***Name**: Souham Biswas*   ***CWID**: A20365242* ***Email**: sbiswas7@hawk.iit.edu*

CS 597 – Report 8

GPUDirect enabled MPI requires complete re-haul and installation of MPI on cluster; therefore distribution implemented using traditional MPI which uses host memory as intermediate.

All types of layers along with their activation GPU kernels (both forward and backward) have been verified to be working and been checked to be free from memory leakages.

Memory leakage checking was done by the following 2 steps –

1. By using macros and inline functions to verify the return values from CUDA calls and to check for errors right after the launch of a CUDA kernel. The macros are presented below –

```
#define CUDA_ERROR_CHECK


#define CudaSafeCall( err ) __cudaSafeCall( err, __FILE__, __LINE__ )
#define CudnnSafeCall( err ) __cudnnSafeCall( err, __FILE__, __LINE__ )
#define CublasSafeCall( err ) __cublasSafeCall( err, __FILE__, __LINE__ )
#define CudaCheckError() __cudaCheckError( __FILE__, __LINE__ )


#ifdef CUBLAS_API_H_
// cuBLAS API errors
static const char* cublasGetErrorString(cublasStatus_t error) {
  switch (error) {
    case CUBLAS_STATUS_SUCCESS:
      return "CUBLAS_STATUS_SUCCESS";
    case CUBLAS_STATUS_NOT_INITIALIZED:
      return "CUBLAS_STATUS_NOT_INITIALIZED";
    case CUBLAS_STATUS_ALLOC_FAILED:
      return "CUBLAS_STATUS_ALLOC_FAILED";
    case CUBLAS_STATUS_INVALID_VALUE:
      return "CUBLAS_STATUS_INVALID_VALUE";
    case CUBLAS_STATUS_ARCH_MISMATCH:
      return "CUBLAS_STATUS_ARCH_MISMATCH";
    case CUBLAS_STATUS_MAPPING_ERROR:
      return "CUBLAS_STATUS_MAPPING_ERROR";
    case CUBLAS_STATUS_EXECUTION_FAILED:
      return "CUBLAS_STATUS_EXECUTION_FAILED";
    case CUBLAS_STATUS_INTERNAL_ERROR:
      return "CUBLAS_STATUS_INTERNAL_ERROR";
  }
  return "<unknown>";
```

```
}
#endif


inline void __cudaSafeCall(cudaError err,
                                         const char *file,
                                         const int line) {
  #ifdef CUDA_ERROR_CHECK
    if (cudaSuccess != err) {
      fprintf(stderr, "cudaSafeCall() failed at %s:%i : %s\n",
              file, line, cudaGetErrorString(err));
      exit(-1);
    }
  #endif
  return;
}


inline void __cudnnSafeCall(cudnnStatus_t err,
                                          const char *file,
                                          const int line) {
  #ifdef CUDA_ERROR_CHECK
    if (CUDNN_STATUS_SUCCESS != err) {
      fprintf(stderr, "cudnnSafeCall() failed at %s:%i : %s\n",
              file, line, cudnnGetErrorString(err));
    exit(-1);
  }
  #endif
  return;
}


inline void __cublasSafeCall(cublasStatus_t err,
                                           const char *file,
                                           const int line) {
  #ifdef CUDA_ERROR_CHECK
    if (CUBLAS_STATUS_SUCCESS != err) {
      fprintf(stderr, "cublasSafeCall() failed at %s:%i : %s\n",
              file, line, cublasGetErrorString(err));
      exit(-1);
    }
  #endif
  return;
}


inline void __cudaCheckError(const char *file,
                                           const int line) {
  #ifdef CUDA_ERROR_CHECK
    cudaError err = cudaGetLastError();
    if (cudaSuccess != err) {
      fprintf(stderr, "cudaCheckError() failed at %s:%i : %s\n",
              file, line, cudaGetErrorString(err));
      exit( -1 );
    }
  #endif
  return;
}
```

Therefore any call to a generic CUDA API call such as cudaMalloc() must be enclosed within a CudaSafeCall() directive. Similarly, any cuBLAS API call must be enclosed with the CublasSafeCall() directive and all cuDNN API calls be enclosed with in CudnnSafeCall() directive.

However, CUDA kernel calls may be checked for errors by explicitly calling CudaCheckError() after every kernel invocation. This directive basically calls cudaGetLastError() and highlights GPU function execution faults.

These directives have helped eliminate a lot of potential memory leaks which were causing the loss to diverge.

2. Using the cuda-memcheck tool

This tool exactly highlights (even GPU kernel level) violations of memory read/write throughout the execution of the code. It is recommended that the CUDA .cu code files be compiled with the -g -G option to include memcheck debug symbols. A sample error log from this tool looks like this –

```
========= CUDA-MEMCHECK
========= Invalid __global__ write of size 4
=========     at 0x000005e0 in /gpfs/mira-fs1/projects/EnergyFEC_2/users/souham/SHM-
Learn/cooley_workspace/single_node_cifar10/FCLayer.cu:200:ShiftRight_PopulateHelper_GPUKernel
(float*, float*, int, int, int)
=========     by thread (127,0,0) in block (63,0,0)
=========     Address 0x62051f7ffc is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
```

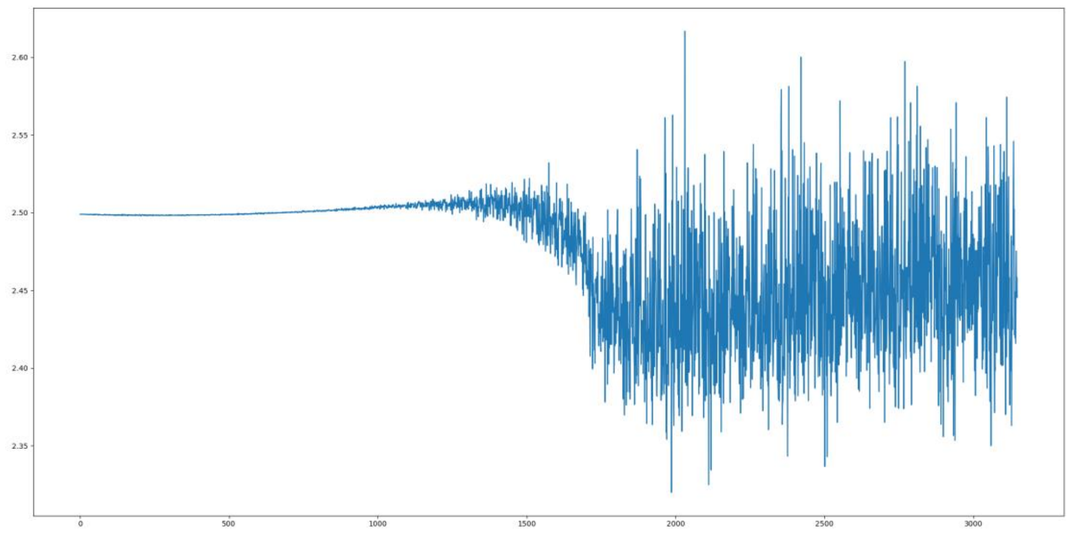
```
=========     Host Frame:/usr/lib64/libcuda.so.1 (cuLaunchKernel + 0x2c5) [0x204235]
=========     Host Frame:/home/souham/.local/cuda-8.0/lib64/libcudart.so.8.0 [0xd23d]
=========     Host Frame:/home/souham/.local/cuda-8.0/lib64/libcudart.so.8.0 (cudaLaunch +
0x143) [0x33783]
=========     Host Frame:convnet [0x9d4c]
=========     Host Frame:convnet [0x960e]
=========     Host Frame:convnet [0x9655]
=========     Host Frame:convnet [0x7bb2]
=========     Host Frame:convnet [0xb3d8]
=========     Host Frame:convnet [0xa9b7]
=========     Host Frame:convnet [0x584f]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ed1d]
=========     Host Frame:convnet [0x36f9]
=========
========= Invalid __global__ write of size 4
=========     at 0x000005e0 in /gpfs/mira-fs1/projects/EnergyFEC_2/users/souham/SHM-
Learn/cooley_workspace/single_node_cifar10/FCLayer.cu:200:ShiftRight_PopulateHelper_GPUKernel
(float*, float*, int, int, int)
=========     by thread (126,0,0) in block (63,0,0)
=========     Address 0x62051f7ff8 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/usr/lib64/libcuda.so.1 (cuLaunchKernel + 0x2c5) [0x204235]
=========     Host Frame:/home/souham/.local/cuda-8.0/lib64/libcudart.so.8.0 [0xd23d]
=========     Host Frame:/home/souham/.local/cuda-8.0/lib64/libcudart.so.8.0 (cudaLaunch +
0x143) [0x33783]
=========     Host Frame:convnet [0x9d4c]
=========     Host Frame:convnet [0x960e]
=========     Host Frame:convnet [0x9655]
=========     Host Frame:convnet [0x7bb2]
=========     Host Frame:convnet [0xb3d8]
=========     Host Frame:convnet [0xa9b7]
=========     Host Frame:convnet [0x584f]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ed1d]
=========     Host Frame:convnet [0x36f9]
=========
```

As is clearly evident, it highlights those threads inside a GPU kernel (along with their thread index) which are making an illegal memory I/O operation which usually results in Segmentation Faults. This tool allowed for their identification and subsequent resolution.

***Name**: Souham Biswas* ***CWID**: A20365242* ***Email**: sbiswas7@hawk.iit.edu*

CS 597 – Report 8

Successfully kicked off distributed training using MPI over 2 nodes. The output suggests the classifier is converging; Sample output presented below –



The noise has to be minimized using better optimization.