Exploiting Modern GPU Architectural Features for Distributed Multi-GPU Graph Analytics

Hochan Lee
Rohan Kadekodi

Abstract

In this project, we explore the use of modern GPU features in D-IrGL to improve the performance of communication of data between the GPUs in a single host and also across multiple hosts. In particular we explore three features that are present in modern GPUs, and we attempt to implement them in the Gluon Substrate of D-IrGL. The first feature is to use asynchronous streaming data transfers to overlap computation at the GPU with the communication between GPU and CPU. The second feature is to use Unified Virtual Addressing to use a single virtual memory address space for host as well as device, to increase the size of the graph partition at each GPU and also to simplify the implementation of Gluon without worrying about the location of data. The third feature is to use GPUDirect RDMA and Peer-to-Peer transfers between GPUs without the involvement of CPU. We implement these features in the context of the bfs_push application, and we discuss about the challenges in implementing them and the future improvements that are possible.

Keywords D-IrGL, Gluon, GPUDirect, Unified Virtual Address, Streaming data transfers

1 Introduction

Graph analytics systems must handle very large graphs such as the Facebook friends graph with more than a billion nodes and 200 billion edges, or the indexable Web graph, which has around 100 billion nodes and trillions of edges. We need parallel computing to process graphs of this size with reasonable performance. GPUs are a popular platform for improving the performance of graph analytical systems, due to their high parallelism. However, these large graphs containing billions of nodes and trillions of edges cannot be processed using the capacity of the memory of a single GPU.

One common solution to process the large graphs is to partition the large graphs across a cluster of machines and GPUs. The graphs are partitioned between the machines, and the communication is handled using a substrate like MPI. The best-performing graph partitioning strategy depends on the algorithm, input graph and number of hosts. While the partitioning of graphs to a cluster of machines makes sure that the GPUs work in parallel and make the computations fast, the performance is bottlenecked by the communication overhead between the different partitions. Also, since different graphs and algorithms need different partitioning policies, a key challenge in distributed graph analytics frameworks is to optimize the communication while supporting heterogeneous partitioning policies.

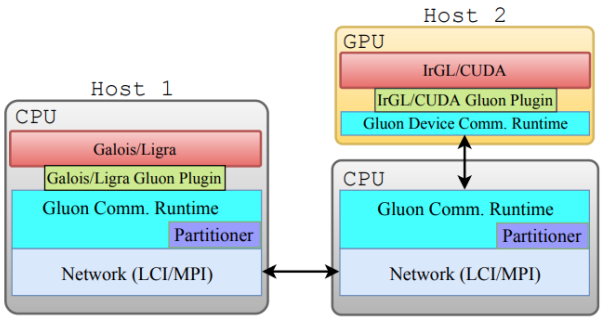

Figure 1. Gluon Overview. The figure provides an overview of the Gluon Communication Substrate.

D-IrGL is a distributed multi-GPU graph analytical framework. It supports multi-host multi-GPU architectures. D-IrGL uses the IrGL compiler for performing computations on each partition independently, and then uses the Gluon communication substrate for communicating and synchronizing data between the different GPUs. The performance of D-IrGL for different graphs depends mainly on the performance of the Gluon communication substrate. Figure 1 shows the basic overview of the Gluon communication substrate.

The limitation in the current Gluon implementation is that it does not use the features that are available in modern GPU architectures such as asynchronous data transfers, unified memory, or direct GPU-GPU communication without the involvement of CPUs. Since these features are independent of the graphs themselves, accommodating these features inside Gluon will improve the performance of all the graphs that use Gluon for communication. In this project, we investigate the different features that have been added to the recent NVIDIA GPUs, and we attempt to apply them to the Gluon substrate and evaluate the performance of a simple BFS push algorithm for large graphs.

The contributions that we make in this project are as follows:

1. We use streaming transfers of data between GPU and CPU of a node to overlap computation of data at the

CS 380C, Fall 2019, UT Austin
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Figure 2. . This figure shows an algorithm for serially performing communication and computation.

GPU along with communication of data between different nodes via CPU.

2. We use Unified Virtual Addressing (UVA) to store the data of the graphs, to store larger partitions on each node.
3. We attempt to implement inter-GPU communication of graph data without the involvement of CPU, among the GPUs located in a single machine as well as GPUs located across multiple machines.

2 Background

In this section, we will provide some background behind D-IrGL, and specifically, the Gluon substrate that is used for communication in the D-IrGL framework. Then we will talk about the bfs_push application that we use as an example for applying the optimizations and also testing the performance.

2.1 D-IrGL and Gluon

[TODO Hochan]

2.2 BFS_PUSH

[TODO Hochan]

3 Overlapping Data Transfers

In this section, we talk about the details of streaming data transfers. Then we go on and talk about how we implement the streaming transfers in Gluon. We then discuss about the implementation and the challenges that we faced during the implementation.

3.1 Streaming Data Transfers

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently. All device operations (kernels and data transfers) in CUDA run in a stream. When no stream is specified, the default stream (also called the âĂIJnull streamâĂİ) is used. The default stream is different from other streams because it is a synchronizing stream with respect to operations on the device: no operation in the default stream will begin until all previously issued operations in any stream on the device have completed, and an operation in the default stream must complete before any other operation (in any stream on the device) will begin.

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset],
                  streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
                  streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHost, stream[i]);
}
```

Figure 3. . This figure shows an algorithm for overlapping communication and computation.

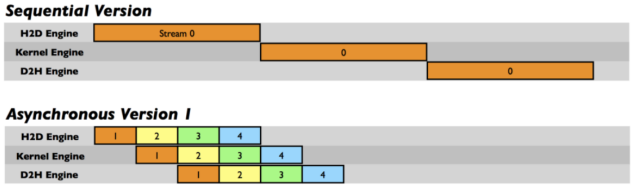

Figure 4. Streaming data performance. This figure shows the difference in performance between serial and overlapping communication with computation

The main goal behind streaming data transfers is that we want to overlap kernel execution along with data transfers. For this to be successful, the kernel execution and the data transfer should both occur in different, non-default streams. Also, the host memory that is involved in streaming data transfers should be pinned memory, if we want to achieve maximum performance. Figure 2 shows the algorithm that uses default stream, which does not overlap computation with communication, whereas Figure 3 shows the same algorithm that overlaps computation with communication using non-default streams. Note that both the algorithms provide current results, but the performance varies significantly.

Figure 4 shows the execution timelines for the sequential algorithm versus the asynchronous algorithm that overlaps computation with communication. As we can see from this figure, the end-to-end latency of the sequential version is roughly 2X compared to the asynchronous version.

3.2 Asynchronous Transfers in Gluon

In the Gluon substrate, during the communication of graph data from a source node (or master node) to a destination node (or mirror node), the following steps take place, in a sequential order:

1. The offset data of the graph and the bitset data of the graph are computed at the GPU of the source
2. The offset data, bitset data and the shared graph data are copied and serialized into a CPU buffer at the source

3. The CPU buffer is transferred from the source to the destination
4. The CPU buffer at the destination is deserialized and copied to the GPU at the destination
5. The corresponding graph data, offset data and bitset data is computed at the GPU of the destination

We modified these steps that occur during the synchronization phase, and we overlapped the computations in step 1 at the source with the data copy from GPU to CPU in step 2 using asynchronous streaming transfers. At the destination, we overlapped the data copy from CPU to GPU in step 4 with the computation on the copied data in step 5 using asynchronous streaming transfers.

3.3 Discussion

The first challenge that we faced while applying this was to find out exactly where it is possible to apply this optimization in the large code base. We solved this by taking an example of the bfs_push algorithm that is currently used for communicating the updates of a node in a graph to all its neighbors. Finally we found the point where there were transfers happening between GPU and CPU at the sender and the receiver, and we found opportunity to use this optimization in that part. We believe that it might be possible to apply this optimization to other parts of Gluon too, and this is just our prototype implementation of overlapping data transfers.

The second challenge was regarding the use of pinned memory for achieving the asynchronous transfers. One limitation of pinned memory is that pinned memory cannot be reallocated or resized once it is allocated. This is an important problem because, in the case of sparse graphs, there is freq uent resizing of buffers in Gluon, and it is impossible to know the size of the graphs or the number of bits in the bitset and offset to statically determine the size of the buffers. We could not use pinned buffers for this feature, and hence we believe that it won't lead to maximum performance improvements on large graphs. There needs to be a comprehensive change in the code base of Gluon to use pinned memory during the transfers, which might lead to better performance.

4 Unified Virtual Addressing

In this section, we talk about the Unified Virtual Addressing (UVA) feature that we use to provide the same virtual memory for both GPU and CPU data, in order to have larger partitions of a graph per GPU. We first talk about the concept of UVA, and then we talk about how we implemented UVA in Gluon and finally we talk about the limitations and future directions with respect to this optimization.

4.1 UVA concept

The GPU memory on servers is typically very limited. So storing graphs in GPUs places a limitation on the size of the graph per node, and requires larger clusters, increasing communication overhead and thus loss in performance. Furthermore, if we store the graphs on the host memory, then we need to copy the graphs from host memory to GPU memory and back during processing, as the address spaces of GPU memory and host memory are different. This leads to complicated implementations, and also cause overheads due to copying of data frequently between the host memory and GPU memory. To avoid this cost of copying and to store large graph partitions per GPU, NVIDIA introduced Unified Virtual Addresses (UVA). In UVA, both the GPU and the CPU memory lie in the same address space. The programmer does not need to worry about creating separate buffers for host memory and GPU memory. This is shown in Figure 5.

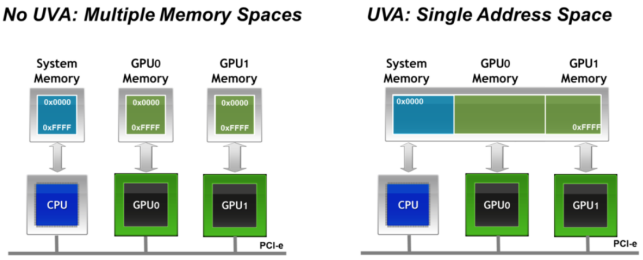

Figure 5. Unified Virtual Address. The figure shows the address spaces with and without using UVA.

In UVA, memory is allocated in a region on the host using a call called cudaHostAlloc(). This memory is pinned memory, and cannot be paged out by the virtual memory system. Once the memory is allocated, it can be accessed from CPU as well as GPU. Note that the access of this pinned memory from GPU can be done without any copying of data from host to GPU. GPU threads can directly access and process data located in the pinned memory on the host. This technique allows you to leverage host memory when your device has limited global memory and also allows you to explicitly transfer data between host and device.

However, sharing zero-copy memory data between host and device involves synchronizing memory accesses across host and device, because modifying data in zero-copy memory simultaneously from host and device could lead to undefined behavior.

4.2 Using UVA in Gluon

We explore the use of UVA in the Gluon substrate in this project. In the original implementation of Gluon, the graph data as well as bitset and offset data is all present in the device memory. So we can imagine that for large graphs, this would lead to a problem, since the device (GPU) memory is very limited. We used Unified Virtual Addressing in which we allocated a the graph, and bitset and offset data on the host memory using cudaHostAlloc(). We then managed to use this pinned data that was allocated in the host directly

for performing GPU computations, with a unified virtual address. We tested our implementation on a server with 2 GPUs, each containing 8GB of device memory. The host itself had 190GB memory. So we could potentially store 10x larger partitions on the host, as opposed to storing the graphs and the bitset and offset data entirely in the GPU.

4.3 Discussion and Limitations

We stored the entire graph and all the bitset and offset data on the host memory using cudaHostAlloc(). Note that this leads to a non-trivial performance drop, since each node has to be fetched from the host through the PCI interface, instead of accessing from the fast bandwidth device memory. There is a lot of opportunity to explore smarter strategies to place only some parts of the graph which might be cold regions that are not accessed frequently using UVA, and store the hot regions in device memory. This would help us get the best of both worlds: get the high performance of device memory and also get the extra capacity from the host memory.

Newer GPUs also support a mode called Unified Memory, which is different from Unified Virtual Addresses. The GPUs that we tested on did not have support for Unified Memory. In Unified Memory, the programmers get a similar interface as that of UVA with a single virtual address space for host and device memory. However, unified memory leads to better performance by transferring data from the host to the device when the device needs it, and managing the synchronization with the host copy of the data transparently without any efforts by the programmer. This has potential to improve the performance significantly by using smart prefetching and readahead strategies to improve the locality of data, and also gain the high capacity of host memory.

5 Inter-GPU communication without CPU Intervention

In this section, we talk about GPUDirect technology for communicating between GPUs without any CPU intervention via Remote Dynamic Memory Access (RDMA). Then we talk about our attempt of implementing the GPUDirect technology in the Gluon communication substrate. We further discuss about the challenges that we faced in this implementation and our insights.

5.1 GPUDirect Technology

Message Passing Interface (MPI) is a standard API for communicating data via messages between distributed processes that is commonly used in High Performance Computing (HPC) to build applications that can scale to multi-node compute clusters. The MPI standard defines a message-passing API which covers point-to-point messages as well as collective operations like reductions.

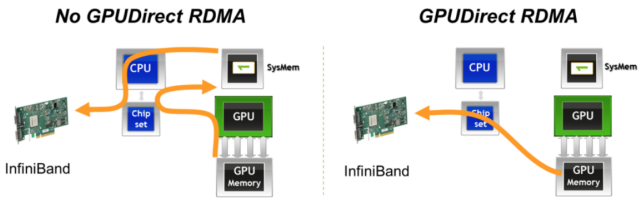

Figure 6. GPUDirect RDMA. The figure shows the data transfers with and without GPUDirect RDMA.

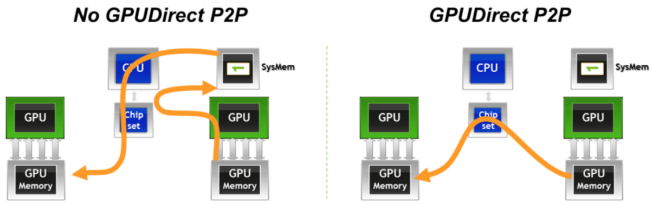

Figure 7. GPUDirect Peer-to-Peer. The figure shows the data transfers with and without GPUDirect Peer-to-Peer.

NVIDIA GPUDirect technologies provide high-bandwidth, low-latency communications with NVIDIA GPUs. GPUDirect is an umbrella name used to refer to several specific technologies. In the context of MPI the GPUDirect technologies cover all kinds of inter-rank communication: intra-node, inter-node, and RDMA inter-node communication. The newest GPUDirect feature is support for Remote Direct Memory Access (RDMA), with which buffers can be directly sent from the GPU memory to a network adapter without staging through host memory. This is shown in Figure 6. Another variant is GPUDirect for Peer-to-Peer (P2P) transfers, which can accelerate intra-node communication. Buffers can be directly copied between the memories of two GPUs in the same system with GPUDirect P2P. This is shown in Figure 7.

If GPUDirect RDMA or GPUDirect P2P is available, the buffers can be directly transferred from the source to the destination without touching the host memory at all. This saves a lot of overhead in serializing/deserializing data and also copying of data between CPU and GPU buffers. Furthermore, the GPUDirect technology can be used along with Unified Virtual Addresses, which further eases the implementation as we do not have to worry about the location of the buffers that are being transferred.

5.2 GPUDirect in Gluon

We attempted to implement the GPUDirect P2P and GPUDirect RDMA transfers in the Gluon Substrate, for transferring the data between GPUs of a single node in the context of the bfs_push application. As explained before, when a source node tries to transfer data to a destination node, the following steps occur:

1. The offset data of the graph and the bitset data of the graph are computed at the GPU of the source
2. The offset data, bitset data and the shared graph data are copied and serialized into a CPU buffer at the source
3. The CPU buffer is transferred from the source to the destination
4. The CPU buffer at the destination is deserialized and copied to the GPU at the destination
5. The corresponding graph data, offset data and bitset data is computed at the GPU of the destination

Using the GPUDirect Technology in this workflow, we can greatly reduce the overheads in these transfers, to the following steps:

1. The offset data of the graph and the bitset data of the graph are transferred using MPI at the source
2. The shared graph data is transferred using MPI at the source
3. The offset and bitset data is received through MPI at the destination
4. The shared graph data is transferred using MPI at the destination

Note that all the steps can occur simultaneously along with other computations at the CPU of source and destination. The data transfers occur asynchronously without any CPU involvement. This feature, if applied effectively, has the potential to significantly improve the performance of data transfers, as compared to the other two features.

5.3 Discussion and Limitations

[TODO Hochan]

6 Conclusion

We explored the use of different powerful features that are present in modern GPU architectures and applied them to D-IrGL. We faced interesting challenges in implementing them and also got ideas for several interesting directions for future work. We hope that the contributions of the project will prove to be valuable additions in the Gluon communication substrate and will improve the performance of distributed graph analytics frameworks significantly.

Acknowledgments

We would like to thank Prof. Keshav Pingali for helping us to acquire the required background knowledge for this project through the course. We would also like to thank our project leaders Vishwesh Jatala and Roshan Dathathri for always being available to help us throughout the project. We would finally like to thank everyone else who directly and indirectly helped us achieve our goals of the project.