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:

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
we need to keep the kernel compilation time manageable, even
by pre-computing simple arithmetic nodes with constant inputs.

networks before rendering the scene by converting them into forms
if the scene has a lot of materials. Also, we optimize the material
can be efficiently handled them is crucial to
rendering performance.

In our implementation, we process the material networks as a
puts of the material nodes instead of pre-processing them. We call
this a material cache which is queried by a combination of several
nodes without an overhead on the rendering performance.

We introduce a progressive method to cache some material
method to update a hash table on the GPU.

We show an efficient approach to handling the UV value to
nodes because it is the value most frequently used to handle material
position, could also be used. We do not consider a node cacheable
although other shading-point information, such as a world-space
depending on the shading point. In this paper, we focus on UV,
that is the index of the node in the material network. Additionally,
node ID

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

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

Figure 3: Virtual mipmapped texture.
As a result, we store the output value as a texel value of the virtual texture from Algorithm 1:

**Update of a hash table on the GPU**

1. cellIdx ← hash(desc) % \(N_c\);
2. hashIdx ← -1;
3. for \(i = 0\) to \(N_d\) do
   4. entryIdx ← cellIdx \(\times N_e\) + \(i\);
   5. current ← matCache[entryIdx];
   6. if current.hash = hashVal ∨ current.hash = 0 then
      7. hashIdx ← entryIdx;
      8. break;
   9. end
10. end
11. if hashIdx ≠ -1 then
    12. encodedVal ← Encode(\(v\));
    13. new ← (hashVal, encodedVal);
    14. old ← CAS(matCache[hashIdx], 0, new);
15. end

Figure 4: 10 \(\times\) 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 ← hash(desc) % \(N_c\); |
| hashVal ← hash(\(2\cdot\)desc); |
| hashIdx ← -1; |
| for \(i = 0\) to \(N_d\) do |
| entryIdx ← cellIdx \(\times N_e\) + \(i\); |
| current ← matCache[entryIdx]; |
| if current.hash = hashVal ∨ current.hash = 0 then |
| hashIdx ← entryIdx; |
| break; |
| end |
| end |
| if hashIdx ≠ -1 then |
| encodedVal ← Encode(\(v\)); |
| new ← (hashVal, encodedVal); |
| old ← CAS(matCache[hashIdx], 0, new); |
| end |

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

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 \(\times\) 1080 screen resolution. Full-resolution images can be found in our supplemental material.

**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 \(\times\) 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 \(N_e\).
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 Trevisan and Anaïs Maamar (JUNK SHOP), Flavio Della Tommas (ITALIAN FLAT), Mike Pan (BMW) and Metin Seven and Blake Stevenson (MONSTER UNDER THE BED).

REFERENCES