# Shayan's Software & Technology

My adventure as a Software Engineer continues..,

## Parallel Programming: CImg Open-Source Library using nVidia CUDA 5.0 Toolkit on GPU

**Overview**

In the last parallel programming blog post. I analyzed and profiled the filled triangles routine and determined that I could do some tasks in the routine in parallel. This blog post is the log of the measures I took to achieve this goal of parallelizing the routine. It was not an easy task but with a little help from the stackoverflow.com community I was able to overcome obstacles. It is now time to give back to the community by publishing my findings, trials and struggles.

**Prerequisites**

In order to implement parallel programming concepts in present day using nVidia CUDA 5.0 Toolkit. There must be a CUDA enabled nVidia graphics card installed on your machine as well as the CUDA Toolkit.

Not sure if you have any of these? Find out here:

https://developer.nvidia.com/get-started-parallel-computing

My GPU has a compute capability of 1.2 (not that good, current max is 3.0) but good enough!

This means that I can run about 512 threads concurrently per block I allocate on the device itself.

**Initializing Parallel Arrays Simultaneously using the GPU**

Let's look at the code that we are going to run concurrently on the GPU:

```
1  /*
2  * Define random properties (pos, size, colors, ..)
3  *for all triangles that will be displayed.
4  */
5  float posx[100], posy[100],
6        rayon[100], angle[100],
7        veloc[100], opacity[100];
8   unsigned char color[100][3];
9
10 for (int k = 0; k<100; ++k) {
11      posx[k] = (float)(cimg::rand()*img0.width());
12      posy[k] = (float)(cimg::rand()*img0.height());
13      rayon[k] = (float)(10 + cimg::rand()*50);
14      angle[k] = (float)(cimg::rand()*360);
15      veloc[k] = (float)(cimg::rand()*20 - 10);
16      color[k][0] = (unsigned char)(cimg::rand()*255);
17      color[k][1] = (unsigned char)(cimg::rand()*255);
18      color[k][2] = (unsigned char)(cimg::rand()*255);
19      opacity[k] = (float)(0.3 + 1.5*cimg::rand());
20   }
```

What's wrong with the above code? It seems very standard to populate an array in such a fashion in the software industry. The drawback is that each element for a given array has to be populated one at a time for a maximum of 100 times in this particular case. This is largely due to the fact that this code is executing on a single thread on a CPU (Intel Core i7 1.6 GHz). What if we could populate each of these arrays all at once concurrently? This would allow us to get the data we need faster without having to wait for the computer to process each element of an array in serial(one at a time).

Can we do this on the CPU? That would involve creating 100 threads and executing them simultaneously on the CPU. My CPU only has 8 cores and 4 multiprocessors. It cannot accomplish this task. The GPU on my nVidia GT 230m has about 500 cores. In our particular case we only need to use 100 of these cores. In order to run this code concurrently, the code must be executed on the device(GPU). nVidia CUDA Toolkit allows for such

transfers from device memory to host memory. Let's look at this technology in action. Here is my code for the above function but in the form of a CUDA Kernel:

```
/*
 * Setup and initialize curand with a seed
 */
__global__ void initCurand(curandState* state){
int idx = blockIdx.x * blockDim.x + threadIdx.x;
curand_init(100, idx, 0, &state[idx]);
__syncthreads();
}

/*
 * CUDA kernel that will execute 100 threads in parallel
 * and will populate these parallel arrays with 100 random numbers
 * array size = 100.
*/

__global__ void initializeArrays
 (float* posx, float* posy,float* rayon, float* veloc,
  float* opacity ,float* angle, unsigned char* color, int height,
  int width, curandState* state, size_t pitch){

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  curandState localState = state[idx];

  posx[idx] = (float)(curand_normal(&localState)*width);
  posy[idx] = (float)(curand_normal(&localState)*height);
  rayon[idx] = (float)(10 + curand_normal(&localState)*50);
  angle[idx] = (float)(curand_normal(&localState)*360);
  veloc[idx] = (float)(curand_uniform(&localState)*20 - 10);
  color[idx*pitch] = (unsigned char)(curand_normal(&localState)*255);
  color[(idx*pitch)+1] = (unsigned char)(curand_normal(&localState)*255);
  color[(idx*pitch)+2] = (unsigned char)(curand_normal(&localState)*255);
  opacity[idx] = (float)(0.3f + 1.5f *curand_normal(&localState));

__syncthreads();
}
```

Upon analyzing the two code examples above, you may be wondering why the **cimg::rand()** function did not get carried over to the function executing concurrently. The answer is that the above function: **initializeArrays** is known as a **kernel function**. A kernel function is a device function and can only execute other variables and functions that are allocated on device memory. Therefore, I needed to use device functions to calculate random numbers. These particular device functions are from the CURAND API which is a library that allows random number generation on the device and host.

nVidia maintains many libraries that execute on the device and these can be found on its website. Notable examples include: CUBLAS, CURAND, and Thrust. I have had the opportunity to work with all of them. In addition, the programmer can write their own device functions by prefixing the function header with the **__device__** or **__global__** for a kernel. Alternatively, a function that executes on the host can be prefixed with the **__host__** prefix. It should be noted that the API's nVidia maintains are optimized, so it is better to use them if necessary.

Another question you may have is: how did I manage to get those arrays on to device memory from the host? The following code will answer these questions by including calls to **cudaMemcpy** and calling the **kernel** function above:

```
// check for any errors returned by CUDA API functions.
void errCheck(cudaError_t err, const char* msg){
if (err != cudaSuccess)
std::cout<< msg << ": " << cudaGetErrorString(err) << std::endl;
}

// Define the same properties but for the device
 float* d_posx, d_posy, d_rayon, d_angle,
        d_veloc,d_opacity;
 unsigned char* d_color;

// CURAND state
 curandState* devState;

// allocate memory on the device for the device arrays,
// check for errors on each call
 err = cudaMalloc((void**)&d_posx, 100 * sizeof(float));
 errCheck(err, "cudaMalloc((void**)&d_posx, 100 * sizeof(float))");
 err = cudaMalloc((void**)&d_posy, 100 * sizeof(float));
 errCheck(err,"cudaMalloc((void**)&d_posy, 100 * sizeof(float))");
```

```
 err = cudaMalloc((void**)&d_rayon, 100 * sizeof(float));
 errCheck(err,"cudaMalloc((void**)&d_rayon, 100 * sizeof(float))");
 err = cudaMalloc((void**)&d_angle, 100 * sizeof(float));
 errCheck(err,"cudaMalloc((void**)&d_angle, 100 * sizeof(float))");
 err = cudaMalloc((void**)&d_veloc, 100 * sizeof(float));
 errCheck(err,"cudaMalloc((void**)&d_veloc, 100 * sizeof(float))");
 err = cudaMalloc((void**)&d_opacity, 100 * sizeof(float));
 errCheck(err,"cudaMalloc((void**)&d_opacity, 100 * sizeof(float))");
 err = cudaMalloc((void**)&devState, 100*sizeof(curandState));
 errCheck(err,"cudaMalloc((void**)&devState, 100*sizeof(curandState))");

size_t pitch;

//allocated the device memory for source array
 err = cudaMallocPitch(&d_color, &pitch, 3 * sizeof(unsigned char),100);
 errCheck(err,"cudaMallocPitch(&d_color, &pitch, 3 * sizeof(unsigned char),100)");

// launch 1 grid of 100 threads
 dim3 dimBlock(100);
 dim3 dimGrid(1);

 /* Kernel call for initializing CURAND */
 initCurand<<<1,100>>>(devState);

// synchronize the device and the host
 cudaDeviceSynchronize();

 /*Kernel for initializing Arrays */
 initializeArrays<<<1, 100>>>(d_posx, d_posy, d_rayon, d_veloc, d_opacity,
 d_angle, d_color, img0.height(), img0.width(), devState, pitch);

// synchronize the device and the host
 cudaDeviceSynchronize();
```

I wrote an error function to ensure each call to a cuda function was not returning any error code. Memory errors are difficult to spot so this extra measure is necessary. I wrote a tiny function to modularize this process called **errCheck()** and I've included it at the very top of the above code example. The above code executes at the speed of 0.150 seconds concurrently.

**Issues**

I had many issues getting this code to execute correctly and upon reading code and framework documentation I decided that I had enough. So I decided to ask some questions on stackoverflow.com. Some friendly developers decided to help me and I am very appreciative of the open-source community. Here are the questions in case you are having any problems:

http://stackoverflow.com/questions/15245723/using-arrayij-in-a-cuda-kernel-memcpy-call

http://stackoverflow.com/questions/15238009/concurrently-initializing-many-arrays-with-random-numbers-using-curand-and-cuda

**Next Steps:**

The next step for this code is to further optimize it. How am I going to accomplish this? I don't know yet but you can be assured I will have a follow up blog post coming up in April! see you then!