Parallel Loop Subdivision with Sparse Adjacency Matrix

Kechun Wang Renjie Chen[†]

University of Science and Technology of China

Abstract

Subdivision surface is a popular technique for geometric modeling. Recently, several parallel implementations have been developed for Loop subdivision on the GPU. However, these methods are built on complex data structures which complicate the implementation and affect the performance, especially on the GPU. In this work, we propose to simply use the sparse adjacency matrix which enables us to implement the Loop subdivision scheme in the most straightforward manner. Our implementation run entirely on the GPU and achieves high performance in runtime with significantly lower memory consumption than the state-of-the-art. Through extensive experiments and comparisons, we demonstrate the efficacy and efficiency of our method.

CCS Concepts

• ***Computing methodologies*** → ***Computer graphics; Mesh models;***

1. Introduction

Subdivision surfaces are widely used for various engineering and science applications, among which, Loop subdivision [Loo87] is one of the most popular subdivision schemes, given the popularity of triangular meshes in computer graphics. Throughout the last few decades, subdivision surface has been widely used in 3D modeling and supported by the major production software packages, among which OpenSubdiv [Pix] is the most popular tool that offers the user the ability to dynamically modify the vertex positions of the mesh. However, in many scenarios, the user often needs a real-time implementation in order to perform interactive modeling. Therefore, it has been an important and challenging task to accelerate the subdivision schemes.

Previous work. Shiue et al. [SJP05] propose to decompose the base mesh of the subdivision surface into patches, and then independently refine these patches to the required subdivision level adaptively. However, this method relies on a proper patch partition of the base mesh. Moreover, due to floating-point inaccuracies, the refined edges may differ between neighboring patches. Brainerd et al. [BFK*16] present a method for real-time rendering of subdivision surfaces by exploiting the bicubic representation using hardware tessellation. They reduced visual errors by subdividing the irregular regions. However, in the irregular regions, the subdivision surfaces remain inaccurate. Mlakar et al. [ZSS17] propose a GPU-adapted data structure for triangle meshes, which includes two matrices encoding the connection between vertices, faces and edges. Built on this data structure, Mlakar et al. [MWS*20] introduce a parallel implementation of the Catmull-Clark subdivision scheme, which runs entirely on the GPU and generates subdivision surface for quad mesh in real-time. They briefly mention that this method could be extended for Loop subdivision. However, it lacks discussion on how the specialized linear algebra GPU kernel programs in their method are generalized from quad meshes to triangular meshes, in order to achieve similar runtime performance. More recently, Dupuy and Vanhoey [VD22] proposed another approach to implement the Loop subdivision scheme using the fully fledged half-edge data structure, which however, is costly for the GPU as it contains redundant operators resulting in excessive memory consumption.

Contribution. In this work, we propose to simply use the adjacency matrix of the mesh in order to implement the Loop subdivision scheme. Our data structure is straightforward to build and use and is more compact than the state-of-the-art. With this simple data-structure, we developed a high-performance parallel implementation of the Loop subdivision scheme running entirely on the GPU, enabling rendering subdivision surfaces in real-time with low memory footprint.

2. Method

In the Loop subdivision scheme, the base mesh is iteratively refined by inserting points on the edges and spliting each triangle into four new ones. The base mesh in the Loop subdivsion is a triangle mesh. More specifically,

Base mesh is given by $\mathcal{M} = \langle \mathbf{P}, \mathbf{F} \rangle$, where $\mathbf{P} = (\mathbf{x}_1, \cdots, \mathbf{x}_n)$ is the

[†] Corresponding author.

This research is supported by the National Key R&D Program of China (2020YFC1523102), the National Natural Science Foundation of China (62072422), and the Anhui Provincial Natural Science Foundation (2008085MF195).

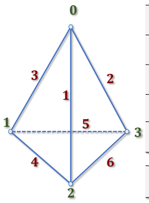
DOI: 10.2312/egs.20231012

vertex list containing the coordinates of all the vertices, i.e., $\mathbf{x_i} = (x_i, y_i, z_i)^T$ is the coordinates of the i$^{\text{th}}$ vertex, and $\mathbf{F} = (\mathbf{f}_1, \cdots, \mathbf{f}_m)$ is the face list containing the indices of the three vertices in each triangular face, with $\mathbf{f_i} = (v_0^i, v_1^i, v_2^i)^T$ being the i$^{\text{th}}$ face of the mesh.

When implementing a subdivision scheme, we need to access the connectivity of the mesh using some mesh data-structure. To minimize the overhead of constructing and provide efficient access for the mesh data-structure, we propose to use the sparse adjacency matrix.

Adjacency matrix. Given a mesh, the adjacency matrix encodes the vertex-vertex adjacency information, e.g., whether two vertices are connected, and in case of yes, the index of the edge between the vertex pair or the face containg this edge can be stored in the mesh. As a sparse matrix, the adjacency matrix can be stored in a compact format, e.g. the Compressed Sparse Column (CSC) format, in which the nonzeros of the matrix are ordered by their column indices. Specifically, we encode the sparse adjacency matrix $\mathcal{E}$ as a triplet $\{columnptr, rowindex, values\}$, the column indices of the nonzeros are compressed into *columnptr* which holds pointers to the first nonzeros in each column of the matrix, while *rowindex* and *values* contain the row indices and values of all the nonzeros.

In our adjacency matrix, for each edge in the mesh, we store its index and the two vertices: $\mathcal{E}_e(v_i, v_j)$ is the index of the edge connecting v_i and v_j, $\mathcal{E}_{verts}(v_i, v_j)$ is (v_i, v_k), where v_i, v_j, v_k are the indices of three vertices on the same triangular face. See the illustration in Algorithm 1 and Fig 1.



ptr	0			3			6			9		
ids	2	3	1	2	3	0	1	3	0	2	0	1
values	1	1	1	1	1	0	0	1	0	0	0	0
edge	1	2	3	4	5	–	–	6	–	–	–	–
verts	(0, 1)	(0, 2)	(0, 3)	(1, 3)	(1, 0)	(1, 2)	(2, 0)	(2, 1)	(2, 3)	(3, 0)	(3, 1)	(3, 2)

F	0	2	1	0	3	2	0	1	3	1	2	3

Figure 1: *The adjacency matrix $\mathcal{E}$ and face list* **F** *of a simple triangular mesh.*

2.1. Parallel Loop Subdivision

Each iteration of Loop subdivision consists the following 4 steps.

Adjacency Matrix is constructed in the first step of each subdivision step. The edge number and vertex index in the matrix are recorded at the same time. The *columnptr* is easy to compute since the valences of the new edge points are all equal to 6. $\mathcal{E}_e$ is the prefix sum of $\mathcal{E}_{value}$, i.e. $\mathcal{E}_e[n] = \sum_{i=1}^{n} \mathcal{E}_{values}[i]$ for each n. More details are shown in Algorithm 1.

Topology Refinement. Each triangle in the mesh splits into four new ones, and the inserted edge points can be numbered by the edge indices offset by the number of vertices before the mesh is subdivided. The index of edge between v_i and v_j is given by $\mathcal{E}_e(v_i, v_j)$. Algorithm 2 provides pseudo-code for topology refinement.

Algorithm 1: Construction of the Adjacency Matrix

```
Input: Mesh face list $\mathbf{F}$, $columnptr$ of $\mathcal{E}$, face number $m$
Output: Adjacency Matrix $\mathcal{E}$
foreach $i \in [0, m)$ do
    $(v_0, v_1, v_2) \longleftarrow \mathbf{F}[i]$;
    $\mathcal{E}_{rowindex}[columnptr[v_0]] \longleftarrow v_1$;
    $\mathcal{E}_{values}[columnptr[v_0]] \longleftarrow v_0 < v_1$;
    $\mathcal{E}_{verts}[columnptr[v_0]] \longleftarrow (v_0, v_2)$;
end
$\mathcal{E}_e \longleftarrow prefixSum(\mathcal{E}_{values})$
```

Algorithm 2: Topology Refinement

```
Input: adjacency matrix $\mathcal{E}$, mesh face list $F$, vertex count $n$,
       face count $m$
Output: face list of the refined mesh $F_o$
foreach $i \in [0, m)$ do
    $(v_0, v_1, v_2) \longleftarrow F[3i + (0:2)]$;
    $e_0 \longleftarrow \mathcal{E}_e(v_1, v_2) + n$;
    $e_1 \longleftarrow \mathcal{E}_e(v_2, v_0) + n$;
    $e_2 \longleftarrow \mathcal{E}_e(v_0, v_1) + n$;
    $F_o[12i + (0:2)] = (v_0, e_1, e_2)$;
    $F_o[12i + (3:5)] = (v_1, e_2, e_0)$;
    $F_o[12i + (6:8)] = (v_2, e_0, e_1)$;
    $F_o[12i + (9:11)] = (e_0, e_2, e_1)$;
end
```

Vertex Vertex Update. The coordinates of each vertex after subdivision can be computed from its neighbors,

$$v_i = (1 - k\beta)\, v_i + \beta \sum_{v_j \in \Omega_i} v_j \tag{1}$$

where Ω_i is the set of the vertices in the neighborhood of v_i, and $\beta = \frac{1}{k}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{k}\right)^2\right)$ depends on the valence k. To calculate the vertex position after subdivision, we need the list of adjacent vertices of each vertex, which is a column of the adjacency matrix $\mathcal{E}$. Hence we can compute the vertex coordinates via vector summation and scalar multiplication. More details are given in Algorithm 3.

Algorithm 3: Vertex Vertex Update

```
Input: adjacency matrix $\mathcal{E}$, vertex list $\mathbf{P}$, vertex number $n$
Output: Refined vertex list $\mathbf{P}_o$
foreach $i \in [0, n)$ do
    $u \longleftarrow \sum_{j \in \Omega_i} \mathbf{P}[j]$;
    /* $\Omega_i$ is the neighborhood of $\mathbf{P}[i]$, and
       their indices can be found between
       $\mathcal{E}_{ids}[\mathcal{E}_{ptr}[i]]$ and $\mathcal{E}_{ids}[\mathcal{E}_{ptr}[i+1]]$.          */
    $\beta \longleftarrow \frac{1}{k}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{k}\right)^2\right)$;
    $\mathbf{P}_o[i] \longleftarrow (1 - k\beta)\,\mathbf{P}[i] + \beta u$;
end
```

Edge Vertex Insertion. For each edge, a new vertex is inserted at the weighted average of four related vertices, all of which are available in matrix $\mathcal{E}$. $\mathcal{E}_{verts}(v_i, v_j)$ and $\mathcal{E}_{verts}(v_j, v_i)$ are the indices of the two vertices opposite to the edge connecting v_i and v_j. Algorithm 4 provides the pseudo-code for computing the edge vertices.

Algorithm 4: Edge Point Calculation

```
Input: adjacency matrix $\mathcal{E}$, vertex list $\mathbf{P}$, edge number $l$,
       vertex number $n$
Output: Refined vertex list $\mathbf{P}_o$
foreach $i \in [0, l)$ do
    $v_1 \longleftarrow \mathcal{E}_{ids}[i]$;
    $(v_2, v_3) \longleftarrow \mathcal{E}_{verts}[i]$;
    $e \longleftarrow \mathcal{E}_e[i]$;
    if $v_2 < v_1$ then
        $(v_2, v_4) \longleftarrow \mathcal{E}_{verts}(v_1, v_2)$;
        $\mathbf{P}_o[e+n] \longleftarrow \frac{1}{8}(3\mathbf{P}[v_1] + 3\mathbf{P}[v_2] + \mathbf{P}[v_3] + \mathbf{P}[v_4])$;
    end
end
```

Parallelization. Algorithms 2, 3 and 4 are embarrassingly parallelizable and easy to implement on both CPU and modern GPU. It is less trivial to implement Algorithm 1 in parallel due to the prefixSum routine involved, yet it is provided by the standard libraries of C++ and NVIDIA CUDA, which makes our method easy to implement with high performance.

3. Evaluation

We implement our method on the GPU using NVIDIA CUDA and on the CPU using C++/OpenMP. To ease reproduction, we release our reference implementation at https://github.com/USTC-wkc/LoopSubdiv. In this section, we analysis the complexity of our method and compare its performance against the state-of-the-art. However, for some competing methods, it is difficult to implement and no public implementation is available. In particular, Mlakar et al. [MWS*20] provided a method for GPU parallel Catmull-Clark subdivision and indicated that it can be extended to Loop subdivision, however, the implementation details are unclear and no public implementation is available. Dupuy and Vanhoey [VD22] propose a method based on the half-edge data-structure and achieves the-state-of-the-art performance. Therefore, we compare our method with their implementation in terms of memory cost and running time. Throughout our experiments, we use the following hardware configuration: an Intel i9-10900k CPU, 128GB of memory and an NVIDIA RTX3060 with 12GB of memory.

3.1. Memory Consumption

In addition to the face list $\mathbf{F}$ and the vertex list $\mathbf{P}$, our method requires memory for the adjacency matrix $\mathcal{E}$ during runtime. The storage cost of these three parts can be expressed in terms of the number of faces F_i, number of vertices V_i and number of edges E_i, where i is the subdivision level. Notice that:

$$F_{i+1} = 4F_i, \quad V_{i+1} = V_i + E_i, \quad E_{i+1} = 2E_i + 3F_i, \tag{2}$$

we can derive that,

$$\begin{cases} F_n = 4^n F_0 \\ E_n = 2^n E_0 + 3 \cdot (2^{2n-1} - 2^{n-1}) F_0 \\ V_n = V_0 + (2^n - 1) E_0 + (2^{2n-1} - 3 \cdot 2^{n-1} + 1) F_0 \end{cases} \tag{3}$$

At subdivision level n, the memory cost (in bytes) involves two parts, the adjacency matrix $\mathcal{E}_{n-1}$ for computation, and face list array $\mathbf{F}$ and vertex list $\mathbf{P}$ for output, whose costs can be estimated as follows,

$$\begin{aligned} &\text{memory cost for } \mathcal{E}_{n-1}: \ 4 \cdot (5 \cdot 2E_{n-1}) \approx 30 \cdot 2^{2n-1} F_0 \\ &\text{memory cost for } \mathcal{F}_n \text{ and } \mathcal{V}_n: \ 4 \cdot (3V_n + 3F_n) \approx 36 \cdot 2^{2n-1} F_0. \end{aligned} \tag{4}$$

Dupuy and Vanhoey [VD22] also provided an estimation of memory cost (in bytes) for their method:

$$12 \cdot \sum_{i=0}^{n} (F_i + V_i) \approx 112 \cdot 2^{2n-1} F_0. \tag{5}$$

which means the memory cost of our method is only 60% of theirs. This is mainly due to that we directly output the face list of the subdivided mesh $\mathbf{F}$, instead of the heavy half-edge structure, and the extra memory cost of our method comes from the adjacency matrix of the mesh before subdivision (level $n-1$), which is smaller than the subdivided mesh (level n). We note that based on our estimation, 55% memory cost in our method is for the subdivided mesh. We performed several experiments to verify the theoretic memory cost 4, and compared it with the method of Dupuy and Vanhoey [VD22] in Figure 2.

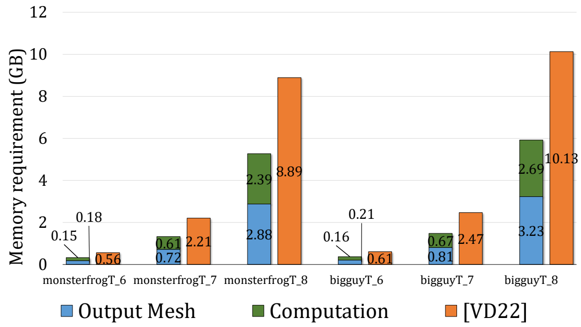


Figure 2: *The memory cost of subdividing two meshes, MonsterfrogT and BigguyT models, into different levels $n = 6, 7, 8$. The memory costs for computing and outputting the subdivision mesh are listed separately.*

3.2. Runtime Consumption

We also compare the runtime performance of our method with that of Dupuy and Vanhoey [VD22], whose implementation is kindly provide by the authors. This comparison is conducted for the running times with both GPU and CPU implementations. Figure 3 and Figure 4 show that our method runs about 40% faster. We conjecture that this is mainly due to that our implementation has the following advantages:

- The neighboring vertices of each vertex are stored together in the adjacency matrix, which makes it more cache friendly, leading to higher parallelization efficiency of our method.
- We directly use the face list **F** to represent the input and output meshes instead of a more complicated data structure such as the half-edge, which makes our method more efficient to load and write the mesh.

Figure 3 and 4 show the running times for different meshes under different subdivision levels. We can see that our method runs in real-time rate, and can therefore be used in interactive modelling.

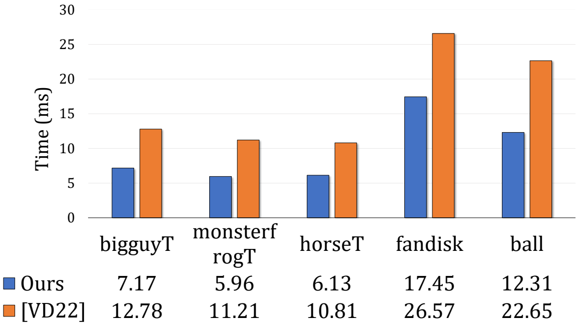


Figure 3: *Running times (ms) of Loop subdivision on an NVIDIA RTX3060 GPU, for a few different shapes. Our approach requires* $35\% - 45\%$ *less time than the state-of-the-art.*

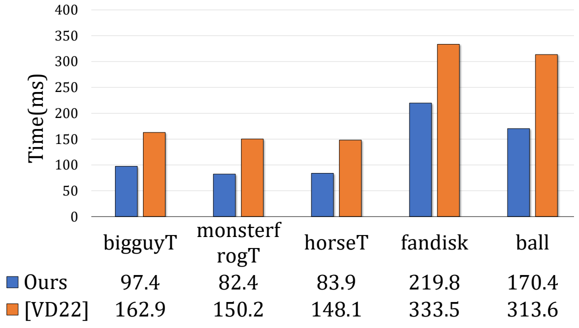


Figure 4: *Running times (ms) of parallel Loop subdivision on an Intel i9-10900k CPU, for the same shapes from Figure 3. Our approach requires* 45% *less time than the state-of-the-art.*

To further test the robustness of our method, we perform subdivision for a few meshes with different subdivision levels. Table 1 shows that the performance of our method is consistent for a range of meshes under different scales. Figure 5 reports the running times of each GPU kernel executed by our method. As can be seen, the 4 stages of our method are well balanced.

Input Mesh	**BigguyT**		**MonsterfrogT**	
	time (ms)		time (ms)	
subdivision level	ours	[VD22]	ours	[VD22]
2	0.04	0.08	0.04	0.08
3	0.14	0.21	0.11	0.20
4	0.45	0.78	0.39	0.68
5	1.79	3.19	1.49	2.80
6	7.16	12.83	5.97	11.21
7	28.16	52.26	24.15	46.09

Table 1: *Loop subdivision time in ms for two meshes with different levels. Computed on an NVIDIA RTX3060 GPU.*

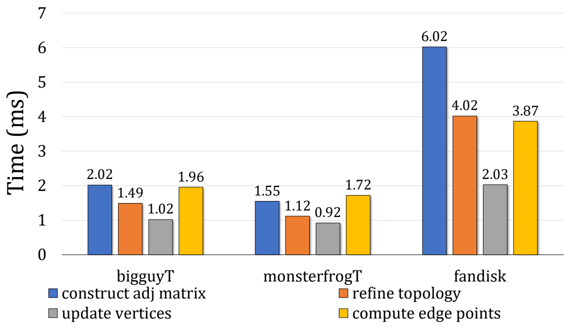


Figure 5: *The running times of each CUDA kernel in our method for subdivision down to level 6.*

Conclusion and limitations. We have presented a parallel implementation of the Loop subdivision scheme based on a simple mesh data structure. Through extensive experiments, we demonstrate that our algorithm achieves higher performances than the state-of-the-art in both running time and memory consumption. Currently our method is limited to manifold meshes. This is due to that there exist more than 2 edges connecting the same pair of vertices, and this is impossible to be encoded by a matrix. One possible fix for this problem is to split the non-manifold mesh into several manifold patches.

References

[BFK*16] Brainerd W., Foley T., Kraemer M., Moreton H., Niessner M.: Efficient GPU rendering of subdivision surfaces using adaptive quadtrees. *ACM Trans. Graph. 35* (jul 2016). doi:10.1145/2897824.2925874. 1

[Loo87] Loop C.: Smooth subdivision surfaces based on triangles. *PhD. Thesis* (1987). 1

[MWS*20] Mlakar D., Winter M., Stadlbauer P., Seidel H.-P., Steinberger M., Zayer R.: Subdivision-specialized linear algebra kernels for static and dynamic mesh connectivity on the GPU. In *Computer Graphics Forum* (2020), vol. 39, Wiley Online Library, pp. 335–349. doi:https://doi.org/10.1111/cgf.13934. 1, 3

[Pix] Pixar: Opensubdiv from research to industry adoption. In *ACM SIGGRAPH 2013 Courses*, Association for Computing Machinery. doi:10.1145/2504435.2504451. 1

[SJP05] Shiue L.-J., Jones I., Peters J.: A realtime GPU subdivision kernel. In *ACM SIGGRAPH 2005 Papers* (2005). doi:10.1145/1186822.1073304. 1

[VD22] Vanhoey K., Dupuy J.: A Halfedge Refinement Rule for Parallel Loop Subdivision. In *Eurographics 2022 - Short Papers* (2022), Pelechano N., Vanderhaeghe D., (Eds.), The Eurographics Association. doi:10.2312/egs.20221028. 1, 3, 4

[ZSS17] Zayer R., Steinberger M., Seidel H.-P.: A GPU-adapted structure for unstructured grids. In *Computer Graphics Forum* (2017), vol. 36, Wiley Online Library, pp. 495–507. doi:https://doi.org/10.1111/cgf.13144. 1