Sunghun Jo, Yuna Jeong, and Sungkil Lee. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY Vol. 0 No. 0: 1–15 Jan. 2018

GPU-Driven Scalable Parser for OBJ Models

Sunghun Jo, Yuna Jeong, and Sungkil Lee, Member, ACM

Department of Software, Sungkyunkwan University, Suwon, 16419 South Korea

E-mail: {shjo, jeongyuna, sungkil}@skku.edu

Received 2016-01-02

Revised 2018-01-025

Abstract This paper presents a scalable parser framework using graphics processing units (GPUs) for massive text-based files. Specifically, our solution is designed to efficiently parse Wavefront OBJ models texts of which specify 3D geometries and their topology. Our work bases its scalability and efficiency on chunk-based processing. The entire parsing problem is subdivided into subproblems the chunk of which can be processed independently and merged seamlessly. The within-chunk processing is made highly parallel, leveraged by GPUs. Our approach thereby overcomes the bottlenecks of the existing OBJ parsers. Experiments performed to assess the performance of our system showed that our solutions significantly outperform the existing CPU-based solutions and GPU-based solutions as well.

Keywords 3D Model, Wavefront OBJ, Parser, GPU

1 Introduction

In computer graphics and its associated areas, 3D scene information such as geometry, topology, hierarchical layouts, and surface appearance, often requires to be represented in an interchangeable format for many applications (such as 3D viewers, modelers, and renderers), which can be typically stored in files. There are numerous file formats for this purpose. Similar to other areas, it is also common to use two types of file formats, binary and text-based specifications. The binary specification is easier to directly load, thereby faster to load, but often less flexible for further editing and extensions. The text-based ones are more flexible in its expression and extensions, and thus, more common

This work was supported in part by the Mid-career and Global Frontier (on Human-centered Interaction for Coexistence) Research & Development programs through the National Research Foundation (NRF) grants (Nos. 2015R1A2A2A01003783, 2012M3A6A3055695), the Information Technology Research Center (ITRC) program (IITP-2017-2016-0-00312) supervised by the Institute for Information and Communications Technology Promotion (IITP), funded by the Korea Government (Ministry of Science, ICT and Future Planning), and Faculty Research Fund, Sungkyunkwan University, 2011. Correspondence author: Sungkil Lee.

in practice. However, it is generally slow to load such formats, because they require to be additionally parsed to be loadable in the computer memory.

One of the most common text-based 3D file formats is the object (OBJ) file format, which was originally developed by Wavefront Technologies [①] (now merged with Alias/Autodesk). Its specification is relatively simple. It uses tags or commands to distinguish different geometric elements (e.g., '`v`', '`vt`', '`vn`', and '`f`' for vertex positions, texture coordinates, normals, and faces, respectively). Such elements are well mapped to the majority of geometric and visual applications, but as indicated already, the format is not directly loadable to the memory and incompatible with particular applications due to its text-based nature.

As the computing power of both CPUs and GPUs improves, sizes of 3D models become increasingly larger. Their text-based specifications are much worse in loading speed and storage effectiveness. Existing CPU-based parsers [1] cannot handle such large 3D models well. Typical loading time takes up to a few seconds for small files, but such large data may take up to dozens of minutes, impeding their use in interactive applications. Parallel parsers may help, but the serial nature and the inter-element dependency of the OBJ format make the parallelization difficult. This inspired us to explore how to efficiently parallelize an OBJ parser.

Apart from OBJ parsers, general text processing has been already utilizing parallel processing. The early approach used multi-core CPUs [2], but recent ones used GPUs. GPUs utilize massive (up to thousands of) cores for parallel processing [3, 4]. Their values for text processing have been already identified in many areas such as extensible markup languages (XMLs) [5], natural language processing [6], and the structured query language (SQL) processing [7].

The first GPU-based OBJ parser was presented by Possemiers and Lee in their seminal work [8]. They proposed how to utilize GPUs for parallel loading of OBJ files. The linefeeds, which are atomic units for subsequent steps, are found in initialization. Then, elementary attributes are found in parallel, and then, vertices are packed and sorted, followed by the parallel indexing of faces. All the steps are made parallel using NVIDIA Compute Unified Device Architecture (CUDA) [②] and Thrust library [9]. The work by Possemiers and Lee attained impressive speedup (up to 6–8×) compared with the existing CPU implementations. Nevertheless, we observed further room for higher performance, with which we deal in this work.

This paper presents a scalable and efficient framework for GPU-based parallel OBJ parsing. Our solution is partially similar to the previous work [8] in the structure of within-chunk processing. However, our solution is entirely chunk-based, which is designed for out-of-core processing to significantly improve the scalability to larger models. Chunks in our work are unorganized and split by an arbitrary memory size. This brings about additional challenges in chunk-based processing and integration, which we solve in this work. To integrate chunk-based processing well in our framework, we propose additional optimization schemes, including

[①] http://www.fileformat.info/format/wavefrontobj/egff.htm, Jul. 2015.
[②] http://docs.nvidia.com/cuda/, Dec. 2016.

tag-based line splitting and table-based parallel vertex buffer indexing.

To be more precise, our work is distinguished from the previous GPU-based OBJ parser [8] for what follows. First, ours better handles chunks, which is scalable to out-of-core data and also avoids the redundant memory copies. While the previous work used chunks only for file reading, our work uses chunks for the entire pipeline. Second, our solution delimits the line in the OBJ file by the *first* characters (tag) instead of the *last* characters (e.g., linefeeds). This avoids an additional operation for trimming comments. Third, in the vertex buffer indexing, our solution performs sorting on the basis of chunks instead of the entire vertices. Also, we use an integer-based key for sorting instead of vertex values. Typically, the GPU sorting has super-linear-time complexity, and thus, this strategy significantly improves the performance, making a great difference with the previous work.

The major contributions of our work include:

- a scalable chunk-based scheme for OBJ parsers;
- a tag-based efficient line splitting scheme;
- a table-based efficient vertex buffer indexing scheme.

2 Related Work and Preliminary Background

We first briefly review previous studies on general text file parsers. We then explain the specification of the OBJ file format and previous attempts for its parsers.

2.1 Parallel Text Parsers

Text parsers are one of the crucial components for many applications. The applications start data processing from file parsing, and the speed of parsers is highly relevant to overall performance in the presence of frequent accesses to external documents. With the advent of big data and very large datasets, high-speed text processing has been of a growing interest, in particular for parallel processing.

Major applications of parallel text processing include XML parsing, natural language parsing, SQL query processing in databases. The most basic form of parallel processing starts from the utilization of multiple CPU cores [2, 10, 11]. Commonly, they first do pre-parsing to understand the structure of documents, and parse individual lines/chunks in parallel. Another mainstream on acceleration has attempted with SIMD (single-instruction multiple-data) capabilities of modern CPUs, such as a bit stream based parsing scheme [12].

To gain more speedup, parallel text/grammar processing has been directed to GPUs, inspired by many successful examples of GPU-based data processing. The effectiveness of GPU-based text processing has been proven to great extents, including XML processing (query optimization [5] and X3D parsing [13]), SQL query processing [7], and natural language processing [14, 15]. To resolve real-world issues of applying GPU processing to applications, many studies have been also dedicated to improvements of the load balancing between multi-core CPUs and GPUs [5, 6], and dense-to-sparse problem transformation [16].

2.2 Wavefront OBJ File Format

The OBJ file format uses tags to specify geometric elements in a 3D model. 'v' tag defines 3D position of a vertex. 2D texture coordinates and 3D normals are specified by 'vn' and 'vt', respectively. 'f' tag defines face elements with respect to integer indices/references to the previously defined vertex attribute lists; the starting index is one instead of zero. '#' tags indicate comments. Fig. 1 shows a simple OBJ file that specifies four vertex positions, four texture coordinates, one (shared) vertex normals, and the topology of vertices (faces); a quad of two triangles is here specified. Optionally, 'g' tag can be used to specify hierarchical grouping of faces (not shown here for simplicity).

```
# reference to an external material file
mtllib quad.mtl
# vertex positions
v -1.0 -1.0 0.0
v 1.0 -1.0 0.0
v 1.0 1.0 0.0
v -1.0 1.0 0.0
# texture coordinates
vt 0.0 0.0
vt 1.0 0.0
vt 1.0 1.0
vt 0.0 1.0
# vertex normals
vn 0.0 0.0 1.0
# face elements (ordered by v/vt/vn)
f  1/1/1 2/2/1 3/3/1
f  1/1/1 3/3/1 4/4/1
```

Fig. 1. An example OBJ file of two triangles forming a quad.

The OBJ models typically accompany material template library (MTL) files that specify materials such as diffuse and specular reflectance. They are usually small and easy to parse, and we do not take them into account in this work.

2.3 Previous OBJ Parsers

Due to the simple specification of the OBJ format, there have been many available implementations of OBJ parsers, including TinyOBJ [3] and MeshLab [1]. They are mostly based on CPU processing and similar in structures (see Fig. 2 for overview). An OBJ file is read into a file stream, and is parsed for each line. The vertex attributes (positions, texture coordinates, and normals) are simply accumulated as arrays, and the array of faces/polygons stores references (typically, integer indices) to the vertex attributes.

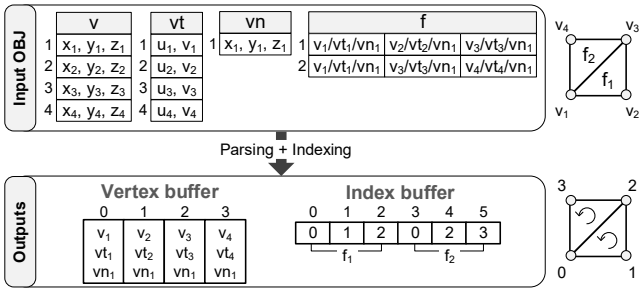

Fig. 2. Overview of OBJ parsing in typical CPU parsers.

The original references to vertex attributes in faces contain repeated vertices, and thus, need to be translated to a new single vertex buffer only of unique entries. This often requires a hash table to efficiently avoid the redundancy. Also, the three different attribute indices need to be consolidated as a new unified index buffer, because the majority of modern rendering APIs (e.g., OpenGL and Microsoft Direct3D) allow a single index buffer for high-performance rendering. This often requires interleaved arrays of aggregate vertex attributes to use a single index buffer for multiple attributes.

The previous CPU-based parsers are generally

[3] http://syoyo.github.io/tinyobjloader/, Dec. 2016.

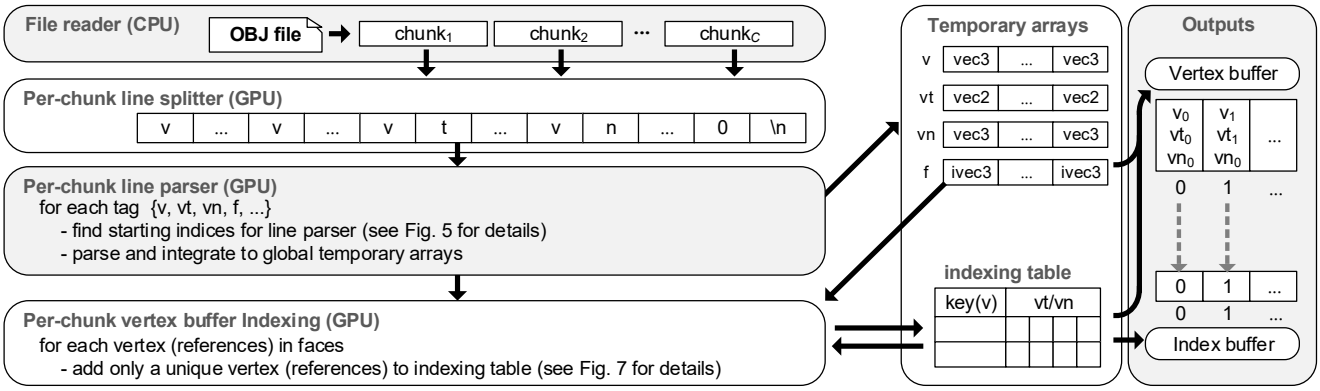

Fig. 3. Overview of our OBJ parser framework.

slow, and a GPU-based scheme (with NVIDIA CUDA and Thrust [9]) has been recently proposed by Possemiers and Lee [8]. The parsing is greatly parallelized, which proves the benefit of GPUs. This work motivates us to better utilize GPUs and optimize the scheme for higher performance. Particularly, we introduce an entirely chunk-based scheme for scalability. Also, the vertex buffer indexing, requiring sorting, was significantly improved with the table-based lightweight scheme.

3 Overview of Framework

This section provides a preliminary overview of our framework for parallel OBJ parsing (see Fig. 3 for illustration). Our framework consists of four stages: the chunk-based file reader, the tag-based line splitter, the element-wise line parser, and the vertex buffer indexer. Our framework is entirely chunk-based, and processes each chunk in a scalable way. We provide a detailed overview below and details in Sections 4, 5, and 6.

The first step is the chunk-based file reading. A typical buffer-based reading from a raw file stream, by default, incurs significant overhead and redundant memory copy (to the buffer). For better file access, we use an alternative method, the memory-mapped file to utilize the efficient *virtual memory* management provided by the operating systems. The file is repeatedly read on the basis of chunks, the chunk of which is partially mapped to virtual memory addresses. The chunks are usually small and can be directly loaded into GPU memory for further in-memory chunk parsing.

The next step is line splitting, which delimits lines in the chunk. Our line reader delimits the lines by the first characters (i.e., element tags in OBJ file formats) instead of the last characters in the line (i.e., line feeds). This approach allows us to avoid an additional step to trim redundant lines, improving the speed of reading lines.

Then, the next step is parsing/translating individual lines to binary arrays from the text representations. We use our own ASCII-to-binary type conversion functions. Although there are many tag types in the OBJ file, this work focuses only on the most basic elements of OBJ tags, for geometric attributes and face topology.

The vertex buffer indexing is the last and the most important step to accelerate the entire loading. The ver-

tices for forming faces are randomly associated with multiple indices of the binary arrays of positions, normals, and texture coordinates. A typical CPU-based parser can use hashing, but it cannot be easily parallelized on GPUs. To cope with this, we propose an efficient chunk-based scheme to make this parallel, with less overhead of sorting. The sorting in our indexing scheme is performed on the basis of individual chunks. Hence, its overhead is much smaller than the previous one that requires to sort an array of entire vertices [8].

4 Scalable File Loader

This section describes preprocessing step of our framework that is performed before parsing. This stage includes the chunk-based file reading and tag-based line splitting.

4.1 Chunk-Based File Reader

The major difference of our work from the previous work [8] is the use of chunks through the entire pipeline (see Figs. 3 and 4). We first split a file to chunks by a user-defined size and repeatedly process each chunk until no chunks are read from the file. This scheme enables serial chunk processing, which is scalable to large-size models. Note that, however, each chunk is processed in parallel on GPUs.

As already mentioned in Section 3, a typical buffer-based file streaming may not be efficient due to the use of a temporary buffer and its copies. Even a single additional copy for each line access may incur non-trivial overhead. So, we use an alternative method for faster chunk-based file access.

Our chunk-based scalable scheme is realized using a memory-mapped file (MMF; e.g., `mmap` in Linux). The memory mapping of the file enables direct access to the file content of a finite range (here, the chunk size), as the chunks reside in the main memory. Hence, this avoids the use of temporary buffer and its repeated copies. This way is even more efficient owing to the automatic cache management of the virtual memory system, which is provided by operating systems. This simple change to the loader brings us a non-trivial (actually, quite high) speedup.

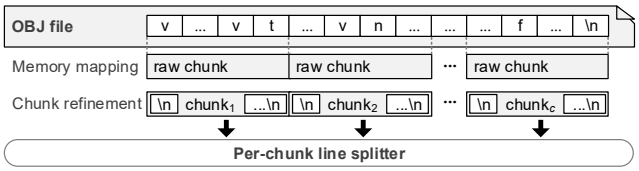

Fig. 4. Our chunk-based file reader for scalable parsing.

For each chunk, we do line splitting in the next step, but the last line of each chunk requires to be handled with care. When the chunk does not end up with the linefeed, the line needs to be padded with missing information to the next linefeed. This can be done at negligible cost; doing this in GPUs is obviously inefficient. Since such missing line content is not that big, we add a slight margin (e.g., 128 bytes) to the chunk for file mapping. Finding such a line feed is performed only for the ending of the chunk.

The chunk size matters in tuning for performance. Intuitively, too small or large sizes might not be efficient, and finding a moderate size is necessary. The size of the chunk is also related to the number of threads for GPU parsing, but it is hardly possible to explicitly formulate. We instead use a rule of thumb by an empirical

approach. We report the effects of the chunk size in our experiments (Section 7).

4.2 Tag-Based Parallel Line Splitter

Unlike the typical line-based file reading (e.g., using `fgets` in the C language), our scheme directly reads a chunk block. Hence, we need to split the chunks to lines in a separate GPU pass. GPU threads as many as the number of characters in the chunk are launched, and mark delimiting characters as 1 and others as 0. The former technique [8] used the linefeed ('`\n`') as a delimiting character. This works well, but a robust reader requires to trim meaningless lines (e.g., comments and blanks) and additional memory copies as well.

We improve the former technique using a tag-based splitting scheme. We use a heading character (i.e., tags) instead of the linefeed. The OBJ format declares a small number of tags, and we can easily exploit this scheme. Specifically, our idea is to find '`v`' and '`f`' in the line ('`vt`' and '`vn`' share the same heading '`v`'). Then, the line parser can start from the headings, and parse (a fixed number of) numeric entities. Hence, we do not have to process the entire lines to the end without trimming comments and blanks.

For robust reading, ambiguity in the comments needs to be avoided. In other words, '`v`' and '`f`' may exist in the comments, and thus, it is necessary to additionally verify that the preceding non-whitespace character is the linefeed. As for the exception in the first line, we prepend a linefeed for the beginning of each chunk (shown in Fig. 4).

Once the heading of each line is found, we mark the valid heading as 1 and the others 0. These marks are refined for processing of each tag in the line parsing.

5 Parallel Line Parser

This section first describes how to identify each element of the OBJ file separately and how to parse individual lines to binary representations.

5.1 Parallel Element Identification

Since we know the specific location each line begins, we can directly parse individual lines. However, one problem here is that we need to identify the indices of individual elements to store the binary output of the parser to the (intermediate) attribute arrays. In the CPU-based processing, the index identification is easy using a per-element counter. However, applying the index identification on GPUs requires atomic accesses to the counters, which serializes entire threads with significant stalls.

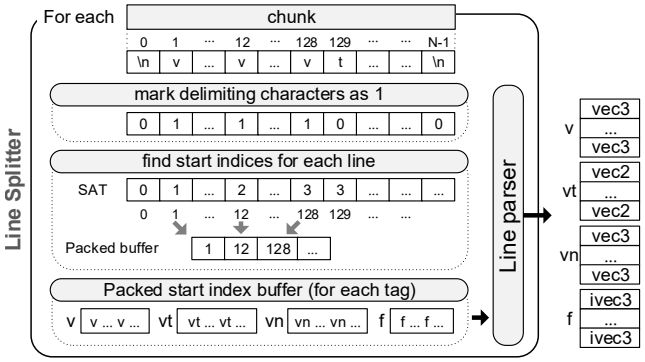

Fig. 5. Identification of the attribute start indices for the line parser.

To identify individual indices of line elements in parallel without atomic counters, we use a GPU-based parallel indexing scheme (see Fig. 5 for illustration and

Algorithm 1 for the pseudocode), which is proposed by Possemiers and Lee [8] for their vertex buffer indexing scheme. Briefly saying this, we separate the parsing for each attribute element and faces. In the beginning of each parsing, we mark only the tag we process as 1 and others 0. For instance, when we parse `v` tag only, we mark lines with `v` tag as 1. Then, we apply the *parallel prefix sum* (e.g., summed area table [17]), which accumulates forward elements from indices; we used `thrust::inclusive_scan` for implementation [9]. The resulting output indicates the starting positions (having non-zero indices) of each line for the element and where to store output (to the indices of the output buffer). We again tightly pack the output buffer using the indices and trigger parsing. In addition, this allows us to find the number of total lines to invoke the GPU threads for line parsing.

Algorithm 1 Line Splitter

```
Input:  C[N]: file chunk                                   ▷ N: size of chunk
Output: P[M]: tag index buffer                             ▷ M: size of tag index buffer
/* Kernel Code */
procedure MarkTag(C)
    B[tid] ← 0                                             ▷ B: temp. buffer to store marks
    if C[tid] is tag then B[tid] ← 1                       ▷ tid: thread ID
    return B
/* Kernel Code */
procedure Packing(C, B)
    if C[tid] is tag then P[B[tid] − 1] ← tid              ▷ store tag index
    return P
/* Host Code */
procedure LineSplitter
    B ←MarkTag(C)
    B ← Thrust::inclusive_scan(B)                          ▷ parallel prefix-sum
    P ←Packing(C, B)
```

We note that lengths of individual lines need not to be found. This is because each tag has a fixed number of scalar values and we can serially parse the line until we find the same number of whitespace characters.

5.2 Line Parsing

Line parsing follows the same approach as [8] does. The parsing can be classified into vertex attribute (geometry) parsing and face (topology) parsing. While the vertex attributes are largely similar, based on vectorized real numbers, face information is based on the integer indices that are indirect references to the arrays of the vertex attributes. The pseudocode for the line parsing is given in Algorithm 2.

Vertex attributes are parsed only for real numbers, while faces only for integers. Since we know where each line starts, we can directly access each element. We used our own type conversion function; CUDA, used for our implementation, does not provide standard ASCII-to-float conversion (e.g., `atof`). After parsing is complete, we store the binary values to tightly packed arrays for each element (i.e., position, normals, texture coordinates, and faces).

Algorithm 2 Line Parser

```
Input:  C[N]: file chunk                                   ▷ N: size of chunk
Input:  P[M]: tag index buffer                             ▷ M: size of tag index buffer
Output: V[M]: vertex attribute buffer                      ▷ one of {pos, tex, norm}
Output: F[K]: face (index) buffer                          ▷ K: number of faces × 3
/* Kernel Code */
procedure ParseLine
    if C[P[tid]] is a vertex tag then                      ▷ parse for vertex
        for each token t[i] do v[i] ← atof(t[i])
        V[tid] ← v
    else if C[P[tid]] is a face tag then                   ▷ parse for triangle
        for each token t[i] do f[i] ← atoi(t[i])
        for each vertex attribute j do                     ▷ {0,1,2} for "v/vt/vn"
            F[3 ∗ tid + j] ← (f[3j], f[3j + 1], f[3j + 2])
```

While vertex attribute formats are nearly fixed by the types of tags (e.g., 3 for position and normals and 2 for texture coordinates), face formats can be variable by the types of polygons and optional references. However, modern OpenGL deprecates general polygons types except triangles, and we assume that only up to three

vertices can be present in a single line. As for the optional references, the references to vertex positions are crucial, but those for normals and texture coordinates are optional. We handle the four possible cases, including "`v/vt/vn`", "`v/vt/`", "`v//vn`", and "`v//`" in the face definition. For the integer type conversion, we also used our own `atoi()`, similar to our `atof()`.

6 Scalable Parallel Indexing of Vertex Buffers

This section describes the problem of vertex buffer indexing and our approach for parallel indexing.

6.1 Background on Vertex Buffer Indexing

The last stage builds new vertex and index buffers readily available for rendering by vertex buffer indexing (see Fig. 2 for illustration). In the previous stage, we are given the vertex attributes of positions, normals, and texture coordinates, and their indices/references in the face elements. However, modern rendering APIs typically allow only a *single* index buffer, and we need to build a new vertex buffer, interleaving three vertex attributes (e.g., see Fig. 6), and use references to the new vertex buffer as a new index buffer.

```
struct vertex { vec3 pos; vec2 tex; vec3 norm; };
```

Fig. 6. A simple definition of a `vertex` structure in C++ language.

A straightforward solution to build a new vertex array is to create a new vertex for each face element. However, typical OBJ files contain substantial amounts of repeated vertices (having the same references to vertex attributes). Hence, it is necessary to maintain only unique entries in the vertex buffer. For this, a typical CPU indexing can simply use a *hash table*, which uses the three attribute indices as a key. However, a GPU-based scheme cannot easily exploit the hashing. The hashing requires atomic accesses to a single table, and leads to conflicts in parallel writing. Hence, the main challenge here is how to avoid the use of a hash table and how to avoid conflicts in writing the same entries into the table.

The previous approach [8] used sorting to cluster vertices in proximity, the predicate of which compares the real-number values by positions, normals, and texture coordinates in a row. For implementation, Possemiers and Lee used `thrust::sort`. By this way, the same vertices can be easily detected, also removing repeated entries. However, this requires a heavy sorting of the *entire* vertices, significantly degrading performance. To cope with this problem, we propose a much efficient way based on the indexing table, which is also scalable to large models.

6.2 Table-Based Vertex Buffer Indexing

Our novel scheme for the vertex buffer indexing is table-based, which is scalable and avoids the previous issues as well. First, we use a global indexing table of a fixed size so that the indexing result of each chunk can be seamlessly accumulated, thereby making our indexing scalable. Second, we perform the vertex sorting on the basis of each chunk. Since the number of vertices in each chunk is much smaller, the sorting overhead is greatly reduced; recall that the sorting overhead scales with at least $O(N \log N)$. Third, we use integer indices

(precisely, the position attribute v) as keys to the indexing table, which also lowers key-comparison overhead for sorting. As a result, our indexing highly improves performance in comparison to the previous work [8].

The *indexing table* is a key data structure of our efficient vertex buffer indexing, which interacts with each chunk and is translated to a new vertex buffer and index buffer in the end (see Fig. 7). The indexing table is actually a simple 2D array. Each row is assigned by the position index of each vertex in the face elements, and the columns hold unique combinations of indices of normals and texture coordinates. In practice, the unique combinations are defined by the adjacent faces and not too many. So, we pre-allocate the table with a fixed size (in our experiment, 7 columns suffice in practice). Actually, this serves as a fixed linked list, because GPUs do not support a native linked list due to the lack of pointers.

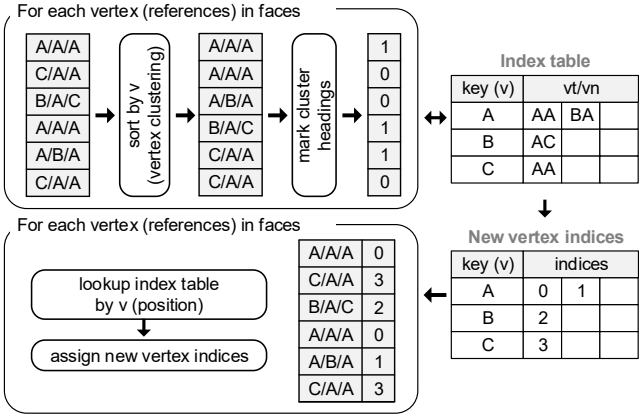

Fig. 7. Our table-based vertex buffer indexing scheme.

Based on the indexing table, our within-chunk indexing is performed as follows (see Algorithm 3 for the pseudocode). We first extract an unpacked vertex array (only of integer attribute references) from each face in each chunk. We then sort the unpacked vertex array using only the vertex position reference as a key. This sorting clusters vertices homogeneous in terms of position; this scheme is similar to that of [8], but much efficient due to less numbers of inputs. Then, we access the indexing table only for the first vertices for each position index. A thread for the same vertex reads the indexing table, and writes only unique pairs of normals and texture coordinate back to the table. Thereby, the vertices of the same key can be accessed in a single GPU thread without atomic accesses, significantly improving the overall performance.

Algorithm 3 Vertex Buffer Indexing

Input: $E[J]$: chunk from face (index) buffer ▷ J: chunk size
Input: $T[U][7]$: indexing table ▷ U: no. unique entries
Input: $C[U]$: index counter buffer ▷ holds no. of entries for each index
Output: $I[K]$: (final) index buffer ▷ K: no. faces × 3

1: */* Kernel Code */*
2: **procedure** UPDATEINDEXTABLE (T, E)
3: **if** $E[tid]$ is heading and not exists in $T[tid]$ **then**
4: $T[E[tid].\text{pos}][C[tid]] \leftarrow E[tid]$
5: $C[tid] \leftarrow C[tid]+1$
6: */* Kernel Code */*
7: **procedure** CREATENEWVERTEXINDEX(T)
8: **for** $i=0$ **to** $C[tid]-1$ **do**
9: **if** $T[tid][i]$ is valid **then** $T_n[tid][i]=1$
10: **else** $T_n[tid][i]=0$
11: **return** T_n
12: */* Kernel Code */*
13: **procedure** LOOKUPINDEXTABLE(T, E, T_n)
14: **for** $i=0$ **to** $C[tid]-1$ **do**
15: **if** $T[E[tid].pos][i]$ is $E[tid]$ **then**
16: **return** $T_n[E[tid].pos][i]$
17: */* Host Code */*
18: **procedure** VERTEXBUFFERINDEXING
19: **for each** chunk E **do**
20: SORT($E.pos$) ▷ using Thrust::sort
21: UPDATEINDEXTABLE(T, E)
22: $T_n \leftarrow$CREATENEWVERTEXINDEX(T) ▷ T_n: vertex index buffer
23: $T_n \leftarrow$ Thrust::inclusive_scan(T_n)
24: **for each** chunk E **do**
25: $I_n \leftarrow$ LOOKUPINDEXTABLE(T, E, T_n)
26: $I \leftarrow$ APPEND(I, I_n)
27: **return** I

When all the chunks are processed and the table is made complete, we create tightly packed vertex and index buffers from the table. The indexing table can

be sparse, because we allocated the over-size table up to a potentially largest column size. Since we know how many cells are filled, the buffer can be packed tightly. This again relies on the summed area table, which performs the prefix sum of the first cell of each row. From the final indices of each cell, we directly create a vertex buffer, which now assigns real values using the attribute references in the table. As for the index buffer, we need to read all the face chunks again, because the index buffer is ordered by the triangle order. For each face, we can easily identify the indices of its vertices by looking up the table using the position index. In this way, face elements can be translated to the final index buffer. One limitation here is that the globally defined table needs to reside in GPU memory, and thus, the maximum size of a file we can process is limited by the size of GPU memory. Nevertheless, this is more efficient than [8], because the scheme by Possemiers and Lee requires the entire file to be read to GPU memory. This scheme can be extended to a scalable way by using chunk-based access to the table (i.e., the index table is also split by its own chunks). Nonetheless, the size of actual vertex buffer is much smaller than its file size, because text formats are usually larger than the binary representations and there are many repeated face elements.

Our indexing scheme fits also with CPUs, and is more efficient than GPU-based schemes. The CPU processing bypasses the costly sorting and directly builds the table, because the table can be directly accessed. We also demonstrate the benefit of this scheme in Section 7.

7 Results

We implemented and experimented our system using NVIDIA CUDA API on an Intel Core i7 machine with an NVIDIA GTX 980 Ti graphics card and 16 GB main memory. Our parallel OBJ parser (POP) used two types of vertex buffer indexing, one with CPU indexing (CPOP) and the other with GPU indexing (GPOP).

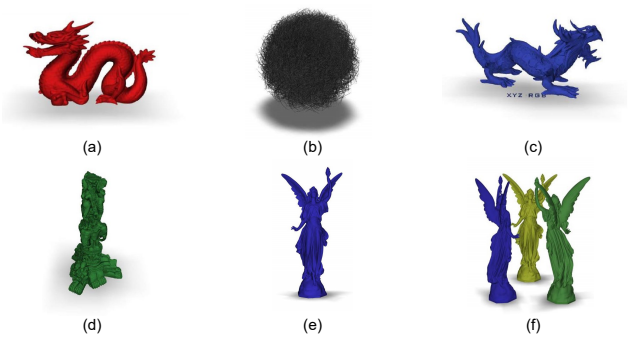

Fig. 8. Six 3D models used for our experiments. (a) Stanford Dragon (63.5MB, 0.4M, 0.9M, 14MB, 10MB). (b) Hairball (219MB, 1.5M, 2.9M, 47MB, 35MB. (c) XYZ Dragon (573MB, 3.6M, 7.2M, 116MB, 87MB. (d) Thai Statue (805MB, 5M, 10M, 160MB, 120MB. (e) Lucy (2.29GB, 14M, 28M, 449MB, 337MB. (f) Lucy3 (7.14GB, 42M, 84M, 1.3GB, 1.0GB). The numbers in the parentheses following the name of each model indicate the file size, the numbers of vertices and faces, the size of the vertex buffer, and the size of the index buffer, respectively.

The time complexity of our algorithms can be a combination of three major stages. Given the size N of the input data, we split them into M and P chunks for the parsing and vertex buffer indexing, respectively. Then, the time complexity of the line splitting is $O(N)$, because the parallel prefix sum of each chunk uses a linear-time implementation. The line parser simply runs in $O(N)$. The vertex buffer indexing sorts each chunk, leading to $O(N \log P)$. Consequently, the overall time complexity of our framework becomes $O(N \log P)$, implying the main bottleneck is the chunk sorting in the

vertex buffer indexing.

The experiments used six 3D OBJ models: Stanford Dragon, Hairball, XYZ Dragon, XYZ Thai Statue, Lucy, and Lucy3. Lucy3 is a tripled copy of Lucy. Fig. 8 shows their geometric complexities and how they look. Lucy and Lucy3 are used to prove the utility of our method for massive models. The models are not assigned materials, which are visualized with arbitrary colors.

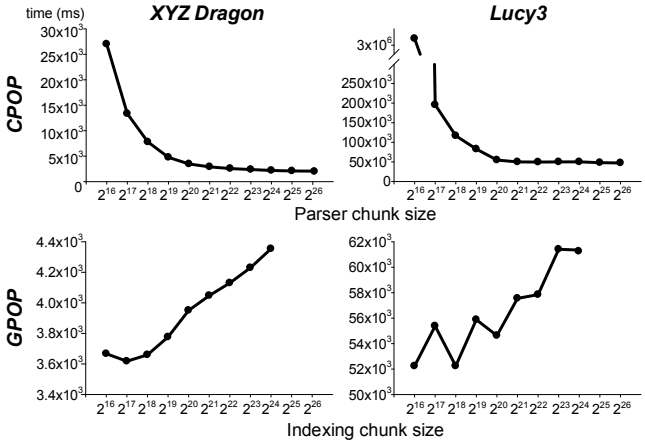

Fig. 9. Effects of chuck size of our OBJ parsers.

We used different chunk sizes of reader/parser and vertex buffer indexing. We chose optimal sizes based on our experiment (Fig. 9). Precisely, the parser chunk used 64MB. The chunk for vertex buffer indexing used 256KB only for GPOP; CPOP did not use chunks, because it can directly access the table in the memory.

We compared our solutions with two previous methods. The first is a well-known CPU-based one, MeshLab [1]. The other is the aforementioned GPU-based one [8], which serves as a reference method (hereafter, REF) in this work. We here excluded another well-known CPU-based implementation, TinyOBJ. Since TinyOBJ does not include the vertex buffer indexing (unlike our solutions and REF), a fair comparison is difficult without additional implementations.

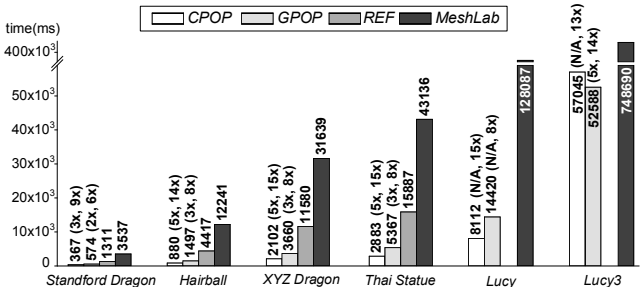

Fig. 10. Performance comparison of the four OBJ parsers. CPOP and GPOP indicate our solutions that use the vertex buffer indexing on CPUs and GPUs, respectively. Speedup factors are given against the REF and MeshLab.

Fig. 10 compares the performances of the four techniques. Overall, both of our methods greatly outperform all the previous ones. CPOP outperforms GPOP by a factor of 1.27–1.86 for all the models except Lucy3. In case of Lucy3, GPOP wins CPOP by a slight margin (with a speedup of 1.08). This shows the sort-less CPU indexing is better for medium-size models, but the GPU indexing better fits with massive models (also for a full GPU implementation). Unlike ours, REF was not able to load the two large models, Lucy and Lucy3, because REF needs the full in-memory processing in GPUs. This proves our solutions are better than REF in terms of scalability. The average speedup factors of CPOP are 4.9 and 13.7 with respect to REF and MeshLab, respectively, and those of GPOP are 2.8 and 9.0.

Fig. 11 shows the performance breakdown, which compares CPOP, GPOP, and REF. While our solutions separate file reading and line splitting, REF merges them as “importing”; REF uses multi-threading for file reading and line splitting. Lucy and Lucy3 are reported

only for ours. Overall, our solutions are more efficient than REF in all the steps. The largest differences are found in the vertex buffer indexing, and other stages are also more efficient. CPOP is more efficient due to its faster vertex buffer indexing, but the difference is less manifested for large models (Lucy3).

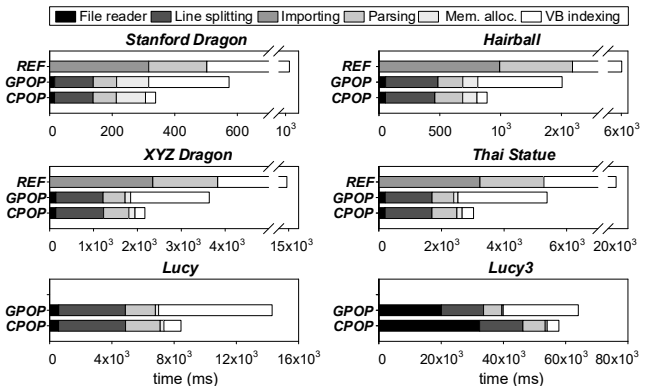

Fig. 11. Performance breakdown of CPOP, GPOP, and REF.

Lastly, we report the effects of chunk sizes on performance (Fig. 9). We use different chunk sizes for the reader/parser and the vertex buffer indexing. Overall, larger sizes of the parser chunk show higher performance but are saturated above 2^{26} (equivalent to 64MB). This results from the higher parallelism from the larger chunk sizes. On the other hand, optimal sizes of the indexing chunk is found around 2^{18} (equivalent to 256KB). Since a larger size of the indexing chunk degrades the performance in particular for the chunk sorting, a smaller (but not too small) chunk is preferred.

8 Discussions and Limitations

In the present work, we focused solely on the parsing of OBJ models. The framework and individual steps were designed for scalable and parallel processing of linewise commands. We envision our work can be extended for other text-based specifications.

At present, our implementation supports only common commands in the specification of OBJ models, i.e., '`v`', '`vn`', '`vt`', and '`f`'. In order for our work to be more practical, we are planning to extend the work to handle 'g' (group) and other commands (e.g., parametric primitives).

Our GPU-based vertex buffer indexing (GPOP) still requires to apply sorting, even though the sorting is efficient. This sorting makes a difference with CPU-based vertex buffer indexing (CPOP). A sort-less vertex buffer indexing would eventually lead to truly efficient OBJ parser. This is a good subject for further performance improvements. Another alternative strategy is a hybrid indexing that employs CPU-based indexing up to medium-size models and GPU-based indexing for huge models.

Another limitation of the vertex buffer indexing is that the global indexing table needs to be maintained in (GPU) memory. Hence, our work can load OBJ models the size of indexing table of which is less than the size of GPU memory. Since the size of modern graphics cards reach up to 6–8 GB, it is sufficient in practice. For further scalability, the global indexing table needs to be re-designed in a scalable way. Also, the table needs to be associated with the scalable rendering modules which can download the loaded vertex buffers into main memory and load them into GPU memory on demand. The subject is of interest for future work.

Our GPU-based approach for an OBJ parser is performed in a synchronous way, where tasks for each

chunk are serially performed either on GPUs or CPUs. While already efficient, we believe there is an additional room for optimizing performance. Such examples include asynchronous scheduling of tasks (modern GPUs support simultaneous threading and memory copy) and thereby hiding latencies in CPU-GPU communications.

One way to further improve the scalability of our framework is to employ multiple GPUs to distribute workload across multiple GPUs. Our line splitter and line parser can be improved with this scheme, because they perform in parallel for each chunk. However, our vertex buffer indexer interacts with the global indexing table. This might cause non-trivial communication overhead for synchronization across GPUs, which requires a well-designed parallel indexing scheme. Alternatively, the communication overhead can be reduced using a virtualized-GPU scheme (e.g., NVIDIA's scalable link interface), but this scheme is limited in its platform dependency. Further work and experiments are encouraged to explore optimal multi-GPU schemes of the vertex buffer indexing.

9 Conclusion

In this paper, we presented a scalable efficient parser framework for the Wavefront OBJ file format. All the components were designed on the basis of chunk processing, which is seamlessly scalable for many serial between-chunk processing. Within-chunk processing is made highly efficient, overcoming a few limitations of the previous study. The improvements include better file handling, tag-based line splitting, and table-based efficient vertex buffer indexing. To our knowledge, this work is currently the fastest parser framework for loading OBJ models.

In the future, we are planning to extend and improve our work in terms of scalability (full out-of-core processing, also with scalable rendering framework), a full support for OBJ commands, sort-less vertex buffer indexing, and an asynchronous scheduling to further hide latency.

Finally, we note that our work starts from one of the text-based file specification but can be extended to a more general text processing. Well-designed processing strategy and performance optimization techniques would be crucial components for such general text file parser.

Acknowledgment Stanford Dragon, XYZ Dragon, XYZ Thai Statue, and Lucy 3D models are provided by the courtesy of the Stanford 3D Scanning Repository and the Hairball model by Samuli Laine, Tero Karras, and Morgan McGuire at NVIDIA.

References

[1] Cignoni P, Corsini M, Ranzuglia G. Meshlab: an open-source 3D mesh processing system. *Ercim news*, 2008, 73(45–46):6.

[2] Lu W, Chiu K, Pan Y. A parallel approach to XML parsing. In *Proc. IEEE/ACM Int. Conf. Grid Computing*, 2006, pp. 223–230.

[3] Ghorpade J, Parande J, Kulkarni M, Bawaskar A. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[4] Han T D, Abdelrahman T S. hicuda: High-level gpgpu programming. *IEEE Trans. Parallel and Distributed Systems*, 2011, 22(1):78–90.

[5] Si X, Yin A, Huang X, Yuan X, Liu X, Wang G. Parallel optimization of queries in xml dataset using gpu. In *Proc. Int.*

Symp. Parallel Architectures, Algorithms and Programming, 2011, pp. 190–194.

[6] Johnson M. Parsing in Parallel on Multiple Cores and GPUs. In *Proc. Australasian Language Technology Association Workshop*, 2011, pp. 29–37.

[7] Bakkum P, Skadron K. Accelerating SQL database operations on a GPU with CUDA. In *Proc. Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 94–103.

[8] Possemiers A L, Lee I. Fast OBJ file importing and parsing in CUDA. *Computational Visual Media*, 2015, 1(3):229–238.

[9] Hoberock J, Bell N. *Thrust*, 2014. http://thrust.github.io/.

[10] Head M R, Govindaraju M. Parallel processing of large-scale xml-based application documents on multi-core architectures with piximal. In *Proc. IEEE Int. Conf. on eScience*, 2008, pp. 261–268.

[11] Li X, Wang H, Liu T, Li W. Key elements tracing method for parallel XML parsing in multi-core system. In *Proc. Int. Conf. Parallel and Distributed Computing, Applications and Technologies*, 2009, pp. 439–444.

[12] Cameron R D, Herdy K S, Lin D. High performance xml parsing using parallel bit stream technology. In *Proc. Conf. the center for advanced studies on collaborative research: meeting of minds*, 2008, p. 17.

[13] Hou Q, Zhou K, Guo B. BSGP: bulk-synchronous GPU programming. *ACM Trans. Graphics*, 2008, 27(3):19.

[14] Canny J, Hall D, Klein D. A multi-teraflop constituency parser using gpus. In *Proc. Conf. Empirical Methods in Natural Language Processing*, 2013, pp. 1898–1907.

[15] Lewis M, Lee K, Zettlemoyer L. Lstm ccg parsing. In *Proc. Annual Conf. North American Chapter of the Association for Computational Linguistics*, 2016.

[16] Hall D L W, Berg-Kirkpatrick T, Klein D. Sparser, Better, Faster GPU Parsing. In *ACL (1)*, 2014, pp. 208–217.

[17] Hensley J, Scheuermann T, Coombe G, Singh M, Lastra A. Fast Summed-Area Table Generation and its Applications. *Computer Graphics Forum*, 2005, 24(3):547–555.

Sunghun Jo received the B.S. degree in computer engineering at Hansei University (2016). He is a M.S. student in computer engineering at Sungkyunkwan University. His main research interest is real-time rendering.

Yuna Jeong received the B.S. degree in computer engineering at Korea Polytechnic University (2012). She is a Ph.D. student in computer engineering at Sungkyunkwan University. Her main research interest is real-time rendering.

Sungkil Lee received the B.S. and Ph.D. degrees in materials science and engineering and computer science and engineering at POSTECH, Korea, in 2002 and 2009, respectively. He is currently an associate professor in the Software Department at Sungkyunkwan University, Korea. He was a postdoctoral researcher at the Max-Planck-Institut Informatik (2009–2011). His research interests include GPU rendering, GPU algorithms, virtual/augmented reality, perception-based rendering, and information visualization.