Using NVLINK for multi GPU environments

Introduction

One of the key problems in ZKP protocols for hardware (GPU) are memory bottlenecks. As problem sizes grow, the memory scales exponentially often leading to significant time spent in transfering data between host and device. This also remains a key hurdle in ever getting an end to end Prover running inside a device, without any data transfer to the host.

The general way the host to device data transfer occurs is via a PCIe standard. It is a connection that consists of one or more lanes. with each lane consisting of two wires, one for sending and another for recieving. The bandwidth scales with the number of lanes linearly with number of lanes x1 … x16 etc

Unidirectional Bandwidth: PCIe 3.0 vs. PCIe 4.0				
PCIe Generation	x1	x4	x8	x16
PCIe 3.0	1 GB/s	4 GB/s	8 GB/s	16 GB/s
PCIe 4.0	2 GB/s	8 GB/s	16 GB/s	32 GB/s

source: see here

The PCIE speed remains a bottleneck for data transfer as problem sizes continue to scale. How it affects certain algorithms used commonly in ZKP depends on the nature of the algorithm.

1. Sometimes, Large size problems (problems that do not fit in memory of a device) can be broken down into small problems independently computable on multiple devices without any explicit data dependency on each other. We call these problems **globally seperable** problems. Examples of this are algorithms that are used in ZKP such as Multi Scalar Multiplications (MSMs), one can always break a large MSM into two smaller MSMs and combine the result at a negligible cost. This usually happens because of some symmetry in the problem such as additive

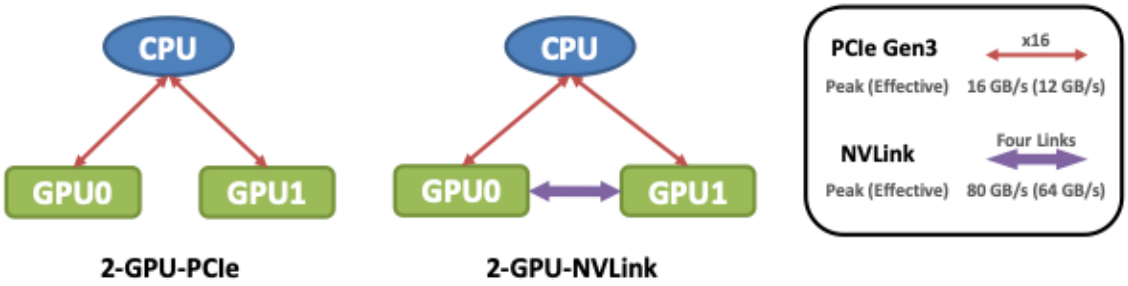
homomorphism (in the case of MSM) or in more general cases high level parallelism in a protocol.

2. On the other hand there are algorithms that have complicated data dependency patterns such as the Number Theoretic Transforms (NTT), while these are parallelizable to a great extent, they often involve data movement or rearrangement such as transposition, which makes it harder to achieve optimal separations, even if for limited part of the algorithm. We refer to these problems as **locally seperable** problems. More examples of such algorithms are NTT, merkle trees, the standard sumcheck protocol for large sizes etc. The amount of data transfer can vary a lot, and in case of NTT split across two devices roughly ¼ of the original data has to be exchanged from each device in the transpose phase.

While all globally separable problems are also locally separable, it is not true the other way in general. A ZKP contains a combination of global and locally seperable algorithms. If want to do a device only prover where problem sizes are larger than device memory, a natural solution is to scale in the device direction by increasing the number of devices, which leads us address the following questions

1. Optimal Device occupancy.
2. Fast data transfer between devices.
3. Efficient data distribution and avoiding race conditions.

In this article we mainly focus on fast data transfer between devices. NVlink is a protocol (link) that enables P2P data transfer between two GPU's. There are other



source: nvidia white paper (old)

Some examples of nvlink usage are

1. Large-Scale Deep learning and AI model training in HP. For eg NVIDIAs repositories use it as an example for bert and GPT., microsoft uses it in High performance LLM inference simulations , dynamic memory management in LLM models,

2. Academic or prototype uses in computational fluid dynamic simulations. In academic works good nvlink performance has been seen in sorting and data exchange processes, large matrix multiplications etc.

Bandwidth

The claimed performance of Nvlink is a 300GB/s bi-directional bandwidth per link (GPU dependent) as opposed to 64 GB/s bi-directional bandwidth per link of PCIE4 x16.

	First Generation	Second Generation	Third Generation
Number of GPUs with direct connection / node	Up to 8	Up to 8	Up to 8
NVSwitch GPU-to-GPU bandwidth	300GB/s	600GB/s	900GB/s
Total aggregate bandwidth	2.4TB/s	4.8TB/s	7.2TB/s
Supported NVIDIA architectures	NVIDIA Volta architecture	NVIDIA Ampere architecture	NVIDIA Hopper Architecture

source: here

On a 3090 pair connected with NVlink, a P2P for about 2^(26) uint32 gets a bandwidth of ~50GB/s

```
root@san1-4g-cgc-xl-1c2c01:~/nvlink_expts/nvlink_testing# ./p2p
cudaDeviceCanAccessPeer(0->1): 1
cudaDeviceCanAccessPeer(1->0): 1
Seconds: 0.012739
Unidirectional Bandwidth: 52.678763 (GB/s)
```

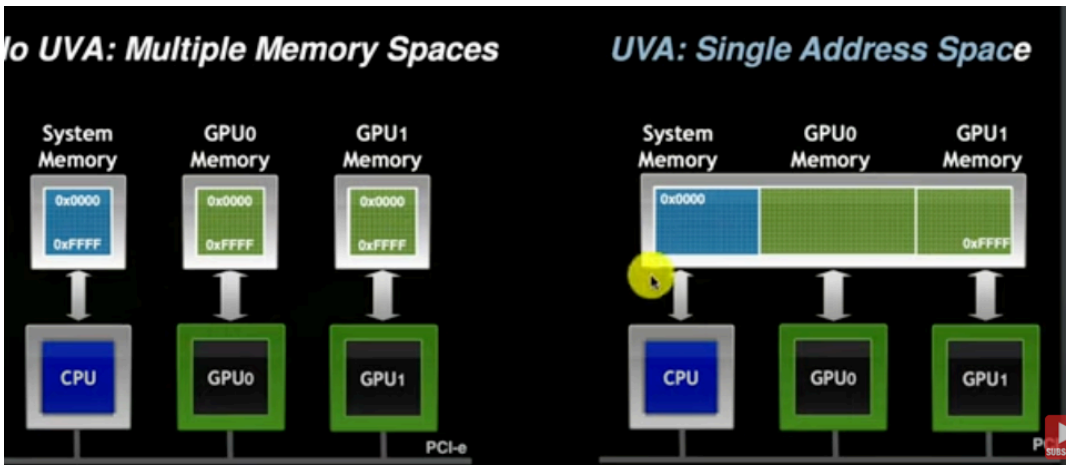
NVidia's own P2P test (with floats) gives a uni-directional bandwidth of ~50GB/s

```
Enabling peer access between GPU0 and GPU1...
Allocating buffers (64MB on GPU0, GPU1 and CPU Host)...
Creating event handles...
cudaMemcpyPeer / cudaMemcpy between GPU0 and GPU1: 48.90GB/s
Preparing host buffer and memcpy to GPU0...
Run kernel on GPU1, taking source data from GPU0 and writing to GPU1...
Run kernel on GPU0, taking source data from GPU1 and writing to GPU0...
Copy data back to host from GPU0 and verify results...
Disabling peer access...
Shutting down...
Test passed
```

The rough bandwidth for bi-directional for 3090 appears to be around 100GB/s.

The method that NVlink uses for connectivity is called Unified Virtual Addressing (UVA), where host and associated devices all share a single continuous virtual address space. In this scenario NVlink merely enables P2P access between the two devices

bypassing host memory. So data stored allocated and stored using $cudamalloc()$ on one device, can be accessed directly using the pointer in another device if there is nvlink enabled across the devices. This is similar to pinned memory where we access data from host pointers using pcie, except that nvlink does so between devices and is claimed to be faster. In principle NVlink can connect between CPU-GPU or GPU-GPU depending on the setup.



source: [this video]

Usage: how to bypass the host?

1. The usage is typically in 4 steps. First identify the nvlink topology using $nvidia - smi\ top$

```
        GPU0    GPU1    GPU2    GPU3    NIC0    NIC1    CPU Affinity    N
GPU0     X      NV4     SYS     SYS     SYS     SYS     0-127   0
GPU1    NV4      X      SYS     SYS     SYS     SYS     0-127   0
GPU2    SYS     SYS      X      NV4     SYS     SYS     0-127   0
GPU3    SYS     SYS     NV4      X      PHB     PHB     0-127   0
NIC0    SYS     SYS     SYS     PHB      X      PIX
NIC1    SYS     SYS     SYS     PHB     PIX      X

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect betwe
  NODE = Connection traversing PCIe as well as the interconnect between P
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typica
  PXB  = Connection traversing multiple PCIe bridges (without traversing
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

2. Enable P2P

```
cudaSetDevice(gpuid[0]);
cudaDeviceEnablePeerAccess(gpuid[1], 0);
cudaSetDevice(gpuid[1]);
cudaDeviceEnablePeerAccess(gpuid[0], 0);
```

3. Allocate memory on both devices. In this case we do a sum reduce by copying two arrays ONLY to device 0, and calling the sum reduce kernel in parallel in device 0 and device 1. Device 1 access the data from device 0 using the UVA

```
cudaSetDevice(gpuid[0]);
uint64_t* d_Input0;
uint64_t* d_Input1;
uint64_t* d_Result0;

cudaSetDevice(gpuid[1]);
//   unsigned int* d_Input1c;
uint64_t* d_Result1;

cudaSetDevice(gpuid[0]);
cudaMalloc(&d_Input0, (DATA_N/2) *sizeof(uint64_t));
cudaMalloc(&d_Input1, (DATA_N/2) *sizeof(uint64_t));
cudaMalloc(&d_Result0,sizeof(uint64_t)* ((DATA_N/2+255)/256));
// Copy full data from host to device 0

cudaSetDevice(gpuid[1]);
cudaMalloc(&d_Result1,sizeof(uint64_t)* ((DATA_N/2+255)/256));
```

4. Copy data into whichever device (in this example we copied in GPU0)

```
cudaSetDevice(gpuid[0]);
// Copy full data from host to device 0
int current_device;
cudaGetDevice(&current_device);
std::cout << "Copying data of size " << DATA_N/2 << " from host to device" << current_device <<std::endl;
copySliceToDevice(d_Input0,h_Input,0,DATA_N/2);
cudaGetDevice(&current_device);
std::cout << "Copying data of size " << DATA_N/2 << " from host to device" << current_device <<std::endl;
copySliceToDevice(d_Input1,h_Input,DATA_N/2,DATA_N/2);
```

5. compute kernels

```
//only compute halfi in device 0
std::cout << "computing sum reduction in device"<<current_device <<std::endl;
sumreduceKernel<<<numBlocks, blockSize, blockSize*sizeof(uint64_t)>>>(d_Input0,d_Result0,DATA_N/2);
cudaDeviceSynchronize();
```

here we pass in to the kernel in device 1, a pointer to the data in device0, the rest

happens via nvlink, bypassing the host.

```
//device 1
cudaSetDevice(gpuid[1]);
//compute other half
cudaGetDevice(&current_device);
std::cout << "computing sum reduction in device"<<current_device <<std::endl;
sumreduceKernel<<<numBlocks, blockSize, blockSize*sizeof(uint64_t)>>>(d_Input1,d_Result1,DATA_N/2);
cudaDeviceSynchronize();
```

Where not to use nvlink?

For **globally seperable** algorithms, it is better off with just dividing into parts where each part runs on a seperate GPU, and there is a minimal recombination overhead.

In this example we applied it for the sum reduce algorithm, The results for sum reduce using 2GPUs without/with nvlink are

```
Test3: 2 GPU:(P2P not enabled) (N/2 data in each) , Parallel sums+comb
Copying data of size 536870912 from host to device0
computing sum reduction in device0
Copying data of size 536870912 from host to device1
computing sum reduction in device1
Sum: 67108839 Time: 647 Milli seconds

Test 4: 2 GPU: (N data in dev 0, dev 1 access by P2P (Nvlink)) Parallel sum+comb
P2P Capable GPUs: device0 and device1
Copying data of size 536870912 from host to device0
Copying data of size 536870912 from host to device0
computing sum reduction in device0
computing sum reduction in device1
Sum: 67108839 took 722 Milli seconds
```

The difference is essentially due to the fact that in the nvlink test, there is an additional p2p time added to the device copy time. The 2GPU time is slightly faster than doing it in 1 GPU (see the sum reduce repo) with a single kernel.

Similarly algorithms such as Multi column hashing are better done with global splitting. See for instance the 11*2^(30) poseidon column hashes in ICICLE evenly split into two GPUs. This is again a case of globally seperable, and it is not worth any P2P read benefits in such cases.

```
Available GPUs: 4
Device ID: 0, Type: NVIDIA GeForce RTX 3090, Memory Total/Free (MiB) 24576/24252
Device ID: 1, Type: NVIDIA GeForce RTX 3090, Memory Total/Free (MiB) 24576/24252
Device ID: 2, Type: NVIDIA GeForce RTX 3090, Memory Total/Free (MiB) 24576/24252
Device ID: 3, Type: NVIDIA GeForce RTX 3090, Memory Total/Free (MiB) 24576/24252
Required Memory (MiB) 6144
Allocate and initialize the memory for layers and hashes
Parallel execution of Poseidon threads
2 GPUs: 2473 ms
Output Data from Thread 0: 0x038b709f4d22b48a6135d07148720d5f0aadd169eb5327ab755c08a8a0520fb6
Output Data from Thread 1: 0x087f453b41ecc05a3a5f5ea67aafe684a7bb788ebd15f576436d447e6790daa6
Sequential execution of Poseidon threads
1 GPU: 4421 ms
Output Data from Thread 2: 0x038b709f4d22b48a6135d07148720d5f0aadd169eb5327ab755c08a8a0520fb6
Output Data from Thread 3: 0x087f453b41ecc05a3a5f5ea67aafe684a7bb788ebd15f576436d447e6790daa6
```

Where to use nvlink?

The concieved scenarios for ZKP are where there is significant data movement

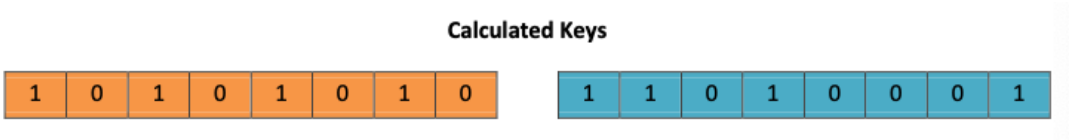
1. Large NTTs out of device size. - Most of NTT can be parallelized well, except the transpose part, **that is exactly where we believe that nvlink will become useful.**.
2. high level protocol workload division. Cryptographic protocols have complicated data dependencies. Protocols such as Groth16/Plonk may be run in a multi GPU environment with NVLINK for faster e2e proving.

Sorting algorithm experiment

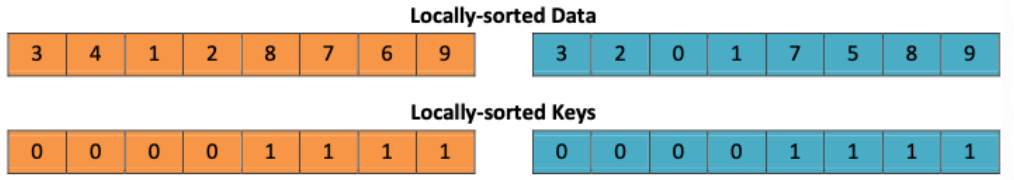
An algorithm similar to transposition is the sorting example covered in [NVLINK white paper](), Where data is stored in multiple GPU's and is sorted in a specific order. It involves the steps key-> local sort -> data exchange based on key -> local sort. The pseudo code is

1. Given data

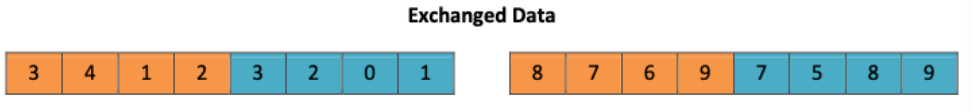
2. hash data to determine which GPU's bin each element should be placed in. Examples: extract highers order bits, arbitrary hash functions, to determine destination bins. In Nvidia eg: $ceil(x/5)$ is used. In this case, we get the keys



3. locally sort data and keys



4. Our eventual goal is to send data associated with all 0's to GPU orange, and all 1's to GPU blue. bidirectional Exchange (use NVLINK)



5. Sort locally again

It has been observed in literature that multi GPU P2P sorting algorithms have only a moderate performance boost compared to multiGPU heterogeneous sort that has an even workload distribution on multicore CPU and GPU without P2P. (see algorithm 1 and 2 in the reference for sorting with arbitrary number of GPUs).

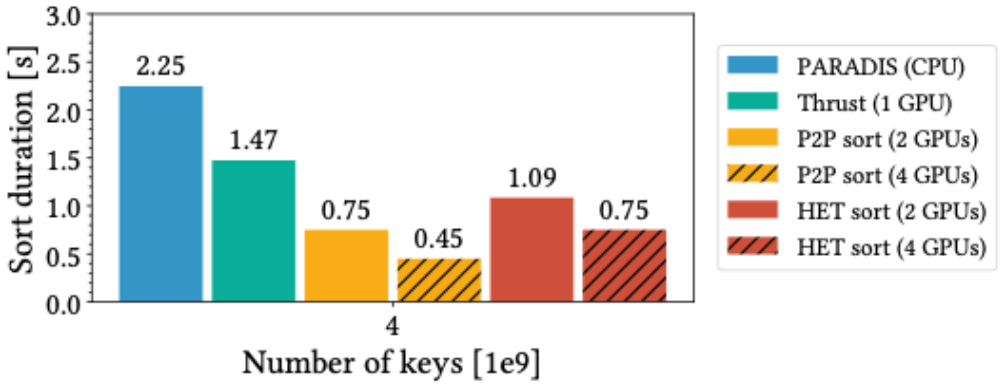
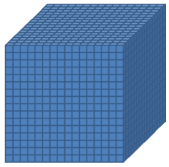


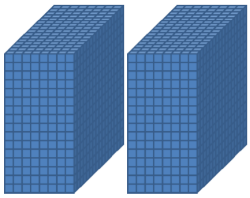
Figure 1: Sorting 16 GB on the DGX A100: CPU vs. GPUs

Large NTT experiment

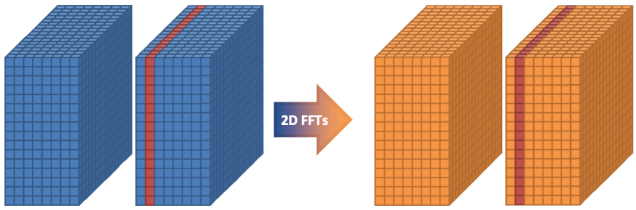
1. Take a large NTT: of size N = k x k x k (or in any hypercube form), here we will use 3d for illustration and consider a 2 GPU with NVLINK setting



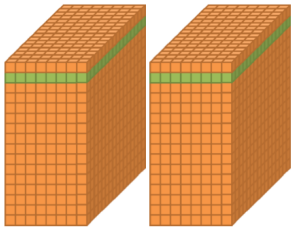
2. Split the cube into two halves, copy each half to GPU0 and GPU1 respectively and enable P2P access with nvlink.



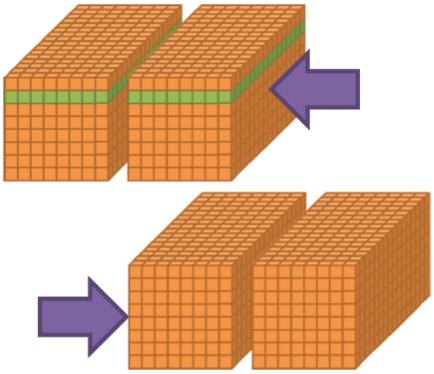
3. Compute 2d FFTs



4. To finish the 3d part, each GPU needs data from the other GPU. This step is known as transposition, and is a known bottleneck in NTTs, since it requires complicated data movement.



5. We can easily achieve transposition by bidirectional P2P, this data transfer will happen via nvlink and should be faster (for u64 size: 2^(30) we got 3x for data transfer) than host2device.



A simple comparison for copying 2^(30) uint64 random integers between host-device and device-device (using nvlink) gives atleast a 3x speedup. This speedup should be VISIBLE in the transposition stage.

```
DATA size (uint_64) 1073741824
H2D_time : 593.877 ms.
P2P Capable GPUs: device0 and device1
D2D_time (NVLINK) : 163.383 ms.
```

6. Thus the overall time spent for transposition in an NTT of very large size will be reduced by a speed factor of 3x-4x. This is roughly equal to the ratio of unidirectional NVLINK bandwidth/PCIExbandwidth, which in the example seen above is roughly 50 GBPs/16GBPs ~ 3x - 4 x.
7. The catch seems to be that, you might be able to get a similar performance as this by using efficient task distribution with multicore CPU+GPU similar to heterogeneous sorting.

Groth16 parallelizing experiment

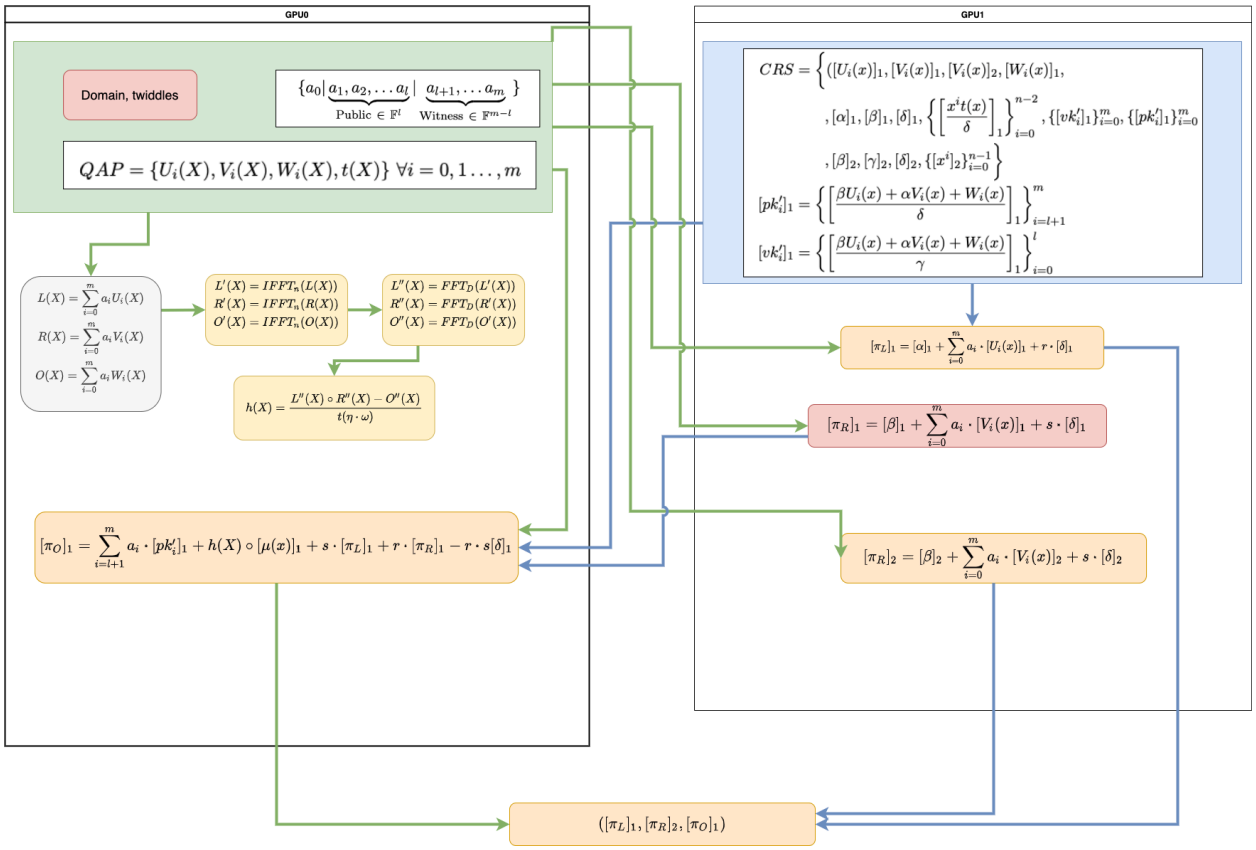
In this part, we propose an experiment with groth16 (backend only) where we try to use protocol seperability to deal with GPU parallelism and use NVLINK to minimise host communication. This necessarily does not assume large sizes, and perhaps can work with tasks within GPU memory capability as well. Below we give a template as to how one can potentially do a full Groth16 proof in 2 GPU;s with NVLINK. As per the msm document, the actual memory needed for a single MSM (BLS12-377 curve) of size 2^(24) is 7 GB for c=18 with no precomputes. Thus at any given point, we can atmost keep enough data for points in a GPU including G1, and possibly G2 which is roughly twice as big.

We suggest the following ordering of the computation

1. Enable P2P access in GPU0 and GPU1 and concurrently
    a. Copy all witness, R1CS, QAP and twiddle factors in GPU0

   b. copy all Points G1. G2 in GPU1.
2. Concurrently run
   a. GPU0: FFT leading to quotient poly, retain quotient poly and witness in memory.
      i. Delete R1CS and QAP data, and domain data at the end of computation
   b. GPU1: Access witness via P2P and do MSM for $[\pi_L]_1$ in GPU1
   c. GPU1: Access witness via P2P do MSM for $[\pi_R]_1$ in GPU1 (can be done simultaneously if memory permits)
3. Concurrently compute
   a. GPU0: P2P access G1 points and $[\pi_L]_1$, $[\pi_R]_1$ and compute $[\pi_O]_1$
   b. GPU1: P2P access witness data and begin G2 computation. Might need splitting and use GPU0 concurrently.
   c. device synchronize
4. Output Proof.

The blue print is given below. If things work out within memory of 2 GPUs, there is no need to go to host. This can be done even with our APIs I believe, as long as the pointers and memory allocations are correctly given. G2 performance might not work



A similar experiment can be done with Plonk, though the division of labor is much more involved.