Published in GPGPU

Georgii Evtushenko Follow
Jul 21, 2020 · 24 min read · Listen

Multi-GPU Programming

Two RTX 2080 connected with NVLink-SLI

The RTX GPU series has introduced an ability to use NVLink high-speed GPU-to-GPU interconnect in a user segment. Ever since I saw the news, I couldn't help thinking about getting one. I had too many questions about some features and the performance of this technology. Is it possible to use remote atomic operations? Where are remote accesses cached? How should communication over NVLink look like if there is no direct connection between GPUs?

By the time I've got answers to those questions, I realized that I need to organize my multi-GPU programming knowledge in general. This post is an attempt to do so. Therefore, I'll highlight some general multi-GPU programming principles before diving

Avoiding multi-GPU support

First of all, why and when do you need to invest your time into multi-GPU support? There is no reason to waste your time unless you need to accelerate a particular instance of your application. Another reason for multi-GPU programming is memory limitations. If a single application instance doesn't fit into a single GPU's memory, it is a case for multi-GPU programming.

In other words, if you have a set of relatively small tasks you'd better run them independently on different GPUs. To run multiple instances of a single-GPU application on different GPUs you could use CUDA environment variable CUDA_VISIBLE_DEVICES. The variable restricts execution to a specific set of devices. To use it, just set CUDA_ VISIBLE_DEVICES to a comma-separated list of GPU IDs. I've written a helper script for this purpose.


```
usage= usage: $0 executable inp1 inp2 inp3 ... inpN

if [ "$#" == "0" ]; then
        echo "$usage"
        exit 1
fi

assigned_gpu=0
gpus_count=$(nvidia-smi.exe -L | wc -l)
executable="$1"
shift

while (($#)); do
        job=$1
        echo "Run ${job} on ${assigned_gpu}"
        export CUDA_VISIBLE_DEVICES=${assigned_gpu}
        export WSLENV=CUDA_VISIBLE_DEVICES/w

        ${executable} ${job} &

        assigned_gpu=$(($assigned_gpu + 1))

        if [ $assigned_gpu -eq $gpus_count ]; then
                wait
        fi

        assigned_gpu=$(($assigned_gpu % $gpus_count))
        shift
done

wait
```

It runs an executable on multiple GPUs with different inputs. Although I used CUDA_VISIBLE_DEVICES to avoid multi-GPU programming, it could be used to facilitate it. Application with multi-GPU support could require this variable in case it doesn't support partially-connected topologies. With CUDA_VISIBLE_DEVICES it's possible to restrict execution to GPUs that are connected with NVLink. I'll tell you why it's important later.

Spend your time on optimizations before multi-GPU support

In this part, I would like to show that it's possible to achieve the same performance improvement by code optimization instead of multi-GPU support. The result of such

Example of a grayscale image

Let's start with a simple kernel. I assigned each thread to one pixel. Although this code performs better than a multi-threaded CPU one, it's far from optimal. It achieves only a fraction of memory utilization. There is no reason to spend time on multi-GPU support here. Instead, let's try to optimize it.

```
1  template <int div>
2  __global__ void gpu_div_kernel (
3    unsigned int pixels_count,
4    const unsigned char * __restrict__ input,
5          unsigned char * __restrict__ output)
6  {
7    const unsigned int pixel = threadIdx.x + blockDim.x * blockIdx.x;
8
9    if (pixel < pixels_count)
10     output[pixel] = input[pixel] / div;
11 }
```

memory utilization. It is more than twice faster than the previous one.

```
struct char_vec
{
public:
  __device__ char_vec (const unsigned char * __restrict__ input)
  {
    *reinterpret_cast<int2*>(data) = *reinterpret_cast<const int2*> (input);
  }

  __device__ void store (unsigned char * output) const
  {
    *reinterpret_cast<int2*> (output) = *reinterpret_cast<const int2*>(data);
  }

public:
  static constexpr int size = 8;
  unsigned char data[size];
};

template <int div>
__global__ void gpu_div_kernel_vec (
  const unsigned char * __restrict__ input,
  unsigned char * __restrict__ output)
{
  auto tid = threadIdx.x + blockDim.x * blockIdx.x;

  char_vec vec (input + tid * 8);

  unsigned char *vec_data = vec.data;
  for (int i = 0; i < vec.size; i++)
    vec_data[i] /= div;

  vec.store (output + tid * 8);
}
```

I assure you that optimization is frequently easier before multi-GPU support. Besides, you'll know what to expect from further optimizations by determining bottlenecks of your code. At this point, the performance of the optimized version is limited by a GPU's memory bandwidth. It's just the right time to switch to multiple GPUs.

up processing. This method is called a domain decomposition.

Image partitioning between multiple GPUs

From a programming point of view, there are different ways of utilization of multiple GPUs. We'll start with the simplest one to show how this simplicity affects performance. The large part of the further description is common for any model, though.

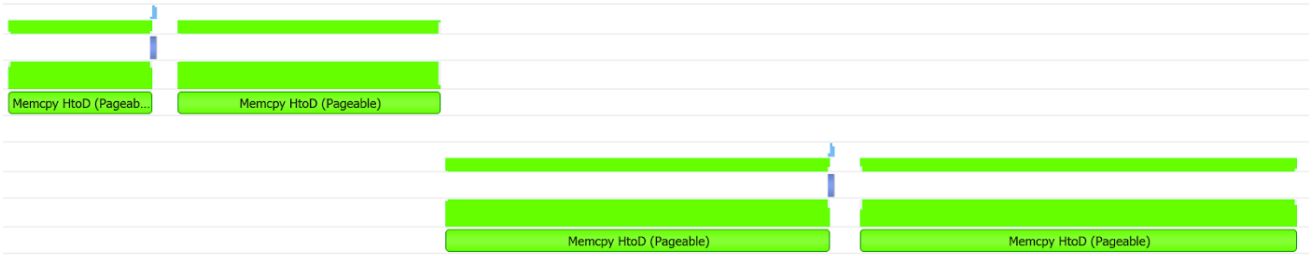
As you may have noticed, there is no such parameter in CUDA API as GPU ID. Launching kernels, transferring data, creating streams, creating, and recording events don't require it. That's because of CUDA's API design that implies a concept of current GPU. All CUDA API calls are issued into a current GPU. It's possible to change the current GPU by cudaSetDevice function call, which receives a GPU's ID. GPU IDs are always in a range [0, number of GPUs). You can get GPUs count with cudaGetDeviceCount.

As you know, kernel calls and asynchronous memory copying functions don't block CPU thread. Therefore, they don't block switching GPUs. You are free to submit as many non-blocking calls into GPU's queue as you want and then switch into other GPU. Calls to the next GPU will be executed concurrently with respect to all others.

```
for (unsigned int device_id = 0; device_id < devices_count; device_id++)
{
  cudaSetDevice (device_id);
  const unsigned int chunk_size = chunk_ends[device_id] - chunk_begins[device_id];
  const unsigned char *host_chunk_input = img->data.get () + chunk_begins[device_id]
  unsigned char *host_chunk_output = host_result + chunk_begins[device_id];
  cudaMemcpy (devices_inputs[device_id], host_chunk_input, chunk_size * sizeof (unsi
  gpu_div_kernel_vec<div> <<<block_sizes[device_id], threads_per_block>>> (devices_i
  cudaMemcpy (host_chunk_output, devices_outputs[device_id], chunk_size * sizeof (un
}
```

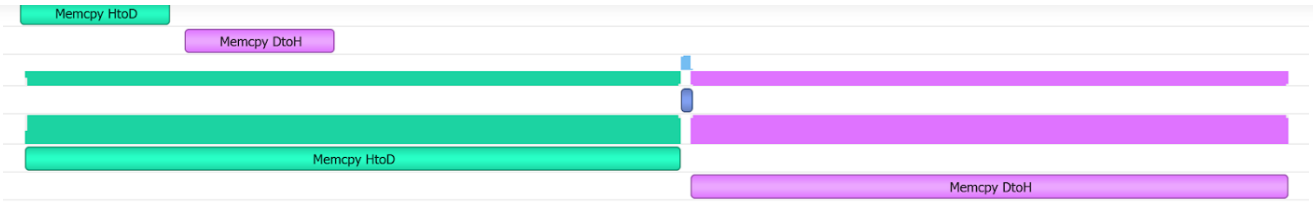
multi_gpu_mistake.cu hosted with ❤ by GitHub view raw

The code above provides no performance improvement over a single GPU. The reason for such behavior is presented below. The thread simply doesn't enqueue further calls until it completes blocking cudaMemcpy calls.



Blocking cudaMemcpy calls

An obvious solution to this issue is to use cudaMemcpyAsync instead of cudaMemcpy. Unfortunately, it won't solve the problem. The asynchronous version of cudaMemcpy is asynchronous with respect to a CPU thread. There are a lot of restrictions that could force CUDA runtime to use blocking version of cudaMemcpy within. One of them is a requirement for host memory to be pinned. It's clear from the profiling above that I used pageable memory. Although the cudaMemcpyAsync call doesn't block CPU thread, there is no difference between cudaMemcpy and cudaMemcpyAsync usage on pageable memory from the CUDA runtime point of view. Pinned memory usage with cudaMemcpyAsync allows performing cudaSetDevice without blocking the second GPU's queue. The result of the discussed changes is presented below. Note that memory copying is here just to illustrate single-thread multi-GPU programming issues. I'll get rid of them soon.

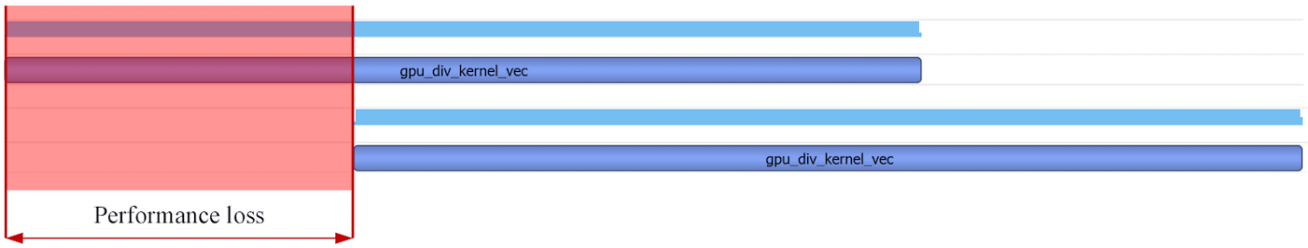


Correct multi-GPU execution

Another important thing to note is that CUDA streams and events are per GPU. Calls to stream can be issued only when its GPU is current. Each GPU has its own default stream. That is clear from the profiling above. I called kernels and memory operations using the default stream of both GPUs. If the default stream was one per process, I'd observe some sort of serialization.

Although an event can be recorded only when its GPU is current, it can be queried and synchronized with when a different GPU is current. This feature is widely used for GPU synchronization purposes. We'll need it later.

For now, this information should be sufficient to dive into the next feature. To demonstrate it, I've extracted expensive transfers outside the kernel-calling loop. Before looking at this feature, let's think about the performance of a multi-GPU run. Although it's not always clear what speedup should be expected from GPU, things are easier with a multi-GPU case. Execution time for kernel above as about 0.026s on my RTX 2080 for an 8000x6000 image. I would expect my multi-GPU code to complete in half of this time (0,013s). Unfortunately, the code is quite slower than expected — 0.015s. That number corresponds to 86% efficiency. But why? To answer this question the NVIDIA Nsight Systems is needed. Profiling shows us that our kernels start with a huge gap. That performance loss is a result of a GPU switch within one CPU thread.



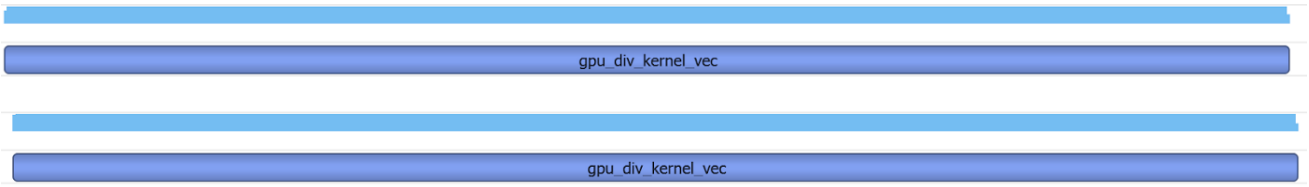
Performance loss on GPU switching

To solve this issue we need to abandon the single-thread multiple GPUs programming

just records the events for time measurement, and call the kernel. Call to sync function synchronizes with the recorded event.

```
std::vector<std::thread> threads;

for (unsigned int device_id = 0; device_id < devices_count; device_id++)
  {
    threads.push_back (std::thread ([&,device_id] () {
        cudaSetDevice (device_id);

        for (unsigned int step = 0; step < 100; step++)
          gpu_data[device_id]->launch<div> ();
        gpu_data[device_id]->sync ();
    }));
  }

for (auto &thread: threads)
  thread.join ();
```

The new multi-GPU programming model performs just fine. An elapsed time for the code above is near the expected values (0,013s). The value corresponds to 100% efficiency of multi-GPU execution.



Efficient multi-GPU execution

At this point, we've discussed the most important features of multi-GPU programming for a simple case of independent kernels. Discussed changes provided us with 100% utilization of a multi-GPU system. Nevertheless, it's impossible to avoid inter-GPU communications in many applications. For example, explicit numerical algorithms for solving partial differential equations require access to the neighboring cells. Domain decomposition technique implies that some neighboring cells are stored in different GPU's memory. In order to compute border cells of a current GPU, their neighbors

As you understand, we are getting closer to the interesting part — multi-GPU communications. Before diving into code, I'll describe technologies that are used for inter-GPU communications.

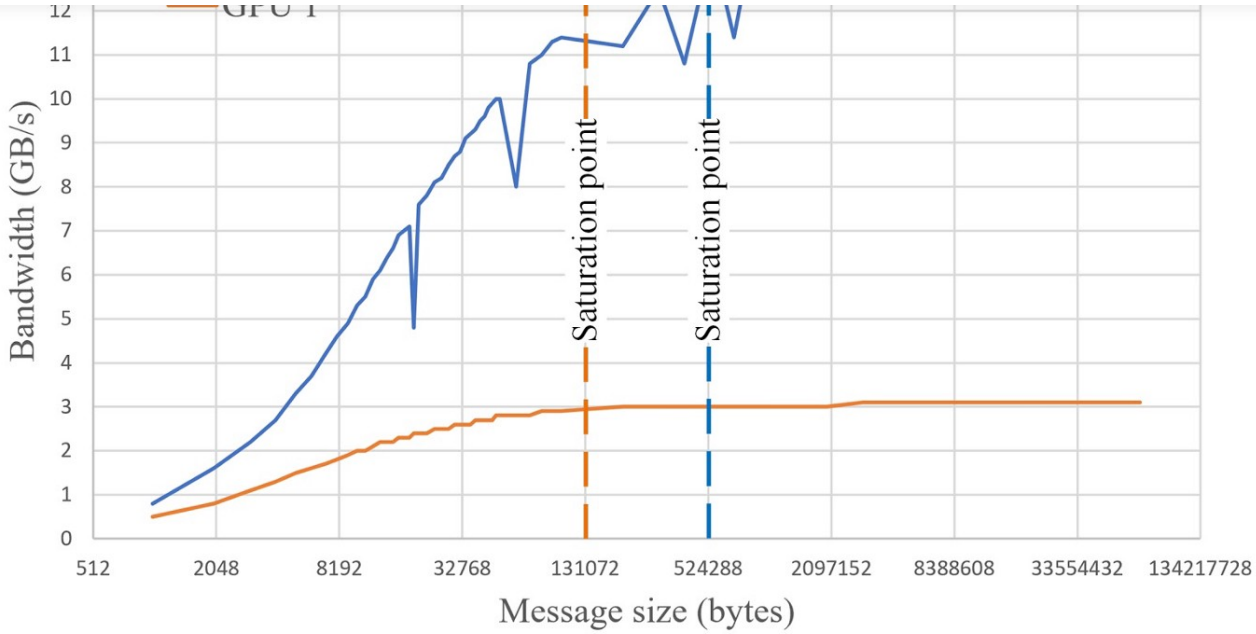
Multi-GPU Communications

Most of the algorithms whose result is affected by other's GPU results require multi-GPU communications. It's often possible to rewrite a multi-GPU problem as a sequence of independent tasks executed on multiple GPUs that are followed by collective communications. For example, you could execute some complex compaction algorithm on each GPU. Afterward, you could use all-gather communication primitive to assemble the results of each other GPU. If your problem suits this scheme, The NVIDIA Collective Communications Library (NCCL) could be interesting for you. NCCL provides multi-GPU and multi-node topology-aware collective communication primitives. You could find optimized versions of many primitives such as all-gather, all-reduce, broadcast, reduce, and reduce-scatter there. If you are not satisfied with the performance of communications extraction, you'll need to understand underlying technologies to write efficient multi-GPU programs. In this case, the following material is for you.

There is a variety of ways for data to transfer. For example, data can be transferred between GPUs through a host memory buffer. Although it's the slowest possible way, I'll describe it to demonstrate some of the multi-GPU programming features.

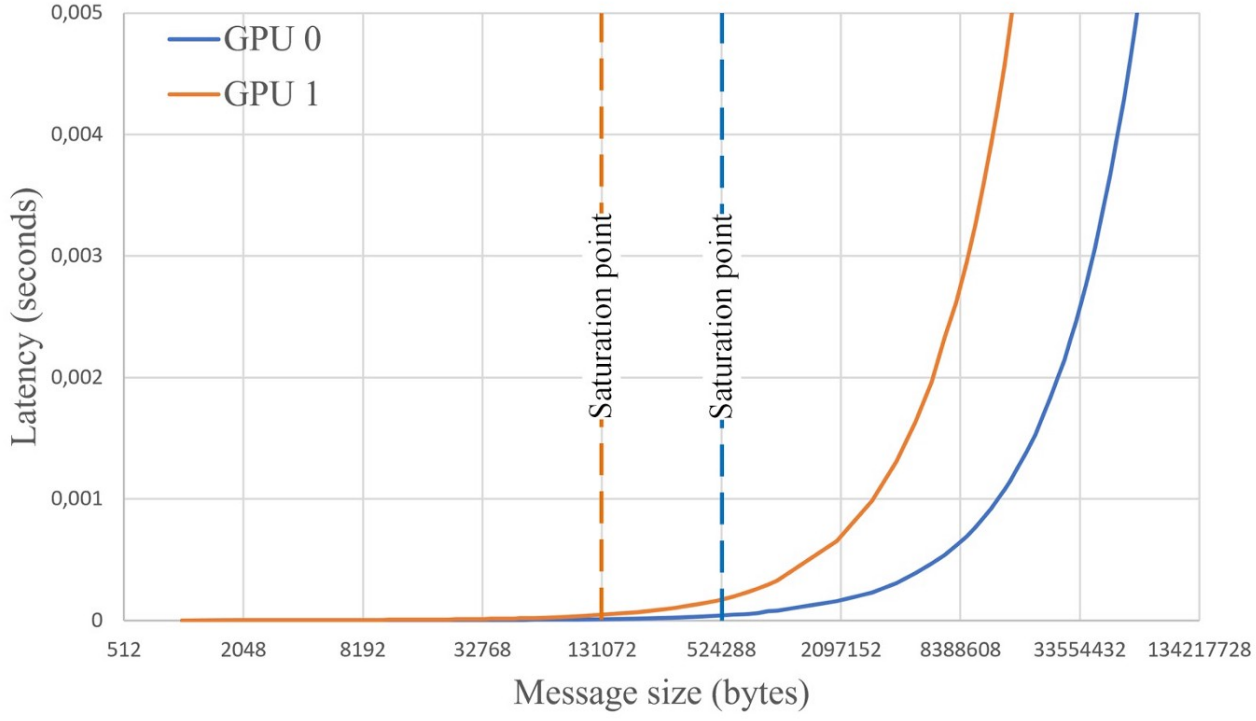
PCIe

If you aren't a Summit user, your host-device transfers use PCIe (Peripheral-Component-Interconnect-Express-Bus). PCIe is a high-speed serial bus standard. PCIe connection consists of one or more lanes. Each lane consists of two pairs of wires, one for receiving and one for transmitting. You may have noticed x1, x4, x8, x16 labels in an nvidia-smi -q output (PCI — GPU Link info — Link Width). These numbers represent PCIe lanes count. Bandwidth scales linearly with respect to the number of links. A single PCIe 3.0 lane has a bandwidth equal to 985 MB/s. In x16 mode, it should provide 15 GB/s.



PCIe CPU-GPU bandwidth

Bandwidth test on my configuration demonstrates 13 GB/s. As you may have noticed from the profiling results above, data transfers to my second GPU took much more time to complete. The reason is that the second GPU is connected with PCIe x4. This gives me 3.3 GB/s for the second GPU (expected slowdown).



PCIe CPU-GPU latency

First of all, if you have many small transactions you don't saturate the PCIe bus. For example, if you have 512 messages of 1 KB, it's better to gather them and send them as one 512 KB transaction. To measure the difference you could run the code below. On my machine, the batched send of 512 KB is 130 times faster.

```
double send_n (unsigned int messages_count, size_t message_size)
{
  std::unique_ptr<unsigned char[]> host_memory (new unsigned char[messages_count * m

  unsigned char *device_memory {};
  cudaMalloc (&device_memory, messages_count * message_size);
  cudaDeviceSynchronize ();

  auto begin = std::chrono::high_resolution_clock::now ();

  for (unsigned int message = 0; message < messages_count; message++)
    cudaMemcpy (
          device_memory + message_size * message,
          host_memory.get () + message_size * message,
          message_size, cudaMemcpyDeviceToHost);

  auto end = std::chrono::high_resolution_clock::now ();

  cudaFree (device_memory);
  return std::chrono::duration<double> (end - begin).count ();
}
```

Now, let's try to copy data between two GPUs while using CPU as an intermediate. To copy data from one GPU to another through CPU with PCIe it's sufficient to call cudaMemcpy function with cudaMemcpyDeviceToDevice flag. It's important to note, that a copy between the memories of different GPUs doesn't start until all commands previously issued to either GPUs have completed. Also, commands in either GPU can't start until the copy between them issued into the default stream has completed. In other words, the default stream extends a synchronization semantics to the multi-GPU case.

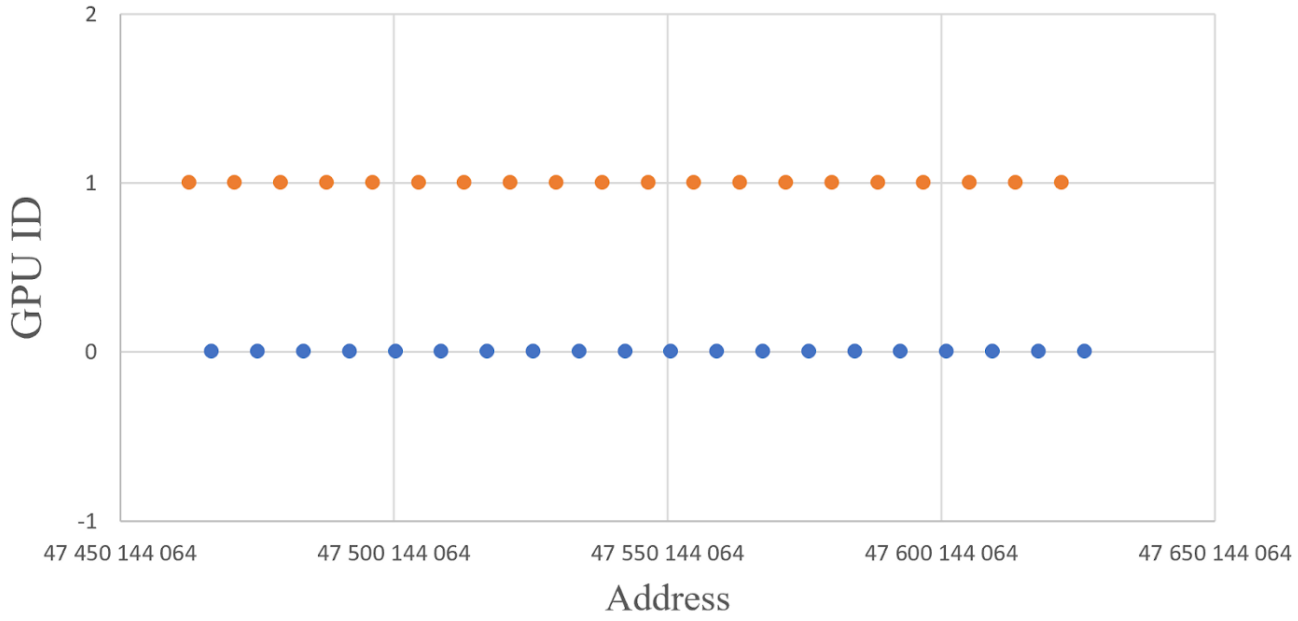
It's fine to use pointers to a memory of a different GPU thanks to Unified Virtual Addressing (UVA). UVA is supported within 64-bit processes only. UVA supports single virtual address space for all devices and the host memory allocated with CUDA API. It

```
int get_device_by_ptr (void *ptr)
{
  cudaPointerAttributes pointer_attributes;
  cudaPointerGetAttributes (&pointer_attributes, ptr);
  return pointer_attributes.device;
}
```

get_gpu_by_pointer.cu hosted with ❤ by **GitHub** **view raw**

If UVA is supported (i.e. your code is within a 64-bit application) you can actually forget about specifying the kind of memory transfer and use cudaMemcpyDefault everywhere. This also works for host pointers not allocated through CUDA. For some reason, UVA is illustrated as continuous virtual address ranges for each device. If it was like that, it would be possible to implement cudaPointerGetAttributes as a range checker. Clearly, this can't be true. To check this, you could run the code below.

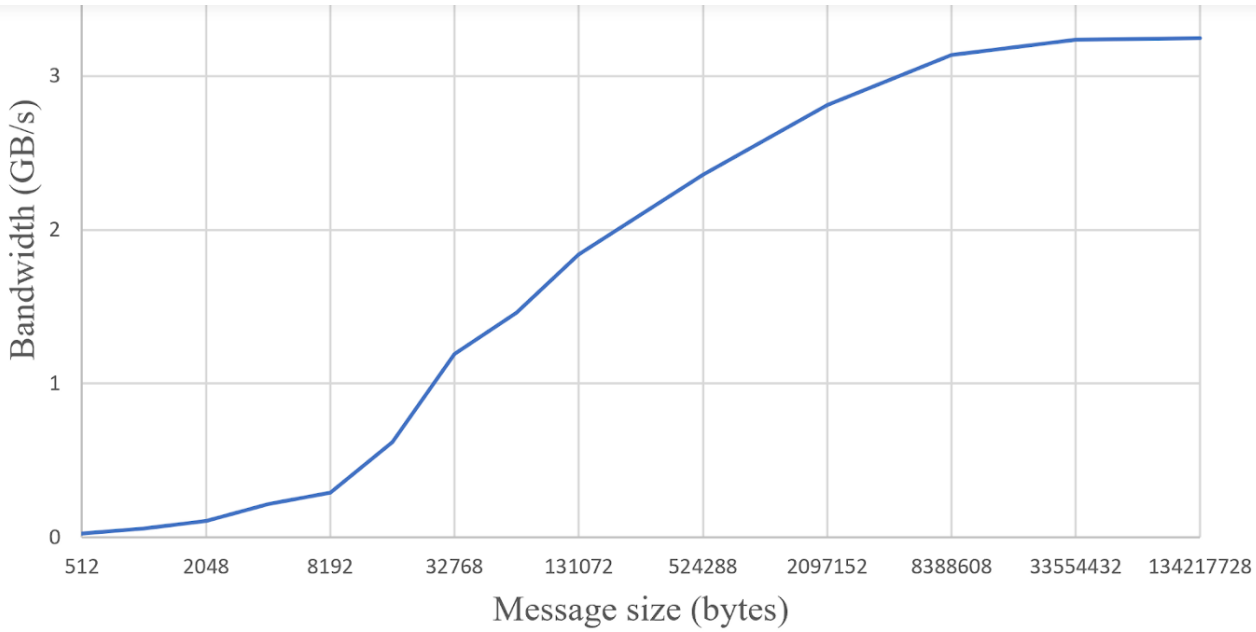
```
void print_ptr_dev (int max_depth, int current_depth = 0)
{
  cudaSetDevice (current_depth % 2);

  void *ptr {};

  cudaMalloc (&ptr, 1024 * 1024 * 4);

  cudaPointerAttributes pointer_attributes {};
  cudaPointerGetAttributes (&pointer_attributes, ptr);

  std::cout << (size_t)ptr << ", " << pointer_attributes.device << "\n";

  if (max_depth > current_depth)
      print_ptr_dev (max_depth, current_depth + 1);

  cudaFree (ptr);
}

int main ()
{
  print_ptr_dev (40);
  return 0;
}
```



Multi-GPU allocations distribution

UVA support is required for further examples. If you have to work with 32-bit applications, there are explicit functions for you. Functions like cudaMemcpyPeer and cudaMemcpyPeerAsync have additional arguments, specifying a source and destination GPU IDs. You don't need UVA to use them.

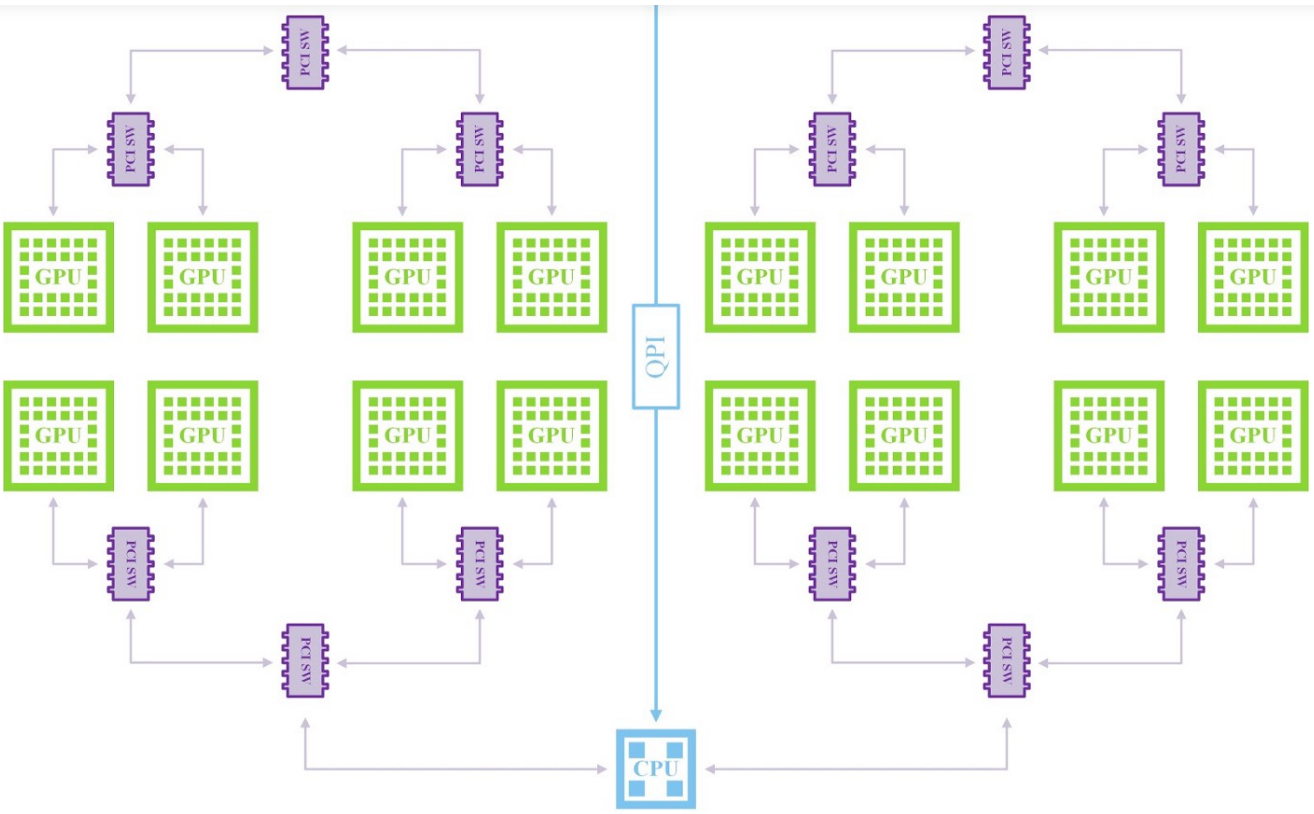
The results of copying data through CPU can be drastically improved. The figure below illustrates that the bandwidth plateau is reached much further than in the case of transfers to the second GPU. Fortunately, there is a way of excluding CPU from GPU-GPU transfers.



Indirect GPU-GPU transfers bandwidth

Throwing out CPU from GPU-GPU memory transfers

If multiple GPUs are connected to the same PCIe hierarchy, it's possible to avoid CPU in the previous scheme. Let's look at the example of PCI topology that allows direct transfers between some GPUs. In the image below you can see a simplified version of a DGX-2 station's PCI topology. The topology consists of two PCIe trees, connected to each other with QPI.



PCI topology of DGX-2

Apparently, binding GPU to a thread from a different NUMA node leads to QPI traffic. If you can't extract host-device traffic from the hot path of your application, there is a piece of good news. It's possible to partition GPUs depending on the best CPU affinity (nvmlDeviceGetCpuAffinity) or bind thread to the best-suited NUMA node (nvmlDeviceSetCpuAffinity). These functions are provided by NVIDIA Management Library (NVML) and work only on Linux systems. Unfortunately, it's impossible to completely avoid NUMA effects in GPU-GPU transfers, because one of the GPUs could have a different affinity. To avoid this headache and get discussed speedup it's better to throw out CPU from GPU-GPU transfers. To do so, PCIe has decent support for performing direct memory accesses (DMA) between two devices on a bus. By PCI specification any PCI device may initiate a transaction, so in some cases, we don't need CPU here. The type of transaction is therefore called Peer-to-Peer (P2P).

Peer-to-peer memory access

Depending on the system properties devices are able to address each other's memory without the need to use main memory as temporary storage or use of the CPU to move data. PCIe's P2P communications significantly reduce communication latency. For

The biggest problem with PCI specification here is that it doesn't require forwarding transactions between hierarchy domains. In PCIe, each root port defines a separate hierarchy domain. The P2P is supported only when the endpoints involved are all behind the same PCI hierarchy domain. CUDA provides the cudaDeviceCanAccessPeer function to check if P2P access is available between two GPUs. CUDA API doesn't distinguish PCIe from NVLink P2P, so cudaDeviceCanAccessPeer returns true if two GPUs don't belong to one PCIe domain but can access each other over NVLink.

If P2P access is available between two GPUs, it doesn't mean that it is used. Allocations mapping to target GPU is required to enable P2P access to the current GPU. There are two possible ways for this. The simplest way is to call cudaDeviceEnablePeerAccess. It's simple because it enables access to all previous allocations on the current GPU to the target GPU. Moreover, it forces all future allocations to be also mapped to the target GPU. The code below illustrates a simple case of P2P enabled copying between two GPUs.

```
int* pointers[2];

cudaSetDevice (0);
cudaDeviceEnablePeerAccess (1, 0);
cudaMalloc (&pointers[0], size);

cudaSetDevice (1);
cudaDeviceEnablePeerAccess (0, 0);
cudaMalloc (&pointers[1], size);

cudaEvent_t begin, end;
cudaEventCreate (&begin);
cudaEventCreate (&end);

cudaEventRecord (begin);
cudaMemcpyAsync (pointers[0], pointers[1], size, cudaMemcpyDeviceToDevice);
cudaEventRecord (end);
cudaEventSynchronize (end);

float elapsed;
cudaEventElapsedTime (&elapsed, begin, end);
elapsed /= 1000;

cudaSetDevice (0);
cudaFree (pointers[0]);

cudaSetDevice (1);
cudaFree (pointers[1]);

cudaEventDestroy (end);
cudaEventDestroy (begin);

return elapsed;
```

NVIDIA Programming Guide states that on non-NVSwitch enabled systems, each device can support a system-wide maximum of eight peer connections.

Although it's quite easy to develop multi-GPU programs with cudaDeviceEnablePeerAccess, it can cause some problems. The main problem is the performance loss of memory allocations. A runtime complexity of a cudaMalloc call is

difference in time with the code below. On my machine with two GPUs, I got 0.23 seconds after enabling peer access instead of 0.1 seconds without it.

```
double measure_allocation_time (unsigned int max_allocations)
{
  std::vector<char *> pointers (max_allocations, nullptr);

  const auto begin = std::chrono::high_resolution_clock::now ();
  for (unsigned int alloc_id = 0; alloc_id < max_allocations; alloc_id++)
    cudaMalloc (&pointers[alloc_id], 2 * 1024 * 1024);
  const auto end = std::chrono::high_resolution_clock::now ();

  for (auto &ptr: pointers)
    cudaFree (ptr);

  return std::chrono::duration<double> (end - begin).count ();
}

void p2p_alloc_bench ()
{
  const unsigned int allocations_count = 1000;
  const double basic_time = measure_allocation_time (allocations_count);
  cudaDeviceEnablePeerAccess (1, 0);
  const double p2p_time = measure_allocation_time (allocations_count);

  std::cout << basic_time << " s => " << p2p_time << std::endl;
}
```

Besides performance issues, it's better to have strict control over inter-GPU memory accesses. I would like to have a runtime error in case of access to memory that wasn't implied as P2P.

Both problems can be solved with a magnificent new API for low-level virtual memory management. The API was introduced in CUDA 10.2. It includes cuMemCreate, cuMemAddressReserve, cuMemMap, and cuMemSetAccess functions for physical memory allocation, virtual address range reservation, memory mapping, and access control respectively. These functions can be used simultaneously with the CUDA runtime functions. I've created some RAII wrappers for these functions in order not to spend your time on the subject. I'll stop only on access control. The code below

some GPUs.

```
template <typename data_type>
class memory_mapper
{
  const virtual_memory<data_type> &virt;

public:
  memory_mapper (
    const virtual_memory<data_type> &virt_arg,
    const physical_memory<data_type> &phys_arg,
    const std::vector<int> &mapping_devices,
    unsigned int chunk)
    : virt (virt_arg)
  {
    const size_t size = phys_arg.padded_size;
    const size_t offset = size * chunk;
    chck (cuMemMap (virt.ptr + offset, size, 0, phys_arg.alloc_handle, 0));

    std::vector<CUmemAccessDesc> access_descriptors (mapping_devices.size ());

    for (unsigned int id = 0; id < mapping_devices.size (); id++)
      {
        access_descriptors[id].location.type = CU_MEM_LOCATION_TYPE_DEVICE;
        access_descriptors[id].location.id = mapping_devices[id];
        access_descriptors[id].flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
      }

    chck (cuMemSetAccess(virt.ptr + offset, size, access_descriptors.data (), access
  }

  ~memory_mapper ()
  {
    chck (cuMemUnmap (virt.ptr, virt.padded_size));
  }
};
```

Furthermore, now it's possible to allocate per-GPU physical memory and map it into a continuous virtual address range. One use case for this feature would be multi-GPU sparse matrix-vector multiplication. Within the previous API, we were restricted to two approaches. The first way of implementing it was by storing the whole vector on all

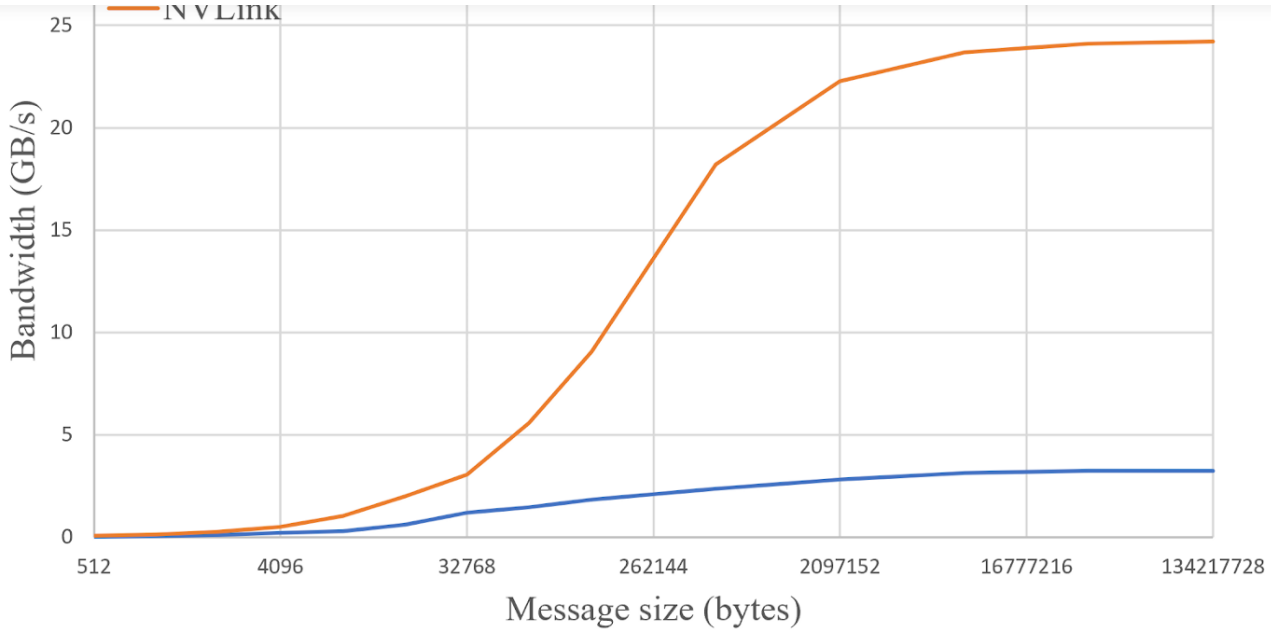
I won't stop here to show benchmark results for PCIe P2P because something more interesting is waiting for us ahead. It would be sufficient to say that in the discussed configuration, device-to-device bandwidth was limited by the second GPU and equal to 3.3 GB/s. Not much. Fortunately, I have NVLink in my configuration.

NVLink

NVLink is a common name for a family of high-speed bridges. NVLink supports CPU-GPU or GPU-GPU linking. It's bidirectional, so each link consists of two sublinks — one for each direction. Each sublink further contains eight lanes. An NVLink can be viewed as a cable with two terminal-plugs. GPU incorporates several NVLink slots. Multiple cables could be used together to improve bandwidth by linking the same endpoints. The bandwidth scales linearly in this case. For example, Pascal-P100 GPU has four NVLink slots. So, it's possible to connect two GPUs with four NVLinks to get 4x bandwidth of a single link. On the other hand, these slots could be used to create complex topologies to link more GPUs.

Another useful feature is atomic operations support. But, of course, the main feature of NVLink is bandwidth. Before diving into benchmarks, let's get used to family members of these bridges.

The first NVLink is called NVLink 1.0. It's used in P100 GPUs. Each NVLink provides a bandwidth of around 20 GB/s per direction. This technology was improved with the second generation of NVLink — NVLink 2.0 which improves per-link bandwidth by 25% (25 GB/s per direction). Each V100 GPU features six NVLink slots. And finally, Turing architecture brings with it NVLink-SLI — NVLink 2.0 based bridge. It could pair two GPUs with one NVLink 2.0 link. NVLink 3.0 was announced recently. It should double NVLink 2.0 bandwidth and provide 50 GB/s per link per direction. I'll consider NVLink-SLI further in this post. The difference between NVLink-SLI P2P and PCIe bandwidth is presented in the figure below.

NVLink GPU-GPU bandwidth

Besides higher bandwidth, NVLink-SLI gives us lower latency than PCIe. It's about 1.3 microseconds, in comparison with 13 microseconds of PCIe. You can get access to some NVLink performance counters from your code. To do so you need to use NVML's API. The interface has changed in CUDA 11, so I'll show you two versions. The code below works for any CUDA version prior to 11.

```
nvmlInit ();
nvmlDeviceGetCount (&devices_count);

nvmlDevice_t device;
nvmlDeviceGetHandleByIndex (0, &device);

nvmlNvLinkUtilizationControl_t utilization_control;
utilization_control.units = NVML_NVLINK_COUNTER_UNIT_BYTES;
utilization_control.pktfilter = NVML_NVLINK_COUNTER_PKTFILTER_ALL;
nvmlDeviceFreezeNvLinkUtilizationCounter (device, 0, 0, NVML_FEATURE_DISABLED);
nvmlDeviceSetNvLinkUtilizationControl (device, 0, 0, &utilization_control, 1);

unsigned long long int tx_before {};
unsigned long long int rx_before {};
nvmlDeviceGetNvLinkUtilizationCounter (device, 0, 0, &rx_before, &tx_before);

// code to measure

unsigned long long int tx_after {};
unsigned long long int rx_after {};
nvmlDeviceGetNvLinkUtilizationCounter (device, 0, 0, &rx_after, &tx_after);

const unsigned long long int tx = tx_after - tx_before;
const unsigned long long int rx = rx_after - rx_before;
```

And here is the version for CUDA 11. If you don’t need such a fine-grained measurement, you could use nvidia-smi nvlink -gt d before and after your application run.

```
nvmlInit ();
nvmlDeviceGetCount (&devices_count);

nvmlDevice_t device;
nvmlDeviceGetHandleByIndex (0, &device);

nvmlFieldValue_t field;
field.scopeId = 0;
field.fieldId = NVML_FI_DEV_NVLINK_THROUGHPUT_DATA_TX;
nvmlDeviceGetFieldValues (device, 1, &field);
const unsigned long long int initial_tx = field.value.ullVal;

field.fieldId = NVML_FI_DEV_NVLINK_THROUGHPUT_DATA_RX;
nvmlDeviceGetFieldValues (device, 1, &field);
const unsigned long long int initial_rx = field.value.ullVal;

// code to measure

field.fieldId = NVML_FI_DEV_NVLINK_THROUGHPUT_DATA_TX;
nvmlDeviceGetFieldValues (device, 1, &field);
const unsigned long long int final_tx = field.value.ullVal;

field.fieldId = NVML_FI_DEV_NVLINK_THROUGHPUT_DATA_RX;
nvmlDeviceGetFieldValues (device, 1, &field);
const unsigned long long int final_rx = field.value.ullVal;

const unsigned int rx = final_rx - initial_rx;
const unsigned int tx = final_tx - initial_tx;
```

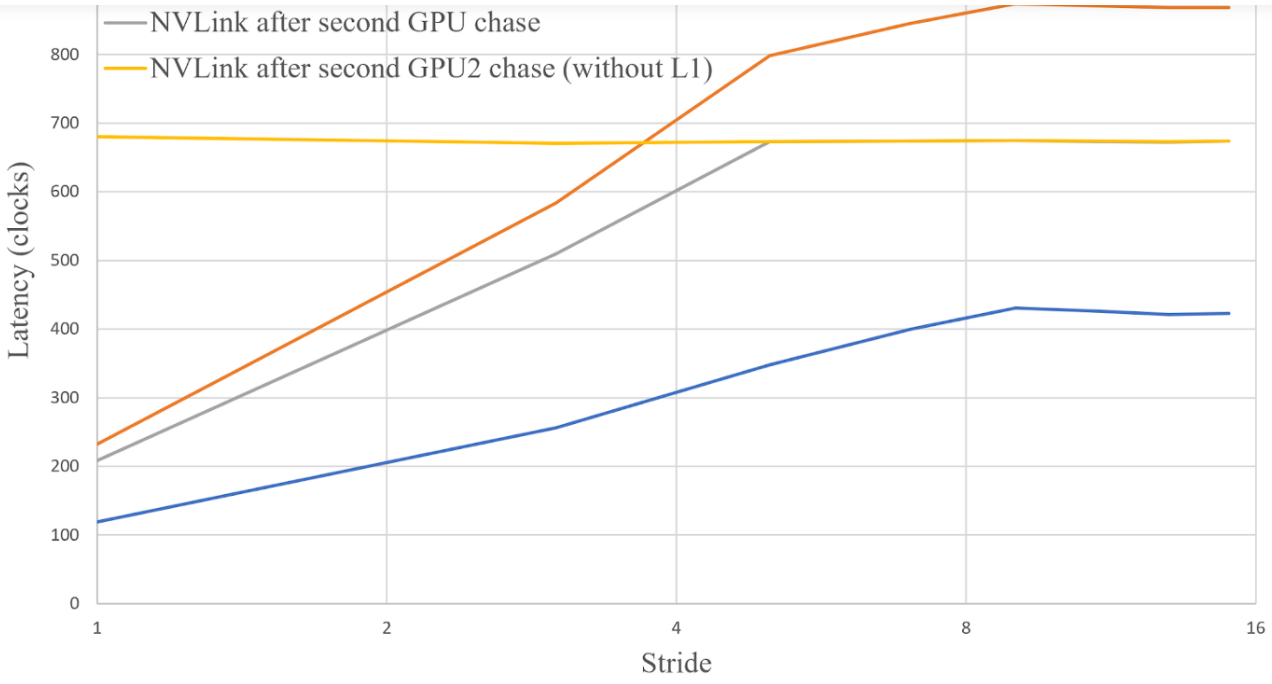
These metrics could give you some interesting insights. I've been trying to understand the numbers that they gave me when I realized that warp-wide atomicAdd is actually compiled to __ballot_sync and one atomic operation per warp (godbolt). By the way, to test that NVLink actually supports atomic operations you could use the code below. I pass pointers to arrays on different GPUs here. In that case, both GPUs work on the same data simultaneously. After a sufficient amount of iterations, resulting values are checked and compared with expected ones.

```
unsigned int *gpu_0, unsigned int *gpu_1)
{
  for (unsigned int step = 0; step < iterations; step++)
    {
      if (step % 2 == 0)
        {
          for (unsigned int i = 0; i < times; i++)
            {
              atomicAdd_system (gpu_0, 1);
              atomicAdd_system (gpu_1, 1);
            }
        }
      else
        {
          for (unsigned int i = 0; i < times; i++)
            {
              atomicSub_system (gpu_0, 1);
              atomicSub_system (gpu_1, 1);
            }
        }
    }
}
```

It answers the question that I asked at the beginning of this post. NVLink does support remote atomic operations. It's time to go to the second question. Where are remote accesses cached? Well, measuring memory accesses is never easy. We need to eliminate any instructions that are not memory operations. There is a classical trick for this particular case that is called pointer chasing.

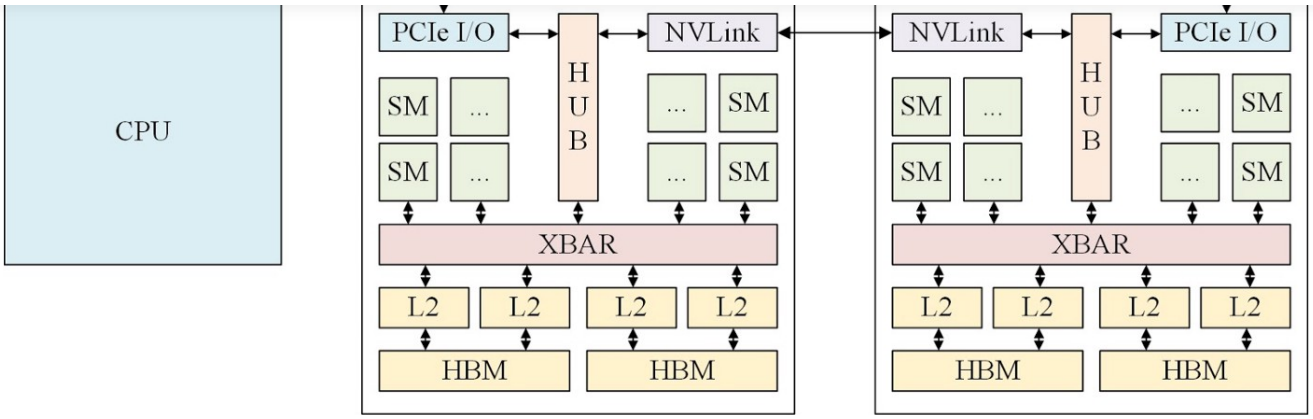

```
#define REPEAT16(x) REPEAT8(x) REPEAT8(x)
#define REPEAT32(x) REPEAT16(x) REPEAT16(x)
#define REPEAT64(x) REPEAT32(x) REPEAT32(x)

struct node { node *ptr; };

static __device__ __inline__ uint32_t __clk ()
{
  uint32_t mclk;
  asm volatile("mov.u32 %0, %%clock;" : "=r"(mclk) :: "memory");
  return mclk;
}

__global__ void pointer_chase (unsigned int n, node * __restrict__ head, int32_t *re
{
  node *ptr = head;

  const int32_t begin = __clk ();

  REPEAT64(ptr = ptr->ptr)

  const int32_t end = __clk ();

  if (result)
    result[0] = (end - begin) / 64;

  head[n] = *ptr;
}
```

Pointer chasing involves a repeated series of irregular memory accesses that require the accessed data to determine the subsequent pointer address to be accessed. In the code above I pass a pointer to a linked list. Then I move the current pointer to the next node in the list 64 times. By controlling dislocation of nodes in memory I can control a stride of memory accesses, and therefore analyze caching effects. To dissect caching effects I've run the kernel above in different ways. By passing a pointer to own memory of GPU, I was able to measure global memory latency. To measure latency of remote memory accesses over NVLink I passed a pointer to remote memory. You could see from the figure below that the latency of NVLink-SLI accesses is approximately equal to two global memory accesses.

NVLink latency

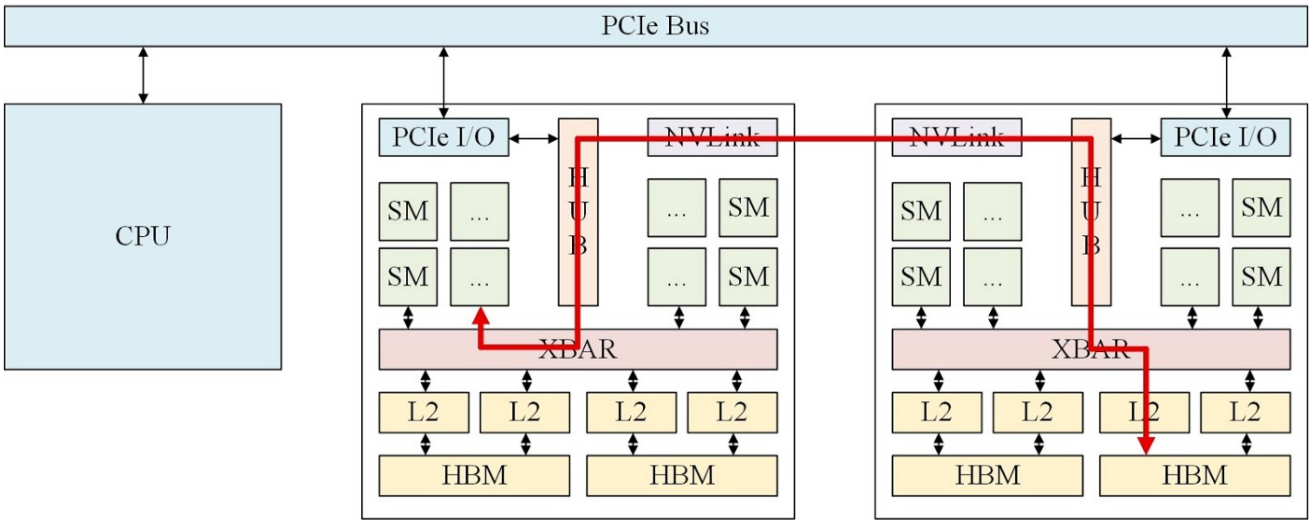
You can see another two lines here. Before describing them, I need to focus on GPU architecture in a more detailed way. Our goal is to understand the underlying hardware. The first step in this journey consists in the understanding of the L1 cache role in NVLink memory operations. As you know, L1 cache resides within a Streaming Multiprocessor (SM). To show if it caches accesses over NVLink I compiled the kernel above with a -Xptxas -dlcm=cg option. The option -dlcm stands for a default load cache modifier. Its default value is 'ca', which means 'cache at all levels'. I've used 'cg', which means 'cache at L2 and below, not L1'. It's also possible to use 'cv', which means 'don't cache and fetch at each load'. Performance degradation with -dlcm=cg clearly indicates that L1 is used with NVLink memory operations. At this point, there is no difference between the reading of own memory and reading of remote memory over NVLink. Is it true for L2? Well, no. The difference in L2 cache is quite interesting. The data is actually cached in L2. The only difference is that it's L2 of a different GPU. To prove that I called the discussed kernel on the second GPU with its own memory. Then I switched to the first GPU and called the kernel on the memory of the second GPU. You could see the performance impact of this test in the figure above ('NVLink after second GPU chase').



NVLink architecture

The path that data takes when the NVLink is used is shown below. It's interesting to note that NVLink is not self-routed. If two GPUs are connected with an intermediate GPU, P2P won't be used. There is a way to overcome this issue. You can run a routing kernel on the intermediate GPU to facilitate the data transfer. In this case, you won't spend extra memory of the intermediate GPU on buffers.

Knowledge about underlying hardware architecture is useful for profiling. Now you can justify increased L2 misses on remote GPU in comparison to single-GPU execution. Fortunately, there is another application of described knowledge. Now, when we know the exact latency of remote memory accesses, we can hide it.



NVLink data path

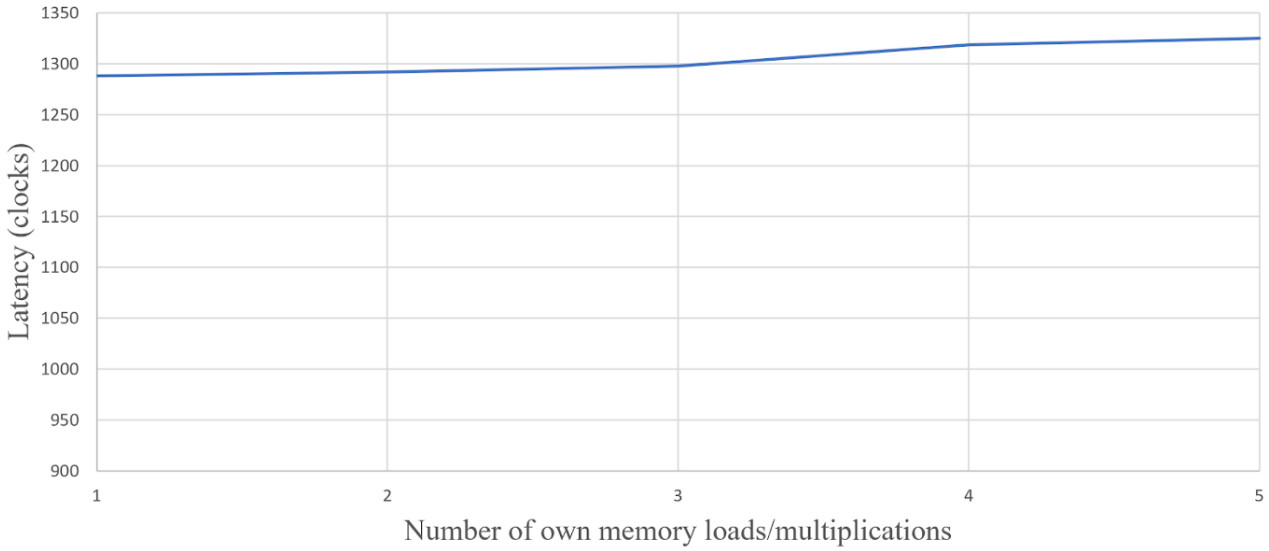
Hiding NVLink latency

Memory loads don't lead to stalls. Actually, it is further instructions that use the loaded

is aware of this feature. In most cases, it manages to reorder memory loads with instructions execution to increase instruction-level parallelism (ILP). In the code below I added own memory loads and some arithmetic. By controlling the ILP we could see how many instructions can be overlapped by remote memory loads.


```
4   }
5
6   __forceinline__ __device__ float load_without_l1 (const float * p)
7   {
8     float out;
9     asm("ld.global.cg.f32 %0, [%1];" : "=f"(out) : "l"(p));
10    return out;
11  }
12
13  <template ILP>
14  __global__ void latency_kernel_1 (
15      float mult,
16      const float *__restrict__ current_gpu_data,
17      const float *__restrict__ remote_gpu,
18      float *__restrict__ result,
19      int32_t *__restrict__ elapsed)
20  {
21    const int32_t begin = __clk ();
22    const unsigned int tid = threadIdx.x;
23
24    float regs[ILP];
25
26    const float remote_val = remote_gpu[tid];
27    // __prefetch_global_l1 (remote_gpu);
28
29    for (int i = 0; i < ILP; i++)
30      regs[i] = load_without_l1 (current_gpu_data + tid + i * gridDim.x * blockDim.x);
31
32    for (int i = 0; i < ILP; i++)
33      regs[i] *= mult;
34
35    float sum = remote_val * mult;
36    for (int i = 0; i < ILP; i++)
37      sum += regs[i];
38
39    const float remote_result = remote_val * mult;
40    result[tid] = sum;
41
42    const int32_t end = __clk ();
43
44    if (elapsed && tid == 0)
45      elapsed[0] = end - begin;
```


GPU is about 1290 cycles. If you don't have enough ILP, it's fine too. It's stated that a sufficient amount of eligible warps can hide 10% of remote memory accesses latency even with PCIe.



If it isn't enough in your case, you need to duplicate remote memory and perform bulk memory operations outside the kernels. But even in this approach, there is room for latency hiding. I'll illustrate remote memory duplication along with some further optimization techniques on a simulation of Maxwell's equations.

Remote memory duplication

Maxwell's equations describe how electric and magnetic fields evolve. To find an approximate solution to Maxwell's differential equations, I use the Finite-Difference Time-Domain (FDTD) or Yee's method. FDTD is a grid-based method. The performance of grid-based methods usually depends on a size of the grid. The computation itself looks like a loop calling two functions that update the H and E field in each cell of the grid.

```
    int cell_id,
4
5     float dx, float dy,
6     const float * __restrict__ ez,
7     const float * __restrict__ mh,
8     float * __restrict__ hx,
9     float * __restrict__ hy)
10  {
11    const float cez = ez[cell_id];
12    const int cell_x = cell_id % nx;
13    const int cell_y = cell_id / nx;
14
15    const int top_neighbor_id = nx * (cell_y + 1) + cell_x;
16    const int right_neighbor_id = cell_x == nx - 1 ? cell_y * nx + 0 : cell_id + 1;
17
18    const float cex =  (ez[top_neighbor_id]   - cez) / dy;
19    const float cey = -(ez[right_neighbor_id] - cez) / dx;
20
21    hx[cell_id] -= mh[cell_id] * cex;
22    hy[cell_id] -= mh[cell_id] * cey;
23  }
```

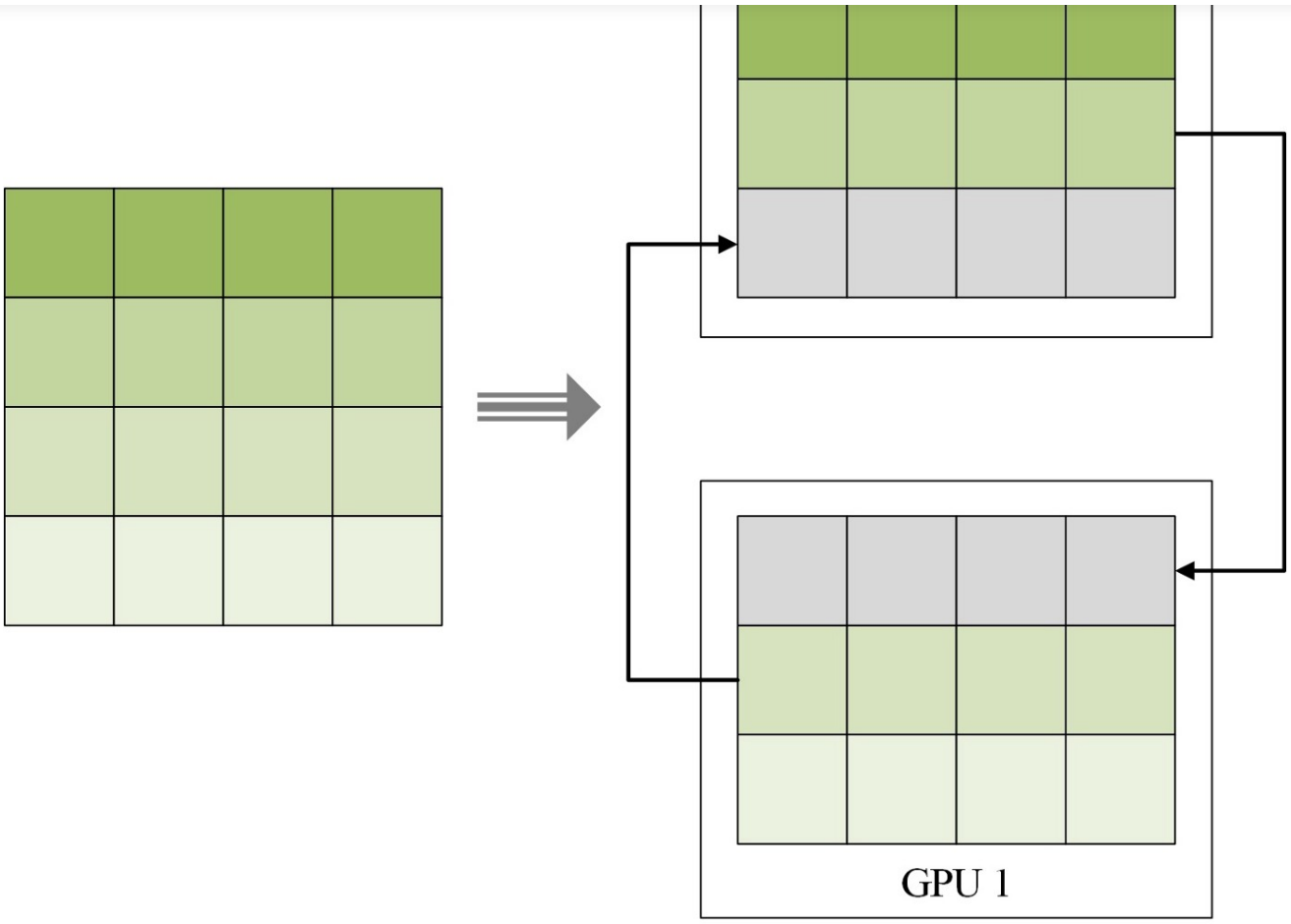
To update H field value in a cell, the value of the H field within the cell is needed along with the values of the top and right neighbors of the cell. A function that updates the E field looks much the same, except it requires values from the bottom and left cells.


```
    int own_in_process_begin,
    int source_position,
    float t, float dx, float dy, float C0_p_dt,
    float * __restrict__ ez, float * __restrict__ dz,
    const float * __restrict__ er,
    const float * __restrict__ hx,
    const float * __restrict__ hy)
{
    const float chx = hx[cell_id];
    const float chy = hy[cell_id];

    const int cell_x = cell_id % nx;
    const int cell_y = cell_id / nx;

    const int left_neighbor_id = cell_x == 0 ? cell_y * nx + nx - 1 : cell_id - 1;
    const int bottom_neighbor_id = nx * (cell_y - 1) + cell_x;

    const float chz = (chy - hy[left_neighbor_id])   / dx
                    - (chx - hx[bottom_neighbor_id]) / dy;

    dz[cell_id] += C0_p_dt * chz;

    if ((own_in_process_begin + cell_y) * nx + cell_x == source_position)
      dz[cell_id] += calculate_source (t, 5E+7);

    ez[cell_id] = dz[cell_id] / er[cell_id];
}
```

You can see the result of the simulation in the video below. In the video, I've created a ground-penetrating radar example with two objects.

Example of the FDTD multi-GPU simulation

Since I'm considering a two-dimensional case of Maxwell's equations, the grid is much alike the grayscale image. We already know how to deal with images in a multi-GPU environment. Let's apply the domain decomposition technique to support multiple GPUs. The only problem here is that cells need field components values of neighbors to compute a value for the next time step. To reduce NVLink traffic within kernel call I created a ghost layer of cells around own elements of the grid chunk for each GPU.



Ghost layers

The code below illustrates the main loop. I've applied all the fixes that we discussed before. It is executed by multiple threads and uses P2P communications over NVLink. And, of course, there is room for optimization.

```
      update_h_kernel<<<blocks_count, threads_per_block>>> (
        nx, n_own_cells, dx, dy, own_ez, own_mh, own_hx, own_hy);

      cudaMemcpy (
        grid_accessor.get_top_copy_dst (fdtd_fields::hx),
        grid_accessor.get_top_copy_src (fdtd_fields::hx),
        grid_accessor.ghost_layer_size_in_bytes(),
        cudaMemcpyDefault);
      cudaMemcpy (
        grid_accessor.get_top_copy_dst (fdtd_fields::hy),
        grid_accessor.get_top_copy_src (fdtd_fields::hy),
        grid_accessor.ghost_layer_size_in_bytes(),
        cudaMemcpyDefault);
      thread_info.sync ();

      update_e_kernel<<<blocks_count, threads_per_block>>> (
        nx, n_own_cells, grid_info.get_row_begin_in_process(), source_position, t, dx,
        C0_p_dt, own_ez, own_dz, own_er, own_hx, own_hy);

      cudaMemcpy (
        grid_accessor.get_bottom_copy_dst (fdtd_fields::ez),
        grid_accessor.get_bottom_copy_src (fdtd_fields::ez),
        grid_accessor.ghost_layer_size_in_bytes (),
        cudaMemcpyDefault);
      thread_info.sync ();

      t += dt;
    }
```

You could notice that I synchronize the threads with a barrier before issuing the next kernel call. The barrier is needed to delay the E field update until memory transaction completion. Otherwise, the kernel would observe a partially updated ghost layer. Threads synchronization after the E field update serves the same purpose.

Everything looks simple so far, but what about performance? If the contact area of grid chunks is smaller than another dimension (for example height of the grid is 8000 cells and the width is about 4000) the code performs at 96% efficiency. This is due to the fact that computation takes the most part of the time. But if we change the ratio to the opposite, we could observe something like 85% of efficiency. It's unacceptable performance. Fortunately, it's possible to hide this latency. Just like before, we need to

extracted transfers of ghost layer fields into a separate method. It calls cudaMemcpyAsync and records an event. The modified main loop is illustrated below.

```
for (int step = 0; step < steps; step++)
  {
    /// Compute bulk
    cudaStreamWaitEvent (compute_stream, e_border_computed, 0);
    update_h_bulk_kernel<<<bulk_blocks_count, threads_per_block, 0, compute_stream>>> (
      nx, grid_info.get_n_own_y (), dx, dy, own_ez, own_mh, own_hx, own_hy);
    cudaEventRecord(h_bulk_computed, compute_stream);

    /// Compute boundaries
    cudaStreamWaitEvent (push_top_stream, e_bulk_computed, 0);
    cudaStreamWaitEvent (push_top_stream, *grid_accessor.get_bottom_done (grid_accessor.
    update_h_border_kernel<<<borders_blocks_count, threads_per_block, 0, push_top_stream
      nx, grid_info.get_n_own_y (), dx, dy, own_ez, own_mh, own_hx, own_hy);
    cudaEventRecord(h_border_computed, push_top_stream);

    grid_accessor.async_send_to                                    after_h, push_top_stream);
    thread_info.sync ();

    /// Compute bulk
    cudaStreamWaitEvent (compute_stream, h_border_computed, 0);
    update_e_bulk_kernel<<<bulk_blocks_count, threads_per_block, 0, compute_stream>>> (
      nx, n_own_cells, grid_info.get_row_begin_in_process(), source_position, t, dx, dy,
        C0_p_dt, own_ez, own_dz, own_er, own_hx, own_hy);
    cudaEventRecord(e_bulk_computed, compute_stream);

    /// Compute boundaries
    cudaStreamWaitEvent (push_bottom_stream, h_bulk_computed, 0);
    cudaStreamWaitEvent (push_bottom_stream, *grid_accessor.get_top_done (grid_accessor.
    update_e_border_kernel<<<borders_blocks_count, threads_per_block, 0, push_bottom_str
      nx, grid_info.get_row_begin_in_process(), source_position, t, dx, dy,
        C0_p_dt, own_ez, own_dz, own_er, own_hx, own_hy);
    cudaEventRecord(e_border_computed, push_bottom_stream);

    grid_accessor.async_send_bottom (fields_to_update_after_e, push_bottom_stream);
    thread_info.sync ();

    t += dt;
  }
```


CUDA JIT Compilation

Cuda 3 min read

Block Sparse Matrix-Vector Multiplication with CUDA

Programming 7 min read

I've increased the priority of tran


```
int least_priority {};
int highest_priority {};
cudaDeviceGetStreamPriorityRange (&least_priority, &highest_priority);

cudaStream_t compute_stream, push_top_stream, push_bottom_stream;
cudaStreamCreateWithPriority (&compute_stream, cudaStreamDefault, least_priority);
cudaStreamCreateWithPriority (&push_top_stream, cudaStreamDefault, highest_priority);
cudaStreamCreateWithPriority (&push_bottom_stream, cudaStreamDefault, highest_priority);
```

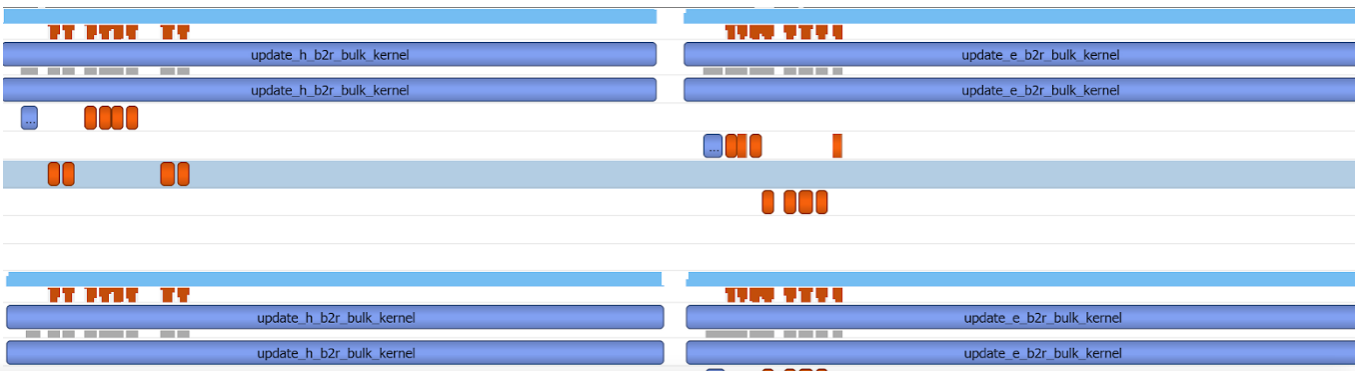
streams_priority.cu hosted with ❤ by **GitHub** **view raw**

It's often better to pay only for something that you use. I used events only to synchronize streams. As you know, events also record time by default. To improve events performance it' possible to disable timing.


```
cudaEvent_t h_bulk_computed, h_border_computed;
cudaEvent_t e_bulk_computed, e_border_computed;

cudaEventCreateWithFlags (&h_bulk_computed, cudaEventDisableTiming);
cudaEventCreateWithFlags (&h_border_computed, cudaEventDisableTiming);

cudaEventCreateWithFlags (&e_bulk_computed, cudaEventDisableTiming);
cudaEventCreateWithFlags (&e_border_computed, cudaEventDisableTiming);
```

disable_timing.cu hosted with ❤ by **GitHub** **view raw**

This version performs much better — around 99% efficiency with two GPUs. You could see from the profiling below that the copying is completely overlapped with computation.



The same approach could be used in multi-node systems. It's possible to send border cells w MPI while computing the rest part of the grid.

Cooperative Groups

There is another approach that I haven't mentioned yet. The Cooperative Groups (CG) programming model describes synchronization patterns both within and across CUDA threa blocks. With CG it's possible to launch a single kernel and synchronize all threads of GPU between different stages. In other words, CG extends __syncthreads to GPU scope. But what important for our subject — it's possible to extend it even further — to the multi-GPU scope.

You can see a CG interface in the kernel below. I rewrote the simulation of Maxwell's equatic so that it could fit into one kernel. After H and E fields update, I synchronize all threads of GPU with the sync method of a grid group. To extend this into a multi-GPU case it would be sufficient to call the sync method of multi grid group. You can get an object of this type with this_multi_grid ().


```
__global__ void fdtd_cg (
    int nx, int ny,
    int steps, int n_cells, int source_position,
    float dt, float initial_t,
    const float dx, const float dy, const float C0_p_dt,
    float * __restrict__ ez, float * __restrict__ dz,
    const float * __restrict__ er, const float * __restrict__ mh,
    float * __restrict__ hx, float * __restrict__ hy)
{
  const int first_cell_id = blockIdx.x * blockDim.x + threadIdx.x;

  cg::grid_group grid = cg::this_grid ();

  float t = initial_t;

  for (int step = 0; step < steps; step++)
    {
      for (int cell_id = first_cell_id; cell_id < n_cells; cell_id += blockDim.x * gridDim.
        {
          // update H
          const float cez = ez[cell_id];
          const int cell_x = cell_id % nx;
          const int cell_y = cell_id / nx;

          const int top_neighbor_id = nx * (cell_y == ny - 1 ? 0 : cell_y + 1) + cell_x;
          const int right_neighbor_id = cell_x == nx - 1 ? cell_y * nx + 0 : cell_id + 1;

          const float cex =  (ez[top_neighbor_id]   - cez) / dy;
          const float cey = -(ez[right_neighbor_id] - cez) / dx;

          hx[cell_id] -= mh[cell_id] * cex;
          hy[cell_id] -= mh[cell_id] * cey;
        }
      grid.sync ();

      for (int cell_id = first_cell_id; cell_id < n_cells; cell_id += blockDim.x * gridDim.
        {
          const int cell_x = cell_id % nx;
          const int cell_y = cell_id / nx;

          // update E
          const float chx = hx[cell_id];
          const float chy = hy[cell_id];
```

```
52                              - (chx - hx[bottom_neighbor_id]) / dy;
53
54            dz[cell_id] += C0_p_dt * chz;
55
56            if (cell_id == source_position)
57            {
58              dz[cell_id] += calculate_source (t, 5E+7);
59              t += dt;
60            }
61
62            ez[cell_id] = dz[cell_id] * er[cell_id];
63          }
64        grid.sync ();
65      }
66  }
```

fdtd_cg_kernel.cu hosted with ❤ by **GitHub** **view raw**

CG requires kernel launch to be changed. The kernel should be launched with cudaLaunchCooperativeKernel in order to use grid-wide barriers. To use multi-GPU barriers you'll need to use cudaLaunchCooperativeKernelMultiDevice instead.

```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, 0 /* dev */);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&num_blocks_per_sm, fdtd_cg, num_threads, 0);
dim3 dim_block (num_threads, 1, 1);
dim3 dim_grid (deviceProp.multiProcessorCount * num_blocks_per_sm, 1, 1);

const int kernel_iterations = write_each >= 0 ? write_each : steps;

void *kernelArgs[] = {
  (void*)&nx, (void*)&ny, (void*)&kernel_iterations, (void*)&n_own_cells,
  (void*)&source_position, (void*)&dt, (void*)&t, (void*)&dx, (void*)&dy,
  (void*)&C0_p_dt, (void*)&own_ez, (void*)&own_dz, (void*)&own_er,
  (void*)&own_mh, (void*)&own_hx, (void*)&own_hy
};

for (int step = 0; step < steps; step++)
  {
    cudaLaunchCooperativeKernel((void*)fdtd_cg, dim_grid, dim_block, kernelArgs);

    t += dt * write_each;
    step += kernel_iterations;
  }
```

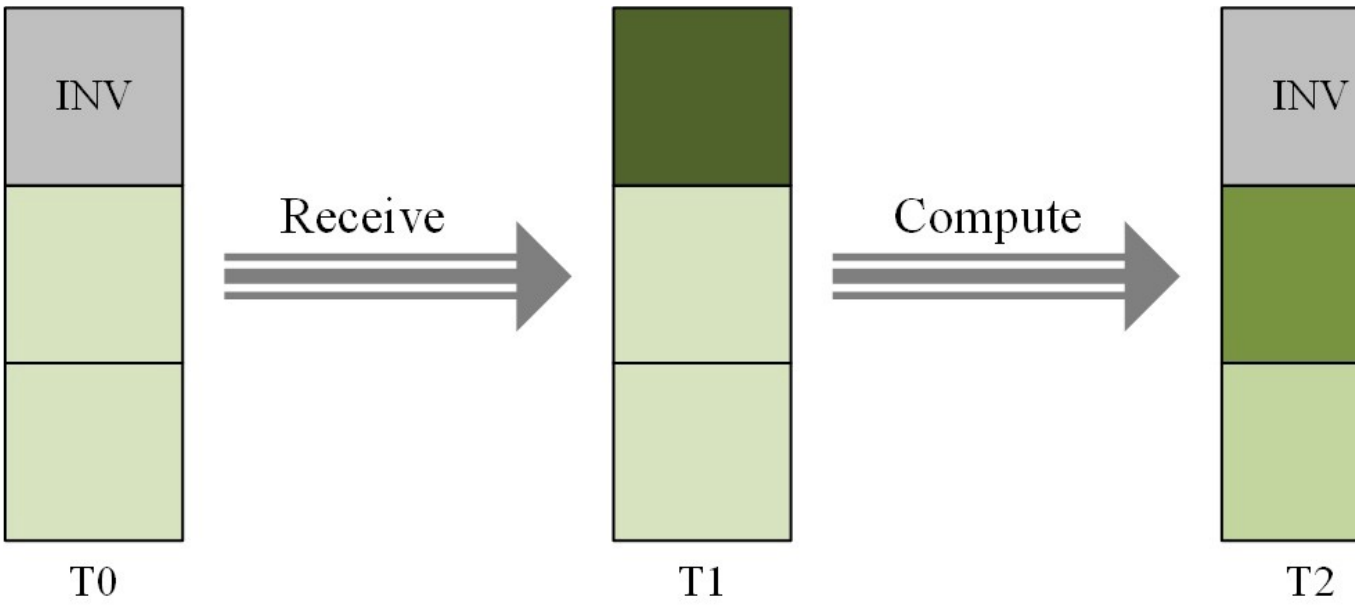
cg_launch.cu hosted with ❤ by **GitHub** **view raw**

After these changes, you'll see a single kernel call in your profiler. Although this technology quite convenient from the programming point of view, you should understand clearly when should be used. First of all — there are a lot of restrictions. If you are fine with them, you should ask the next question — is the code faster with CG? Well, it depends. If you have a lo of small kernels — you could get a performance improvement. In the case of the discussed code, I got around 9% improvement on small grids. I also got a 40% slowdown on big ones. I important to understand that you can't use dynamic parallelism with CG launches. If your small kernels are followed by big ones, it would be difficult to outperform the classical approach. Thereby, I would suggest CUDA Graphs in this case. CUDA Graphs are out of the scope of this post. Instead of them, I'd like to share the last multi-GPU optimization techniqu with you.

Latency bandwidth exchange

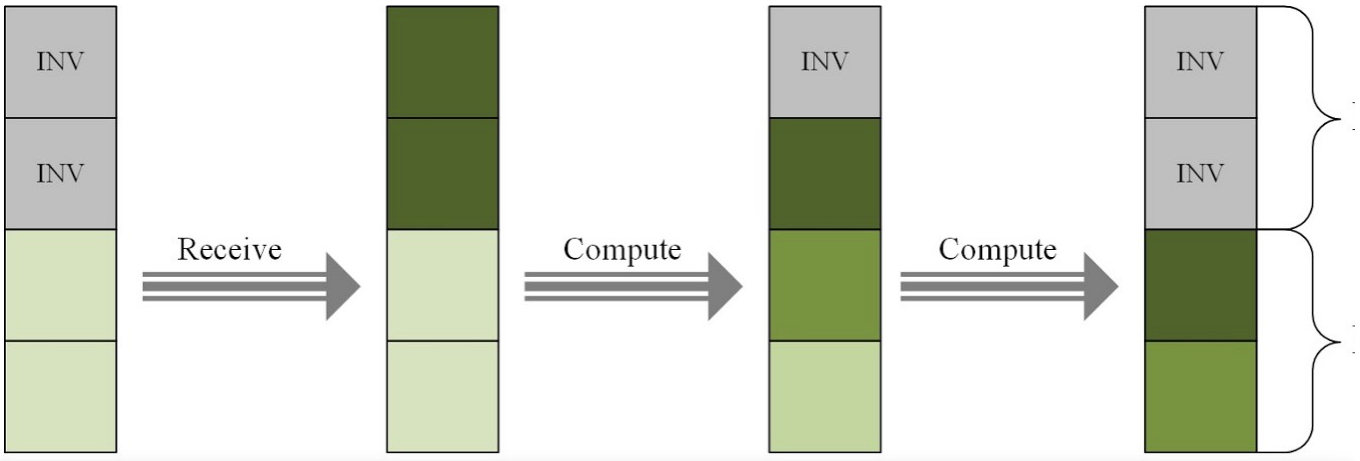
Do you remember that saturation points in the bandwidth plots? We could take advantage o

Let's look at the current state of the code from the data update perspective. In the beginning we have actual values in the own part of the grid. The ghost layer is in an invalid state.



Ghost layer invalidation stages

There is a paper that proposes a way of hiding latency by performing redundant computatio The paper is called "Efficient Simulation of Agent-Based Models on Multi-GPU and Multi-Cc Clusters". The authors achieved over four orders of magnitude speedup on multi-GPU syster The method is called the B+2R latency hiding scheme. It consists in extending the ghost par of the grid chunk from one layer to R layers. By extending the ghost part by R layers we coul postpone transfers to R steps. These R steps could be computed without any synchronizatior with different GPUs.



Conclusion

I'd like to say that multi-GPU programs are a collection of single-GPU programs only in the simplest case of independent kernels. In all other cases, it's a thing on its own. Try multi-GPU programming yourself, and I wish you to achieve 100% efficiency.