A Framework to Transform In-Core GPU Algorithms to Out-of-Core Algorithms

Takahiro Harada*
Advanced Micro Devices, Inc.

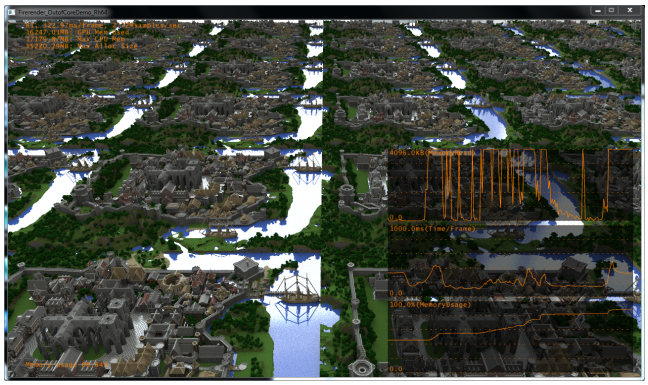

Figure 1: *A screenshot of an out-of-core GPU path tracer developed using the framework rendering global illumination. The scene geometry (32GB, 428M triangles) is rendered on an AMD FirePro™ W9100 GPU using 15GB geometry cache at 1280 × 720 resolution. The graphs on the left in the screenshot show the history of the amount of the page copy, the frame time, and the physical memory usage over frames.*

Keywords: GPU, out of core, global illumination, path tracing

Concepts: •**Computing methodologies** → *Graphics systems and interfaces;* **Rendering;**

1 Introduction

Porting an existing application to the GPU requires a lot of engineering effort. However, as the GPU memory of today is smaller compared to the host memory, some application which requires to access to a large data set needs to implement an out-of-core logic on top of the GPU implementation which is additional engineering work [Garanzha et al. 2011]. We present a framework to make it easy to transform an in-core GPU implementation to an out-of-core GPU implementation. In this work, we assume that the out-of-core memory access is read only. The proposed method is implemented using OpenCL thus we use the terminology of OpenCL in this document.

2 Implementation

Overview

To make a kernel execution out of core, changes need to be done for the host and device code. They are

1. Data management (Host)
2. Kernel implementation (Device)
3. Kernel launch (Host).

The kernel implementation requires the most engineering effort as development environment is not as good as development on the host (e.g., debuging). We developed macros to transform in-core kernel to out-of-core kernel described below.

Data Management

We essentially need to implement a software virtual memory system which receives page requests from device and fill the page from the source of the data which can be host memory or disk which is used when the data does not fit to the host memory. The reason why we implemented it in software is because we do not have full control of the GPU virtual memory. On the device side, we need to prepare buffers for page table, data storage (physical memory), and page requests. Once they are prepared, we execute the kernel. The CPU is responsible for serving page requests. It copies pages to the physical memory until it gets full. When there is no space available in the physical memory, it evicts older pages in our implementation. A pseudo code for an execution is shown in below.

```
while True do
    Requests ← Execute kernel, passing physical memory and page table to the kernel.
    Break if there is no requests from work items.
    Serve the requests. Update physical memory and page table.
end while
```

Kernel Implementation

We need to make changes for the kernel to access out-of-core data. The first one is the memory access. It needs to read the page table first to find if the memory resides on the physical memory or not. If it is not there, we need to suspend the kernel execution and ask the system for the data which requires a suspension of the kernel execution. The mechanism to suspend and resume the kernel is the second change. We need to prepare storage for work item context and add the logic for the kernel to load and store it to the buffer so that work item can suspend and resume the execution. Since these operations are general, we developed a few macros for these which makes it easy to transform an in-core kernel to an out-of-core kernel. If the kernel has one memory access to the out-of-core buffer, we only need to change 3 lines (1 mod, 2 add) for the body of the kernel as shown in the appendix. A few kernel examples and the macros are provided in the supplemental material.

Kernel Launch

The only change we need for an out-of-core execution, we need to pass a few additional buffers (physical memory, page table and page request).

3 Results

We transformed an in-core GPU path tracer to an out-of-core GPU path tracer performing out-of-core accesses to the scene geometry using the framework. Fig. 1 is a screenshot of a 32GB scene rendered on a GPU using 15GB physical memory in 720p resolution.

*e-mail:takahiro.harada@amd.com

I3D '16, February 26-28, 2016, Redmond, WA, USA
ISBN: 978-1-4503-4043-4/16/02
DOI: http://dx.doi.org/10.1145/2856400.2876011

Table 1: *Comparison of ray casting time.*

	Primary ray cast	Secondary ray cast
In core	29.4ms	15.8ms
Out of core	58.3ms	39.8ms

The evaluation of the proposed method should be done in 1. simplicity of the kernel transformation, 2. execution performance. Although 1 is subjective, we believe the framework made it simple enough (please see the appendix and the supplemental material). Although the code developed by the proposed method can render a scene which does not fit to the GPU memory, the execution performance of the out-of-core kernel is expected be worse than the original if all the data fits in the physical memory because of one additional memory reads of the page table for an out-of-core memory access, complexity of the code, and loading and storing work item contexts.

We compared the time for ray casting of our in-core and out-of-core GPU path tracing on an AMD FirePro W9100 GPU on a workstation with dual Intel® Xeon® CPUs. The renderer does out-of-core access to the geometry data which includes triangles (vertices, normals, texture coordinates, connectivities) and the spatial acceleration structure (BVH) while others, such as textures, are stored in core. We set the page size to 32KB, and the VM serves top 1024 most-requested pages per iteration. We tried several combinations of these parameters manually and select the best among them, but automatic setting of these is a future work.

The out-of-core implementation is about 2× slower than the in core implementation in the experiment although the ratio depends on many factors such as computational complexity and memory access pattern. This would be mainly from the difference in the GPU occupancy, 79% (using 36VGPR) for the in core and 30% (using 70VGPR) for the out of core. Since the modification we need for the transformation is simple, we believe that it could be done automatically for example when compiling the code. Fully automatic transformation of the code is a future work. Other future works include optimization of the compiler to reduce the VGPR usage, application to the R&W data, and extension to multiple GPUs.

References

GARANZHA, K., BELY, A., PREMOZE, S., AND GALAKTIONOV, V. 2011. Out-of-core gpu ray tracing of complex scenes. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, 21:1–21:1.

Appendix

Here we show a kernel transformation example. More examples can be found in the supplemental material. There are in core and out of core impementations. The modification made for the out of core implementation is highlighted in **bold**. This kernels are slightly simplified from the one used for the benchmark of the paper for illustration purpose.

Here we show kernels computing the geometric and the shading normals for a ray which has a hit to a triangle. The geometry data is in the out-of-core storage for implementation 2.

1. In core implementation

```
__kernel
void ComputeNormalKernel(
  __global Ray* gRays,
  __global Hit* gHits,
  __global HitNormal* gHitNormalOut,
  __global Triangle* gTriStorage )
{
  const int gIdx = GET_GLOBAL_IDX;
  Triangle t;

  if( hasHit( gHits[gIdx] ) )
  {
    t = gTriStorage[ gHits[gIdx].m_idx ];
    const float4 ng =
      normalize3( cross3(t.v1-t.v0, t.v2-t.v0) );
    const float4 hp = gRays[gIdx].getHitPoint();
    const float4 bCrd =
      calcBaryCrd( hp, t.v0, t.v1, t.v2 );
    const float4 ns = normalize3(
      bCrd.x*t.n0+bCrd.y*t.n1+bCrd.z*t.n2 );
    gHitNormalOut[gIdx].m_ng = ng;
    gHitNormalOut[gIdx].m_ns = ns;
  }
}
```

2. Out of core implementation

```
__kernel
void ComputeNormalKernel(
  __global Ray* gRays,
  __global Hit* gHits,
  __global HitNormal* gHitNormalOut,
  VM_KERNEL_ARGS ) //ooc
{
  const int gIdx = GET_GLOBAL_IDX;
  Triangle t;

  VMInitialize;

  if( hasHit( gHits[gIdx] ) )
  {
    VMLoad( Triangle, gHits[gIdx].m_idx * sizeof(Triangle), &t, 0 );
    const float4 ng =
      normalize3( cross3(t.v1-t.v0, t.v2-t.v0) );
    const float4 hp = gRays[gIdx].getHitPoint();
    const float4 bCrd =
      calcBaryCrd( hp, t.v0, t.v1, t.v2 );
    const float4 ns =
      bCrd.x*t.n0+bCrd.y*t.n1+bCrd.z*t.n2 );

    gHitNormalOut[gIdx].m_ng = ng;
    gHitNormalOut[gIdx].m_ns = ns;
  }

  VMFinalize;
}
```