Progressive Material Caching

Shin Fujieda
Advanced Micro Devices, Inc.
Japan
Shin.Fujieda@amd.com

Takahiro Harada
Advanced Micro Devices, Inc.
USA
Takahiro.Harada@amd.com

(a) Without caching (b) Our method (c) Difference between (a) and (b) (d) Nodes found in the cache

Figure 1: Equal-sample comparisons (128 samples per pixel) with Classroom, Junk Shop **and** Italian Flat **scenes. (a) Images rendered without caching. (b) Images rendered with our method. (c) Differences between images (a) and (b) multiplied by 5. (d) Visualizations of the average number of material nodes per sample which can be found in the cache by lookup. As the color changes from black to yellow in the viridis colormap, the respective average increases from 0 to 20.**

ABSTRACT

The evaluation of material networks is a relatively resource-intensive process in the rendering pipeline. Modern production scenes can contain hundreds or thousands of complex materials with massive networks, so there is a great demand for an efficient way of handling material networks. In this paper, we introduce an efficient method for progressively caching the material nodes without an overhead on the rendering performance. We evaluate the material networks as usual in the rendering process. Then, the output value of part of the network is stored in a cache and can be used in the evaluation of the next materials. Using our method, we can render the scene with performance equal to or better than that of the method without caching, with a slight difference in the images rendered with caching and without it.

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; **Ray tracing**.

KEYWORDS

material evaluation, ray tracing

ACM Reference Format:
Shin Fujieda and Takahiro Harada. 2022. Progressive Material Caching. In *SIGGRAPH Asia 2022 Technical Communications (SA '22 Technical Communications), December 6–9, 2022, Daegu, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3550340.3564223

SA '22 Technical Communications, December 6–9, 2022, Daegu, Republic of Korea

ACM ISBN 978-1-4503-9465-9/22/12…$15.00
https://doi.org/10.1145/3550340.3564223

1 INTRODUCTION

Modern 3D modeling software allows artists to create their desired scenes using node-based materials, which make editing these materials more intuitive and flexible. Thanks to this node-based system, most scenes in production rendering usually contain hundreds or thousands of materials with large material networks. Rendering these scenes requires us to evaluate complex materials, which is

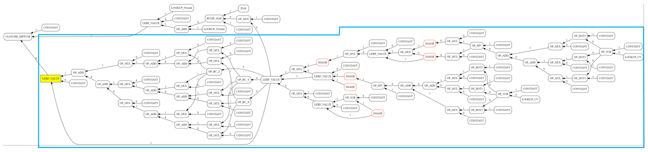


Figure 2: An example of a material. The yellow and red nodes are the cacheable and UV-dependent nodes, respectively.

generally a very resource-intensive operation that often can be the bottleneck of the rendering performance. Besides, we need to evaluate the same complex network many times, which is redundant and can be optimized. To alleviate this bottleneck and redundancy, one can bake all inputs of BSDF nodes to texture maps [Adobe 2022] or per-vertex data [Fascione et al. 2018] as a pre-process and use them when evaluating the materials in the rendering process. However, this pre-process requires some additional resources and cannot handle everything properly because some material nodes require information about the object at a shading point.

In this paper, we propose a method to progressively cache outputs of the material nodes instead of pre-processing them. We call this a material cache which is queried by a combination of several keys considering shading-point information using a hash table (see Sec. 3) as we render the scene. Then, we can reuse the cache when a ray hits the surface with the material under the same condition next time. We also present our GPU implementation that inserts the output value of a material node into a fixed-size cache with two different hashes. This progressive way of handling the material cache does not add any overhead on the rendering performance, except for the additional memory for the cache. To summarize, the contributions of our work are as follows:

- We introduce a progressive method to cache some material nodes without an overhead on the rendering performance.
- We show an efficient approach to handling the UV value to store material nodes in the cache.
- This paper also illustrates the implementation details of our method to update a hash table on the GPU.

2 MATERIAL NETWORK EVALUATION

Fig. 2 shows one of the material networks in the Classroom scene. Larger image can be found in our supplemental material. This material is composed of a massive network having more than 100 nodes. To render the scene, we need to evaluate such massive networks many times, so efficiently handling them is crucial to the rendering performance.

In our implementation, we process the material networks as a form of stack machine on the GPU. More specifically, nodes in a material graph are converted into data that is read and executed in the order they are stored in the memory. Although there are other ways of processing the material networks on the GPU, such as a dynamic kernel generation per material, we choose it because we need to keep the kernel compilation time manageable, even if the scene has a lot of materials. Also, we optimize the material networks before rendering the scene by converting them into forms that can be easily placed in and treated by the stack, for example, by pre-computing simple arithmetic nodes with constant inputs.

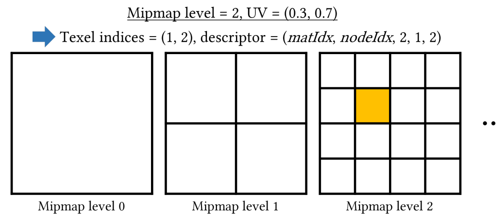


Figure 3: Virtual mipmapped texture.

3 PROGRESSIVE MATERIAL CACHING

Our method progressively caches the outputs of some material nodes as we render the scene with little pre-process of material networks. Before rendering the scene, we traverse the material networks from the bottom to the top to define cacheable nodes at the highest possible level in the network. A cacheable node, where a result of its sub-graph is cached, is a node whose sub-graph has no shading-point dependent node as a descendant other than texture coordinates (UVs), as we are going to describe in Sec. 3.1. And then, when reaching the cacheable node during a material evaluation in the rendering process, we first look it up in the cache, and, if it is found, we can directly use the cached value without processing its sub-graph under the cacheable node. Otherwise, we manage its sub-graph in the standard way and store the resulting value in the cache. Thus, by defining cacheable nodes at highest possible levels in a network, we can skip the material evaluation of most parts of this network. Fig. 2 shows an example of a material where a cacheable node is marked with yellow and its sub-graph is outlined with the blue polygon. In this material, the cacheable node is defined at the highest level, so we do not need to process most parts of the network once it is cached. We can have multiple cacheable nodes for a single material. Note that this process of defining the cacheable nodes is a lightweight computation, because we can execute it at the same time as we optimize the material networks as described in Sec. 2.

3.1 Node Descriptor with UV

We need to uniquely identify cacheable nodes to provide for caching their output values. For each cacheable node, we define a descriptor that contains at least identifiers such as *material ID* and *node ID* that is the index of the node in the material network. Additionally, as some material nodes depend on information about the object at a shading point, the descriptor should include it as well, to be unique depending on the shading point. In this paper, we focus on UV, although other shading-point information, such as a world-space position, could also be used. We do not consider a node cacheable if its sub-graph contains any shading-point dependent nodes other than UV. We want to extend our method to them in the future.

We employ the UV value as one of the descriptors of cacheable nodes because it is the value most frequently used to handle material nodes. With UV, the output value of the cacheable node can be cached as a texel value. However, the challenge lies in defining the proper cache resolution, which is not obvious only from the UV value. To properly define it, we also need other information, such as texture resolution and the mipmap level. However, we do not

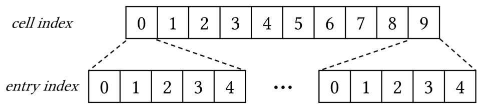


Figure 4: 10×5 size of a hash table. The numbers of cells and entries on the hash table are 10 and 5 for each.

Algorithm 1: Update of a hash table on the GPU

Input: Descriptor of the cacheable node $desc$, the cacheable node's value v, array of material cache $matCache$, the number of cells N_c and the number of entries N_e.

$cellIdx \leftarrow$ `Hash`($desc$)$\%N_c$;
$hashVal \leftarrow$ `Hash2`($desc$);
$hashIdx \leftarrow -1$;
for $i \leftarrow 0$ **to** N_e **do**
  $entryIdx \leftarrow cellIdx \times N_e + i$;
  $current \leftarrow matCache[entryIdx]$;
  if $current.hash = hashVal \cup current.hash = 0$ **then**
    $hashIdx \leftarrow entryIdx$;
    $break$;
  end
end
if $hashIdx \neq -1$ **then**
  $encodedVal \leftarrow$ `Encode`(v);
  $new \leftarrow \{hashVal, encodedVal\}$;
  $old \leftarrow$ `CAS`($matCache[hashIdx], 0, new$);
end

know it when managing the cacheable node during the material evaluation at a shading point. As seen from Fig. 2, for example, the UV-dependent nodes are different from the cacheable node and only exist on its sub-graph, so it is impossible to retrieve the texture information at the cacheable node. Therefore, instead of using the actual texture information, we propose to introduce a virtual mipmapped texture for the cacheable node using the footprint of the ray cone at a shading point [Akenine-Möller et al. 2021]. We use the word "virtual" for this texture because texel allocation is on-demand due to the sparse texel access, i.e., we do not need to allocate a series of two-dimensional textures but only allocate texels we need. This allows us to cache the information at the appropriate resolution. The intersection of a ray cone and a surface gives an elliptic footprint, and we can compute the texture-coordinate gradients, $\vec{g}_1$ and $\vec{g}_2$, along with both axes of the ellipse, as derived in [Akenine-Möller et al. 2021]. With these gradients, we compute the mipmap level λ for the virtual texture as follows:

$$\lambda = -\log_2(min(\|\vec{g}_1\|, \|\vec{g}_2\|)), \tag{1}$$

Considering this mipmap level, we calculate the resolution of the virtual texture from 2^{λ}. So, unlike the standard mipmap hierarchy, the resolutions of virtual textures increase as the mipmap level becomes higher in our definition, as shown in Fig. 3. Then, using the UV value, the texel indices on the virtual texture are calculated. As a result, we store the output value as a texel value of the virtual texture in the cache using the mipmap level λ and the texel indices as the descriptor of the cacheable node.

3.2 Hashing Material Nodes

Using the unique descriptor discussed in Sec. 3.1 as the key, we store and look up the cacheable node's value in a hash table to cache it in constant time. Although the number of values of the cacheable nodes stored in the hash table is unknown because we store them progressively, we use a fixed-size hash table for the GPU efficiency. As the size of the hash table is limited, hash collisions may occur, because different descriptors may be mapped to the same index in the table. To differentiate each descriptor, we apply two different hash functions to the same descriptor as described in [Binder et al. 2022]. In our implementation, the first hash function defines the *cell index* on the fixed-size hash table with a modulo operator, whereas the second hash function checks the equality of keys and allocates the entry on each cell with the *entry index*, as shown in Fig. 4. These numbers of cells and entries are chosen empirically. Then, during the rendering, we progressively insert the cacheable node's value into the hash table using these two different hash values per the descriptor of the cacheable node. However, even with this approach, we cannot completely avoid hash collisions. In this case, we keep the first inserted value on the hash table without replacing it with the new value to simplify our implementation.

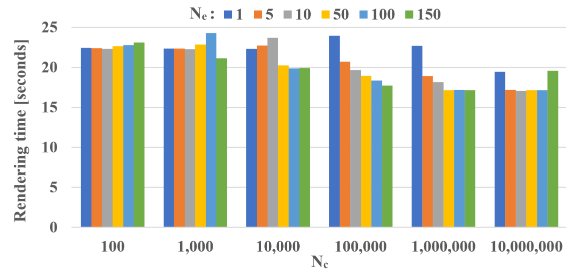


Figure 5: Cache size evaluation. N_c and N_e are the numbers of cells and entries on the cache, respectively.

3.3 Cache Update on the GPU

The proposed method is implemented in a GPU path tracer, which can assign more than one thread for different samples in a pixel. Therefore, we use a lock-free algorithm using a compare-and-swap (CAS) instruction that avoids data races to insert the cacheable node's value into the cache. For this CAS instruction, we encode the output value as a single 32-bit value, and then the second hash is packed with the encoded output value as a single 64-bit variable, which allows us to update the cache in a single CAS operation. Algorithm 1 illustrates how to store the cacheable node's value in the cache using two different hashes with a 64-bit CAS instruction.

4 RESULTS

We implemented our method of progressively caching materials using OpenCL™. All the images in this paper are rendered on an AMD Radeon™ RX 6900 XT GPU at 1920×1080 screen resolution. Full-resolution images can be found in our supplemental material.

Cache Size Evaluation. We use the fixed-size cache in our implementation. So, we investigate how the cache size affects the rendering performance. We have two parameters, such as the numbers of cells N_c and entries N_e, in the material cache. Fig. 5 shows the results of the performance evaluations with several combinations of N_c and N_e on the Classroom scene with 32 samples per pixel. In general, the larger N_c improves the performance, while the larger N_e usually shows better performance than the smaller N_e.

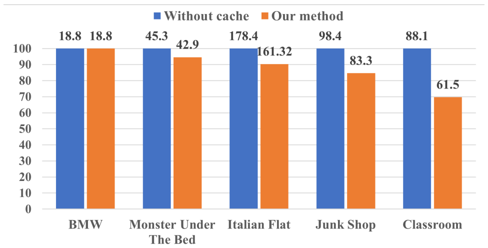
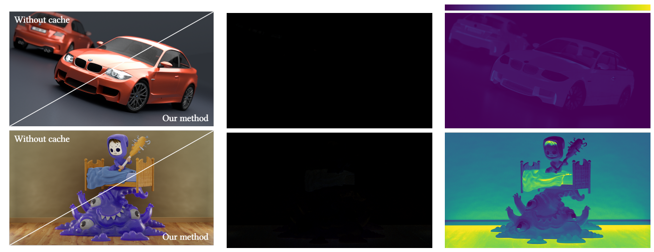


Figure 6: Performance comparison of five scenes.



(a) Rendered images **(b) Image difference** **(c) Number of nodes**

Figure 7: Equal-sample comparisons (128 samples per pixel) with BMW and MONSTER UNDER THE BED scenes. (a) Images rendered without the cache (top-left) and with our method (bottom-right). (b) and (c) are the same as Fig. 1c and Fig. 1d

However, the performance improvement hits its peak with some large size of the cache. In this paper, we use $10^7 \times 10$ as the size of the cache that requires about 763 MB of the memory, because it represents the best performance in this experiment.

Performance Comparison. Fig. 6 compares the performance on five scenes with 128 samples per pixel. The vertical axis represents the percentage in the rendering time of our method as compared to that of the method without caching, taken as 100%. Numbers on each bar show the actual rendering times in seconds. On CLASSROOM, JUNK SHOP, ITALIAN FLAT and MONSTER UNDER THE BED scenes, our method improves the performance by about 31%, 16%, 10%, and 5% for each. The BMW scene shows the worst case of our method with no gain in performance. However, this result demonstrates that our method performs at least equally to the method without caching, because we can render the scene as it is, even if the material value cannot be stored or looked up in the cache.

Fig. 1 and Fig. 7 represent rendered results with the scenes on the experiment on Fig. 6. In Fig. 1a, Fig. 1b and Fig. 7a, we cannot see critical visual differences in the results, while the rendering time is reduced by 5 – 31% except for the BMW scene. Fig. 1c and Fig. 7b visualize those differences multiplied by 5. As seen from these images, they do not show much discrepancy, although some surfaces with textures are slightly different due to the virtual texture's resolution. As shown in Fig. 3, the virtual texture becomes more precise as the mipmap level becomes higher. Therefore, by offsetting the mipmap level with a small value, we can obtain a bit more accurate result while it is a trade-off between the accuracy and the performance. Also, Fig. 1d and Fig. 7c visualize the average number of material nodes per sample which we can find in the cache by lookup. The color from black to yellow in the viridis colormap represents 0 to 20 times. In the CLASSROOM scene, we find more nodes in the material cache over the whole scene compared to the JUNK SHOP scene. This can explain the difference in the performance improvement for these scenes. However, comparing CLASSROOM, ITALIAN FLAT and MONSTER UNDER THE BED scenes, we note that the rendering time on the CLASSROOM scene is significantly reduced, while we can see more red in ITALIAN FLAT and MONSTER UNDER THE BED scenes. This means we can cache the material nodes at the higher level in the network which contains more complex sub-graphs with our proposed method on the CLASSROOM scene.

5 CONCLUSIONS AND FUTURE WORK

We presented an efficient approach to evaluate the material networks by progressively caching them without an overhead. Our proposed method uses a fixed-size hash table to store and lookup the information about the cacheable nodes which can be effectively updated on the GPU. The experiments show that our method can outperform significantly or at least perform equally to the method without caching. The rendered images are slightly different, but the error does not appear an artifact.

In this paper, we demonstrated our method with the UV information as an additional descriptor of the cacheable node. However, in the material networks, a material node can depend on other types of shading-point information which we can use as the descriptor. We would like to extend our method to them in the future. We expect supporting them will make the performance better because we can cache more material nodes in the network. Our method uses fixed-size storage and keeps the first inserted values all the time, even if hash collisions happen. This could be improved, because limited storage might be filled with data that is not reused often. Thus, we also would like to investigate a more effective way of storing the material information in the cache, for example, by prioritizing directly visible materials over indirectly visible ones.

ACKNOWLEDGMENTS

We thank Brian Savery at AMD for his inspiring study and valuable comments, Oleksandr Kupriyanchuk for proofreading. We also thank the following people for their scenes used in our experiments: Christophe Seux (CLASSROOM), Alex Treviño and Anaïs Maamar (JUNK SHOP), Flavio Della Tommas (ITALIAN FLAT), Mike Pan (BMW) and Metin Seven and Blake Stevenson (MONSTER UNDER THE BED).

REFERENCES

Adobe. 2022. Substance 3D. https://www.adobe.com/products/substance3d/3d-augmented-reality.html

Tomas Akenine-Möller, Cyril Crassin, Jakub Boksansky, Laurent Belcour, Alexey Panteleev, and Oli Wright. 2021. Improved Shader and Texture Level of Detail Using Ray Cones. *Journal of Computer Graphics Techniques (JCGT)* 10, 1 (25 January 2021), 1–24. http://jcgt.org/published/0010/01/01/

Nikolaus Binder, Sascha Fricke, and Alexander Keller. 2022. Massively Parallel Path Space Filtering. In *Monte Carlo and Quasi-Monte Carlo Methods*, Alexander Keller (Ed.). Springer International Publishing, Cham, 149–168.

Luca Fascione, Johannes Hanika, Mark Leone, Marc Droske, Jorge Schwarzhaupt, Tomáš Davidovič, Andrea Weidlich, and Johannes Meng. 2018. Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production. *ACM Trans. Graph.* 37, 3, Article 31 (aug 2018), 18 pages. https://doi.org/10.1145/3182161