different representation formats, a conversion is required for approach switching. If the visitation status bitmap is updated correctly, no conversion is required when switching from top-down BFS to bottom-up BFS. However, conversion from a bitmap to an array is required when switching from bottom-up BFS to top-down BFS. The proposed implementation uses population count instruction and prefix scan to convert from a bitmap to an array.

4 Multi-node Algorithm

The proposed hybrid BFS implementation for GPU clusters is based on the hybrid BFS algorithm proposed by Beamer et al. [4] for a distributed memory system. We modified the proposed implementation in order to use GPUs for computation. In each computation kernel, differences between the algorithm for a single GPU and the algorithm for GPU clusters are small. We added a reordering kernel and a format conversion kernel to simplify communication. In the proposed implementation, sending from device memory and receiving to device memory occurs as MPI communication. We use CUDA extensions of MVAPICH2 to overlap copying over PCI Express and MPI communication.

4.1 Top-Down BFS

Since the result of our top-down BFS kernel is represented as an array of vertices, sequences for communication can be constructed by generating sequences of their parents. Data that will be sent to the same process must be consecutive in memory in order to achieve efficient communication. However, the results of the top-down BFS kernel are unordered. Therefore, these data must be sorted according to the destination process number.

Table 1. Suite of benchmark graphs

Benchmark	Reference	Benchmark	Reference
packing-500x100x100-b050	[7] [9]	wikipedia-20070206	[7]
com-YouTube	[8]	com-LiveJournal	[8]
soc-Pokec	[8]	wiki-Talk	[8]
random.1Mv.64Me	[6]	rmat.1Mv.64Me	[6]
kron_g500-logn20	[7] [9]		

4.2 Bottom-Up BFS

Since the results of the proposed bottom-up BFS kernel are represented as a bitset, a kernel to convert from a bitset to an array is required in order to construct sequences for communication. We implemented a new kernel for this conversion that consists of an atomic add operation and conversion in a single thread block. The procedure for conversion in a single thread block is similar to the procedure for generating an array of unvisited vertices in the bottom-up BFS kernel. The visitation status bitmap is also sent by MPI communication in bottom-up BFS. No additional processing to send the visitation status bitmap is required because the visitation status bitmap is the same bitmap that is used for computation.

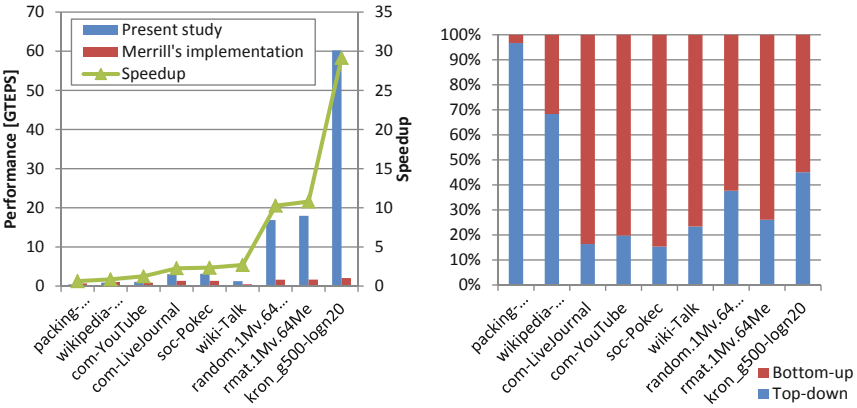
5 Evaluation

5.1 Single-GPU Algorithm

We evaluated the proposed single-GPU implementation by graphs listed in Table 1. These graphs are generated by GTGraph [6] or selected from the University of Florida Sparse Matrix Collection [7] and the Stanford Large Network Dataset Collection [8]. This dataset does not contain a graph that can be traversed without the use of bottom-up BFS when using the proposed implementation. We used NVIDIA Tesla M2050 for evaluation and evaluated the performance of Merrill's implementation [10] for comparison. Merrill's implementation consists only of an algorithm corresponding to top-down approach. Therefore, the most important difference between it and the proposed implementation is whether to use the bottom-up BFS.

The performance of the traversal is measured by building a BFS tree from randomly selected vertices and calculate the performance in TEPS (traversed edges per second) 64 times. The overall performance of the traversal is the median value of each measured performance. Copying from device memory to host memory is performed in order to verify the results after each construction, but the time required for this task is not included in the build time.

The overall performance for traversing each graph is shown in Fig. 4(a). The proposed implementation achieves a maximum speedup of 29x, as compared to Merrill's implementation, but does not work efficiently for some graphs. We show the ratio of the computation time of top-down BFS to the computation



(a) Overall performance of traversal

(b) Ratio of computation time of top-down BFS and bottom-up BFS

Fig. 4. Performance characteristics of the single GPU implmentation

time of bottom-up BFS in Fig. 4(b), which indicates that the time required for top-down BFS increases when the efficiency of the proposed implementation decreases. As mentioned in Subsection 3.2, the proposed implementation of top-down BFS cannot traverse more efficiently than Merrill's implementation. This is one possible cause of the observed relative performance degradation. On the other hand, the proposed implementation works efficiently when it uses bottom-up BFS appropriately. This results shows hybrid BFS is effective for improving graph traversal performance on GPUs.

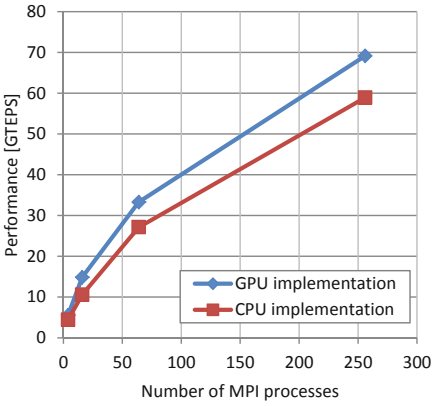
5.2 Multi-node Algorithm

We used the Graph500 benchmark to evaluate the multi-node implementation and evaluated the proposed implementation on the HA-PACS cluster. The specifications of HA-PACS are described in Table 2. Each node of HA-PACS has two CPUs and four GPUs. Each MPI process is assigned four CPU cores and one GPU, and each computation node is assigned four MPI processes. The problem scale is determined such that the number of vertices is equal to $2^{21}\times$ the number of MPI processes. We also measured the performance of the proposed hybrid BFS implementation that does not use GPUs as the target of comparison.

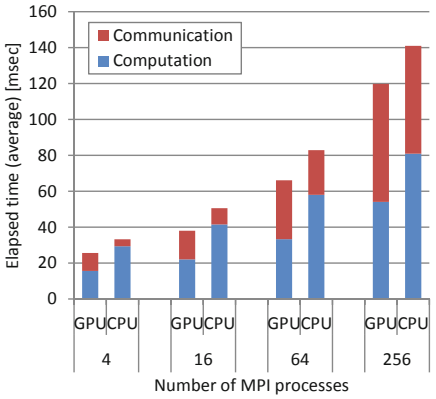
The overall performance of each implementation is described in Fig. 5(a). The results indicate that GPUs can accelerate the hybrid BFS. When running BFS on a multi-node system, MPI communication often takes a great deal of time. The time required for computation and communication of each implementation is shown in Fig. 5(b). This result indicates that the computation time is shortened by using GPUs, but the communication time become longer than the CPU implementation time. This difference in the communication time is thought to be due to the overhead of copying between host memory and device memory.

Table 2. Specifications of HA-PACS

CPU	Intel Xeon E5-2670 2.6 GHz × 2
Main Memory	DDR3 1,600 MHz 128 GB
GPU	NVIDIA Tesla M2090 × 4
GPU Memory	GDDR5 24 GB (6 GB per GPU)
Interconnection	InfiniBand x4 QDR × 2
Compiler	GCC 4.4.5 (-O2)
CUDA Toolkit	5.0
CUDA Compiler	nvcc release 5.0, V0.2.1221 (-gencode arch=compute_20,code=sm_20 -O2)
MPI	MVAPICH2 1.8
Number of computation nodes	268



(a) Overall performance of traversal



(b) Details of elapsed time

Fig. 5. Performance characteristics of multi node implmentations

6 Related Research

On multicore systems, Agarwal et al. [3] demonstrated that using a bitmap to manage the visitation status is effective for achieving higher performance. In the GPU-based system, Harish and Narayanan [11] introduced the first implementation of BFS on a GPU. Merrill et al. [5] proposed an efficient algorithm for graph traversal on GPUs.

On the cluster system, the communication time is often longer than the computation time and often becomes a bottleneck. Satish et al. [12] presented an efficient compression algorithm and pipelined computation and communication in order to achieve good scalability. On the GPU cluster system, Bernaschi et al. [13] reported that accelerating GPU-GPU communication by APEnet+ is beneficial for BFS.

7 Conclusion

In the present paper, we presented a hybrid BFS algorithm that can improve the performance of BFS on GPU-based architecture. The proposed implementation achieves a speedup of approximately 29x compared to the existing BFS implementation for the GPU. However, the proposed implementation cannot work efficiently in some cases. Improvement of the proposed implementation so as to achieve efficient traversal in such cases will be investigated in the future.

Moreover, GPUs were demonstrated to be beneficial for traversing a graph by the hybrid BFS on a cluster system. Using GPUs requires additional time for communication but shortens the computation time to the extent that the overall performance is improved. We have not yet considered communication in the proposed implementation. Improvement of communication (e.g., data compression and pipelining) is needed in order to achieve higher performance.

Acknowledgments. This research was partially supported by Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency (JST).

References

1. Brief Introduction of Graph 500, `http://www.graph500.org/`
2. Beamer, S., Asanović, K., Patterson, D.: Direction-Optimizing Breadth-First Search. In: Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, No. 12 (2012)
3. Agarwal, V., Petrini, F., Pasetto, D., Bader, D.A.: Scalable Graph Exploration on Multicore Processors. In: Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11 (2010)
4. Beamer, S., Buluç, A., Asanović, K., Patterson, D.A.: Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. Technical Report UCB/EECS-2013-2, EECS Department, University of California, Berkeley (2013)
5. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. In: Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012), pp. 117–128 (2012)
6. GTgraph: A suite of synthetic random graph generators, `http://www.cse.psu.edu/~madduri/software/GTgraph/`
7. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices/`
8. Stanford Large Network Dataset Collection, `http://snap.stanford.edu/data/`
9. 10th DIMACS Implementation Challenge, `http://www.cc.gatech.edu/dimacs10/`
10. back40computing - Fast and efficient software primitives for GPU computing - Google Project Hosting, `http://code.google.com/p/back40computing/`
11. Harish, P., Narayanan, P.J.: Accelerating Large Graph Algorithms on the GPU Using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)

12. Satish, N., Kim, C., Chhugani, J., Dubey, P.: Large-Scale Energy-Efficient Graph Traversal: A Path to Efficient Data-Intensive Supercomputing. In: Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, No. 14 (2012)
13. Bernaschi, M., Bisson, M., Mastrostefano, E., Rossetti, D.: Breadth first search on APEnet+. In: Proc. 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC 2012), pp. 248–253 (2012)