Technical University of Denmark

Large scale computations on multiple-GPU architectures

The effects of topology on computational performance

Simon Aertssen (s181603)
Louis Hein (s181573)

July 20, 2020

Contents

1 Abstract

We investigate the influence of computational hardware topology on the throughput performance in terms of double precision floating point operations per second (FLOPS). Two commercially available GPU-accelerated compute nodes are compared using the multiplication of dense matrices as a compute-bound mathematical problem. We find that, the higher host-device memory bandwidths provided by NVLink-enabled CPUs (compared to PCIe connections) significantly improve the overall performance. We also compare our own implementation with an NVIDIA benchmark and report substantial speedups, especially for very large matrices.

The code accompanying this report is available on GitHub.

2 Introduction

Many scientific problems, from training deep neural networks to weather simulations, can be broken down to a multiplication of large dense matrices [1]. In contrast to performing the multiplication on the central processing unit (the CPU), it can be significantly faster to transfer the data to dedicated computational hardware like GPUs, multiply them there and send back the result. When multiple GPUs (*devices*) are connected to the CPU (*host*), we refer to the structure and bandwidth of their connections as topology. In this report, we will investigate the performance of the distributed matrix multiplication of matrices $\mathbf{A} \in \mathbb{R}^{\mathsf{M}\times\mathsf{K}}$ and $\mathbf{B} \in \mathbb{R}^{\mathsf{K}\times\mathsf{N}}$ on two hardware systems with different topologies. Distributed computing is necessary when the memory of a hardware component is too small to hold all the required data to perform the computation, and so it has to be shared in different ways. If we want to make use of the dedicated computational performance of GPUs, which usually have less memory than CPUs, we need to distribute **A** and **B** in some way.

2.1 Matrix multiplications as computational challenge

The transfer of a dense $\mathsf{N} \times \mathsf{N}$ double precision square matrix requires $\mathsf{O}(\mathsf{N}^2)$ operations, while the multiplication of two $\mathsf{N} \times \mathsf{N}$ matrices is bound by $\mathsf{O}(\mathsf{N}^3)$. This type of so called *compute-bound* problem is ideal for investigating the influence of the system's topology on the throughput performance in (*giga*) floating-point operations per second (GFLOPS). Performing matrix operations of order $\mathsf{O}(\mathsf{N}^3)$ requires a heavily optimised routine, like the level-3 basic linear algebra subprograms (L3 BLAS), which includes general matrix multiplication algorithms of different number types (`<type>`GEMM) like single- and double-precision floating-point numbers. Specifically, the GEMM routine includes a matrix multiplication of **A** and **B** and matrix addition with **C**:

$$\mathbf{C}_{\mathsf{mn}} = \alpha \sum_{\mathsf{k}=1}^{\mathsf{K}} \mathbf{A}_{\mathsf{mk}} \mathbf{B}_{\mathsf{kn}} + \beta \mathbf{C}_{\mathsf{mn}} \qquad \mathsf{m} \in \mathsf{M}, \mathsf{n} \in \mathsf{N} \tag{1}$$

The results are gathered in **C** for efficient memory use. For of $\mathbf{A} \cdot \mathbf{B}$, we need $\mathsf{M} \cdot \mathsf{K} \cdot \mathsf{K}$ multiplications and $\mathsf{M} \cdot \mathsf{N} \cdot (\mathsf{K} - 1)$ additions. Multiplication with α requires another $\mathsf{M} \cdot \mathsf{N}$ operations. Adding $\beta \cdot \mathbf{C}$ requires another $\mathsf{M} \cdot \mathsf{N}$ multiplications and $\mathsf{M} \cdot \mathsf{N}$ additions. The total number of operations required for GEMM of real number types is $\mathsf{M} \cdot \mathsf{N} \cdot (2 \cdot \mathsf{K} + 2)$, with which we can now measure the performance of the algorithm.

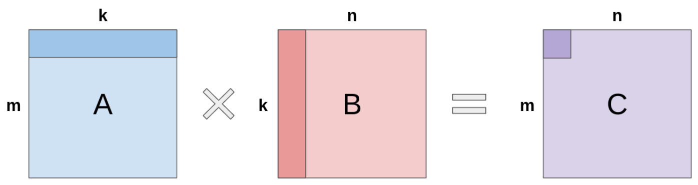


Figure 1: At its core, the matrix-matrix multiplication from equation 1 requires rows of **A** to be multiplied with columns of **B** (both colored darker).

2.2 The CUDA API

The GPUs we will be working with are produced by NVIDIA, and using the CUDA API operating a GPU is heavily simplified. Concurrency on the GPU is obtained by using CUDA streams, the GPU equivalent of a CPU thread. Commands like memory transfers and computations can be assigned to a stream, which will operate in parallel with other streams and the CPU. Commands issued to a stream will be performed in the order of their submission.

There is no upper limit to the number of streams one can create, though it has been documented that more than four streams does not improve performance [2]. A minimum of two streams is required to perform communication and computation in parallel. Between computations, synchronisation may be necessary. We can synchronise all commands on the current device, or on the current stream, or we can use CUDA events, which can pinpoint a specific moment on a stream.
CUDA also provides a wrapper for the GEMM routines, cuBLAS, and we will be timing the performance of the GEMM routine with the highest memory demand: the double-precision floating-point number type, simply *double type*, with cublasDgemm. A double type number has a high precision and consists of 8 bytes (or 64 bits). The main challenge when using double types in distributed computing is that they are *expensive* to transfer, meaning that due to their size the time needed for transfer between systems is high, and this is the true bottleneck we need to improve upon.

3 Hardware

The efficiency of any algorithmic implementation depends on the underlying hardware. This section briefly discusses the arithmetic processing properties most relevant to multi-GPU matrix multiplication.

3.1 Computer memory hierarchy and cache

Computer memory consists of different hierarchical levels to increase the speed at which the memory can be accessed. Higher levels are large but slow to access, while lower levels are small but can usually be accessed within one clock cycle of the CPU. Most of these levels are found in the form of hardware or software cache, which are located closely to the processor cores. The optimal performance of a program can then be obtained by maximizing the use of data that is stored within the same cache. Requesting data that is not in that cache results in a *cache miss* and the cache will be reloaded from a lower level. Since the processors will be idle until the correct data is fetched, cache misses are to be kept low.

The memory hierarchy usually goes from fast to slow, as well as from expensive to more affordable in the following manner:

$$\text{Processor registers} \rightarrow \text{L1-L2-L3-(L4) cache} \rightarrow \text{Main physical memory (RAM)} \rightarrow \text{File system}$$

To efficiently access matrix elements we therefore need to be aware of their *spatial locality* in memory and the pattern by which they are accessed. When programming in C, matrices are stored in a row-major order. This means that when element $\mathbf{A}_{\mathrm{mn}}$ is requested from memory, it is likely that $\mathbf{A}_{\mathrm{m}\,(\mathrm{n}+1)}$ lies in the same cache, and therefore it can be presented more quickly upon a new request. As the element $\mathbf{A}_{(\mathrm{m}+1)\,\mathrm{n}}$ is n bytes away in memory (a full row in this case), its request is likely to result in a cache miss. The study of caches and registers for other usage than data access falls out of the scope of this report. It is worth noting that CPUs usually have fewer registers, but up to 4 levels of cache [3]. Further discussions of memory types will follow in section 5.

3.2 GPU architecture: NVIDIA Tesla V100

All experiments described in this report were conducted using Tesla V100 GPUs by NVIDIA. Every V100 contains 84 streaming multiprocessors (SM) made of four processing blocks as illustrated in Figure 2. Each processing block contains 8 double-precision cores, a 64 KB register file can share 96 KB of memory with the other three blocks to reduce latency [4]. The memory hierarchy of an NVIDIA Tesla V100 GPU is structured as follows [5], [6]:

- 65.536 GPU registers of 64 bits, distributed over 84 Streaming Multiprocessors (SMs): immediately accessible, at the speed of the inner processor cores.
- L1-cache of 32 to 128 KiB, depending on the amount of shared memory. Fast access, with the speed of the memory bus of each processor core.
- L2-cache of 6144 KiB: slower access. This is the maximum size of memory cache stored by a single memory transfer between the host and a device.

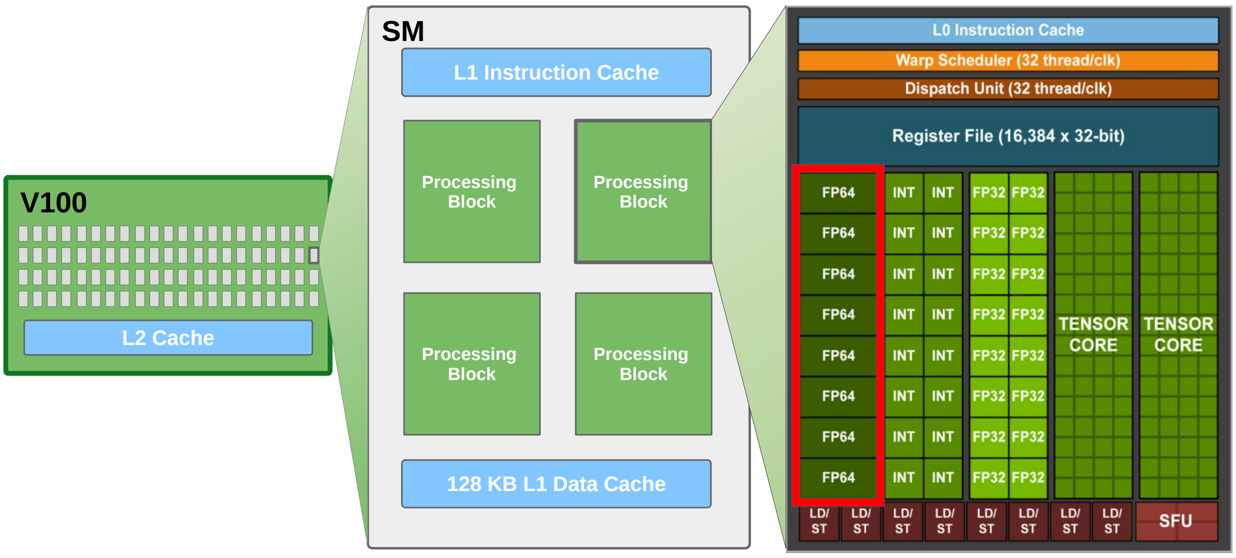


Figure 2: Simplified architecture of the Nvidia V100 GPU. The whole device can access the same L2 cache (left), each of the 84 streaming multiprocessors can access a shared L1 cache (middle). For double-precision arithmetic, only the highlighted 64-bit floating point (FP64) cores are utilized (right).

Since multiplying dense matrices can require large amounts of data to be transferred, the CPU-GPU connection bandwidth also influences the overall program execution time. Transfers to and from a V100 GPU can happen using either a PCIe 3.0 [16 GB/s] or the NVLink [25 GB/s] interconnect. Due to its impact on computational performance, the following section describes the connectivity between the processors in greater detail.

3.3 Multi-GPU topologies

Multiplying two matrices outside of the CPU requires both of them to be fully transferred from host memory to device memory, as well as the result to be transferred back. This makes the bandwidth of the connections [GB/s] an important factor for overall program execution time. When using V100 GPUs, the data transfer can happen using either a PCIe 3.0 [16 GB/s] or the NVLink [25 GB/s] interconnect. Using more than one device therefore requires special attention to how the matrices and computations are distributed across devices to use the available bandwidth efficiently.

This report investigates two commercially available 4-device topologies. They are accessible on the DTU cluster through *SXM2SH* for the DGX-1 and *P9SH* for the Power9. We will use the different names for the topologies interchangeably.

NVIDIA DGX-1
The DGX-1 unit investigated in this report consists of an Intel Xeon Gold 6142 CPU, connected to four V100 GPUs via PCIe bus. The devices themselves are interconnected via single-or double NVLink, allowing for a bandwidth of up to 25 or 50 GB/s respectively. Figure 3 depicts the exact configuration of these connections and Table 1 shows both the linking scheme and measured transfer rates on the node. It is immediately visible that the bandwidth bottlenecks in this configuration are the comparably slow host-device PCIe connections, all measuring less than 13 GB/s.

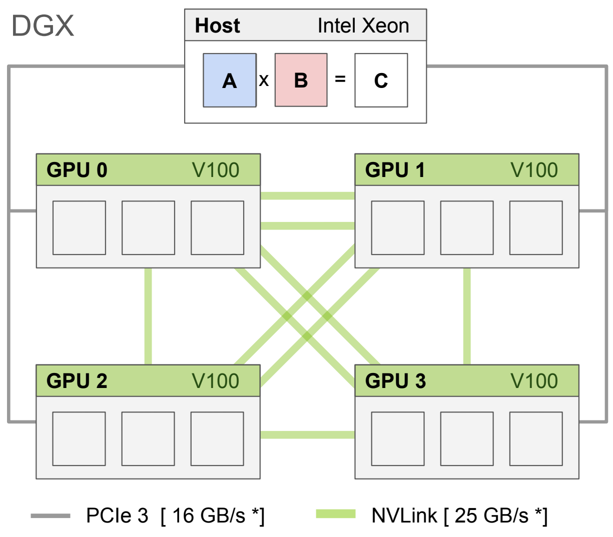


Figure 3: DGX-1 topology showing 4 interconnected V100 GPUs and a third-generation PCI express bus from host to all devices. The three squares in each device indicate allocated memory for array structures containing matrix parts.
* For exact values refer to measurements in Table 1.

	GPU0	GPU1	GPU2	GPU3	CPU
GPU0	X	NV1	NV1	NV2	SYS
GPU1	NV1	X	NV2	NV1	SYS
GPU2	NV1	NV2	X	NV2	SYS
GPU3	NV2	NV1	NV2	X	SYS
CPU	SYS	SYS	SYS	SYS	X

	GPU0	GPU1	GPU2	GPU3	host
GPU0	X	48.22	24.11	48.23	12.77
GPU1	45.29	X	44.60	23.81	12.48
GPU2	24.11	48.23	X	23.76	12.78
GPU3	48.23	24.11	23.07	X	12.85
CPU	12.77	12.48	12.78	12.85	X

Table 1: Direct comparison of the topology and measured transfer speed of DGX-1.
left: DGX-1 topology, taken from the command `nvidia-smi topo -m`. NV# = # of NVLinks between nodes in the network. SYS = PCIe bus.
right: measured connection speed in GiB/s using 12.0 GiB of data. These results are obtained using pinned memory on the host, and reflect the topology exactly.

IBM Power9

The second topology investigated addresses the described host-device bandwidth bottleneck by using an AMD POWER9 CPU that allows either triple NVLink connections or a fast PCIe 4 - based system interconnect (SYS) to the devices. Each of the four V100s is connected to one other device via triple NVLink, and to the remaining two others via SYS as detailed in Table 2 and visualized in Figure 4.

	GPU0	GPU1	GPU2	GPU3	mlx5_0	mlx5_2		GPU0	GPU1	GPU2	GPU3	host
GPU0	X	NV3	SYS	SYS	NODE	SYS		X	72.23	33.43	33.46	35.49
GPU1	NV3	X	SYS	SYS	NODE	SYS		72.22	X	32.68	31.99	33.92
GPU2	SYS	SYS	X	NV3	SYS	NODE		30.74	33.10	X	72.23	72.14
GPU3	SYS	SYS	NV3	X	SYS	NODE		33.20	33.34	72.20	X	72.14
mlx5_0	NODE	NODE	SYS	SYS	X	SYS		35.49	33.92	72.14	72.14	X
mlx5_2	SYS	SYS	NODE	NODE	SYS	X						

Table 2: Direct comparison of the topology and measured transfer speed on the P9 node.
left: P9 topology, taken from running `nvidia-smi topo -m`. NV# = # of NVLinks between nodes in the network. SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI). NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node.
right: measured connection speed in GiB/s using 12.0 GiB of data. These results are obtained using pinned memory on the host, and reflect the topology exactly.

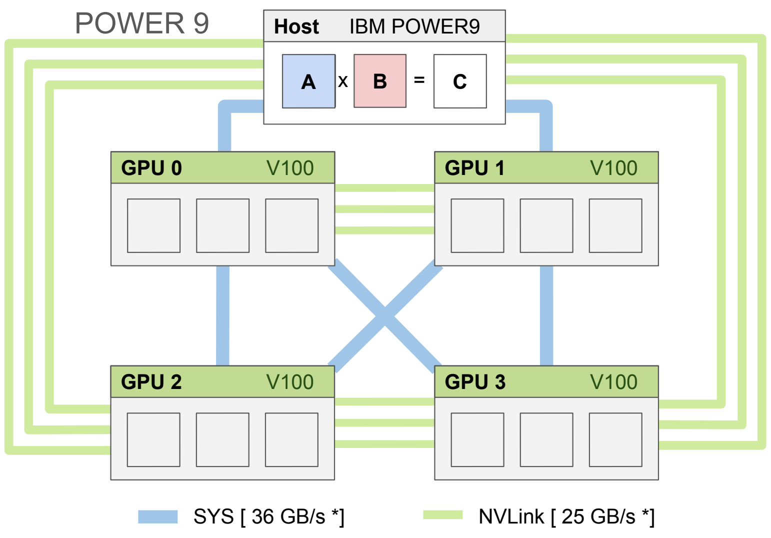


Figure 4: POWER9 topology with 4 V100 GPUs. 2 of them are connected to the CPU via NVLink, the other two use the system innterconnect involving a PCIe 4 bus.
* For exact values refer to measurements in Table 2.

4 Distributed DGEMM algorithms and related work

Due to the memory access pattern (see section 2), loading a (whole) row of a matrix in memory is faster, but a matrix multiplication also requires a (whole) column, which is much slower. In such cases, subdividing the matrix into square tiles can be a good trade-off between rows and columns. In what follows, all algorithms use a subdivision of the matrices **A**, **B** and **C** into square tiles of size $\mathsf{T} \times \mathsf{T}$. It is clear that T is a whole number and we will always take $\mathsf{T}\%\mathsf{N} = \mathsf{T}\%\mathsf{M} = 0$. A matrix which is subdivided into 25 tiles then has five rows and five columns of tiles, like in Figure 5. We can now also redefine (1) in terms of tiles:

$$\mathbf{C}_{\mathsf{mn}} = \alpha \sum_{\mathsf{k}=1}^{\mathsf{K}/\mathsf{T}} \mathbf{A}_{\mathsf{mk}} \mathbf{B}_{\mathsf{kn}} + \beta \mathbf{C}_{\mathsf{mn}} \qquad \mathsf{m} \in \mathsf{M}/\mathsf{T}, \mathsf{n} \in \mathsf{N}/\mathsf{T} \tag{2}$$

One could argue that for the multiplication of two matrices we would not need the addition of **C** in (2). However, as we are distributing **A** and **B** across devices through tiles, the computation of different parts of **C** will occur in different steps. **C** can then work as an accumulator, and we use `memset` to clear the tile. The results can also be accumulated on the CPU, in which case synchronisation is required to avoid race conditions, though the PCIe will be very crowded, leading to inefficiencies. The problem is that even though we can parallelise moving the contents from the transfer buffer to the host memory, the whole buffer acts as an accumulator, as the elements only accumulate the results of a set number of K/T tile operations. A reduction is therefore not evident.

4.1 Static job scheduling policies

The most straightforward method to distribute the computational load is to assign operations on tiles of **C** to the different devices in the topology. Tile $\mathbf{C}_{\mathsf{mn}}$ is then computed from all tiles in the m-th row of **A** and the n-th column of **B**. In that way, reading and sending tiles of **A** and **B** to each device is independent of the device, and writing the resulting **C** tile is free of race conditions. This operation is referred to as a task: a memory transfer from host to the device (D2H), a computation, and a reverse transfer (H2D).
Other static job schedules reuse tiles that are already on the device and only change a tile of **A** or **B** to limit the use of the PCIe over time. This requires serious synchronization across the devices, as the current accumulating **C** tile has to be transferred between devices. In Figure 5, one can see how such jobs might be scheduled for four devices. The advantage of this simple method is that tiles of **A** can be sent from the host to a device, and then broadcast to the other devices, as peer to peer communication is generally much faster.

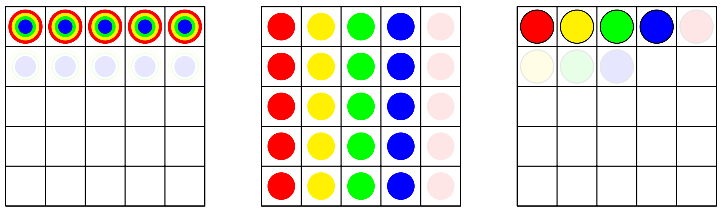


Figure 5: Example of the direct matrix multiplication algorithm of matrices **A**, **B** and **C** using 25 tiles and four devices. With the use of color, we can distinguish what tiles are used by what device: • Device 0, • Device 1, • Device 2 and • Device 3

4.2 Multi-GPU BLAS libraries

There exist a few powerful platforms to perform distributed computation of large dense matrices on NVIDIA hardware. These APIs are constructed in a similar fashion: every device in the available architecture is asked to compute certain tiles of **C**, and the device is given a set of commands to get the right tiles of **A** and **B**. The devices are managed through CPU threads.

4.2.1 cuBLASXt

With cuBLASXt, NVIDIA made an in-house platform for distributed computation of large matrices using its signature hardware. It uses a static job scheduling policy as described above, where each device is managed through a CPU thread. Little is known about the inner workings of cuBLASXt, though we will be using the API as a benchmark. According to the documentation, there is no restriction on matrix sizes, as long as the host memory is not exceeded.

4.2.2 BLASX

In 2016, a multi-GPU level-3 BLAS library was released under the name of BLASX [2]. BLASX mainly stood out from its competitors due to a dynamic job scheduler and an ingenious use of device memory. The algorithms adjust the distribution of tasks at runtime, so that also heterogeneous network topologies can perform at maximum efficiency.
Two systems were put in place to minimize communication between host and devices. Building on the cache hierarchy (see section 3), two more levels of cache are constructed on the devices. A L1-tile cache is implemented on each device using the GPU onboard RAM, on which a cache hit enables the tile to be reused immediately. An Approximate Least Recent Used (ALRU) algorithm is used to load and discard tiles in the L1-tile cache of each device, using the information of the current tiles on the device. An L2-tile cache is implemented by using the combined memory of GPUs that share the same communication channel with the host, so that a L2-tile cache hit reduces the CPU-GPU communication to GPU-GPU communication from the adjacent GPU. This is much faster because of NVLink and due to low latency of the GPU RAM. The L2-tile cache is implemented using a custom MESI protocol, where the ALRU's of the adjacent devices decide on the states as follows:

- M (modified): a device is writing a intermediate result of tile $\mathbf{C}_{mn}$. It should be noted that intermediate results get copied back to host immediately, after which the state is set to I.
- E (exclusive): the tile is used by one device.
- S (shared): the tile is used by multiple devices.
- I (invalid): the tile is not in use, discard.

Before computation starts, all tasks are constructed using the necessary metadata: indices m and n dimensions and host address of $\mathbf{C}_{mn}$. Tasks are then sent to the devices through their dedicated CPU thread, and using the new caches, tasks with better temporal locality are prioritized on the same device. This priority happens at runtime and uses the ALRUs to check potential cache misses. A small fraction of the tasks is also sent to the host. As BLASX uses four streams per device, four prioritized tasks are sent to a device at once. A special buffer, the Reservation Station (RS) also stores the next four prioritized tasks for continuity, and devices are allowed to exchange tasks in their RS if the priority matches. In that way, the computational load is adjusted according to the real time demand of the individual processors.

4.2.3 XKBLAS

Very recently, a paper describing the multi-GPU library XKBLAS was released [7]. XKBLAS works in a similar fashion as BLASX, where the CPU queue tasks for the devices through dedicated threads. A number of tasks is executed in accordance with the predicted level of concurrency, through a data flow graph. When tasks are finished, they activate successor tasks that use the same matrix tile. As there are dependancies between tasks, it seems like XKBLAS uses a schedule in which tiles of **C** have to be shared between devices, like described before.
In contrast to BLASX, different streams are used for the D2H and H2D transfer and the computation, these are shared among the active threads. This seems to yield promising results, though synchronisation among streams is not discussed. Concerning the GPU software cache, read-only data is prioritized for removal from the cache, and an L1-tile cache miss automatically sources from the L2-cache of a peer device with the fastest connection.

4.3 Our method, DIEKUHDA

We have gathered our work in the DIEKUHDA library ("*d-cuda*"), which bears its name as a tribute to the double type matrix multiplication using CUDA. DIEKUHDA is built in a lighweight format that is readable and understandeable for users with experience in `c`. Functions are parallelised with OpenMP, instead of `p_threads`, so that all parallel instructions are right there for the user to read. Matrix (and tile) data is stored in the custom `matrix` struct, as the information required to perform a task is received at runtime, at the moment **C** is split into tiles. Most of the functionality of DIEKUHDA consists of function wrappers, like safely (de-)allocating host and device memory and memory transfers.

4.3.1 Under the hood

DIEKUHDA uses a static job scheduler as described in Figure 5. Before beginning the multiplication, the devices are briefly warmed up by a cuBLAS handle allocation. The available memory on each of the devices is then gauged, and the tile dimension is appropriately chosen as to allocate as much device memory as possible. As each task is executed by a single CUDA stream, we need a minimum of three tiles per stream on a device. A priority is given to less streams and larger tiles per stream, as this should minimize bookkeeping overhead. For each stream, two pinned memory buffers are created on the host, to increase transfer speed. For very large applications, it is possible to decrease this number to one, or to use even smaller buffers.
Instructions to all streams on all devices are given in parallel, using the indices of **C**. For every task on a stream the **C** tile is cleared, the appropriate row and column of **A** and **B** respectively are loaded tile by tile and the results are added in the **C** tile of the device, using the addition in the DGEMM routine. The result is then sent back to the host. For the memory transfer the function `cudaMemcpy2DAsync` is used, which showed superior performance during timing experiments. For a H2D transfer, the pinned buffer of the said stream is filled using 16 `omp` threads, then a transfer is initiated through a CUDA stream. For the reverse D2H, synchronization is necessary to ensure that the buffer is filled. Synchronisation is also performed right before the DGEMM routine, to ensure correct results, and after the task loops to ensure correct performance timing. The results are then tested, as every element in the result is unique. A cleanup loop frees all dependencies, and resetting the device is a good practice. [8]. Using four streams per device, and T = 8192, one needs to allocate $3\ \frac{\text{tiles}}{\text{stream}} \cdot 4\ \frac{\text{streams}}{\text{device}} \cdot 8\ \frac{\text{bytes}}{\text{double}} \cdot 8192^2$ doubles = 6.14 GiB of data. This grows exponentially with the tile dimension: using T = 16384 requires 24.58 GiB per device.
Lastly, performance was optimised by using two separate parallel `omp` regions as then the required number of threads could be tuned. Usually, one would only start a single parallel region and use different `omp` for-loops.
The code can be found here. The script name is different due to internal bookkeeping.

4.3.2 D2D implementation

The aim of this project was the development of an algorithm capable of including device-to-device (D2D) transfers, to minimize communication over the comparably slow PCIe bus. Using the scheme in Figure 5, we can see how tiles of **A** can be efficiently shared among (streams of) devices, upon which only tiles of **B** will have to be loaded from the host memory. Tiles of **A** can be shared between devices if the next tile (using another column of **B**) is on the same row of **A**.
Using the DGX-1 topology from Figure 3, one could use the PCIe to send tiles of A to devices 0 and 1 (which should not influence each other) and then have device 0 send over its tile of **A** to device 3, and have device 1 send to device 2. This scheme uses the fastest connections in the network. Synchronisation is required to communicate when a D2D transfer is finished. Using `cudaEventRecord`, we can make stream 0 on device 0 record the event that

the tile of **A** has been transferred. Using cudaStreamWaitEvent, we can make stream 0 on device 3 wait for this event to occur. However, there is a problem with this implementation: when issuing these commands with omp, device 3 will reach the request for synchronisation before the event has occured, because device 0 is still performing the transfer. As evens are created in an *unset* state, recording the event always has to come first. Using different flags did not help, nor did cudaEventSynchronize.

To make the D2D implementation work, the algorithm has to be split in three parallel parts: one where the D2D transfer command is issued and the event of that transfer is recorded, one where the other devices wait for the event to occur, and one where all devices send their results back to the host - as this can only happen once all operations are executed. This scheme requires the parallel loops over the devices and streams to be reversed. To keep the elegance of the implementation, a permutation of the counter is performed inside every loop, so that we can map the counter to the desired device and perform permutations according to the network topology.
Using the D2D implementation, we only expect a large improvement in performance on the SXM2SH, where D2D transfers between certain devices are up to four times faster. The best D2D configuration on the P9SH can only improve on one connection (host $\longleftrightarrow$ device 0 $\longleftrightarrow$ device 1) by about 1/3; devices 2 and 3 already use the fastest connections. Although benchmark measurements still need to be conducted in the future, the implementation should increase overall performance on both topologies.

4.3.3 Limitations

DIEKUHDA currently only supports matrices with even dimensions ($\mathrm{M}\%2 = \mathrm{N}\%2 = 0$), as for $\mathrm{M}\%2 = 1$ we can pad the matrix with a column and a row of zeros, which should not affect performance noticeably. The tile dimension also has to be even, and the number of tiles has to be even. The scheduler assumes all GPUs are identical and performance is maximized when the number of tiles is also a multiple of the number of devices. The current D2D implementation only works when using one tile per device, more investigation is needed in the future.

5 Results

Here, we will perform a comparison between the performance of DGEMM of the different algorithms and libraries. Due to the difficulties in linking the libraries to the OpenBlas and CUDA installations, we were unable to install BLASX and XKBLAS on the DTU cluster. The difference in performance between SXM2SH and P9 is therefore the main subject of this chapter, benchmarked on cuBLASXt. We expect these differences to stem from the following:

- E1: as the lowest D2H transfer speed in the P9 topology is three times as fast as on the DGX-1 (Table 2), we expect much better performance.
- E2: as the DGX-1 CPU has twice as much memory as the one on the P9, we expect much larger tiles to be present on the devices of the former.

As SXM2SH is an interactive node on the DTU cluster, other users can use the devices at any time. For this reason, three repetitions per experiment were recorded, which is a tradeoff between accuracy and computation time (with possible interference from other users). As private access to the node was requested but not granted, benchmarking proved difficult. For fair comparison, experiments on the P9SH are recorded in the same fashion. In the following plots, the mean $\bar{x}$ and the 90% confidence intervals are communicated, using $\bar{x} \pm t_{(0.05,2)}\frac{\sigma}{\sqrt{3}}$. A red colour will be used for results on the SXM2SH, and results on the P9SH will be in blue.

5.1 cuBLASXt

5.1.1 Finetuning the optimal tile dimension

In Figures 6 and 7, we can see how the performance evolves when using different tile dimensions. As it is unclear how cuBLASXt uses memory on the host, measuring the optimal tile dimension for use on large matrices proved to be very difficult. For large tile dimensions the program execution terminates with an error due to insufficient memory.

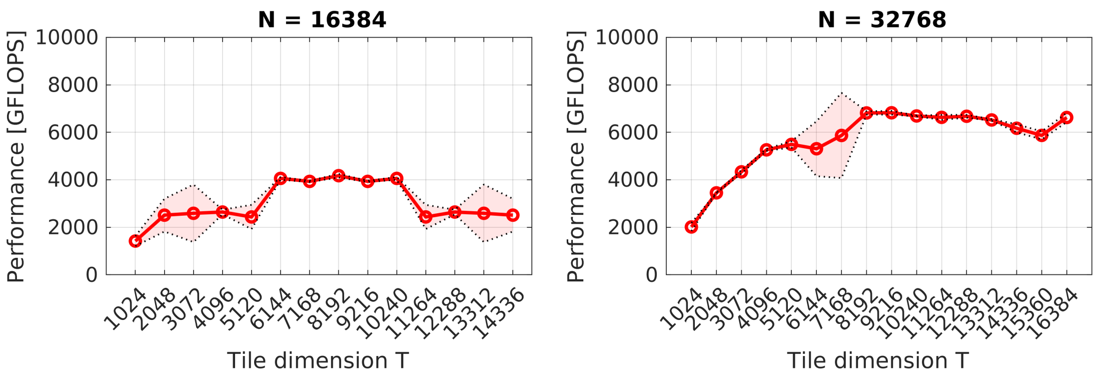


Figure 6: Finding the optimal tile dimension for cuBLASXt on the SXM2SH using different matrix sizes.

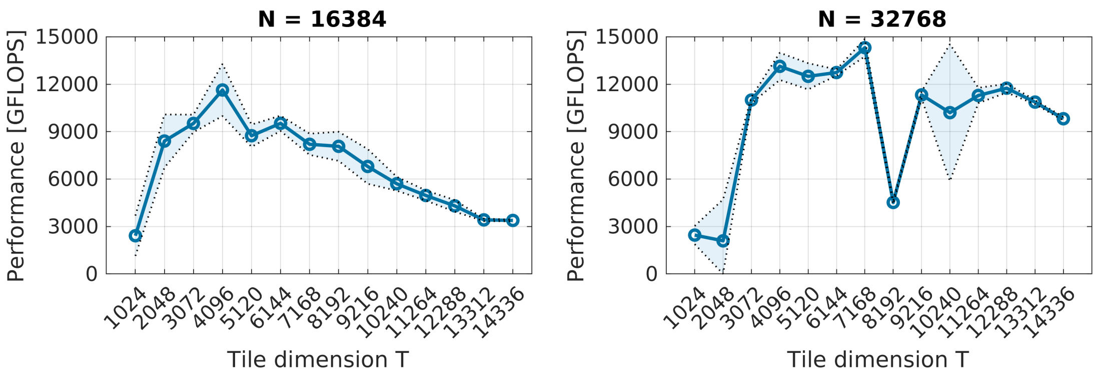


Figure 7: Finding the optimal tile dimension for cuBLASXt on the P9SH using different matrix sizes.

Given the above data, we chose to limit the maximum tile dimension to 8192 for SXM2SH and 4096 on the P9SH, in accordance with E2. This is not perfect, as it is not the memory on the device that is doubled, only the tile dimension, but it is a heuristic to prevent the algorithm from crashing.

5.1.2 Performance

We now compare the performance of the cuBLASXt routine on both topologies. In accordance with E1, the performance on the P9SH is higher, with the exception of very large matrices ($N = 2^{16} = 65536$), where performance drops significantly. It is worth noting, that three dense matrices of this size stored in double precision together require roughly 100 GB of memory. We therefore chose it as an upper limit for our experiments, to not run out of memory on the host.

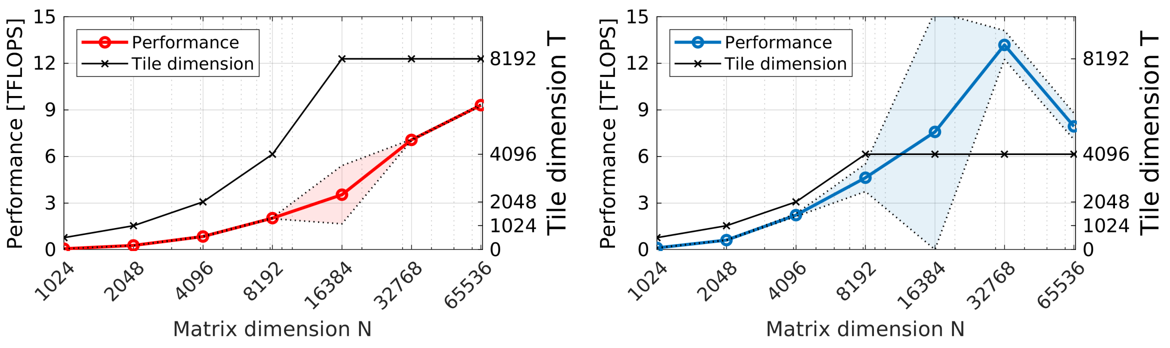


Figure 8: Results of using cuBLASXt on SXM2SH (left) and P9 (right). The tile dimension is shown in black. The large variance in the P9SH performance at N = 16384 is caused by a potential outlier at around 2 TFLOPS - the other two measurements resulted in roughly 10 TFLOPS.

Overall, cuBLASXt reaches higher performance on the Power9 for smaller matrices. Since peak performance on the SXM2SH continues to increase throughout the experiment, it performs better for $N = 65536$.

5.2 DIEKUHDA

5.2.1 Finetuning the optimal tile dimension

As with the baseline library, we compare the performance for two selected matrix sizes across a range of tile dimensions. The figures 9 and 10 indicate that, in contrast to cuBLASXt, the performance increases monotonically with larger tiles.

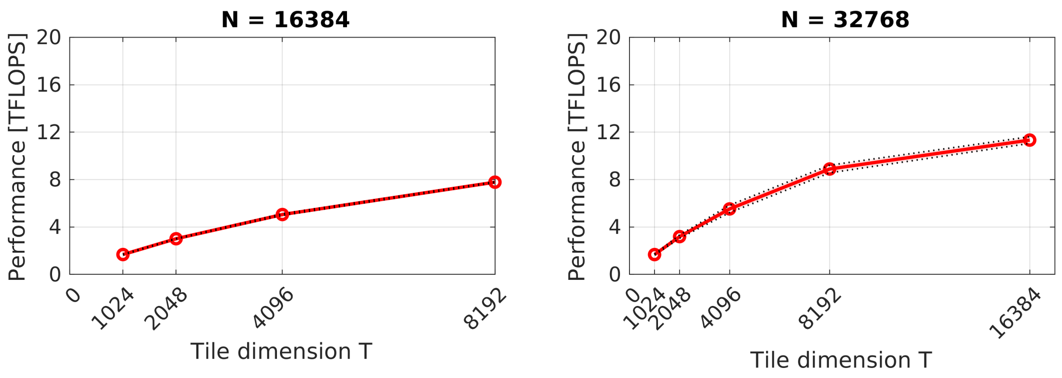


Figure 9: Finding the optimal tile dimension for DIEKUHDA on the SXM2SH using different matrix sizes.

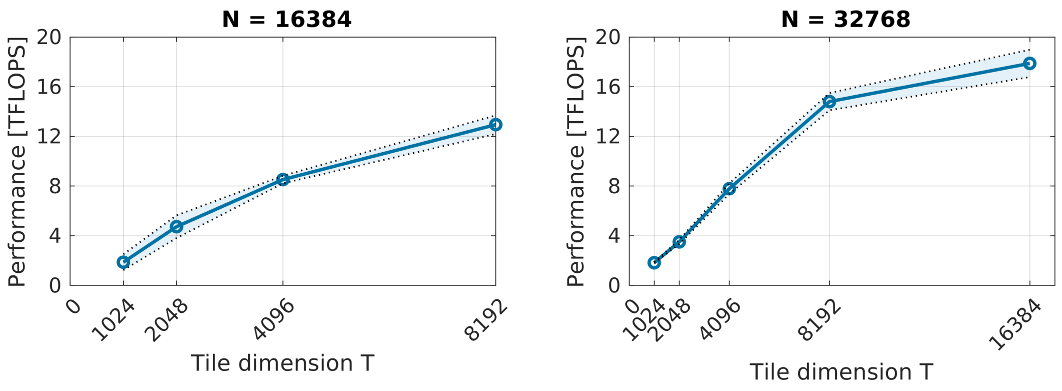


Figure 10: Finding the optimal tile dimension for DIEKUHDA on the P9SH using different matrix sizes.

We can both see a much lower variance on each measurement, as well as more alike looking performance profiles of the topologies. Furthermore, the results indicate that optimal performance for both chosen matrix sizes is reached when using quarter tiles, regardless of the topology.

5.2.2 Finetuning the optimal stream count

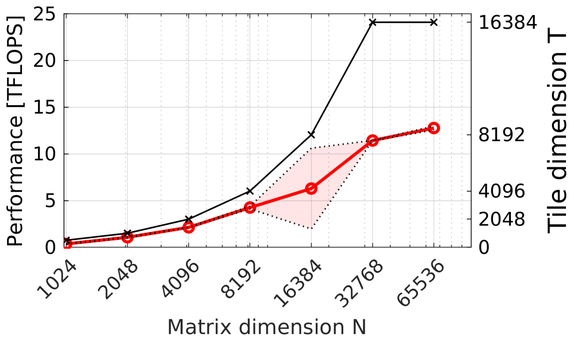
During testing, it was hypothesised that using less CUDA streams per device would improve the performance as well. Less streams use less memory and larger tiles can be used per stream, which was shown to be beneficial. In Table 3 we can see how on SXM2SH there is virtually no improvement in performance, as according to E2 there is a lot of memory available; we will be using four CUDA streams to maximise parallel work here. For P9SH the performance increases by about 60% when using two streams.

N	2 streams	4 streams
16384	7.790	7.766
32768	11.443	11.428
65536	12.996	12.756

N	2 streams	4 streams
16384	12.764	12.975
32768	18.035	14.019
65536	22.158	14.625

Table 3: Mean performance in GFLOPS for different CUDA streams for SXM2SH (left) and P9SH (right).

5.2.3 Performance



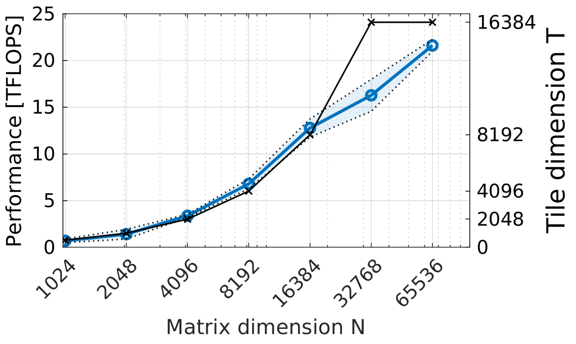
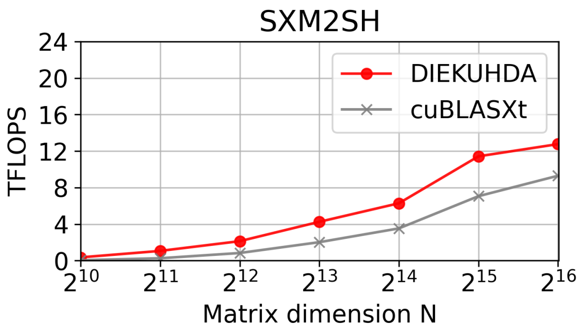


Figure 11: Results of using DIEKUHDA on SXM2SH (left) and P9 (right). The tile dimension is shown in black and only depends on the matrix dimension, not on the topology. DIEKUHDA uses a tilesize of $T = \frac{1}{2}N$ With the exception of $N = 2^{16}$, where $T = \frac{1}{4}N$, due to memory limitations on the devices.

The resulting curves have a similar shape as the ones produced by cuBLASXt and, at least on the DGX-1, also show large variance for $N = 16384$. Figure 12 shows the performance of both multiplication methods on both topologies in direct comparison.



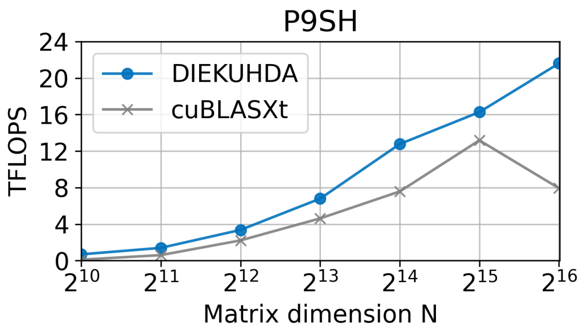


Figure 12: Mean performance values of both discussed multiplication routines in comparison.

On average, DIEKUHDA consistently performs better than cuBLASXt on both topologies. The most significant difference for large matrices is seen on the Power9 topology, where DIEKUHDA brings a 2.7x performance increase. In addition, compared to the DGX-1 results, the Power9 performance curve shows no sings of flattening out.

5.2.4 Lessons from the profiler

Using the NVIDIA visual profiler, one can investigate the (lack of) concurrency in an executable using NVIDIA GPUs. Let us now look under the hood of cuBLASXt and our work, on device 0 of SXM2SH, using M = N = 32768 and T = 1024. The first thing that is clear is that cuBLASXt is also using four streams per device, and that it allocates all pinned memory buffers in sequence. This procedure is slow, and therefore it was parallelised in DIEKUHDA. The concurrency in communication and computation is very clear, as we can see in Figure 13. As we can see, communication only occurs through the PCIe. When computations are done, all streams are synchronised, which lasts quite a while, and a brief deallocation is spotted, also not in parallel.

Using DIEKUHDA, all devices have access to the same resources and these are all created on the devices in parallel. A signature set of `cudaFree` commands is issued during cuBLAS handle creation. As we can see in Figure 14, the D2H transfers do not run in parallel across streams of the same device.

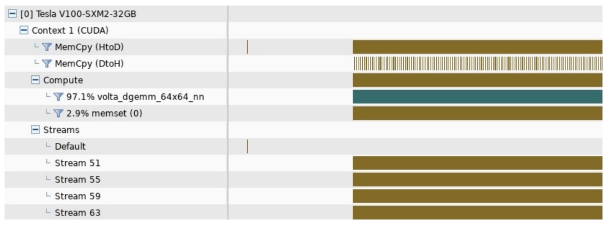


Figure 13: Profiler results of cuBLASXt.

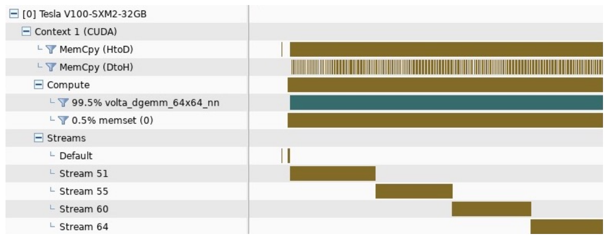


Figure 14: Profiler results of DIEKUHDA.

This might be due to our use of `omp`: all commands are issued to all streams in parallel, so stream 51 continues to receive commands and will continue to perform them in order of their submission. Stream 55 can only start when all commands on stream 51 have been executed, and we can see how the four streams perform about a quarter of the work. It was not possible to solve this artefact (to run all commands perfectly in parallel), and we suspect our use of `omp` is the only culprit. However, the performance of DIEKUHDA was already shown to be better in the last section. Using DIEKUHDA, the devices spends more time computing: 99.5% versus 97.1%, which is better. The fact that the device streams do not have the exact same numbers might also be due to `omp`, and that is not significant. The reason why P9 performs better with two streams might also be due to this artefact.

6 Discussion

This report compared the performance of matrix multiplication routines on two differently hosted and interconnected compute-topologies. We benchmarked both topologies with the cuBLASXt implementation by NVIDIA and developed our own routine, DIEKUHDA, to compare the results with. The experiments show, that our implementation utilizes the compute resources more efficiently and thereby produces better results. The overall peak performances are 12.8 and 21.6 TFLOPS using DIEKUHDA on the SXM2SH and the Power9 respectively, making the latter almost twice as performant in the given usecase.

Unfortunately, cuBLASXt provides very little information about the workings under the hood and the other two interesting implementations discussed in section 4.2 require prohibitively complicated setups. With DIEKUHDA we hope to contribute an easy-to-use, yet performant alternative for the multiplication of large double-precision matrices.

6.1 Network connections on the DTU cluster nodes

Our results show that it is better to have a few carefully selected links with very high bandwidth, than to link every device in the network with a poor connection. In general, connections with the host need the best performance, and if only a PCIe exists, connections between individual GPUs have to be concentrated between devices using the same PCIe to prevent overloading. Commands usually sent to two devices over the PCIe can also occur in parallel when using one device on the PCIe and another through NVLink.
For the SXM2SH node, we suggest not connecting all devices, but to connect devices 0 and 1 and devices 2 and 3 with up to 6 NVLinks for each connection. For the P9SH node, devices 2 and 3 are already well connected to each other and to the host. Here, we suggest NVLink connections between the host, devices 0 and 1, and connections between the host, devices 2 and 3. This is basically the same topology as suggested for SXM2SH. These suggestions only come from the point of this report and exclude the need for connections for other tasks. These suggestions would also improve the performance of BLASX, as the L2-cache is now much faster.

6.2 Future work

Throughout the development of the DIEKUHDA library, many promising ideas for improvement have come up that stayed outside the scope of this project. This last section is intended as an inspiration for future projects building on top of the presented work.
First of all, we suggest a re-benchmarking of both the algorithms described in this report, as well as the D2D implementation discussed in section 4.3.2, on exclusively reserved cluster nodes for more accurate results. For

further performance gains, the number of required streams should be adjusted according to the available memory, as sometimes two streams need so much memory that the tile size is halved, while using three streams could yield better performance. Another suggestion is to allocate all available memory on the devices, and use tiles that are as large as possible. When a cleanup loop is required, the remaining rows and columns can be gathered in a tile format as well. This requires some good bookkeeping. Finally, when creating a dynamic task scheduler, the average tile size should be reduced, as there is little use of the L1- and L2-tile cache if only one or two tiles fit on the device.

References

[1] P. Warden, *Why GEMM is at the heart of deep learning.* April, 2015. https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/.

[2] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, *BLASX: A High Performance Level-3 BLAS Library for Heterogeneous Multi-GPU Computing* in *Proceedings of the 2016 International Conference on Supercomputing*, p. 20, ACM. 2016.

[3] I. Corporation, *Enterprise, Scale-out and Accelerated Servers with POWER9 Processor Technology* 2018. https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.

[4] *NVidia Tesla V100 GPU Architecture* August, 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[5] NVIDIA, *NVIDIA Tesla V100 GPU architecture. The world's most advanced data center GPU.* 2017.

[6] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking* 2018.

[7] T. Gautier and J. V. F. Lima, *XKBlas: a High Performance Implementation of BLAS-3 Kernels on Multi-GPU Server* in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 1–8. 2020.

[8] *CUDA C++ Best Practice Guide* June, 2020. https://docs.nvidia.com/pdf/CUDA_C_Best_Practices_Guide.pdf.