Realization and Improvement of GPU Streaming BDPT

Fu-jun YE* and Gen-yuan ZHANG
Zhejiang University of Media and Communications, Zhejiang Province, China
*Corresponding author

Keywords: GPU, Streaming BDPT.

Abstract. In order to speed up the Streaming BDPT, this paper first introduces the implementation of Streaming BDPT algorithm in Tracy, and then puts forward the improvement of this algorithm, including using MIS one–sample estimator to construct simplified path, and an algorithm for constructing a uniform selection path, which improves the speed of GPU streaming BDPT.

Implementation of Streaming BDPT
As a comparison, the Streaming BDPT algorithm is implemented in Tracy[1,2,3,4], which uses light source, camera, material and other key classes that have been finished before. The implementation of the algorithm is very straightforward. Before introducing its implementation, the structure of the vertex of the path is first given as follows[5,6]:

```cpp
struct PathVert{
    float3 position, shading_normal;
    float3 wo; // random walk direction
    float3 throughput; //path contribution flux
    int light_index; //the radiation from the camera hit the light source
    BSDFInfobsdf_info;//BSDF information at the vertex
    // See formula 2.22
    struct {
        float d;
        float inv_cur_forward_pdf; //the probability of sampling the current vertex forward (based on area measurements)
    }mis;
    __device__ TRACY_INLINE void sampleCameraVertex(uint& seed);
    __device__ TRACY_INLINE void sampleLightVertex(uint& seed);
    __device__ TRACY_INLINE bool randomWalk(uint& seed, const uint2& pixel_xy);
};
```

The implementation of iterative updating MIS weights is given below:

```cpp
__device__ float PathVert::misPartailWeight(int vindex, //the index of the current vertex on its child path
int connect_index, //the index of the connected vertex on its child path
const PathVert& connect_with) // the connected vertex
{
    if(vindex == 0) return 0.f; //the initial value of the recursive formula
    float3 W_nextToCur = normalize(this->position – connect_with.position);
    float reverse_pW = connect_with.evalPdfW(connect_with.wo, W_nextToCur);
    float reverse_pA = Measure::pdfSolidAngle2Area(reverse_pW, connect_with.position, this->position, this->shading_normal);
    float reverse_prev_pW = (vindex>1?this->evalPdfW(-W_nextToCur, this->wo):0.f);
    return reverse_pA*(mis.Inv_cur_forward_pdf+mis.d*reverse_prev_pW);
}

__device__ float3 evalConnection(int s, int t, const PathVert& y_endpoint,
const PathVert& z_endpoint)
```

233
```c
int num_techs = s + t + 1;
float3 connection_factor = c_st(s, t, y_endpoint, z_endpoint);
float3 contrib = connection_factor*y_endpoint.throughput*z_endpoint.throughput;
float pLs = y_endpoint.misPartialWeight(s, t, z_endpoint);
float pEt = z_endpoint.misPartialWeight(t, s, y_endpoint);
float miswgt = 1.f/(1.f + pLs + pEt);
return miswgt*contrib;
```

The implementation of SBDPT algorithm is as follows:

```c
RT_PROGRAM void sbdpt()
{
  uint seed = ...;
  float3 res = make_float3(0.f);
  uint2 pixel_pos = ...; // position of pixel
  PathVert light_supernode; // s = 0
  float3 implicit_res = make_float3(0.f);
  int nl = 0;
  PathVert y_endpoint; // the endpoint of sub path of light source
  y_endpoint.sampleLightVertex(seed);
  bool y_terminated = false;
  for(int s=1; !y_terminated&s<max_depth;++s)
  {
    PathVert z_endpoint; // The endpoint of sub path of camera
    z_endpoint.sampleCameraVertex(seed);
    bool z_terminated = false;
    int t=2;
    z_terminated = z_endpoint.randomWalk(seed, pixel_pos);
    int path_len = s + t - 1;
    while(!z_terminated&&path_len<= max_depth)
    {
      if(nl == 0)
        implicit_res += evalConnection(0, t, light_supernode, z_endpoint);
      float3 connectionContrib = evalConnection(s, t, y_endpoint, z_endpoint);
      res += connectionContrib;
      z_terminated = z_endpoint.randomWalk(seed, pixel_pos);
      ++t, ++path_len;
    }
    ++nl;
    y_terminated = y_endpoint.randomWalk(seed, pixel_pos);
  }
  res += implicit_res;
}
```

### One-sample BDPT

The MIS estimation model used by MISBDPT and SBDPT is called the multi-sample estimator (See the ninth chapter of the literature[7]), that is, a variety of sampling methods generate several samples at the same time, and then use MIS weights to combine. The general formula is as follows:

\[
F = \sum_{i=1}^{n_t} \sum_{j=1}^{n_i} W_i(X_{i,j}) \frac{f(x_{i,j})}{p_i(x_{i,j})}
\]  

(1)
Where \( n \) is the number of sampling methods, \( n_i \) is the number of samples generated by the \( i \)-th method, \( p_i \) is the probability density function of the \( i \)-th sampling method, \( W_i \) is the weight of MIS, \( X_{ij} \) is \( j \)-th sample generated by the \( i \)-th sampling method.

One-sample Estimation Model

Instead generating several samples for each sampling method in advance, one-sample estimator of MIS(see ninth chapter of the literature[7]) first selected a sampling method randomly, and then used this method to generate one sample. This estimating method could also be weighted by MIS, and the goal of noise reduction was achieved. The formula is as follows:

\[
F = \frac{w_i(X_i)f(X_i)}{c_i p_i(X_i)}
\]

(2)

Where \( I \in \{1, \ldots, n\} \) is a random variable based on the \( c_i \) probability distribution, \( \sum_{i=1}^{n} c_i = 1 \), and \( X_i \) is a sample that is sampled according to the probability density function \( p_i \). Note that the \( c_i \) in the formula refers to the selected probability of the \( i \)-th sampling method, and it is a function that irrelevant to sample \( X \). That is the function \( c_i \) should be defined before sampling, instead requiring information in sampling process. Literature [7] does not give unbiased proof of this estimate, so here gives a brief proof:

\[
E[F] = \sum_{i=1}^{n} c_i \int_{\Omega} \frac{w_i(x)f(x)}{c_i p_i(x)} p_i(x) d\mu(x)
= \sum_{i=1}^{n} \int_{\Omega} w_i(x)f(x)d\mu(x)
= \int_{\Omega} \sum_{i=1}^{n} w_i(x)f(x)d\mu(x)
= \int_{\Omega} f(x)d\mu(x) = 1
\]

(3)

Path Generation Method

Based on the one-sample estimation model, we can get a new algorithm for BDPT path construction. This paper named it one-sample BDPT, and its steps are described in Figure 1, which is described below:

![Figure 1. The path connection method of one-sample BDFT presented in this paper.](image)

1. At initialization, both the light source sub path and the camera sub path are null.
2. Randomly select a sub path to expand.
3. Connect the endpoints of the two sub paths and accumulate the lightpath contribution.
4. If sampling ends (for example, the maximum path length is reached), the algorithm terminates, otherwise go to step 2 and continue.

As shown above, in the path construction algorithm, because the sub path can make the total path length increased by 1 each time extended, and the connection always occurs in the sub path endpoint, so for the light path of arbitrary length, there will be only one method generating it, which corresponds exactly one-sample estimation model. Step 2 above does not specify how to choose a sub path to expand, so the next two sections will describe two methods for this.

Naïve Selection Method

First, we introduce a simplest way to select an extended light source sub path or a camera sub path at the same probability (0.5). Because of its simplicity, this paper calls it the naïve selection method. The selection is determined, and then we calculate the probabilities of the alternatives.

For example, generate two sub paths in the following order, then connect them:
In Figure 2, \( s = 3, t = 1 \), note that the probability here is not \( \frac{c_1}{g_1^2} \cdot 0.5 \cdot 0.5 = 0.25 \), because the \( c_i \) in the formula refers to the probability that the i-th sampling method is selected, rather than the probability of the sampling process in this particular order. The correct \( c_i \) is calculated as follows:

\[
c_{3,1} = \frac{c_1}{2^s}
\]

Generally,

\[
c_{s,t} = \frac{c_{s+t}}{2^{s+t}}
\]

**Uniform Selection Method**

The problem of above selection method is that it’s more inclined to choose \( s \approx t \) sampling methods. Because of its greater probability, the greater noise may be introduced potentially. As discussed in Veach[7], when there is no more prior information about the sampling object, it’s best to make all sampling methods to generate same quantity of samples (that is the probability of being selected is same), therefor this paper designs a simple method to achieve the goal:

Suppose that the current light source sub path has \( s \) vertices and the camera sub path has \( t \) vertices, then the probability of selecting the extended sub path of the light source is set to \( \frac{s+1}{s+t+2} \). The following is a simple proof that this method generates sampling methods with the same probability:

\[
c_{s,t} = \frac{C_{s+t}}{2^{s+t}} \cdot \frac{s!t!}{(s+t+1)!} = \frac{(s+t)!}{s!t!(s+t+1)!} \cdot \frac{s!t!}{(s+t+1)!} = \frac{1}{s+t+1}
\]

Where \( C_{s+t} \) represents the quantity of the \( (s, t) \) combinations, the numerator and denominator of \( \frac{s!t!}{(s+t+1)!} \) derives from the multiplicative of above probabilities. This result shows that for any path containing \( (s+t) \) vertices, the probability of any combination of \( (i, s+t-i) \) is the same. Figure 3 shows all the order of generation and its probability of the \( (3, 1) \) combination:
of each vertex, $c_{3,1}$ equals the sum of 4 path generation methods’ probability. It’s easy to verify that $c_{3,1} = \frac{1}{5}$.

Here is the implementation code for this selection method:

```c
__device__ bool chooseLightPath(int s, int t, uint&seed)
{
    return (rnd(seed)*(s+t+2-BDPT_EXCLUDED_TECHNIQUES)) <= (float)(s+1);
}
__device__ float Cs(int s, int t)
{
    return 1.f / (s+t+1 -BDPT_EXCLUDED_TECHNIQUES);
}
```

**Implementation Details**

According to the above path generation algorithm, one-sample BDPT program framework is as follows:

```c
#define BDPT_EXCLUDED_TECHNIQUES 2
#define LIGHT_PATH 0
#define EYE_PATH 1
RT_PROGRAM void one_bdpt()
{
    unit seed=...;
    float3 res = make_float(0.f); // final result
    uint2 pixel_pos = ...; // pixel position
    PathVertendpoints[2]; // endpoints of two paths
    verts_cnt[2];
    #define y_endpoint endpoints[LIGHT_PATH]
    #define z_endpoint endpoints[EYE_PATH]
    #define s verts_cnt[LIGHT_PATH]
    #define t verts_cnt[EYE_PATH]
    y_endpoint.init(MTL_LIGHT_SUPERNODE);
    z_endpoint.sampleCameraVertex(seed);//The lenses are usually small, so the camera wasn’t modelled into a part of the scene, and the t=0's sampling technique was removed
    s=0, t=2;
    bool no_primary_hit = z_endpoint.randomWalk(seed, pixel_pos);
    int path_len = no_primary_hit ? (max_depth + 1):(s+t-1);//if the primary fails to hit, terminate
    for(; path_len<= max_depth; ++path_len_)
    {
        float3 connectionContrib = evalConnection(s, t, y_endpoint, z_endpoint);
        float ci = Cs(s, t); // one-sample estimator
        connectionContrib /= ci;
        res += connectionContrib;
        int which_path = (chooseLightPath(s, t, seed)? LIGHT_PATH:EYE_PAHT);
        if(which_path == LIGHT_PATH&&s == 0)
        {
            endpoints[LIGHT_PAHT].sampleLightVertex(seed);
        }
        else
        {
```
if(endpoints[which_path].randomWalk(seed, pixel_pos)) break;
    ++verts_cnt[which_path];
}

Acknowledgement
This research was financially supported by Zhejiang Province Project [2017C31044] "Research on key technologies and system development of modeling and display of multi spectral materials for high fidelity 3D printing".

References