NVIDIA Hopper H100 GPU: Scaling Performance

Jack Choquette, *NVIDIA, Santa Clara, CA, USA*

The H100 Tensor Core GPU is NVIDIA's latest flagship GPU. It has been designed to provide industry leading performance for high-performance computing, artificial intelligence, and data analytics datacenter workloads. Notable new features include a fourth-generation Tensor Core, new Tensor Memory Accelerator unit, a new CUDA cluster capability, and HBM3 dynamic random-access memory.

H100 is NVIDIA's next-generation GPU based on the new Hopper architecture.[1] It is built on TSMC's 4 N custom 4-nm process and contains over 80 billion transistors.

H100 includes a new memory system that features the world's first implementation of HBM3 and a large 50-MB L2 cache. H100 features five HBM3 sites with a total memory capacity of 80 GB and over 3 TB/s of memory bandwidth, twice the throughput of our previous generation A100 GPU.[2]

To support Multi-GPU SuperPOD and Cloud designs, H100 introduces a variety of new system architecture features. It includes the fourth generation of NVLink with 900 GB/s of total bandwidth. H100 includes NVIDIA's second-generation multi-instance GPU technology, as well as new accelerated confidential computing support for secure accelerated computing.

The H100 GPU used in the SXM board form-factor contains 132 streaming multiprocessors (SMs), each delivering twice the performance per clock over A100's SMs and introduces multiple new features. H100 features a 2× clock-for-clock improvement in traditional FP32 and FP64 throughput. It supports 256 KB of unified L1 data cache and shared memory storage, which is 33% more than A100. It contains the new fourth-generation Tensor Core that is 2× faster and more efficient. The SM in the Hopper architecture introduces a new dedicated instruction set for dynamic programming called DPX, which provides advanced operand fusion for the inner loop of many dynamic programming algorithms. The Hopper architecture adds a tensor memory accelerator (TMA) for efficient asynchronous movement of multidimensional tensor data.

The Hopper architecture also introduces a new level of hierarchy between the CUDA hierarchy of thread blocks and grids called thread block clusters. Thread block clusters enable applications to take advantage of locality to dramatically improve efficiency.

Figure 1 shows the performance for key mainstream high-performance computing (HPC) and artificial intelligence (AI) models, including many vision neural networks or smaller language models. H100 delivers two to 30 times the performance of NVIDIA's previous generation A100 when both use InfiniBand. On top of that, H100 introduces a new NVLink network interconnect. The improved interconnect enables more efficient scaling across multiple GPU-accelerated nodes and provides another major boost of performance across HPC and AI applications. With the new NVLink, H100 provides up to an additional 2× to 3× performance improvement for HPC and AI training.

ACCELERATING PRINCIPLES FOR PERFORMANCE

Hopper architecture development looked closely at two key principles for achieving high performance with parallel programs: data locality and asynchronous execution and data transfer.

Data locality is about moving and keeping program data as close to the execution units as possible. A programmer can exploit the performance that comes from having lower latency and higher bandwidth access to local data. Locality can occur spatially or temporally. With spatial locality, data and parallel execution have a spatial relationship, which can be thought of as computation and data being co-located. With temporal locality, data and parallel execution have a temporal

0272-1732 © 2023 IEEE
Digital Object Identifier 10.1109/MM.2023.3256796
Date of publication 14 March 2023; date of current version 15 May 2023.

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.

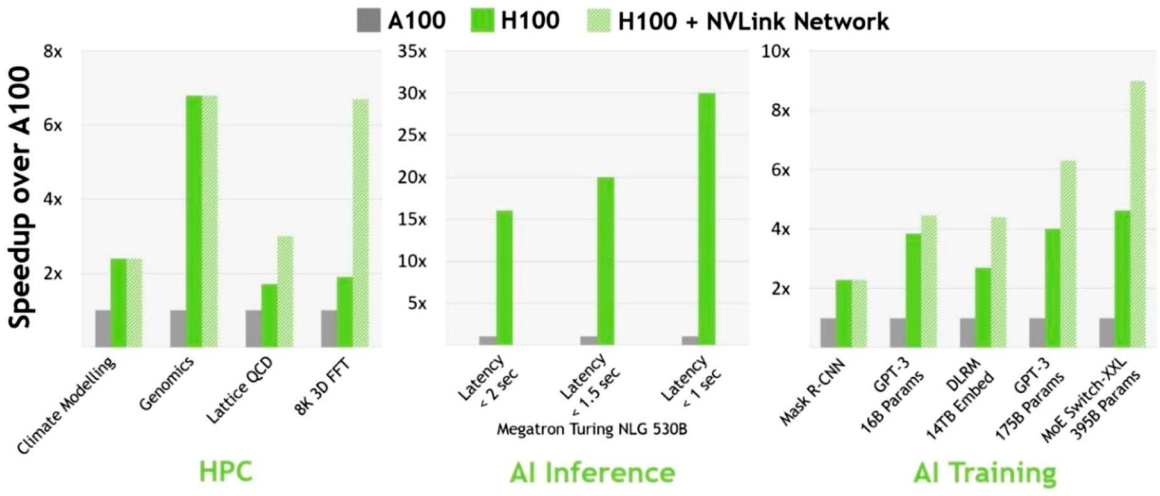


FIGURE 1. H100 performance.

relationship, which can be thought of as computation passing over data. Asynchronous execution and data transfer is about finding and allowing independent tasks and data movement to overlap as much as possible. The goal is to keep all the units in the GPU fully utilized.

Spatial Locality

With spatial locality, data and parallel execution have a spatial relationship. One example of a computation reusing data spatially is with a spatial halo overlap. A computation operates not only on the data in its rectilinear tile but also shares data with computations operating on adjacent tiles. To exploit natural spatial locality in a GPU design, the Hopper architecture supports multiple levels of hierarchy in the CUDA programming model (Figure 2).

In NVIDIA architectures prior to Hopper, three levels of are supported: GRID, thread block, and threads. The lowest level of locality is a thread. Threads are mapped onto hardware thread lanes. Thread level computation uses thread-local register file storage, up to 1 KB per thread, which translates to 64 KB per SM Warp and 256 KB per SM.

The next level of locality that can be expressed is a thread block, which is a cooperative group of up to

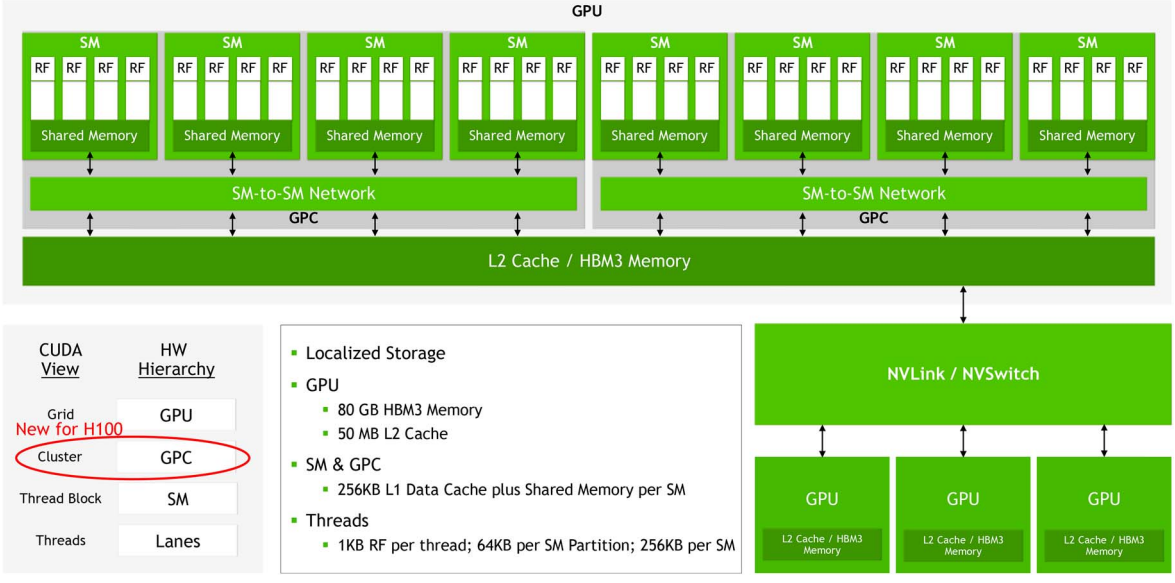


FIGURE 2. CUDA and GPU locality hierarchy.

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.

1024 threads. A thread block is mapped onto the SM hardware hierarchy. Thread block cooperative computation can employ 256 KB of SM localized storage in the L1 data cache plus shared memory.

The third level of locality that can be expressed is a Grid, which is a collection of Thread Blocks. A Grid of work is broken into fragments of work that are assigned to individual Thread Blocks. Hardware supports splitting up of the work in one, two, or three dimensions. The programmer specifies whatever splitting is best for the underlying algorithms. A Grid-level computation uses the storage local to that GPU, which for H100 is 50 MB of L2 cache and 80 GB of HBM3 memory.

These three levels of GPU hierarchy have existed since CUDA was first introduced and has enabled parallel programs to take advantage of the locality that exists in their algorithms. However, GPUs have scaled by orders of magnitude. Figure 3 shows the NVIDIA Kepler GK110 GPU, introduced 10 years ago, compared against the new Hopper H100 GPU. The NVIDIA GK110 GPU has 15 SMs. The NVIDIA H100 GPU has 132 SMs, an order of magnitude increase in 10 years. The entire GK110 GPU could fit in one corner of the H100 GPU. The GK110 GPU is roughly the same size as Hopper's GPC level of hierarchy.

To exploit locality, the Hopper architecture has added a new level of hierarchy to CUDA: the thread block cluster, also simply referred to as a cluster. Here, computation can take advantage of the GPC hardware hierarchy and the 256 KB of localized storage in the SM.

In CUDA, the thread block cluster is a collective of up to 16 thread blocks. Each thread block is guaranteed to execute on a separate SM and to run at the same time. In CUDA code, kernels are annotated with cluster size and dimensions. Just like thread blocks and grids, the thread block cluster dimension can be one, two, or three dimensional.

ASYNCHRONOUS EXECUTION AND DATA TRANSFER IS ABOUT FINDING AND ALLOWING INDEPENDENT TASKS AND DATA MOVEMENT TO OVERLAP AS MUCH AS POSSIBLE.

Thread blocks in a cluster no longer need to execute independently. Like threads in a thread block, the thread blocks in a cluster can execute cooperatively on their fragment of a problem. This approach enables programs to target a larger localized subset of the grid, enabling more opportunities for programmers to easily express execution cooperation and locality for more performance.

The Hopper architecture provides clusters with a direct SM-to-SM communication network. Using this network, threads in one thread block can directly access the shared memory of another thread block. Accesses are performed through a distributed shared memory model, laid out as a partitioned global address space. All memory operations are supported including loads, stores, atomics, and synchronization operations. The communication network also supports accelerated synchronization and data exchange. Threads and thread blocks in a cluster can synchronize directly with each other through barriers in distributed shared memory. The architecture also supports asynchronous direct memory access operations between thread blocks using the new TMA unit.

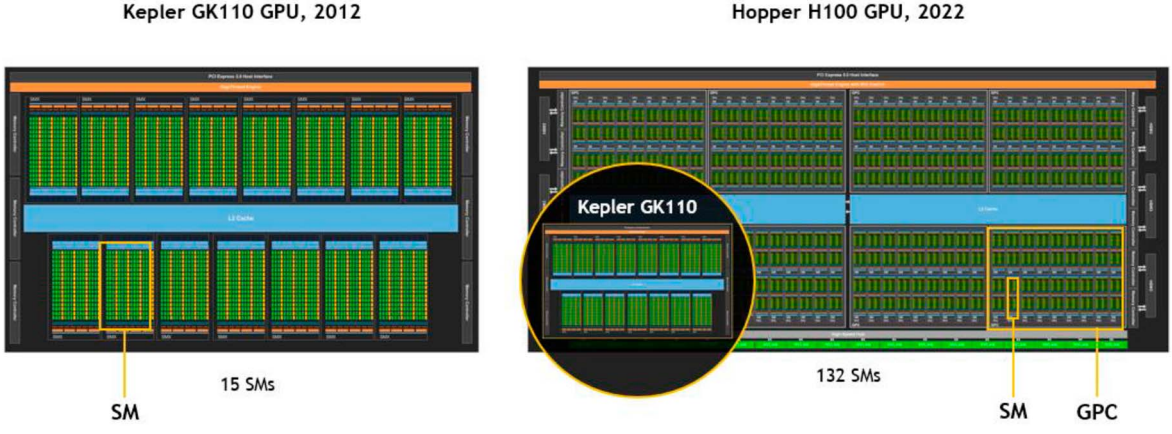


FIGURE 3. Orders of magnitude GPU scaling.

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.

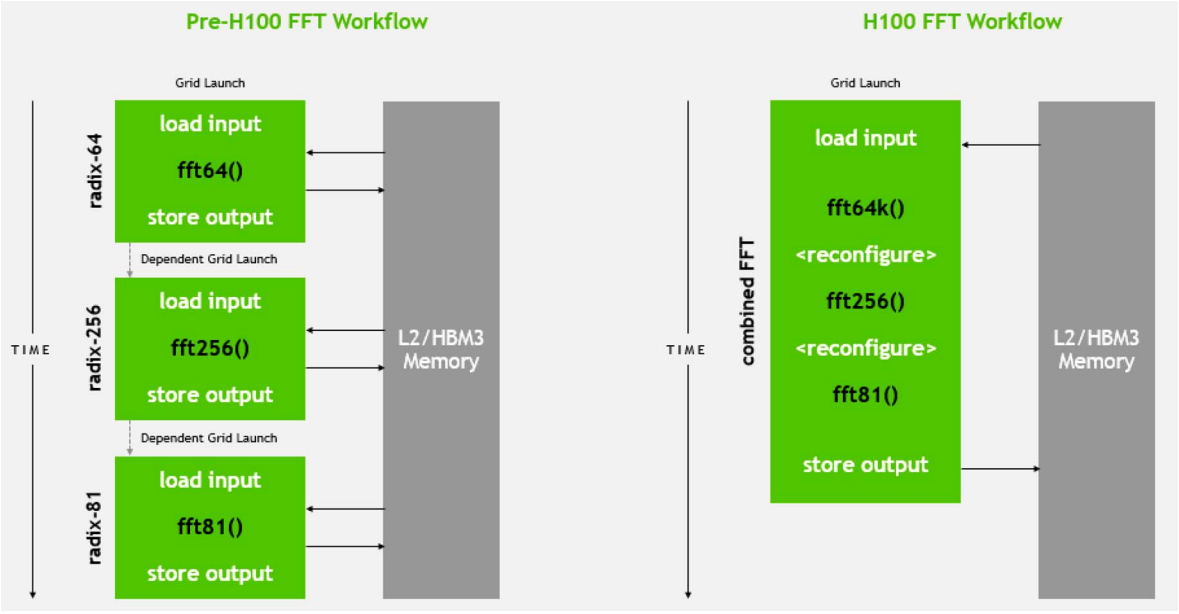


FIGURE 4. Temporal locality for FFT.

Temporal Locality

One form of temporal locality supported in existing NVIDIA GPUs is for multiple grids to execute on and process data in L2 and HBM3 memory. Figure 4 illustrates an example of a fast Fourier transform (FFT) workflow. Multiple computation steps are needed to perform the transform on a set of data, each with a different radix. In pre-H100 GPUs, data is moved into HBM memory that is local to the GPU. From there, each step is performed by reading the data from the GPU's memory, performing the computation, and writing the result back to GPU memory so that it is available to the next step.

The limitation of this sequence is that each step must be a different kernel launch, because the radix algorithm for each step requires a different configuration of the SM. This limitation also means that between each step, data stored local to the SM must be flushed back to GPU memory.

To improve the efficiency of temporal locality, the Hopper architecture has added a new capability: Thread block reconfiguration. The H100 FFT workflow example (Figure 4), demonstrates how this works. There is a single grid launch for the entire transform. For each radix step, the SM thread and register file resources can be reconfigured to match what is needed for the radix at each step. At the beginning of the transform, data is loaded from memory into shared memory and distributed shared memory, and each radix step is performed without data leaving that memory. Instead of the execution passing over data in GPU memory, the execution passes over SM shared memory, which is much closer and more efficient.

Asynchronous Execution

To perform a parallel computation, multiple threads must work together. One way for threads to work together is for them to work cooperatively. This occurs when threads are doing the same computation but are working on different parts of the data. At each phase of the computation, the threads will synchronize to ensure the data they are working on is ready for the next phase of computation on the data.

Another way for threads to work together is to employ pipelining using producer/consumer communication and synchronization. This happens when data produced by one thread is consumed by another thread. In such cases, the consumer must synchronize with the producer to ensure the data has been produced and is ready to be consumed.

With a synchronous machine approach, all threads must arrive and wait at the barrier until all other threads have also arrived at the barrier. The threads are effectively running in a lock-step manner. This leads to inefficiencies where thread and execution resources are idle waiting for the barrier to clear.

With an asynchronous machine design approach, the synchronization is split. Threads arrive and wait at the barrier. When threads are cooperatively executing, additional threads continue to arrive and wait at the

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.

barrier. However, between the arrive and the wait, each thread can schedule thread-independent work. So instead of being idle waiting for other threads to arrive, each thread can be busy doing useful work. In the producer/consumer model, the threads operate in a decoupled manor. After the producer produces the first batch of data, it arrives and then immediately starts to produce the second batch of data. For the consumer, if the data it needs is available, the thread immediately starts to consume the data without any unnecessary synchronization. An asynchronous design approach leads to much more efficient parallel execution.

In the previous generation A100 GPU, NVIDIA provided an asynchronous barrier. A100's asynchronous barriers split the synchronization process into two steps. First, threads signal Arrive when they are finished producing their portion of the shared data. This Arrive is non-blocking so the threads are free to execute other independent work. Eventually, the threads need the data produced by all other threads. At this point they do a Wait which blocks them until every thread has signaled Arrive.

In A100, waiting threads spin on the barrier object in shared memory. H100 adds hardware acceleration for the asynchronous barrier to make it more efficient by doing things like putting waiting threads to sleep until all other threads arrive. These improvements allow the asynchronous barrier synchronization to be lower overhead and lower latency.

Asynchronous Data Transfer

The Hopper architecture also adds acceleration support for asynchronous data transfers. Hopper adds a new form of barrier called an asynchronous transaction barrier, that when combined with the new thread block clusters and new TMA unit, improves the efficiency of data movement and data exchange.

The asynchronous transaction barrier is also a split barrier, but instead of only counting thread arrivals it also counts memory transactions. There are new commands for writing memory that pass both the data to be written and a barrier update. In effect, the barrier can be configured to track not only threads, but asynchronous memory transactions as well. The asynchronous transaction barrier will block threads at the Wait command until all the producer threads have Arrived and all asynchronous memory transactions have completed. This barrier is a very powerful new primitive and is a key building block for performing asynchronous memory copies or data exchanges.

This new barrier support combined with thread block clusters dramatically improves the efficiency of thread block to thread block data exchange. This efficiency can be illustrated by comparing the data exchange using these new features versus a more traditional approach.

TO IMPROVE THE EFFICIENCY OF TEMPORAL LOCALITY, THE HOPPER ARCHITECTURE HAS ADDED A NEW CAPABILITY: THREAD BLOCK RECONFIGURATION.

Traditional data exchange occurs through global memory and the L2 cache using a barrier stored in global memory. The sequence for the producer is to:

- write the data to global memory
- do a memory ordering fence
- write the barrier in global memory.

The sequence for the consumer is to:

- poll the barrier by reading the barrier in global memory, checking the barrier, and re-reading it if not cleared
- once the barrier has cleared, read the data from global memory.

Performing this data exchange requires three to four round trips through global memory.

The Hopper architecture accelerates memory copies using its new TMA unit. The TMA unit performs asynchronous memory copies between global and shared memory, as well as shared- memory-to-shared-memory copies between thread blocks in a thread block cluster. Completion of TMA memory copies are tracked using the new asynchronous transaction barrier. As individual memory transactions complete, the barrier is updated. Once all copies have been completed, threads waiting on that barrier will become unblocked. The TMA is fully asynchronous with thread execution. There is no address generation or data movement and synchronization management overhead requirement on the thread. From the thread's perspective, it is simply fire-and-forget. This design makes for a simplified and efficient programming model.

Using the TMA unit, thread blocks in a thread block cluster can now efficiently copy data directly into each other's shared memory. That copy can be done using a memory transaction where the data and the barrier synchronization travel together. This design enables a minimum latency data exchange of a one-way trip from

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.

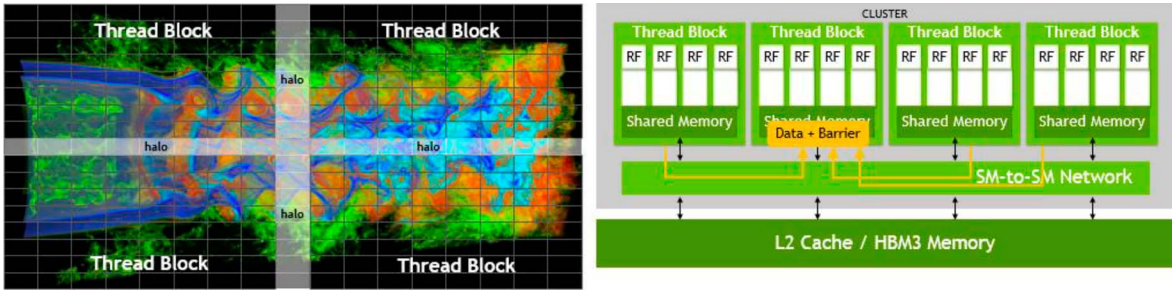


FIGURE 5. Halo data exchange with minimal latency.

the producer to the consumer, resulting in a 7× latency reduction when compared to performing the transfer through global memory.

Figure 5 shows an example of how halo data exchange is performed on Hopper with its new capabilities. The figure shows a thread block cluster of four thread blocks, each working on a different part of the data. When one thread block needs to obtain halo data from the other thread blocks, the other thread blocks write the data directly into its shared memory, updating a local barrier indicating the data is ready. Halo data exchange is efficiently done asynchronously with minimal latency.

Combining Hopper's improvements for locality and asynchronous execution and data movement provides dramatic performance improvements. Figure 6 shows the performance of three algorithms running on Hopper H100 GPU with and without using the new features. Taking advantage of locality and asynchrony provides 1.7× to 2.7× performance benefit on these key algorithms.

ACCELERATING DEEP LEARNING

At the heart of the H100 SM is the new fourth-generation Tensor Core. The new Tensor Core has

FIGURE 6. Performance with clusters and asynchronous execution and data movement.

double the throughput per SM, clock-for-clock, for all data formats compared to A100. TF32, FP16, BFLOAT16, and INT8 have increased to 1024, 2048, 2048, and 4096 MACs/clock/SM, respectively. H100, like A100, supports sparse tensor arithmetic enabling an additional 2× throughput when one of the operands is sparse. To keep these Tensor Cores fed and to keep GPU power in check, H100's Tensor Cores have also improved operand delivery efficiency by 30%.

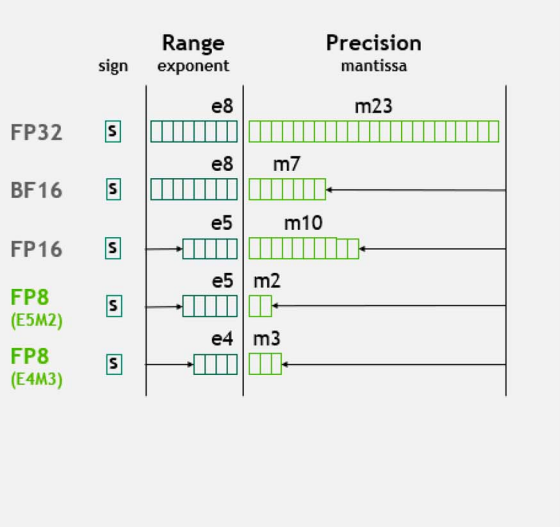
The H100 Tensor Cores have also added support for a new 8-bit floating-point format.[3] It provides twice the throughput of the FP16 and BFLOAT16 formats, matching the throughput of 8-bit integer.

Figure 7 shows the exponent range and mantissa precision provided by each of the Tensor Core floating point formats. The figure also shows the precision of the tensor processing done by the Tensor Core and the SM. The Tensor Cores support two FP8 formats. The E5M2 format provides an extra bit of exponent range and matches the exponent range of FP16. This support can be thought of as a truncated mantissa version of FP16, similar to how BFLOAT16 is a truncated mantissa version of FP32. The E4M3 format trades off one bit of range for an additional one bit of precision. When performing the matrix multiplication of FP8 data, it can be accumulated into either FP32 or FP16 format. Once the matrix multiply is done, various common neural network functions, such as adding a bias or applying an activation function, can be performed by the SM in the higher FP32 or FP16 precision. The final result is then converted to the desired output format before being stored back to memory.

The increased throughput and power efficiency of FP8 can be used to provide a significant boost to training performance for the very compute-intensive Transformer models. However, there are some numerical challenges that must be overcome to maintain accuracy.

The numerical distribution for a variety of network layers in a language model can vary significantly. Some layers have a wider exponent range and are best

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.



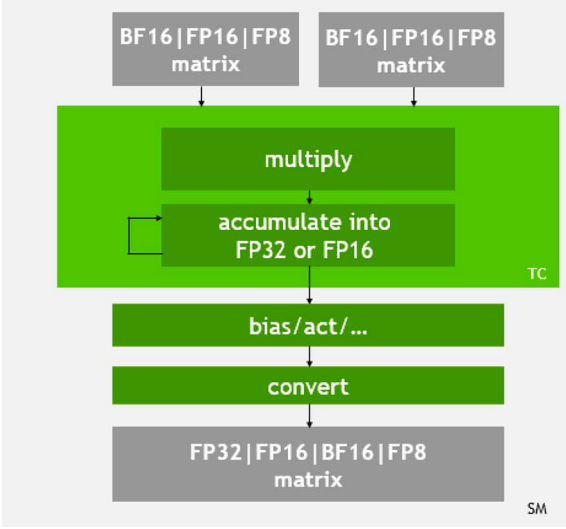


FIGURE 7. Tensor math with FP8.

mapped to the E5M2 FP8 format. Others have a narrow exponent range and require more precision to differentiate between values and are best mapped to the E4M3 FP8 format. During the linear math computation, the range of the result can be outside of the FP8 representable range. By performing the matrix multiply accumulation in the FP16 or FP32 formats, we can safely complete the computation without loss. When converting to the FP8 format for use in the next layer of the network, we scale the result into the FP8 representable range.

The goal of mixed precision is to intelligently manage the precision to maintain accuracy, while still gaining the performance of smaller, faster numerical formats. Statistics of the output values of each layer in a network can be analyzed to determine which FP8 format is optimal, and what scaling factor is needed when converting the final computation into that FP8 format. Figure 8 shows that we can successfully train GPT-3[4] and other models using the FP8 format compared to equivalent accuracy as native 16-bit floating point training. The last two columns of Figure 8 (left) shows that FP8 achieves approximately the same accuracy as 16-bit for two transformer networks and BERT. Figure 8 (right) shows that FP8 very closely matches the training error (perplexity) of native BF16 or FP16 over the duration of GPT-3 training.

H100 provides approximately 6× the training tensor math throughput over the previous generation A100.

Architecture	Network	Dataset	Metric	16-bits	FP8
Transformer	**Vaswani Base**	WMT	BLEU	26.87	26.76
	Vaswani Large	WMT	BLEU	28.43	28.35
Transformer	**XL Base**	WikiText	PPL	22.71	22.76
	XL Large	WikiText	PPL	17.90	17.85[1]
BERT	**BERT Base**	Wikipedia	Loss	1.352	1.357[1]
	BERT Large	Wikipedia	Loss	1.163	1.167

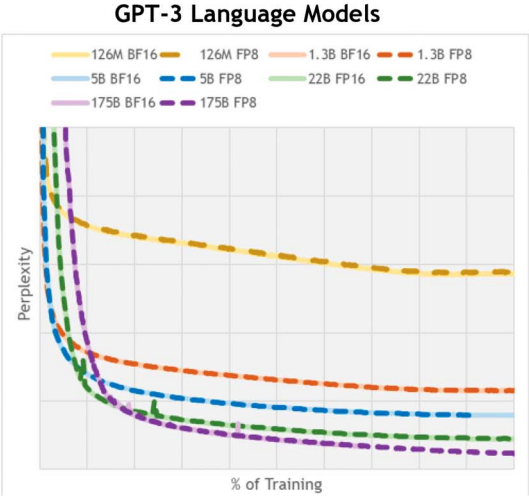


FIGURE 8. Accuracy of transformer models trained with FP8.

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.

H100 achieves this by combining the use of FP8, 2× higher throughput Tensor Cores, a 1.2× increase in the number of SMs, and a 1.3× increase in frequency.

To support this higher throughput math, H100's new TMA unit has been optimized to efficiently manage deep learning tensor matrices in memory. The TMA unit automatically computes the addresses, strides, and bounds checking needed for tensors of up to rank 5. The TMA unit also supports automatic boundary padding for out-of-bounds access. TMA copies are triggered by a single SM thread in a fire-and-forget manner, with no iteration or bounds checking code required for the SM thread.

SCALING UP AND OUT

Using NVLink and NVSwitch chips, eight H100 GPUs can be combined into a DGX H100 node (Figure 9). Thirty-two DGX H100 nodes can be connected over NVLink to create a 256-GPU SuperPOD. Inter-node NVLink connectivity is provided by top-of-rack NVLink Network appliances which make use of a physical form-factor similar to standard 1 U Ethernet and InfiniBand network switches. A SuperPOD provides 1 ExaFLOP of AI compute, and 70.4 TB/s of bisection bandwidth. Multiples of these SuperPODs can be assembled into even larger machines with many thousands of GPUs. Interconnect between multiple SuperPODs is provided by standard network protocols, such as Ethernet and InfiniBand, and DGX H100 nodes can be configured with up to 400 Gbs of standard network bandwidth per GPU.

NVIDIA developed the Grace CPU that was designed with the H100 GPU to be tightly integrated into a new Grace Hopper module. The Grace and Hopper chips are directly connected with a hardware-coherent CPU-GPU interconnect that provides 900 GB/s of bandwidth, allowing the Hopper GPU to directly access Grace's 512 GB of LDDDR5X memory as an extension of its own memory. The Grace Hopper architecture provides a

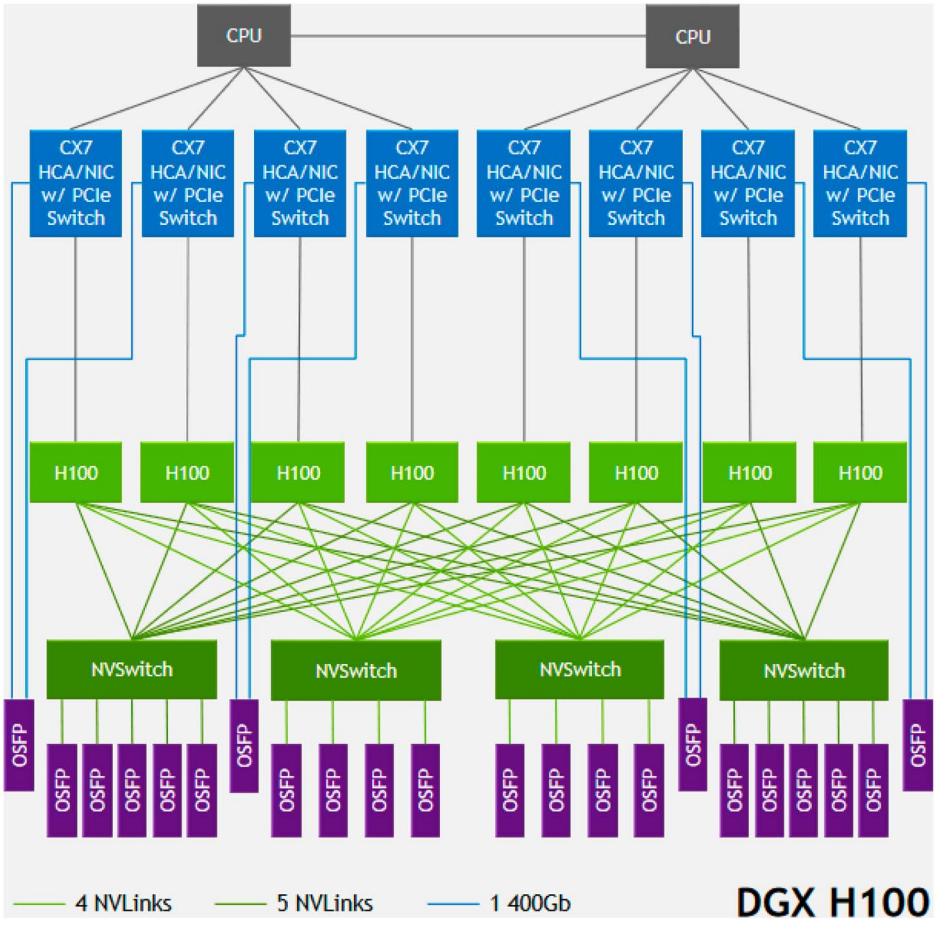


FIGURE 9. DGX H100 node.

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.

unified memory for the GPU and CPU along with a shared virtual address space. The system allocator has support for allocating GPU memory, including both allocated (malloc) and memory-mapped pointers. The Grace Hopper Superchip provides native atomics to the CPU and GPU and includes support for standard C++ atomics.

CONCLUSION

The H100 Tensor Core GPU is NVIDIA's next-generation, highest-performing data center GPU. It represents the product of the collaboration of thousands of engineers whose work spanned the entire system stack, including transistors, architecture, systems integration, software architecture, and programming systems standards. H100 will accelerate AI training and inference, HPC, and data analytics applications in cloud data centers, servers, systems at the edge, and workstations.

REFERENCES

1. "NVIDIA H100 tensor core GPU architecture." Nvidia. Accessed: Nov. 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/h100/
2. J. Choquette et al., "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar./Apr. 2021, doi: 10.1109/MM.2021.3061394.
3. P. Micikevicius et al., "FP8 formats for deep learning," Sep. 2022, *arXiv:2209.05433*.
4. T. Brown et al., "Language models are few-shot learners," May 2020, *arXiv:2005.14165*.

JACK CHOQUETTE is a senior distinguished engineer at NVIDIA, Santa Clara, CA, USA. Choquette received his M.S. degree in computer engineering from the University of Illinois Urbana-Champaign. He is a Member of IEEE. Contact him at jchoquette@NVIDIA.com.

Authorized licensed use limited to: University Duisburg Essen. Downloaded on August 01,2025 at 09:06:54 UTC from IEEE Xplore. Restrictions apply.