

**EXHIBIT 8 – U.S. PATENT NO. RE49461 VS NVIDIA ACCELERATED COMPUTING PRODUCTS & SERVICES –
HOPPER ARCHITECTURE**

I. INTRODUCTION

The chart below demonstrates how Defendant Nvidia Corporation (“Nvidia” or “Defendant”), as well as Defendant’s partners, customers, and end users, directly infringe, either literally or under the doctrine of equivalents, claims 21-31 of U.S. Patent No. RE49461 (“the ’461 Patent”).

Nvidia has committed acts of infringement within this District. Nvidia uses the Accused Products in this District in manners that practice the ’461 Patent, including by testing the Accused Products and by using the Accused Products at its offices and premises in this District. Defendant makes, uses, advertises, offers for sale, and/or sells hardware for accelerated computing, including GPUs, CPUs, and SoCs; computers for accelerated computing (e.g., supercomputers, servers, and data centers for high performance computing); and computer platform software-as-a-service (“SaaS”) that implements accelerated computing (including the Accused Products) in the State of Texas and in this District directly and/or through its partnerships

Defendant also provides data center and HPC services that practice the ’461 Patent in the State of Texas and in this District directly and/or through its partnerships with businesses in the State of Texas and in this District.

Nvidia sells, offers for sale, advertises, makes, installs, and/or otherwise provides hardware, software, firmware, and/or computer platforms for accelerated computing and data center and HPC services, including the Accused Products, the use of which infringes the ’461 Patent in this District and the State of Texas. (*See* <https://www.nvidia.com/en-us/data-center/solutions/accelerated-computing/>.) Nvidia performs these acts directly and/or through its partnerships with other entities. (*See id.* (“NVIDIA has defined a range of accelerated platforms that each consist of hardware systems designed according to the needs of the use case as well as the software stack that enables the operation and management of the business applications. These hardware systems and software are available from NVIDIA and our partners.”).)

Nvidia also uses a network of partners, which comprise re-sellers, managed service providers, and product and solution experts, to provide the Accused Products and implementation services for the Accused Products to customers in this District. Each of these partners sells, offers for sale, installs, and/or implements Nvidia’s accelerated computing hardware, software, and/or computer platform services. (*See* <https://www.nvidia.com/en-us/about-nvidia/partners/>.)

Nvidia’s partners include “Data Center Provider[s].” (*See* <https://www.nvidia.com/en-us/about-nvidia/partners/>.) Nvidia’s Data Center Provider partners “offer colocation services such as high-density data center facilities, interconnected infrastructure, and state-of-art

cooling technologies for hosting NVIDIA DGX™ servers globally.” (*See id.*) Nvidia’s Data Center Provider partners in the “NVIDIA DGX-Ready Data Center program, built on the NVIDIA DGX™ platform and delivered by NVIDIA partners,” help “accelerate the scaling (*See* <https://www.nvidia.com/en-us/data-center/colocation-partners/#aligned-energy>.)

As further detailed below, Nvidia engages in activities that directly infringe the ’461 Patent within this District. For example, Nvidia’s operation and use of its accelerated computing hardware, software, and/or computer platform services, including its data center-scale accelerated computing platforms, within this District infringe the ’461 Patent.

Nvidia also infringes (directly or indirectly) the ’461 Patent by providing services in connection with the Accused Products including installing, maintaining, supporting, operating, providing instructions, and/or advertising Nvidia’s computer platform, data center, and HPC services within this District. For example, under Nvidia’s cloud and data center line of products and services, the Nvidia DGX platform is a “a fully integrated hardware and software AI platform” and “combines the best of NVIDIA software, infrastructure, and expertise in a modern, unified AI development solution.” (*See* <https://www.nvidia.com/en-us/data-center/dgx-platform/>.) Indeed, “DGX infrastructure is a complete AI solution, and includes NVIDIA AI Enterprise software to accelerate data science pipelines and streamline development and deployment of production-grade AI applications.” (*See id.*) Nvidia platform user and partner customers infringe the ’461 Patent by installing and operating Nvidia’s computer platform software, which performs the claimed methods in the ’461 Patent within this District. (*See also, e.g.*, <https://www.nvidia.com/en-us/data-center/products/ai-enterprise/> (Nvidia AI Enterprise); <https://developer.nvidia.com/cuda-zone> (Nvidia CUDA Toolkit); <https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/> (GPU Cloud Computing).)

Defendant encourages and induces its customers of the Accused Products to perform the methods claimed in the ’461 Patent. For example, Nvidia makes its accelerated platforms and services available on its website, widely advertises those platforms and services, provides applications that allow partners and users to access those platforms and services, provides instructions for installing, and maintaining those platforms and services and supporting software and/or firmware, and provides technical support to users. (*See* <https://www.nvidia.com/en-us/data-center/dgx-support/>.) Nvidia further encourages and induces its customers to operate Nvidia’s hardware and software in an infringing manner, and to use Nvidia’s infringing computer platforms, by providing directions for and encouraging customers to install software, such as software for NVIDIA AI Enterprise and CUDA, (*see* <https://docs.nvidia.com/ai-enterprise/deployment-guide-vmware/0.1.0/software.html>; <https://developer.nvidia.com/cuda-downloads>), which offers evaluation, installation, configuration, customization, and development of Nvidia’s infringing software products and services. Defendant also contributes to the infringement of its customers and end users of the Accused Products by offering within the United States or importing into the United States the Accused Products, which are for use in practicing, and under normal operation practice, one or more of the methods claimed in the ’461 Patent, constituting a material part of the inventions claimed, and not a staple article or commodity of

commerce suitable for substantial non-infringing uses. Indeed, as shown herein, the Accused Products and the example functionality described below have no substantial non-infringing uses and are specifically designed to practice the methods claimed in the '461 Patent.

Nvidia offers, sells, and uses several products that provide and implement GPU-acceleration hardware, software, platforms, and services for individuals and enterprises and incorporate Plaintiff's patented technologies. (See <https://www.nvidia.com/en-us/solutions/ai/inference/>; <https://marketplace.nvidia.com/en-us/data-center/?page=4>; <https://marketplace.nvidia.com/en-us/laptops-workstations/?page=9>; <https://marketplace.nvidia.com/en-us/software/?page=3>.)

The Accused Products, as described *infra*, include Nvidia's hardware, software, and services designed, implemented, and used for hardware acceleration, including Nvidia's GPU accelerators and superchips; Nvidia's computers, supercomputers, data centers, servers, workstations that implement its GPU accelerators and superchips; and Nvidia's software, platforms, and services for accelerated computing.

The Accused Products include Nvidia's GPU accelerators and superchips. (See <https://resources.nvidia.com/en-us-gpu>.) Nvidia's GPU accelerators include Nvidia's GPUs with Nvidia's "Hopper," "Ada Lovelace," "Ampere," "Turing," "Volta," "Pascal," and "Maxwell" GPU architectures. (See <https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-896/support-matrix/index.html>.) These GPUs are specifically designed to run and implement GPU-based hardware acceleration using Nvidia's proprietary CUDA (Compute Unified Device Architecture) platform and CUDA libraries for GPU acceleration. (See *id.* (Nvidia GPU architectures implementing Nvidia's cuDNN (CUDA Deep Neural Network) library for GPU acceleration.); <https://developer.nvidia.com/cuda-gpus>.)

Nvidia's Hopper GPUs include the H100 and H200 GPUs. (See <https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/> (Hopper architecture); <https://www.nvidia.com/en-us/data-center/h100/> (H100); <https://www.nvidia.com/en-us/data-center/h200/> (H200).) In addition, Nvidia's superchips that implement GPU accelerators include the GH200, or Grace Hopper Superchip, which implements the Hopper GPU architecture. (See [https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/\(GH200\)](https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/(GH200)).)

Nvidia's Ada Lovelace (or Lovelace) GPUs include Nvidia Data Center GPUs, including L40, L40S, and L4 GPUs; Nvidia Workstation and Professional Laptop GPUs, including RTX Ada Generations series GPUs and Laptop GPUs (including RTX 6000, RTX 6000 Ada, RTX 5000 Ada, RTX 4500 Ada, RTX 4000 Ada, RTX 4000 SFF, RTX 3500, RTX 3000, RTX 2000, RTX 1000, RTX 500); and GeForce RTX 40 series GPUs and Laptop GPUs (RTX 4090, RTX 4080 SUPER, RTX 4070 Ti SUPER, RTX 4070 SUPER, RTX 4070, RTX 4060 Ti, and RTX 4060; GeForce RTX 4090 Laptop GPU, GeForce RTX 4080 Laptop GPU, GeForce RTX 4070 Laptop GPU, GeForce RTX 4060 Laptop GPU, GeForce RTX 4050 Laptop GPU). (See <https://www.nvidia.com/en-us/technologies/ada-architecture/> (Ada Lovelace architecture). See <https://www.nvidia.com/en-us/data-center/l40/> (L40); <https://www.nvidia.com/en-us/data-center/l40s/>

(L40S); <https://www.nvidia.com/en-us/data-center/l4/> (L4). *See* <https://resources.nvidia.com/en-us-design-viz-stories-ep/l40-linecard> (Nvidia Professional GPUs); <https://www.nvidia.com/en-us/ai-on-rtx/> (RTX GPUs featuring “Accelerated Development”); <https://www.nvidia.com/en-us/design-visualization/desktop-graphics/> (RTX Ada Generation GPUs); <https://www.nvidia.com/en-us/design-visualization/rtx-professional-laptops/compare-table/> (RTX Ada Generation Laptop GPUs). *See* <https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/> (GeForce RTX 40 GPUs); <https://www.nvidia.com/en-us/geforce/graphics-cards/compare/> (GeForce RTX 40 GPUs); <https://www.nvidia.com/en-us/geforce/laptops/compare/> (GeForce RTX 40 Laptop GPUs).)

Nvidia’s Ampere GPUs include Nvidia Data Center GPUs, including A100, A40, A30, A16, A10, and A2 GPUs; Nvidia Workstation and Professional Laptop GPUs, including RTX A series GPUs and Laptop GPUs (A800 40GB Active, RTX A6000, RTX A5500, RTX A5000, RTX A4500, RTX A4000, RTX A2000, RTX A2000 12GB, RTX A1000, RTX A400, RTX A5500, RTX A4500, RTX A3000 12GB, RTX A2000 8GB, RTX A1000 6GB, RTX A500); GeForce RTX 30 series GPUs and Laptop GPUs (GeForce RTX 3090 Ti, GeForce RTX 3090, GeForce RTX 3080 Ti, GeForce RTX 3080, GeForce RTX 3070 Ti, GeForce RTX 3070, GeForce RTX 3060 Ti, GeForce RTX 3060, GeForce RTX 3050 (8 GB), GeForce RTX 3050 (6 GB), GeForce RTX 3080 Ti Laptop GPU, GeForce RTX 3080 Laptop GPU, GeForce RTX 3070 Ti Laptop GPU, GeForce RTX 3070 Laptop GPU, GeForce RTX 3060 Laptop GPU, GeForce RTX 3050 Ti Laptop GPU, GeForce RTX 3050 Laptop GPU); and GeForce MX570 Laptop GPU. (*See* <https://www.nvidia.com/en-us/data-center/ampere-architecture/> (Ampere architecture). *See* <https://www.nvidia.com/en-us/data-center/a100/> (A100); <https://www.nvidia.com/en-us/data-center/a40/> (A40); <https://www.nvidia.com/en-us/data-center/a30/> (A30); <https://www.nvidia.com/en-us/data-center/a16/> (A16); <https://www.nvidia.com/en-us/data-center/a10/> (A10); <https://www.nvidia.com/en-us/data-center/a2/> (A2). *See* <https://www.nvidia.com/en-us/design-visualization/desktop-graphics/> (RTX A GPUs); <https://www.nvidia.com/en-us/design-visualization/rtx-professional-laptops/compare-table/> (RTX A Laptop GPUs). *See* <https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/>; (GeForce RTX 30 GPUs) <https://www.nvidia.com/en-us/geforce/graphics-cards/compare/> (GeForce RTX 30 GPUs); <https://www.nvidia.com/en-us/geforce/laptops/compare/30-series/> (GeForce RTX 30 Laptop GPUs); <https://www.nvidia.com/en-us/geforce/gaming-laptops/mx-570/> (GeForce MX570 Laptop GPU).)

Nvidia’s Turing GPUs include Nvidia Data Center GPUs, including Tesla T4 GPUs and Quadro RTX 8000 (passive) and Quadro RTX 6000 (passive) GPUs; Nvidia Workstation and Professional Laptop GPUs, including T series GPUs and Laptop GPUs, Quadro T series Laptop GPUs, and Quadro RTX series GPUs and Laptop GPUs (Quadro RTX 8000, Quadro RTX 6000, Quadro RTX 5000, Quadro RTX 4000, Quadro RTX 3000, Quadro T2000, T1000 8GB, T1200, Quadro T1000, T1000 (4GB), T600, T550, T500 T400, T400 4GB); Titan series Titan RTX GPU; GeForce RTX 20 series GPUs and Laptop GPUs (GeForce RTX 2080 Ti, GeForce RTX 2080 Super, GeForce RTX 2080, GeForce RTX 2070 Super, GeForce RTX 2070, GeForce RTX 2060 Super, GeForce RTX 2060, GeForce RTX 2500); GeForce GTX 16 series GPUs and Laptop GPUs (GeForce GTX 1660 Ti, GeForce GTX 1660 Super, GeForce GTX 1660, GeForce GTX 1650 Ti, GeForce GTX 1650 Super, GeForce GTX 1650 (G5), GeForce GTX 1650 (G6), GeForce GTX 1650, GeForce GTX 1630); and GeForce MX550, MX450, and MX430 Laptop GPUs. (*See* <https://www.nvidia.com/en-us/geforce/turing/> (Turing

architecture). See <https://www.nvidia.com/en-us/data-center/tesla-t4/> (Tesla T4); <https://www.nvidia.com/en-gb/design-visualization/quadro-data-center/> (Quadro RTX 8000 (passive) and Quadro RTX 6000 (passive)). See <https://www.nvidia.com/en-us/design-visualization/quadro/> (T series GPUs/Laptop GPUs, Quadro T series Laptop GPUs, and Quadro RTX GPUs/Laptop GPUs); <https://www.nvidia.com/en-us/design-visualization/desktop-graphics> (T series GPUs/Laptop GPUs); <https://www.nvidia.com/content/dam/en-zz/Solutions/titan/documents/titan-rtx-for-creators-us-nvidia-1011126-r6-web.pdf> (Titan RTX); <https://www.nvidia.com/en-us/geforce/20-series/> (GeForce RTX 20 GPUs); <https://www.nvidia.com/en-us/geforce/graphics-cards/compare/> (GeForce RTX 20 GPUs and GeForce GTX 16 GPUs); <https://www.nvidia.com/en-us/geforce/gaming-laptops/compare-20-series/> (GeForce RTX 20 Laptop GPUs); <https://www.nvidia.com/en-us/geforce/gaming-laptops/compare-16-series/> (GeForce GTX 16 Laptop GPUs); <https://www.nvidia.com/en-us/geforce/gaming-laptops/mx-550/> (GeForce MX550 Laptop GPU); <https://www.nvidia.com/en-us/geforce/gaming-laptops/mx-450/> (GeForce MX450 Laptop GPU); <https://wccftch.com/nvidia-geforce-mx450-turing-discrete-notebook-gpu-gddr6-pcie-4/> (GeForce M Laptop GPUs).)

Nvidia's Volta GPUs include Nvidia Data Center GPUs, including the Tesla V100 GPU; Nvidia Workstation GPUs, including Quadro GV100; and Titan series Titan V GPU. (See <https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/> (Volta architecture); <https://www.nvidia.com/en-us/data-center/v100/> (Tesla V100); <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-volta-gv100-data-sheet-us-nvidia-704619-r3-web.pdf> (Quadro GV100); <https://nvidianews.nvidia.com/news/nvidia-titan-v-transforms-the-pc-into-ai-supercomputer> (Titan V).)

Nvidia's Pascal GPUs include Nvidia Data Center GPUs, including Tesla P100, P40, and P4 GPUs; Nvidia Workstation and Professional Laptop GPUs, including the Quadro GP100 GPU and Quadro P series GPUs and Laptop GPUs (Quadro P6000, Quadro P5200, Quadro P5000, Quadro P4200, Quadro P4000, Quadro P3200, Quadro P3000, Quadro P2200, Quadro P2000, Quadro P1000, Quadro P620, Quadro P600, Quadro P520, Quadro P500, Quadro P400); Titan series Titan Xp and Titan X GPUs; GeForce GTX 10 series GPUs and Laptop GPUs (GeForce GTX 1080 Ti, GeForce GTX 1080, GeForce GTX 1070 Ti, GeForce GTX 1070, GeForce GTX 1060, GeForce GTX 1050 Ti, GeForce GTX 1050); and GeForce MX300 series, MX200 series, and MX150 Laptop GPUs. (See <https://developer.nvidia.com/pascal>; <https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/> (Pascal architecture). See <https://www.nvidia.com/en-us/data-center/tesla-p100> (Tesla P100); <https://developer.nvidia.com/cuda-gpus> (Tesla P40 and P4); <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-gp100-data-sheet-us-nv-704562-r1.pdf> (Quadro GP100); <https://www.nvidia.com/en-us/design-visualization/quadro/> (Quadro P series GPUs/Laptop GPUs). See https://www.nvidia.com/content/geforce-gtx/NVIDIA_TITAN_X_USER_GUIDE_v02.pdf (Titan X); https://www.nvidia.com/content/geforce-gtx/NVIDIA_TITAN_Xp_USER_GUIDE_v02.pdf (Titan Xp); <https://www.nvidia.com/en-us/geforce/10-series/> (GeForce GTX 10); <https://www.nvidia.com/en-us/geforce/graphics-cards/compare/> (GeForce GTX 10 GPUs); <https://www.nvidia.com/en-us/geforce/news/gfcent/nvidia-geforce-gtx-10-series-laptops/> (GeForce GTX 10 Laptop GPUs); <https://www.nvidia.com/en-us/geforce/gaming-laptops/mx-350/> (GeForce MX350 Laptop GPU); [5](https://www.nvidia.com/en-</p>
</div>
<div data-bbox=)

[us/geforce/gaming-laptops/mx-330/](https://www.wccftech.com/nvidia-geforce-mx450-turing-discrete-notebook-gpu-gddr6-pcie-4/) (GeForce MX330 Laptop GPU); [https://wccftech.com/nvidia-geforce-mx450-turing-discrete-notebook-gpu-gddr6-pcie-4/](https://www.wccftech.com/nvidia-geforce-mx450-turing-discrete-notebook-gpu-gddr6-pcie-4/) (GeForce M Laptop GPUs).)

Nvidia's Maxwell GPUs include Nvidia Data Center GPUs, including Tesla M60, M40, and M10 GPUs; Nvidia Workstation and Professional Laptop GPUs, including Quadro M series GPUs and Laptop GPUs (Quadro M6000 24GB, Quadro M6000 (12GB), Quadro M5000, Quadro M5000M, Quadro M5500, Quadro M4000, Quadro M4000M, Quadro M3000M, Quadro M2200, Quadro M2000, Quadro M2000M, Quadro M1200, Quadro M1000M, Quadro M620, Quadro M600M, Quadro M520, Quadro M500M), the NVS 810 GPU, and Tesla M6 series Laptop GPUs; Titan series GTX Titan X GPU; GeForce GTX 900 series GPUs and Laptop GPUs (GeForce GTX 980Ti, GeForce GTX 980, GeForce GTX 970, GeForce GTX 960, GeForce GTX 980M, GeForce GTX 970M, GeForce GTX 965M, GeForce GTX 960M, GeForce GTX 950M); GeForce GTX 700 series GPUs and Laptop GPUs (GeForce GTX 750 Ti, GeForce GTX 750); and GeForce MX130 series and MX110 Laptop GPUs. (See <https://developer.nvidia.com/blog/maxwell-most-advanced-cuda-gpu-ever-made/> (Maxwell architecture); <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/nvidia-m60-datasheet.pdf> (M60); https://images.nvidia.com/content/tesla/pdf/78071_Tesla_M40_24GB_Print_Datasheet_LR.PDF (M40); <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-m10/pdf/188359-Tesla-M10-DS-NV-Aug19-A4-fnl-Web.pdf> (M10); <https://www.nvidia.com/en-us/design-visualization/quadro/> (Quadro M GPUs/Laptop GPUs); <https://www.nvidia.com/docs/IO/146527/nvs-810-datasheet.pdf> (NVS 810); <https://images.nvidia.com/content/tesla/pdf/188300-Tesla-M6-DS-Aug19-A4-fnl-Web.pdf> (Tesla M6); https://www.nvidia.com/content/geforce-gtx/GTX_TITAN_X_User_Guide.pdf (GTX Titan X); <https://developer.nvidia.com/maxwell-compute-architecture> (GeForce GTX 900 and 700 GPUs/Laptop GPUs); [https://wccftech.com/nvidia-geforce-mx450-turing-discrete-notebook-gpu-gddr6-pcie-4/](https://www.wccftech.com/nvidia-geforce-mx450-turing-discrete-notebook-gpu-gddr6-pcie-4/) (GeForce M Laptop GPUs).

These GPUs and superchips implement, and are specifically designed for, GPU-acceleration for artificial intelligence and neural networks. Nvidia's proprietary CUDA platform for parallel computing, which includes GPU-acceleration libraries such as cuDNN (CUDA Deep Neural Network), is implemented in the Nvidia Hopper, Ada Lovelace, Ampere, Turing, Volta, Pascal, and Maxwell GPU architectures.

The Accused Products further include Nvidia's computers, supercomputers, data centers, servers, workstations that implement its GPU accelerators and superchips. These computer hardware systems include: the DGX line of supercomputers, the HGX line of supercomputers, the OVX line of supercomputers, and the EGX line of servers for data centers and edge devices. (See <https://www.nvidia.com/en-us/data-center/solutions/accelerated-computing/>.)

Nvidia's DGX supercomputers include the DGX H200, DGX BasePOD, and DGX SuperPOD with DGX GB200. (See <https://www.nvidia.com/en-us/data-center/dgx-platform/>; see also <https://www.nvidia.com/en-us/data-center/base-command/>;

<https://resources.nvidia.com/en-us-dgx-software/nvidia-base-command> (DGX Base Command operating system for DGX data centers.) Nvidia’s HGX “AI supercomputing platform brings together the full power of NVIDIA GPUs, NVIDIA NVLink™, NVIDIA networking, and fully optimized AI and high-performance computing (HPC) software stacks.” (See <https://www.nvidia.com/en-us/data-center/hgx/>; <https://nvdam.widen.net/s/5kgbjq2v2t/hpc-hgx-h100-datasheet-nvidia-web>.) One example configuration includes “four or eight H200 or H100 GPUs.” (See *id.*; see <https://nvdam.widen.net/s/5kgbjq2v2t/hpc-hgx-h100-datasheet-nvidia-web>.) Nvidia’s OVX supercomputers implement “L40S GPUs . . . for both complex AI and graphics-intensive workloads.” (See <https://www.nvidia.com/en-us/data-center/products/ovx/>; see <https://resources.nvidia.com/en-us-ovx/ovx-datasheet>.) And Nvidia’s “EGX hardware portfolio” includes “accelerators [that] combine the performance of NVIDIA Ampere GPUs.” (See <https://www.nvidia.com/en-us/data-center/products/egx/>; see <https://www.nvidia.com/en-us/design-visualization/egx-graphics/>.)

The Accused Products further include Nvidia’s software, platforms, and services for accelerated computing. These include CUDA, Nvidia AI Enterprise, the DGX Platform, Nvidia Omniverse, Nvidia Drive, Nvidia Isaac Sim, Nvidia Clara, Nvidia AI Foundation models, and Nvidia NGC.

CUDA is Nvidia’s proprietary “parallel computing platform and programming model.” (See <https://developer.nvidia.com/cuda-zone>.) CUDA is designed to support Nvidia’s GPU accelerators and superchips and includes software specifically for GPU-acceleration such as the cuDNN “GPU-accelerated library.” (See *id.*; <https://developer.nvidia.com/cudnn>.) In addition, Nvidia’s CUDA-X, built on top of CUDA, is a collection of “GPU-accelerated microservices and libraries for AI.” (See <https://www.nvidia.com/en-us/technologies/cuda-x/>.) Nvidia also offers the CUDA Toolkit and SDK Manager for developing GPU-accelerated applications. (See <https://developer.nvidia.com/cuda-toolkit>; <https://developer.nvidia.com/sdk-manager>.)

In addition, Nvidia AI Enterprise is Nvidia’s “end-to-end, cloud-native software platform” for “accelerat[ing] data science pipelines . . . and other generative AI applications.” (See <https://www.nvidia.com/en-us/data-center/products/ai-enterprise/>.) It is Nvidia’s “‘operating system’ for enterprise AI.” (See *id.*)

In addition, Nvidia’s DGX platform is “is a complete AI solution and includes NVIDIA AI Enterprise software.” (See <https://www.nvidia.com/en-us/data-center/dgx-platform/>.) Nvidia DGX Cloud is “an AI-training-as-a-service platform which includes cloud-based infrastructure and software for AI, customizable pretrained AI models, and access to NVIDIA experts.” (See <https://d18rn0p25nwr6d.cloudfront.net/CIK-0001045810/1cbe8fe7-e08a-46e3-8dcc-b429fc06c1a4.pdf>, Nvidia U.S. Securities and Exchange Commission Form 10-K for Fiscal Year Ended January 28, 2024 at 6.)

In addition, Nvidia Omniverse is “a development platform and operating system for building virtual world simulation applications, available as a software subscription.” (See <https://d18rn0p25nwr6d.cloudfront.net/CIK-0001045810/1cbe8fe7-e08a-46e3-8dcc->

b429fc06c1a4.pdf, Nvidia U.S. Securities and Exchange Commission Form 10-K for Fiscal Year Ended January 28, 2024 at 6.) Nvidia Omniverse implements software and services “into existing software tools and simulation workflows for building AI systems.” (See <https://www.nvidia.com/en-us/omniverse/>.)

In addition, Nvidia Drive is a platform that “consists of both the AI infrastructure and in-vehicle hardware and software” for autonomous vehicles. (See <https://www.nvidia.com/en-us/self-driving-cars/>.) “NVIDIA DRIVE Infrastructure encompasses data center hardware, software, and workflows—both on premises and in NVIDIA DGX Cloud & Omniverse.” (See *id.*)

In addition, Nvidia Isaac Sim is a platform that enables “developers to design, simulate, test, and train AI-based robots and autonomous machines in a physically-based virtual environment.” (See <https://developer.nvidia.com/isaac/sim/>.) It is built on Nvidia Omniverse. (See *id.*)

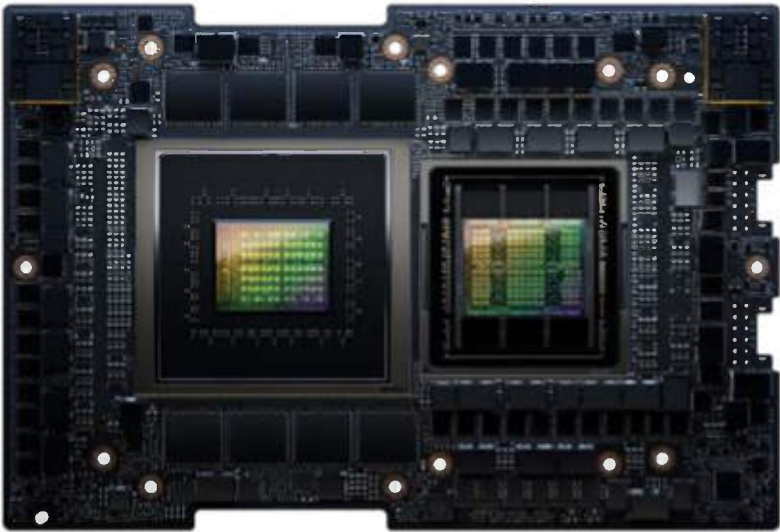
In addition, Nvidia Clara is “a suite of computing platforms, software, and services that powers AI solutions for healthcare and life sciences, from imaging and instruments to genomics and drug discovery” that provides “AI-Powered Solutions for Healthcare.” (See <https://www.nvidia.com/en-us/clara/>.)

In addition, Nvidia AI Foundation models are “community and NVIDIA-built models” that “are NVIDIA-optimized to deliver the best performance on NVIDIA accelerated infrastructure.” (See <https://www.nvidia.com/en-us/ai-data-science/foundation-models/>.)

In addition, Nvidia NGC is a collection of software services and tools that support “end-to-end AI and digital twin workflows” that runs on “NVIDIA GPU-accelerated platforms.” (See <https://www.nvidia.com/en-us/gpu-cloud/>.) NGC “offers a collection of cloud services . . . for generative AI, drug discovery, and speech AI solutions, and the NGC Private Registry for securely sharing proprietary AI software.” (See *id.*)

II. CLAIM CHARTS

Key Features	Accused Products
[21.Pre] A method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition, the method comprising:	The Accused Products perform each step of the method of claim 21 of the '461 Patent. To the extent the preamble is construed to be limiting, the Accused Products perform <i>a method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition</i>, as further explained below. For instance, the Grace Hopper Superchip (GH200) Accused Product “brings together the groundbreaking performance of the NVIDIA Hopper GPU with the versatility of the NVIDIA Grace™ CPU . . . in a single Superchip.” It includes the cuDNN (CUDA Deep Neural Network) library for “[d]eep neural networks.” The “Grace Hopper Superchip is the first true heterogeneous accelerated platform for high-performance computing (HPC) and AI workloads. It accelerates applications with the strengths of both GPUs and CPUs while providing the simplest and most productive heterogeneous programming model to date.” The “Grace Hopper Superchip is the first true heterogeneous accelerated platform for high-performance computing (HPC) and AI workloads. It accelerates applications with the strengths of both GPUs and CPUs while providing the simplest and most productive heterogeneous programming model to date.”

Key Features	Accused Products
	Inside <u>NVIDIA's First GPU-CPU Superchip</u>  The NVIDIA® GH200 Grace Hopper architecture brings together the groundbreaking performance of the <u>NVIDIA Hopper GPU</u> with the versatility of the <u>NVIDIA Grace™ CPU</u>, connected with a high bandwidth and memory coherent <u>NVIDIA NVLink Chip-2-Chip (C2C)®</u> interconnect in a single Superchip, and support for the new NVIDIA NVLink Switch System. * * * * *

Key Features	Accused Products
	An extensive suite of domain-specific libraries and frameworks further accelerates main algorithms in a wide range of application domains, for example: • <u>Deep neural networks (cuDNN)</u> • Linear solvers for simulations and implicit unstructured methods (AmgX) • Quantum computing (cuQuantum) • Data science • Machine learning (RAPIDS) • Data loading and preprocessing for machine learning (DALI) • Real-time 3D simulation and design collaboration (Omniverse) * * * * * The NVIDIA GH200 Grace Hopper Superchip is the first true heterogeneous accelerated platform for <u>high-performance computing (HPC) and AI workloads. It accelerates applications with the strengths of both GPUs and CPUs</u> while providing the simplest and most productive heterogeneous programming model to date, enabling scientists and engineers to focus on solving the world's most important problems. Together with NVIDIA networking technologies, NVIDIA GH200 provides the recipe for the next generation of HPC supercomputers and AI factories, enabling customers to take on larger datasets, more complex models, and new workloads, solving them more quickly than before. (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).) As illustrated below, a diagram describing the architecture of the Grace Hopper Superchip depicts a CPU (“Grace CPU”) with “[u]p to 72 cores” and CPU memory (“CPU LPDDR5X”) and a GPU (“Hopper GPU”) and GPU memory (“GPUHBM3 or HBm3e”).

Key Features	Accused Products
	Figure 1 shows the logical overview of the NVIDIA GH200 Grace Hopper Superchip and Table 1 lists its key features. The diagram, titled "NVIDIA GH200 Grace Hopper Superchip", illustrates the internal architecture. On the left, a green box labeled "High-Speed IO" connects to a central "Grace CPU" (green grid) via "4x 16x PCIe-5 512GB/s". The CPU is flanked by two blue boxes for "CPU LPDDR5X ≤480GB" and is connected to a "Hopper GPU" (green grid) by an "NVLink-C2C 900GB/s" link. The GPU is similarly flanked by two orange boxes for "GPU HBM3 or HBM3e" (96GB HBM3, 144GB HBM3e) and connected to an "NVLink Network ≤ 32 GPUs" on the right via "18x NVLink 4 900GB/s". A dashed green box labeled "Hardware Coherency" encloses the CPU, GPU, and their respective memory blocks. Figure 1. NVIDIA GH200 Grace Hopper Superchip Logical Overview

Key Features	Accused Products																
	Table 1. NVIDIA GH200 Grace Hopper Superchip Key Features <table> <tr> <th>Feature</th><th>Description</th></tr> <tr> <td>Grace CPU cores (number)</td><td>Up to 72 cores</td></tr> <tr> <td>CPU LPDDR5X bandwidth (GB/s)</td><td>Up to 500GB/s</td></tr> <tr> <td>GPU HBM bandwidth (GB/s)</td><td>4TB/s HBM3 4.9TB/s HBM3e</td></tr> <tr> <td>NVLink-C2C bandwidth (GB/s)</td><td>900GB/s total, 450GB/s per direction</td></tr> <tr> <td>CPU LPDDR5X capacity (GB)</td><td>Up to 480GB</td></tr> <tr> <td>GPU HBM capacity (GB)</td><td>96GB HBM3 144GB HBM3e</td></tr> <tr> <td>PCIe Gen 5 Lanes</td><td>64x</td></tr> </table> (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).) The Accused Products perform <i>a method of executing computations representing an artificial neural network on a computer system comprising at least one central processing unit (CPU), a processing unit, a first memory partition, and a second memory partition</i> by implementing Nvidia’s CUDA (Compute Unified Device Architecture) parallel computing platform, including CUDA toolkits and drivers, and deploying that platform on a <i>computer system</i> to perform <i>computations</i>. As previously discussed, <i>supra</i>, in addition to the Grace Hopper Superchip (GH200), the Accused Products include Nvidia GPUs with the Hopper GPU architecture. All of these GPUs are used to <i>perform[] a sequence of computations representing an artificial neural network</i>. The aforementioned Nvidia GPU architectures are compatible with Nvidia’s proprietary CUDA platform for parallel computing, which is “a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).” CUDA further includes the CUDA Toolkit, which “includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.” An exemplary CUDA library specialized for acceleration is cuDNN (CUDA Deep Neural Network), “a GPU-accelerated library of primitives for <i>deep neural networks</i>. <i>cuDNN</i> provides highly tuned	Feature	Description	Grace CPU cores (number)	Up to 72 cores	CPU LPDDR5X bandwidth (GB/s)	Up to 500GB/s	GPU HBM bandwidth (GB/s)	4TB/s HBM3 4.9TB/s HBM3e	NVLink-C2C bandwidth (GB/s)	900GB/s total, 450GB/s per direction	CPU LPDDR5X capacity (GB)	Up to 480GB	GPU HBM capacity (GB)	96GB HBM3 144GB HBM3e	PCIe Gen 5 Lanes	64x
Feature	Description																
Grace CPU cores (number)	Up to 72 cores																
CPU LPDDR5X bandwidth (GB/s)	Up to 500GB/s																
GPU HBM bandwidth (GB/s)	4TB/s HBM3 4.9TB/s HBM3e																
NVLink-C2C bandwidth (GB/s)	900GB/s total, 450GB/s per direction																
CPU LPDDR5X capacity (GB)	Up to 480GB																
GPU HBM capacity (GB)	96GB HBM3 144GB HBM3e																
PCIe Gen 5 Lanes	64x																

Key Features	Accused Products
	implementations for standard routines such as forward and backward convolution, attention, matmul, pooling, and normalization.” CUDA Zone <u>CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).</u> With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs. In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords. The <u>CUDA Toolkit</u> from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime. (See https://developer.nvidia.com/cuda-zone (emphasis added).) NVIDIA cuDNN The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of <u>primitives for deep neural networks.</u> cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, attention, matmul, pooling, and normalization. (See https://developer.nvidia.com/cudnn (emphasis added).) As illustrated in the table below, Nvidia’s GPUs implementing the Hopper architecture (“Supported NVIDIA Hardware”) support Nvidia’s CUDA platform for parallel computing and CUDA libraries such as cuDNN for GPU accelerated. As shown in the table below, columns for “cuDNN Package” and “CUDA Compute Capability” (represented by a version number or SM version, which “identifies the

Key Features	Accused Products
	features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU”) identify the supported functionality the Accused Product GPUs support via their Nvidia GPU architecture. For cuDNN, as an example, package 9.5.1 for CUDA 12.x (e.g., CUDA 12.0 and later) supports Nvidia GPUs with CUDA Compute Capability numbers 9.0, 8.9, 8.6, 8.0, 7.5, 7.0, 6.1, 6.0, and 5.0 including the Accused Products with the Hopper GPU architecture. Nvidia notes that “the recommended configuration for GPUs Volta or later [such as Hopper GPUs] is cuDNN 9.5.1 with CUDA 12.6. For GPUs prior to Volta (that is, Pascal and Maxwell), the recommended configuration is cuDNN 9.5.1 with CUDA 11.8.” As shown below, Hopper GPUs have a CUDA compute Capability of 9.0, which supports CUDA 12 and 11 for cuDNN 9.5.1. In addition, because “CUDA is backward compatible, existing CUDA applications can continue to be used with newer CUDA versions.” Support Matrix GPU, CUDA Toolkit, and CUDA Driver Requirements The following sections highlight the compatibility of NVIDIA cuDNN versions with the various supported NVIDIA CUDA Toolkit, CUDA driver, and NVIDIA hardware versions.

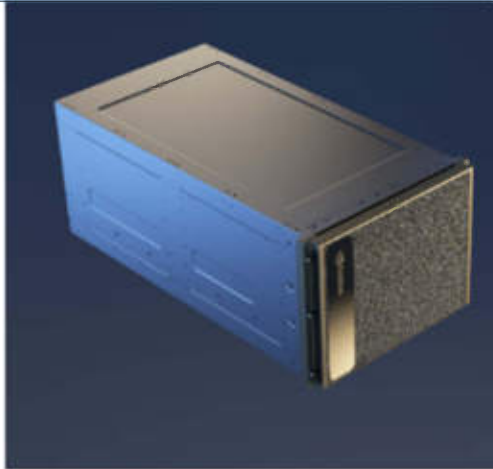
Key Features	Accused Products						
	Supported NVIDIA Hardware and CUDA Version						
	cuDNN Package [1]	CUDA Toolkit Version	Supports static linking? [2]	NVIDIA Driver Version for Linux	NVIDIA Driver Version for Windows	CUDA Compute Capability	Supported NVIDIA Hardware
	cuDNN 9.5.1 for CUDA 12.x	› 12.6	Yes	≥525.60.13	≥527.41	› 9.0 [3]	› NVIDIA Hopper [3]
		› 12.5				› 8.9 [3]	
		› 12.4				› 8.6	› NVIDIA Ada Lovelace architecture [3]
		› 12.3				› 8.0	
		› 12.2				› 7.5	
		› 12.1				› 7.0	› NVIDIA Ampere architecture
		› 12.0				› 6.1	
						› 6.0	› NVIDIA Turing
						› 5.0	› NVIDIA Volta
							› NVIDIA Pascal
							› NVIDIA Maxwell

Key Features	Accused Products						
	cuDNN 9.5.1 for CUDA 11.x	11.8	Yes	>= 450.80.02	>=452.39	› 9.0 [3] › 8.9 [3] › 8.6 › 8.0 › 7.5 › 7.0 › 6.1 › 6.0 › 5.0	› NVIDIA Hopper [3] › NVIDIA Ada Lovelace architecture [3] › NVIDIA Ampere architecture › NVIDIA Turing › NVIDIA Volta › NVIDIA Pascal › NVIDIA Maxwell
(See https://docs.nvidia.com/deeplearning/cudnn/latest/reference/support-matrix.html#gpu-cuda-toolkit-and-cuda-driver-requirements (emphasis added).) › If we build our CUDA application using CUDA 11.0, can it continue to be used with newer NVIDIA drivers (such as CUDA 11.1/R455, 11.x etc.)? Or is it only the other way around? Drivers have always been backwards compatible with CUDA. This means that a CUDA 11.0 application will be compatible with R450 (11.0), R455 (11.1) and beyond. CUDA applications typically statically include all the libraries (for example cudart, CUDA math libraries such as cuBLAS, cuFFT) they need, so they should work on new drivers or CUDA Toolkit installations. In other words, since CUDA is backward compatible, existing CUDA applications can continue to be used with newer CUDA versions. (See https://docs.nvidia.com/deploy/cuda-compatibility/ (emphasis added).)							

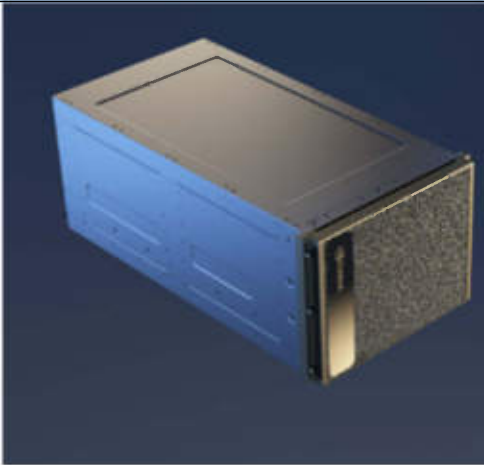
Key Features	Accused Products
	The Accused Products include “CUDA-Enabled Datacenter Products” including “Nvidia Data Center Products” (e.g., “NVIDIA H100”). As shown below, the GPU and “Compute Capability” are displayed for the Nvidia GPU products. As an example, the Nvidia H100 GPU (Nvidia H100 Tensor Core GPU based on the Nvidia Hopper architecture) has a compute capability of 9.0. As previously stated, the Accused Products include Nvidia’s GPUs implementing the Hopper architecture. All of the accused Nvidia GPUs are compatible with CUDA and GPU-acceleration libraries such as cuDNN, that implement infringing functionality as shown throughout this claim chart, <i>infra</i>. The accused Nvidia GPUs and their respective CUDA compute capability for compatibility with Nvidia’s CUDA and CUDA-based GPU-acceleration libraries, such as cuDNN, include the following: Hopper: H100 (9.0), H200 (9.0), GH200 (9.0). (See https://docs.nvidia.com/grace-perf-tuning-guide/index.html/; https://arnon.dk/matching-sm-architectures-arch-and-gencode-for-various-nvidia-cards/.) In addition to cuDNN, Nvidia’s other CUDA-specialized GPU-acceleration libraries include CUTLASS (CUDA Templates for Linear Algebra Subroutines and Solvers, a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication), DALI (NVIDIA Data Loading Library, a library for decoding and augmenting images, videos, and speech to accelerate deep learning applications), cuBLAS (Basic Linear Algebra on NVIDIA GPUs, a GPU-accelerated library for accelerating AI and HPC applications), cuTENSOR (Tensor Linear Algebra on NVIDIA GPUs, high-performance tensor computations and accelerate deep learning training and inference), NPP (NVIDIA Performance Primitives, a library of over 5,000 primitives for image and signal processing), cuSPARSE (GPU accelerated basic linear algebra routines for handling sparse matrices), and cuSOLVER (GPU accelerated library for decompositions and linear system solutions for both dense and sparse matrices). These CUDA libraries are also used to perform methods and implement systems for GPU acceleration and neural networks. (See https://nvidia.github.io/cutlass/; https://developer.nvidia.com/dali; https://developer.nvidia.com/cublas; https://developer.nvidia.com/cutensor; https://developer.nvidia.com/npp; https://docs.nvidia.com/cuda/cusparse/; https://docs.nvidia.com/cuda/cusolver/index.html.) Furthermore, Nvidia’s CUDA-X, built on top of CUDA, is a collection of “GPU-accelerated microservices and libraries for AI.” (See

Key Features	Accused Products
	https://www.nvidia.com/en-us/technologies/cuda-x/.) Nvidia also offers the CUDA Toolkit and SDK Manager for developing GPU-accelerated applications. (See https://developer.nvidia.com/cuda-toolkit; https://developer.nvidia.com/sdk-manager.) (See https://docs.nvidia.com/deploy/cuda-compatibility/index.html/; https://docs.nvidia.com/cuda/cublas/index.html/ (cuBLAS); https://developer.nvidia.com/cutensor/ (cuTENSOR); https://developer.nvidia.com/npp/ (NVIDIA Performance Primitives); https://docs.nvidia.com/cuda/cusparse/index.html (cuSPARSE); https://docs.nvidia.com/cuda/cusolver/index.html (cuSOLVER).) In addition to Nvidia’s GPUs, the Accused Products include Nvidia’s supercomputers, data centers, servers, workstations, and cloud services that embody computer systems specifically designed for GPU acceleration, including: Nvidia DGX products and Nvidia HGX products. For example, Nvidia’s DGX (Deep GPU Xceleration) is Nvidia’s “ready to use” line of premade and complete supercomputers that implement Nvidia GPUs for accelerating workloads such as AI training and inference. Nvidia DGX supercomputers include the DGX H200 and DGX H100 (8x H200 or H100 GPUs), as well as the and DGX GH200 AI Supercomputer (32x GH200 Superchips equating 256x H200 GPUs) and the soon-to-be-released DGX B200 and DGX B100 (soon-to-be-released Blackwell GPU architecture). Nvidia’s DGX supercomputers include the DGX Operating System, “an operating system optimized for AI workloads,” and “NVIDIA Base Command” which “powers the DGX platform” and “leverage[s] the best of NVIDIA software innovation.”

Key Features	Accused Products
	NVIDIA DGX: Pre-Built AI Powerhouse NVIDIA DGX is a ready to use solution—a fully pre-configured system that includes the necessary hardware, software, and networking, making it an out-of-the-box solution for AI tasks. • Use Case: Ideal for organizations looking for a plug-and-play solution to handle AI model training and deep learning. It's excellent for enterprises that need minimal setup and want to avoid complex hardware configurations. • Flexibility: DGX is more of a fixed configuration. You can't easily adjust or modify it like HGX, but this also means it's easier to deploy. • Key Features: It comes with a full NVIDIA software stack, including NVIDIA Base Command and NVIDIA AI libraries for managing and deploying AI workloads seamlessly. • Who Should Use It: Companies or research institutions that want a simple solution for AI, without needing to customize hardware extensively. Think of it as a premium AI system that delivers fast results with minimal effort. (See https://www.server-parts.eu/post/nvidia-ai-platform-dgx-hgx-egx-agx-comparison; see also https://resources.nvidia.com/en-us-dgx-platform/nvidia-dgx-platform-solution-overview-web-us/ (Nvidia's "DGX platform integrates AI software and purpose-built hardware in a comprehensive solution for AI development.").)

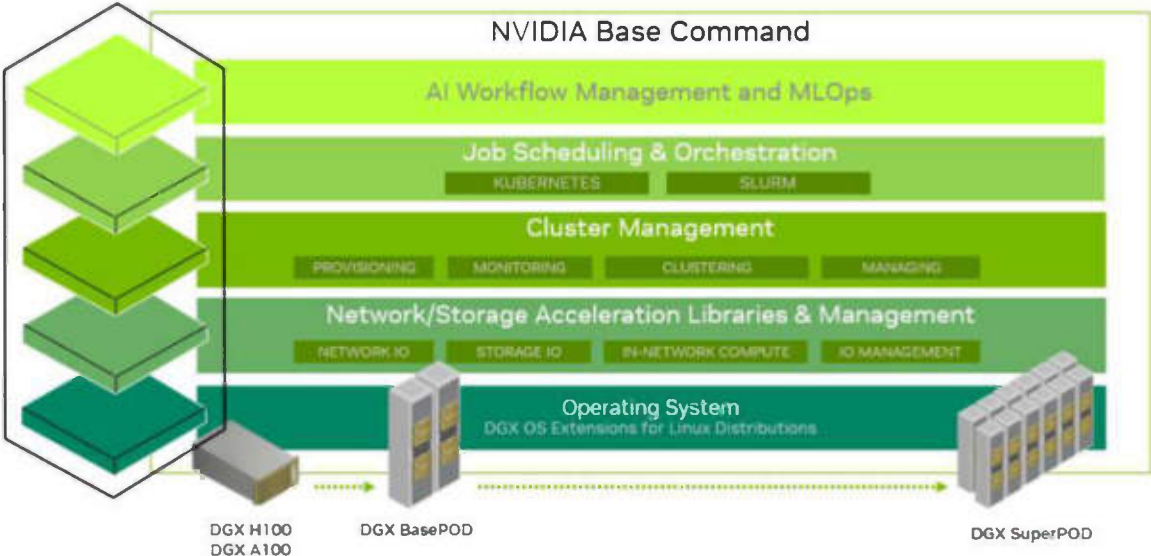
Key Features	Accused Products														
	Datasheet  NVIDIA. NVIDIA DGX H200 The gold standard for AI infrastructure. <table> <tr> <th colspan="2">Specifications</th></tr> <tr> <td>GPU</td><td>8x NVIDIA H200 Tensor Core GPUs, with 141GB of GPU memory each</td></tr> <tr> <td>GPU memory</td><td>1,128GB total</td></tr> <tr> <td>Performance</td><td>32 petaFLOPS FP8</td></tr> <tr> <td>NVIDIA NVSwitch™</td><td>4x</td></tr> <tr> <td>System power usage</td><td>10.2kW max*</td></tr> <tr> <td>CPU</td><td>Dual Intel® Xeon® Platinum 8480C Processors 112 Cores total, 2.00 GHz (Base), 3.80 GHz (Max Boost)</td></tr> </table> (See https://resources.nvidia.com/en-us-dgx-systems/dgx-h200-datasheet/.) 	Specifications		GPU	8x NVIDIA H200 Tensor Core GPUs, with 141GB of GPU memory each	GPU memory	1,128GB total	Performance	32 petaFLOPS FP8	NVIDIA NVSwitch™	4x	System power usage	10.2kW max*	CPU	Dual Intel® Xeon® Platinum 8480C Processors 112 Cores total, 2.00 GHz (Base), 3.80 GHz (Max Boost)
Specifications															
GPU	8x NVIDIA H200 Tensor Core GPUs, with 141GB of GPU memory each														
GPU memory	1,128GB total														
Performance	32 petaFLOPS FP8														
NVIDIA NVSwitch™	4x														
System power usage	10.2kW max*														
CPU	Dual Intel® Xeon® Platinum 8480C Processors 112 Cores total, 2.00 GHz (Base), 3.80 GHz (Max Boost)														

Key Features	Accused Products
	Datasheet  NVIDIA. NVIDIA H200 Tensor Core GPU Supercharging AI and HPC workloads.  Higher Performance With Larger, Faster Memory The NVIDIA H200 Tensor Core GPU supercharges generative AI and high-performance computing (HPC) workloads with game-changing performance and memory capabilities. Based on the NVIDIA Hopper™ architecture, the NVIDIA H200 is the first GPU to offer 141 gigabytes (GB) of HBM3e memory at 4.8 terabytes per second (TB/s)—that's nearly double the capacity of the NVIDIA H100 Tensor Core GPU with 1.4X more memory bandwidth. The H200's larger and faster memory accelerates generative AI and large language models, while advancing scientific computing for HPC workloads with better energy efficiency and lower total cost of ownership. (See https://nvdam.widen.net/s/nb5zzzsjdf/hpc-datasheet-sc23-h200-datasheet-3002446/.) Key Features ➤ 141GB of HBM3e GPU memory ➤ 4.8TB/s of memory bandwidth ➤ 4 petaFLOPS of FP8 performance ➤ 2X LLM inference performance ➤ 110X HPC performance

Key Features	Accused Products														
	Datasheet  NVIDIA. NVIDIA DGX H100 The gold standard for AI infrastructure. <table> <tr> <th colspan="2">Specifications</th></tr> <tr> <td>GPU</td><td>8x NVIDIA H100 Tensor Core GPUs</td></tr> <tr> <td>GPU memory</td><td>640GB total</td></tr> <tr> <td>Performance</td><td>32 petaFLOPS FP8</td></tr> <tr> <td>NVIDIA® NVSwitch™</td><td>4x</td></tr> <tr> <td>System power usage</td><td>10.2kW max</td></tr> <tr> <td>CPU</td><td>Dual Intel® Xeon® Platinum 8480C Processors 112 Cores total, 2.00 GHz (Base), 3.80 GHz (Max Boost)</td></tr> </table> (See https://resources.nvidia.com/en-us-dgx-systems/ai-enterprise-dgx/.) 	Specifications		GPU	8x NVIDIA H100 Tensor Core GPUs	GPU memory	640GB total	Performance	32 petaFLOPS FP8	NVIDIA® NVSwitch™	4x	System power usage	10.2kW max	CPU	Dual Intel® Xeon® Platinum 8480C Processors 112 Cores total, 2.00 GHz (Base), 3.80 GHz (Max Boost)
Specifications															
GPU	8x NVIDIA H100 Tensor Core GPUs														
GPU memory	640GB total														
Performance	32 petaFLOPS FP8														
NVIDIA® NVSwitch™	4x														
System power usage	10.2kW max														
CPU	Dual Intel® Xeon® Platinum 8480C Processors 112 Cores total, 2.00 GHz (Base), 3.80 GHz (Max Boost)														

Key Features	Accused Products
	NVIDIA Announces DGX GH200 AI Supercomputer New Class of AI Supercomputer Connects 256 Grace Hopper Superchips Into Massive, 1-Exaflop, 144TB GPU for Giant Models Powering Generative AI, Recommender Systems, Data Processing May 28, 2023  (See https://nvidianews.nvidia.com/news/nvidia-announces-dgx-gh200-ai-supercomputer; https://www.naddod.com/blog/nvidia-dgx-gh200-ai-supercomputer; https://www.zhaocs.info/wp-content/uploads/2024/04/dgx-gh200-whitepaper.pdf; see also https://www.nvidia.com/en-gb/data-center/dgx-b200/ (DGX GB200 outfitted with soon-to-be-released Grace Blackwell Superchips).)

Key Features	Accused Products
	Furthermore, Nvidia offers DGX AI data center and server systems that stack multiple DGX supercomputers in powerful turnkey solutions. For instance, “DGX SuperPOD” is an “AI data center infrastructure” that offers leadership-class accelerated infrastructure and scalable performance for the most challenging AI workloads, with industry-proven results.” The DGX SuperPOD system can be outfitted with stacks of Nvidia DGX supercomputers such as Nvidia DGX H200 and DGX H100 (supercomputers with H200/H100 GPUs), as well as the soon-to-be-released DGX GB200 and DGX B200 (soon-to-be-released Blackwell GPU architecture). Like the DGX supercomputers, the DGX SuperPOD includes Nvidia Base Command and Nvidia AI Enterprise as well as “White Glove Services.”  NVIDIA DGX SuperPOD Purpose-built for the unique demands of AI. * * * * * The World's First Turnkey AI Data Center The NVIDIA DGX SuperPOD™ is an AI data center infrastructure that enables IT to deliver performance—without compromise—for every user and workload. As part of the <u>NVIDIA DGX™</u> platform, DGX SuperPOD offers leadership-class accelerated infrastructure and scalable performance for the most challenging AI workloads—with industry-proven results. (See https://www.nvidia.com/en-us/data-center/dgx-superpod/; see also https://resources.nvidia.com/en-us-dgx-platform/dgx-superpod-gb200-datasheet (SuperPOD with DGX GB200 outfitted with soon-to-be-released Grace Blackwell Superchips; https://resources.nvidia.com/en-us-dgx-systems/dgx-ai-4 (SuperPOD with DGX H200 or B200 (soon-to-be-released Blackwell GPU architecture))); https://www.naddod.com/blog/how-nvidia-builds-ai-supercomputers-with-superpod.)

Key Features	Accused Products
	In addition, Nvidia’s DGX BasePOD also stacks multiple DGX supercomputers. DGX BasePOD also “provides the underlying infrastructure and software to accelerate deployment and execution of these AI workloads” and includes Nvidia Base Command and Nvidia AI Enterprise. Like the DGX SuperPOD, DGX BasePOD can be outfitted with stacks of Nvidia DGX supercomputers such as Nvidia DGX H200 and DGX H100, as well as the soon-to-be-released DGX GB200 and DGX B200. Figure 2. NVIDIA Base Command features and capabilities with DGX BasePOD  (See https://resources.nvidia.com/en-us-dgx-systems/nvidia-dgx-basepod; see also https://docs.nvidia.com/dgx-basepod/reference-architecture-infrastructure-foundation-enterprise-ai/latest/_downloads/487a2093a4564bef969f38abba12a1f5/ra-11127-001-dbphb100-referencearch.pdf; https://www.nvidia.com/en-us/data-center/dgx-basepod/; https://docs.nvidia.com/dgx-basepod/reference-architecture-infrastructure-foundation-enterprise-ai/latest/index.html; https://docs.nvidia.com/dgx-basepod/reference-architecture-infrastructure-foundation-enterprise-ai/latest/dgx-basepod-overview.html.)

Key Features	Accused Products
	In addition to DGX, Nvidia also offers the HGX (Hyperscale Graphics EXtension) supercomputing platform line that includes Nvidia GPUs and NVSwitch baseboard. HGX is a “modular platform designed for more customizable AI and HPC infrastructure.” “HGX is . . . the baseboard used in the Nvidia DGX system” that is “tweak[able] . . . as needed” for OEMs for customized supercomputer solutions powered by Nvidia GPUs. Nvidia HGX supercomputing platforms include HGX H200 and HGX H100 (8x and 4x H200 or H100 GPUs), and the soon-to-be-released HGX B200 and HGX B100 (soon-to-be-released Blackwell GPU architecture).

Key Features	Accused Products
	NVIDIA HGX: Customizable Supercomputing for AI and HPC NVIDIA HGX is a modular platform designed for more customizable AI and HPC infrastructure. Unlike DGX, HGX isn't a complete system; instead, it's a set of components (like GPUs, networking, and interconnects) that can be built to your exact requirements. • Use Case: Perfect for organizations that need to scale their AI supercomputing or HPC workloads. It's highly flexible, allowing you to integrate as many GPUs as needed, customize networking with InfiniBand, and manage more complex data center requirements. • Flexibility: HGX allows you to build out your infrastructure based on your own custom configurations. For example, you can decide how many GPUs you want to use, whether to employ NVLink or PCIe interconnects, and how to optimize power and cooling. • Key Features: Supports up to 16 GPUs per server and can scale across large data centers. It's designed to be flexible, allowing integration with existing infrastructures, especially when GPU scalability and memory bandwidth are critical. • Who Should Use It: Large enterprises or cloud providers that need scalable AI infrastructure or high-performance computing (HPC) for massive data processing or AI model training. You get more control over your system's configuration and performance. (See https://www.server-parts.eu/post/nvidia-ai-platform-dgx-hgx-egx-agx-comparison.)

Key Features	Accused Products
	 NVIDIA. NVIDIA HGX H100 and HGX H200 The world's leading AI computing platform.  Purpose-Built for AI and High-Performance Computing AI, complex simulations, and massive datasets require multiple GPUs with extremely fast interconnections and a fully accelerated software stack. The NVIDIA HGX™ AI supercomputing platform brings together the full power of NVIDIA GPUs, NVIDIA NVLink™, NVIDIA networking, and fully optimized AI and high-performance computing (HPC) software stacks to provide the highest application performance and drive the fastest time to insights. Key Features NVIDIA HGX H100 and HGX H200 > Transformer Engine > Fourth-generation NVIDIA NVLink (See https://nvdam.widen.net/s/5kgbjq2v2t/hpc-hgx-h100-datasheet-nvidia-web/; https://www.nvidia.com/en-us/data-center/hgx/.) In addition to DGX and HGX, Nvidia AI Enterprise is also deployed on Nvidia-Certified Systems “servers, workstations and laptop certified to accelerate AI workloads” utilizing Nvidia GPUs; and on cloud platforms utilizing Nvidia GPUs. (See https://www.nvidia.com/en-us/data-center/products/ai-enterprise/.) Nvidia’s NVIDIA-Certified Systems “are rigorously tested and validated to deliver enterprise-grade performance, manageability, scalability, and security for <i>AI and accelerated computing</i>.” (See https://www.nvidia.com/en-us/data-center/products/certified-systems/ (emphasis added).) An exemplary list of “NVIDIA-Certified Systems - Data Center Servers” is shown below. Columns for “Partner,” “Data Center Server,” and “Supported NVIDIA GPUs” are shown below. As an example,

Key Features	Accused Products
	partner “Aivres” for data center servers “KR6288E2” and “KR6288X2” is certified for the “HGX H200 8-GPU, HGX H100 8-GPU” Nvidia Supercomputers and GPUs, which implement the Hopper Nvidia GPU architecture. “NVIDIA-Certified systems are tested with each supported NVIDIA GPU,” and “[t]he supported GPUs are certified for Data Center Servers for every generally available GPU version. For example, every system validated for the A100 GPU is certified for both the A100 40 GB and A100 80 GB GPU versions.” An exemplary table of “NVIDIA-Certified Systems - Data Center Servers” is shown below, highlighting Hopper-architecture GPUs.

Key Features

Accused Products

7. List of NVIDIA-Certified Systems - Data Center Servers

The following systems have been validated by NVIDIA as NVIDIA-Certified Systems - Data Center Servers with the supported NVIDIA GPUs listed in the following table. The GPUs listed below are certified for Data Center Servers for every generally available GPU version. For example, every system validated for the A100 GPU is certified for both the A100 40 GB and A100 80 GB GPU versions.

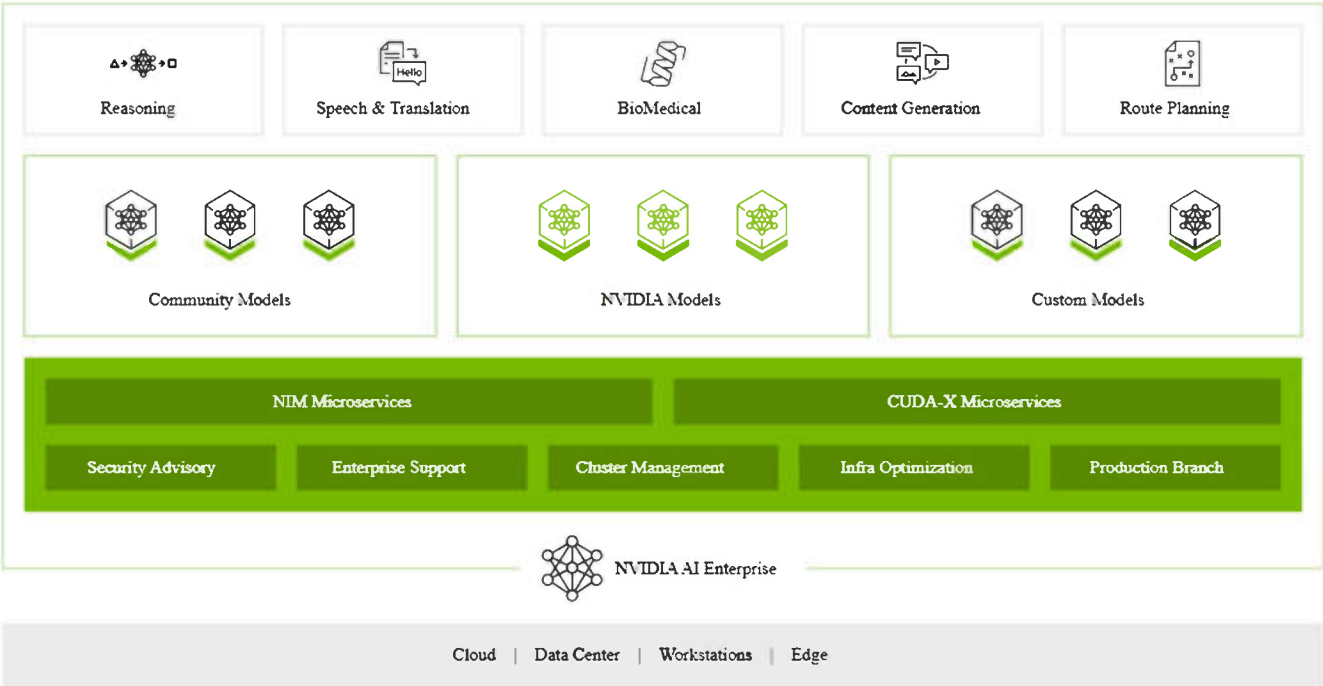
Partner	Data Center Server	Supported NVIDIA GPUs ¹
Aetna	AIS-D422-A1	L4, A2, T4
Advantech	SKY-640V2	A100, A30
Akres	KR6288FZ	HGX H200 8-GPU, HGX H100 8-GPU
Akres	KR6288XZ	HGX H200 8-GPU, HGX H100 8-GPU
Altos Computing	BrainSphere R6M5 F5	RTX A6000, A40
AMAX	AceleMax DGS- 140W	A100, A30, A40
AMAX	AceleMax DGS- 214A	A100, A30, A40
AMAX	AceleMax DGS- 224AS	HGX A100 4-GPU
AMAX	AceleMax DGS- 280W	A100, A30, A40
AMAX	AceleMax DGS- 410W	A100, A30, A40
AMAX	AceleMax DGS- 428A	A100, A30, A40
AMAX	AceleMax DGS- 428AS	HGX A100 8-GPU
AMAX	AceleMax DGS- 428WS	HGX A100 8-GPU
AMAX	AceleMax DL- E440W	A100, A30, A40
ASRock Rack	TU4G-ROME	A100, A30
ASRock Rack	ZU-Open19-m3.xlarge.x86	A100
ASRock Rack	ZU-Open19-n3.xlarge.x86	A100
ASRock Rack	TU4G-ROME/2T	A100, A30
ASRock Rack	4U8G-ROME2/2T	A100
ASRock Rack	4U10G-ICK2/2T	A100
ASRock Rack	4U10G-ROME2/2T	A100
ASUS	ESC4000-E10 / ESC4000-E10S	A100
ASUS	ESC4000A-E10	A100, A30, A40, RTX A6000
ASUS	ESC8000-E11	L40, L4
ASUS	ESC8000-E11P	L40
ASUS	ESC8000A-E11	A100, A40, A30
ASUS	ESC8000A-E12	H100
ASUS	ESC-NB-E11	HGX H100 8-GPU
ASUS	ESC-NB-E11V	HGX H200 8-GPU
ASUS	ESC-NBA-E12	HGX H100 8-GPU

Atipia Technologies	Altezza SX227-32G3	A100, A40
Atipia Technologies	Altezza SX427-32G10	A100, A40
Atipia Technologies	Altezza SX427-32G8	A100, A40
Atipia Technologies	Altezza SX427-32G8SXMM4	HGX A100 8-GPU
Atipia Technologies	Procyon SE218-8G4	A100, T4
Atipia Technologies	Procyon SE218-8G8	A100, A30
Atipia Technologies	Procyon SE228-16G4SXMM4	HGX A100 4-GPU
Atipia Technologies	Procyon SE228-16G8	A100, A40, A30
Atipia Technologies	Procyon SE228-32G3	A100
Atipia Technologies	Procyon SE428-32G10	A100, A30
Atipia Technologies	Procyon SC428-32G8	A100, A30, A40
Atipia Technologies	Procyon SE428-32G8SXMM4	HGX A100 8-GPU
ATOS	BullSequana SA20G	A100
ATOS	BullSequana X410-A5	A100
Boston	ANNA Ampere L1	HGX A100 4-GPU
Boston	ANNA Ampere M1	A100, A40, A30
Boston	ANNA Ampere S2	A100, A40
Boston	ANNA Ampere XL1	A100, A40, A30
BOXX	PAXX P4G	A40
Cisco	UCS C220 M7²	L4
Cisco	UCS C240 M6 Rack Server²	A100, A30
Cisco	UCS C240 M6²	L4
Cisco	UCS C240 M7²	L40, L4, A100
Cisco	UCS C240 M7²	H100, H100 NVL, L40S
Cisco	UCS C245 M6²	H100 NVL, L40S, L40, L4
Cisco	UCS-X²	A100
Cisco	UCS X210c M6²	A40
Cisco	UCS X210c M7²	H100, L40S, L4Q, L4, A100
Cisco	UCS X210c M7²	H100 NVL
Cisco	UCS X215c M6²	H100 NVL, L40S, L4
Colfax	Colfax CX2480a-EX8 2U Rackmount Server	HGX A100 4-GPU
Colfax	Colfax CX41060a-XXB 4U Rackmount Server	A100, A30, A40
Colfax	Colfax CX4860a-EX8 4U Rackmount Server	A100, A30, A40
Colfax	Colfax CX4880a-EX8 4U Rackmount Server	HGX A100 8-GPU
Colfax	Colfax CX4880a-XXB 4U Rackmount Server	HGX A100 8-GPU
Colfax	Colfax ProEdge SXP9000 Workstation	A100, A30, A40

(See <https://docs.nvidia.com/certification-programs/nvidia-certified-systems/index.html#nvidia-certified-systems-list> (emphasis added).)

Key Features	Accused Products																																			
	In addition, Nvidia’s NVIDIA-Certified Edge Systems “run[] accelerated applications outside of a traditional data center environment. NVIDIA-Certified Edge Systems include both Enterprise Edge systems, designed to be deployed in controlled environments like retail stores or manufacturing facilities, and Industrial Edge systems, with rugged designs for operating in a wide range of industrial environments, such as those with elevated temperature.” An exemplary table for “NVIDIA-Certified Systems - Edge Systems” is shown below. <u>11. List of NVIDIA-Certified Systems - Edge Systems</u> The following servers and workstations have been validated by NVIDIA as NVIDIA-Certified Systems - Edge Systems. <table><tr><th>Partner</th><th>Server / Workstation</th><th>Supported NVIDIA GPUs ¹</th><th>Enterprise Edge Server ²</th><th>Industrial Edge Workstation ³</th></tr><tr><td>Aaeon</td><td><u>Boxer-6843-ADS</u></td><td>L4</td><td></td><td>Yes</td></tr><tr><td>Aaeon</td><td><u>BOXER-8332</u></td><td>A2</td><td></td><td>Yes</td></tr><tr><td>Aaeon</td><td><u>Maxer-2100-Q670</u></td><td>L4, RTX 6000 Ada</td><td></td><td>Yes</td></tr><tr><td>Advantech</td><td><u>ACP-2020G</u></td><td>A2</td><td></td><td>Yes</td></tr><tr><td>Advantech</td><td><u>ACP-4340</u></td><td>RTX 4000 Ada SFF</td><td></td><td>Yes</td></tr><tr><td>Advantech</td><td><u>ACP-4340+AIMB-788</u></td><td>RTX 4000 Ada</td><td></td><td>Yes</td></tr></table>	Partner	Server / Workstation	Supported NVIDIA GPUs ¹	Enterprise Edge Server ²	Industrial Edge Workstation ³	Aaeon	<u>Boxer-6843-ADS</u>	L4		Yes	Aaeon	<u>BOXER-8332</u>	A2		Yes	Aaeon	<u>Maxer-2100-Q670</u>	L4, RTX 6000 Ada		Yes	Advantech	<u>ACP-2020G</u>	A2		Yes	Advantech	<u>ACP-4340</u>	RTX 4000 Ada SFF		Yes	Advantech	<u>ACP-4340+AIMB-788</u>	RTX 4000 Ada		Yes
Partner	Server / Workstation	Supported NVIDIA GPUs ¹	Enterprise Edge Server ²	Industrial Edge Workstation ³																																
Aaeon	<u>Boxer-6843-ADS</u>	L4		Yes																																
Aaeon	<u>BOXER-8332</u>	A2		Yes																																
Aaeon	<u>Maxer-2100-Q670</u>	L4, RTX 6000 Ada		Yes																																
Advantech	<u>ACP-2020G</u>	A2		Yes																																
Advantech	<u>ACP-4340</u>	RTX 4000 Ada SFF		Yes																																
Advantech	<u>ACP-4340+AIMB-788</u>	RTX 4000 Ada		Yes																																

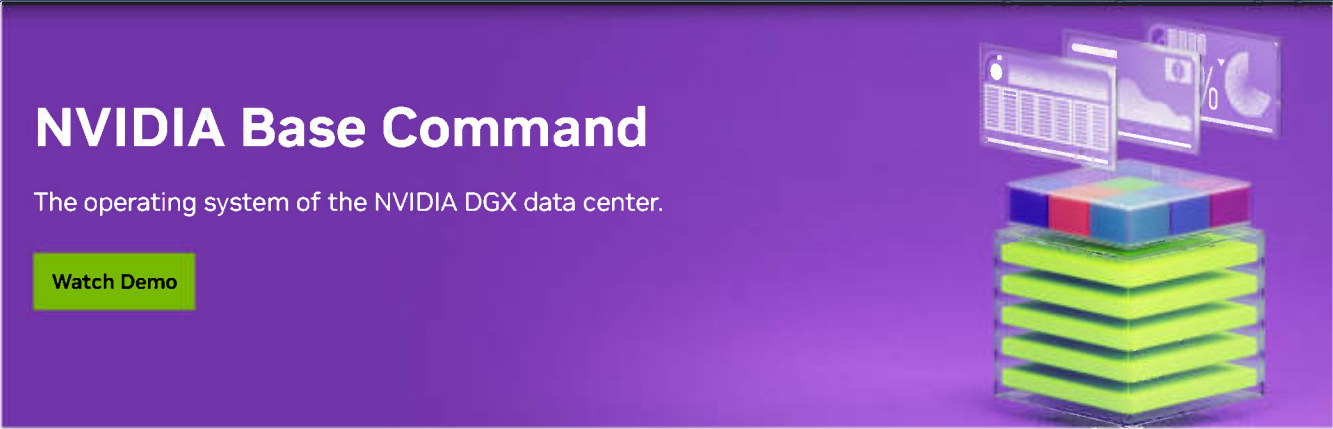
Key Features	Accused Products				
	Cisco	<u>UCS X210c M7</u>	H100, L40S, L40, L4, A100	Yes	
	Cisco	<u>UCS X215c M8</u>	H100 NVL, L40S, L4	Yes	
	Cisco	<u>UCS C220 M7</u>	L4	Yes	
	Cisco	<u>UCS C225 M8</u>	L4	Yes	
	Cisco	<u>UCS C240 M6</u>	L4	Yes	
	Cisco	<u>UCS C240 M7</u>	H100 NVL, H100, L40S, L40, L4	Yes	
	Cisco	<u>UCS C245 M8</u>	H100 NVL, L40S, L40, L4	Yes	
(See https://docs.nvidia.com/certification-programs/nvidia-certified-systems/index.html#nvidia-certified-systems-edge-list/ (emphasis added).) In addition to the aforementioned Nvidia GPUs, supercomputers, datacenters, servers, and hardware, the Accused Products further include Nvidia’s software, platforms, and services for accelerated computing, including Nvidia’s proprietary CUDA platform for parallel computing and Nvidia AI Enterprise. Nvidia AI Enterprise is an operating system for enterprise AI applications that provides accelerated computing services based on Nvidia GPUs and using the CUDA platform, including Nvidia NIM Microservices (which includes Nvidia GPUs and utilizes the CUDA platform and, e.g., libraries cuDNN, cuBLAS, DALI, TensorRT and TensorRT-LLM) and the CUDA-X Microservices.					

Key Features	Accused Products
	 The diagram illustrates the NVIDIA AI Enterprise architecture. At the top, five application areas are shown: Reasoning, Speech & Translation, BioMedical, Content Generation, and Route Planning. Below these are three model categories: Community Models, NVIDIA Models, and Custom Models, each represented by three model icons. The core of the architecture consists of NIM Microservices and CUDA-X Microservices. Below these are five support and management components: Security Advisory, Enterprise Support, Cluster Management, Infra Optimization, and Production Branch. The entire stack is labeled NVIDIA AI Enterprise and is supported across four environments: Cloud, Data Center, Workstations, and Edge. (See https://www.nvidia.com/en-us/data-center/products/ai-enterprise/; see also https://docs.nvidia.com/ai-enterprise/index.html#overview.)

Key Features	Accused Products
	NVIDIA NIM for optimized AI inference NVIDIA NIM is designed to bridge the gap between the complex world of AI development and the operational needs of enterprise environments, enabling 10-100X more enterprise application developers to contribute to AI transformations of their companies. The diagram illustrates the NVIDIA NIM architecture as a multi-layered stack. At the top, a green hexagonal container represents the core inference engine, containing components like TensorRT and TensorRT-LLM, cuBLAS, cuDNN, in-flight batching, memory optimization, and FP8 quantization. Below this, a blue square layer represents the optimized model and customization cache, including single GPU, multi-GPU, and multi-node configurations. The bottom layer is a green square representing the cloud native stack, which includes GPU operators, network operators, and enterprise management (health checks, identity, metrics, monitoring, secrets management). This stack is built on top of the NVIDIA CUDA hardware, which is shown as a green square at the base. To the left of the stack, a list of industry standard APIs (Text, Speech, Image, Video, 3D, Biology) is shown, along with the Triton Inference Server and the Cloud Native Stack. To the right, the TensorRT and TensorRT-LLM components are listed, along with the Optimization Cache and the NVIDIA CUDA hardware. <i>Figure 1. NVIDIA NIM is a containerized inference microservice including industry-standard APIs, domain-specific code, optimized inference engines, and enterprise runtime</i> (See https://developer.nvidia.com/blog/nvidia-nim-offers-optimized-inference-microservices-for-deploying-ai-models-at-scale/.) The Nvidia AI Enterprise cloud-based “End-to-End Software Platform for Production AI” includes “the full stack of NVIDIA AI software,” including Nvidia RAPID Accelerator for Apache Spark for speed data processing; Nvidia NeMo (Neural Modules) for building, customizing, and deploying generative AI; NVIDIA TensorRT, TensorRT-LLM, and NVIDIA Triton Inference Server for AI “inference” (e.g., making AI predictions or solutions) at scale; and Base Command Manager Essentials for managing AI clusters across edge devices and data centers.

Key Features	Accused Products
	The End-to-End Software Platform for Production AI NVIDIA AI Enterprise offers best-in-class development tools, frameworks, and pretrained models for AI practitioners and reliable management and orchestration for IT professionals to ensure performance, high availability, and security. Accelerate your AI pipeline with the full stack of NVIDIA AI software. ➤ Speed data processing with NVIDIA RAPIDS™ Accelerator for Apache Spark, ➤ Develop custom generative AI with NVIDIA NeMo, an end-to-end platform that delivers enterprise-ready models with precise data curation, cutting-edge customization, RAG, and accelerated performance. ➤ Achieve inference at scale with NIMs built on NVIDIA TensorRT™, TensorRT-LLM, and NVIDIA Triton™ Inference Server. ➤ Manage AI clusters at scale, across the edge and data center, with NVIDIA Base Command™ Manager Essentials. (See https://www.nvidia.com/en-us/data-center/products/ai-enterprise/; see also https://www.nvidia.com/en-us/deep-learning-ai/solutions/data-science/apache-spark-3/; https://www.nvidia.com/en-us/ai-data-science/generative-ai/nemo-framework/; https://developer.nvidia.com/tensorrt; https://www.nvidia.com/en-us/ai-data-science/products/triton-inference-server/; https://www.nvidia.com/en-us/ai-data-science/products/base-command-manager-essentials/.) The Accused Products further Nvidia software, platforms, and services that implement and exploit GPU acceleration systems and provide GPU acceleration service, including the DGX Platform, Nvidia Omniverse, Nvidia Drive, Nvidia Isaac Sim, Clara, AI Foundation models, and Nvidia NGC. For instance, the DGX Platform “combines the best of NVIDIA software, infrastructure, and expertise in a modern, unified AI development solution.” As previously discussed with Nvidia’s line of DGX supercomputers, the DGX platform “is a complete AI solution, and includes NVIDIA AI Enterprise software to accelerate data science pipelines and streamline development and deployment of production-grade AI applications.” The DGX platform also includes “direct access to NVIDIA DGXperts who can help optimize [customers’] AI workloads.” Nvidia provides the DGX platform to customers “on-premises, co-located, rented from managed service providers, in private cloud offerings, and more.”

Key Features	Accused Products
	The Proven Standard for Enterprise AI Built from the ground up for enterprise AI, the NVIDIA DGX™ platform combines the best of NVIDIA software, infrastructure, and expertise in a modern, unified AI development solution. Read Solution Overview  The Leading Platform for AI Development Unlock productivity with a fully integrated hardware and software AI platform. DGX infrastructure is a complete AI solution, and includes NVIDIA AI Enterprise software to accelerate data science pipelines and streamline development and deployment of production-grade AI applications.  Infused With NVIDIA AI Expertise Take advantage of the best of NVIDIA software and hardware integrated into one platform, with direct access to NVIDIA DGXperts who can help optimize your AI workloads for faster results and quicker ROI.  World-Class Infrastructure, Perfectly Aligned to Your Needs Experience the power of DGX in a multitude of ways that fit your business: on-premises, co-located, rented from managed service providers, in private cloud offerings, and more. (See https://www.nvidia.com/en-us/data-center/dgx-platform/.) As previously discussed, Nvidia Base Command is the operating system that “powers the NVIDIA DGXTM platform, enabling organizations to leverage the best of NVIDIA AI innovation” including “<i>system software optimized for running AI workloads.</i>” Indeed, Nvidia Command is the “<i>same software that supports NVIDIA’s thousands of in-house developers, researchers, and AI practitioners . . . , featuring best-of-breed developer software, infrastructure management, and accelerated infrastructure libraries.</i>”

Key Features	Accused Products
	 NVIDIA Base Command The operating system of the NVIDIA DGX data center. Watch Demo * * * * * NVIDIA Base Command™ powers the <u>NVIDIA DGX™ platform</u>, enabling organizations to leverage the best of NVIDIA AI innovation. With it, every organization can tap the full potential of their DGX infrastructure with a proven platform that includes AI workflow management, enterprise-grade cluster management, libraries that accelerate compute, storage, and network infrastructure, and system software optimized for running AI workloads. Download Solution Brief (See https://www.nvidia.com/en-us/data-center/base-command/ (emphasis added).) In addition, The DGX Platform cloud is “an end-to-end, scalable AI platform for developers, offering scalable capacity built on the latest NVIDIA architecture and co-engineered with the world’s leading cloud service providers (CSPs).” Your Own AI Factory—in the Cloud NVIDIA DGX™ Cloud is an end-to-end, scalable AI platform for developers, offering scalable capacity built on the latest NVIDIA architecture and co-engineered with the world’s leading cloud service providers (CSPs). (See https://www.nvidia.com/en-us/data-center/dgx-cloud/.)

Key Features	Accused Products
	Nvidia Omniverse is a cloud-based “industrial digitalization and physical AI simulation” platform and operating system for building virtual world simulation applications, available as a software subscription for enterprise use. Using Nvidia RTX GPUs, Omniverse helps industries building physical AI and industrial applications with applications for “Autonomous Vehicle Simulation,” “Configurator Development,” “Reinforcement Learning” (e.g., training robots in virtual environments) “Synthetic Data Generation” (e.g., virtual model training), and “Virtual Facility Integration.” Overview Design, Develop, and Deploy the Next Era of 3D Applications and Services NVIDIA Omniverse™ is a platform of APIs, SDKs, and services that enable developers to easily integrate <u>Universal Scene Description (OpenUSD)</u> and NVIDIA RTX™ rendering technologies into existing software tools and simulation workflows for building AI systems. (See https://www.nvidia.com/en-us/omniverse/.) Nvidia Drive is Nvidia’s automated-driving platform based on Nvidia Omniverse. It is end-to-end development platform for autonomous vehicles that provides “the necessary data center hardware, software, and workflows to support the entire autonomous driving technology development process, including data collection, <i>neural network development, and rigorous testing and validation in simulation environments.</i>” Nvidia acceleration infrastructure “provides the <i>massive compute horsepower crucial for training and fine-tuning self-driving algorithms</i>” as well as “testing and validating in simulation.”

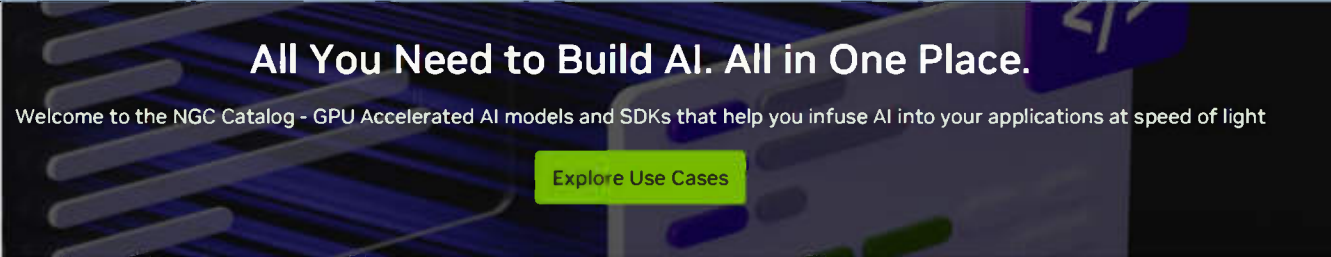
Key Features	Accused Products
	End-to-End Solutions for Autonomous Vehicles NVIDIA provides an end-to-end development platform for autonomous vehicles. The DRIVE AGX™ Developer Kit is an in-vehicle platform designed for developing production-level autonomous vehicles. NVIDIA's AV Infrastructure platform encompasses the necessary data center hardware, software, and workflows to support the entire autonomous driving technology development process, including data collection, neural network development, and rigorous testing and validation in simulation environments. DRIVE AGX Platform The NVIDIA DRIVE AGX Developer Kit is powered by the DriveOS SDK, a reference operating system and associated software stack that includes DriveWorks. The platform, along with supported sensors and accessories, enables developers to build ADAS and in-car AI applications that increase safety and enable convenience features for all occupants. (See https://developer.nvidia.com/drive (emphasis added).) AV Infrastructure NVIDIA's AV Infrastructure provides the massive compute horsepower crucial for training and fine-tuning self-driving algorithms. It also enables testing and validating in simulation.

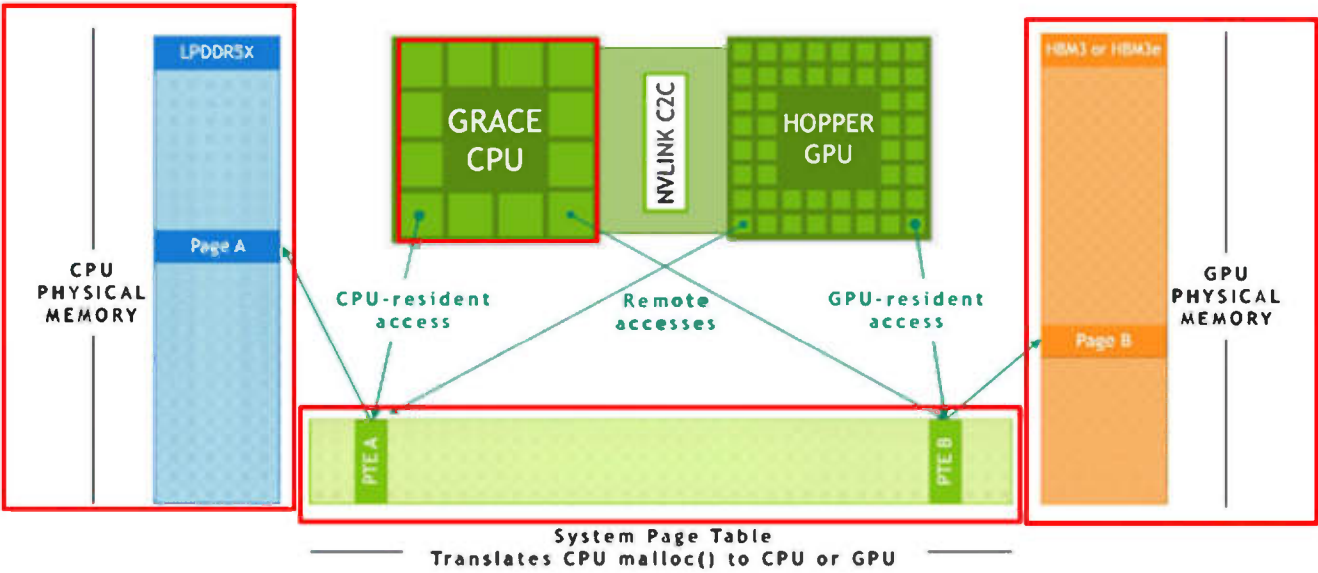
Key Features	Accused Products
	In addition, we offer a scalable data center-based simulation solution, NVIDIA DRIVE Sim, based on NVIDIA Omniverse software, for digital cockpit development, as well as for testing and validating a self-driving platform. Our unique end-to-end, software-defined approach is designed for continuous innovation and continuous development, enabling cars to receive over-the-air updates to add new features and capabilities throughout the life of a vehicle. (See https://d18rn0p25nwr6d.cloudfront.net/CIK-0001045810/1cbe8fe7-e08a-46e3-8dcc-b429fc06c1a4.pdf (Nvidia 10-K for fiscal year ended January 28, 2024).) Nvidia Isaac Sim, also based off Nvidia Omniverse, is a robotics simulation platform that implements Nvidia’s GPU acceleration for training and verifying AI models (e.g., neural networks) virtually. NVIDIA Isaac Sim The NVIDIA Isaac Sim™ robotics developer simulation platform and reference application is designed to help developers design, simulate, test, and train AI-based robots and autonomous machines in a physically based virtual environment. Isaac Sim, built on <u>NVIDIA Omniverse™</u>, is fully extensible. This means you can build your own <u>Universal Scene Description (OpenUSD)</u>-based custom simulators or integrate core Isaac Sim technologies into your existing testing and validation pipelines. Ready to get started? Download the SDK through the Omniverse Launcher, pulled as a container from NGC, or install it using a command line with PIP. (See https://developer.nvidia.com/isaac/sim.) In addition, Nvidia Clara is “is a suite of computing platforms, software, and services that powers AI solutions for healthcare and life sciences, from imaging and instruments to genomics and drug

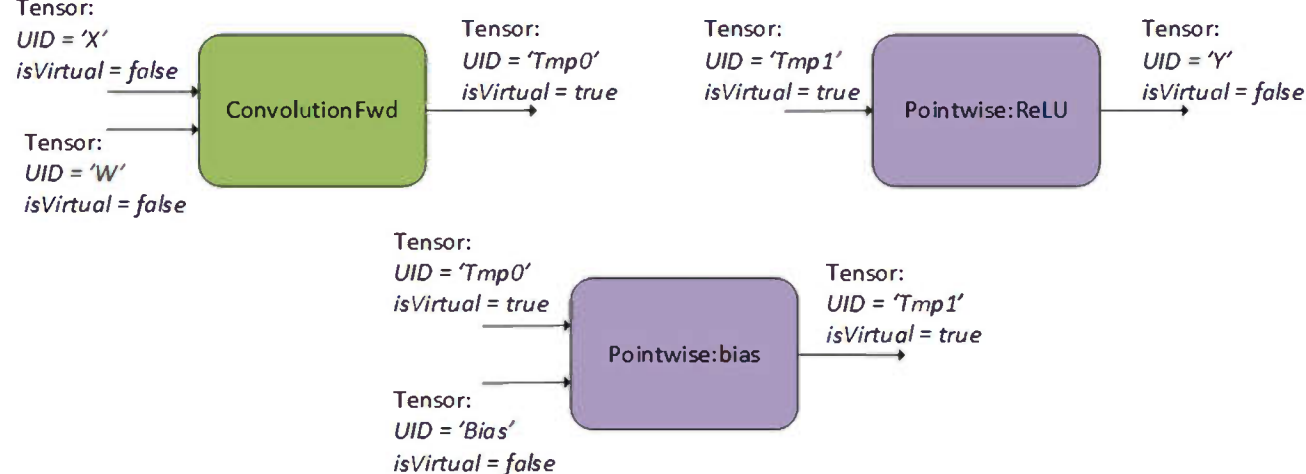
Key Features	Accused Products
	discovery.” AI-powered solution applications provided by Clara include “Biopharma,” “Digital Health,” “Medical Devices,” “Medical Imaging,” and “Genomics.” NVIDIA Clara NVIDIA Clara™ is a suite of computing platforms, software, and services that powers AI solutions for healthcare and life sciences, from imaging and instruments to genomics and drug discovery. AI-Powered Solutions for Healthcare  Biopharma NVIDIA Clara for Biopharma is a collection of AI and accelerated computing frameworks, applications, <u>generative AI platforms</u> and services, and pretrained models for accelerating drug discovery pipelines. Explore NVIDIA Clara for Biopharma >  Digital Health NVIDIA Clara Digital Health is a comprehensive AI toolkit for developing personalized healthcare applications that enhance clinician-patient interactions, streamline documentation, and extract valuable insights from real-world data. Explore NVIDIA Clara for Digital Health >  Medical Devices NVIDIA Clara for Medical Devices is a domain-specific AI computing platform that delivers the full-stack infrastructure needed for building scalable, software-defined medical devices that can process streaming data at the edge in real time. Explore NVIDIA Clara for Medical Devices >

Key Features	Accused Products
	(See https://www.nvidia.com/en-us/clara/.) In addition, Nvidia Foundation models is a “curated collection of enterprise-grade pretrained models” that “give[s] developers a running start for bringing custom generative AI to their enterprise applications.” For example, Nvidia provides “leading community models such as Llama 2, Stable Diffusion XL and Mistral,” and “models [that] have been optimized with NVIDIA TensorRT-LLM to deliver the highest throughput and lowest latency and to run at scale on any NVIDIA GPU-accelerated stack,” such as the “<i>Llama 2 model optimized with [Nvidia] TensorRT-LLM [that] runs nearly 2x faster on NVIDIA H100.</i>” Today’s landscape of free, open-source <u>large language models</u> (LLMs) is like an all-you-can-eat buffet for enterprises. This abundance can be overwhelming for developers building custom generative AI applications, as they need to navigate unique project and business requirements, including compatibility, security and the data used to train the models. <u>NVIDIA AI Foundation Models</u> — a curated collection of enterprise-grade pretrained models — give developers a running start for bringing custom generative AI to their enterprise applications.

Key Features	Accused Products
	NVIDIA-Optimized Foundation Models Speed Up Innovation NVIDIA AI Foundation Models can be experienced through a simple user interface or API, directly from a browser. Additionally, these models can be accessed from NVIDIA AI Foundation Endpoints to test model performance from within their enterprise applications. Available models include leading community models such as Llama 2, Stable Diffusion XL and Mistral, which are formatted to help developers streamline customization with proprietary data. Additionally, models have been optimized with <u>NVIDIA TensorRT-LLM</u> to deliver the highest throughput and lowest latency and to run at scale on any NVIDIA GPU-accelerated stack. For instance, the Llama 2 model optimized with TensorRT-LLM runs nearly <u>2x faster</u> on NVIDIA H100. The new NVIDIA family of <u>Nemotron-3 8B foundation models</u> supports the creation of today's most advanced enterprise chat and Q&A applications for a broad range of industries, including healthcare, telecommunications and financial services. <i>(See https://blogs.nvidia.com/blog/custom-generative-ai-model-development/ (emphasis added); see also https://www.nvidia.com/en-us/ai-data-science/foundation-models/.)</i> Nvidia NGC is Nvidia's catalog of "GPU Accelerated AI models and SDKs." As shown below, examples include "LLMs [large language models, a type of artificial intelligence program that can recognize and generate text, among other tasks] optimized for RTX PCs" which includes a "collection of TensorRT-LLM accelerated Windows RTX PC LLM models." Furthermore, Nvidia provides "Documentation," "AI Enterprise Documentation," "Enterprise Support," and a "Licensing Portal."

Key Features	Accused Products
	 (See https://catalog.ngc.nvidia.com/.)
[21.1] executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition;	The Accused Products perform a method that includes <i>executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition.</i> For instance, as shown in the Grace Hopper Superchip architecture diagram below, the Grace Hopper Superchip is illustrated below with a CPU (“GRACE CPU”) (e.g., <i>the at least one CPU</i>). The CPU “share[s] a single per-process page table” with a GPU (“Hopper GPU”), “enabling all CPU and GPU threads to access all system-allocated memory.” The CPU is depicted as coupled to the GPU via “NVLINK C2C [chip-to-chip],” and can access the “System Page Table” and “CPU PHYSICAL MEMORY” via “CPU-resident access” and “GPU PHYSICAL MEMORY” via “[r]emote access” and “PTE [page table entry] B.” The GPU can also access the System Page Table, and it can access “GPU PHYSICAL MEMORY” via “GPU-resident access” and “CPU PHYSICAL MEMORY” via “[r]emote access” and “PTE A.” Moreover, the “System Page Table” “[t]ranslates CPU malloc() [memory allocation] to CPU or GPU.” “The CPU heap, CPU thread stack, global variables memory-mapped files, and inter-process memory are accessible to all CPU and GPU threads.”

Key Features	Accused Products
	In NVIDIA Grace Hopper Superchip-based systems, Address Translation Service (ATS) enables the CPU and GPU to share a <u>single per-process page table, enabling all CPU and GPU threads to access all system-allocated memory (Figure 8), which can reside on physical CPU or GPU memory.</u> <u>The CPU heap, CPU thread stack, global variables, memory-mapped files, and inter-process memory are accessible to all CPU and GPU threads.</u>  Figure 8. ATS in an NVIDIA Grace Hopper Superchip System (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).) In a related example, a “user passes [operation descriptors] to the operation graph” for tensor operations as shown below (e.g., <i>user interaction stream</i>).

Key Features	Accused Products
	<i>Figure 6. <u>A set of operation descriptors the user passes to the operation graph</u></i>  (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (emphasis added).) The “CUDA programming model” includes programmatic functions, primitives, and executable libraries for CPUs and GPUs that are used by the Accused Products to perform numerical simulations. “The host is the CPU available in the system” and “system memory associated with the CPU is called host memory.” “The GPU is called a device and GPU memory likewise called device memory.” As an example, the first main CUDA program execution step is “[c]opy[ing] the input data from host [CPU] memory to device [GPU] memory, also known as host-to-device transfer” (e.g., <i>executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition</i>).

Key Features	Accused Products
	Let me introduce two keywords widely used in CUDA programming model: <u>host and device</u>. <u>The host is the CPU available in the system</u>. The system memory associated with the CPU is called host memory. <u>The GPU is called a device</u> and GPU memory likewise called device memory. To execute any CUDA program, there are three main steps: • <u>Copy the input data from host memory to device memory, also known as host-to-device transfer.</u> • Load the GPU program and execute, caching data on-chip for performance. • Copy the results from device memory to host memory, also called device-to-host transfer. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).) Examples of publicly available CUDA toolkit documentation for “the memory management functions of the CUDA runtime application programming interface” are shown below. Functions including “[c]op[y]ing <i>data between host [CPU] and device [GPU]</i>” (see “cudaMemcpy”; “cudaMemcpy2D”) and “[a]llocat[ing] an array on the device [GPU]” (see “cudaMallocArray”) (e.g., <i>executing, by the at least one CPU, a user interaction stream, the user interaction stream controlling transfer of inputs to the artificial neural network to the first memory partition and the second memory partition</i>). 6.11. Memory Management This section describes the memory management functions of the CUDA runtime application programming interface. Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module. Functions * * * * *

Key Features	Accused Products
	<pre> __host__ cudaError_t <u>cudaMemcpy</u>(void* dst , const void* src , size_t count , <u>cudaMemcpyKind</u> kind) Copies data between host and device. * * * * * __host__ cudaError_t <u>cudaMallocArray</u>(<u>cudaArray_t</u>* array , const <u>cudaChannelFormatDesc</u>* desc , size_t width , size_t height = 0, unsigned int flags = 0) <u>Allocate an array on the device.</u> * * * * * __host__ cudaError_t <u>cudaMemcpy2D</u>(void* dst , size_t dpitch , const void* src , size_t spitch , size_t width , size_t height , <u>cudaMemcpyKind</u> kind) Copies data between host and device. </pre> (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As another example, operations are executed on the CPU as host code “streams” (e.g., <i>executing, by the at least one CPU, a user interaction stream</i>). <u>The CUDA programming model provides a heterogeneous environment where the host code is running the C/C++ program on the CPU and the kernel runs on a physically separate GPU device.</u> The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively. CUDA code also provides for data transfer between host and device memory, over the PCIe bus. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).) 3.2.8.5. Streams <u>Applications manage the concurrent operations described above through streams.</u> A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (for example, inter-kernel communication is undefined). The commands issued on a stream may execute when all the dependencies of the command are met. The dependencies could be previously launched commands on same stream or dependencies from other streams. The successful completion of synchronize call guarantees that all the commands launched are completed.

Key Features	Accused Products
	(See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=cudaMemcpy#asynchronous-concurrent-execution (emphasis added).)
[21.2] executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network;	The Accused Products perform a method that includes <i>executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network.</i> For instance, the “CUDA programming model” includes programmatic functions, primitives, and executable libraries for CPUs and GPUs that are used by the Accused Products to perform numerical simulations. As previously stated, the host is the CPU and the device is the GPU. After “[c]opy[ing] the input data from host [CPU] memory to device [GPU] memory,” the second main CUDA program execution step is “[l]oad[ing] the GPU program and execut[ing]” (e.g., <i>executing, by the processing unit, a computational stream</i>). Let me introduce two keywords widely used in CUDA programming model: <i>host</i> and <i>device</i>. The host is the CPU available in the system. The system memory associated with the CPU is called host memory. The GPU is called a device and GPU memory likewise called device memory. To execute any CUDA program, there are three main steps: • Copy the input data from host memory to device memory, also known as host-to-device transfer. • <u>Load the GPU program and execute</u>, caching data on-chip for performance. • Copy the results from device memory to host memory, also called device-to-host transfer. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).)

Key Features	Accused Products
	As an example, the Grace Hopper Superchip “is designed to accelerate applications” using “Extended GPU Memory.” As depicted in the gram of the Grace Hopper architecture below, a GPU (“HOPPER GPU”) can access “Local CPU,” “Peer CPU,” and “Peer GPU” memory via “NVLink.” Accelerating Applications with Extended GPU Memory <u>The NVIDIA GH200 is designed to accelerate applications with exceptionally large memory footprints, larger than the capacity of the HBM3 / HBM3e and LPDDR5X memory of a single superchip (see the NVIDIA GH200 Accelerated Applications section below).</u> The Extended GPU Memory (EGM) feature over the high-bandwidth NVLink-C2C enables GPUs to access all the system memory efficiently. EGM provides up to 19.5TBs system memory in a multi-node NVSwitch-connected system. With EGM, physical memory in the system can be allocated to be accessible from any GPU thread. All GPUs can access EGM at the minimum of GPU-GPU NVLink or NVLink-C2C speed. Memory accesses within a Grace Hopper Superchip configuration go through the local high-bandwidth NVLink-C2C at 900GB/s total. Remote memory accesses are performed via GPU NVLink, and depending on the memory being accessed, also NVLink-C2C as shown in Figure 5. With EGM, GPU threads can now access all memory resources available over the NVSwitch fabric, both LPDDR5X and HBM3 or HBM3e, unidirectionally at 450GB/s.

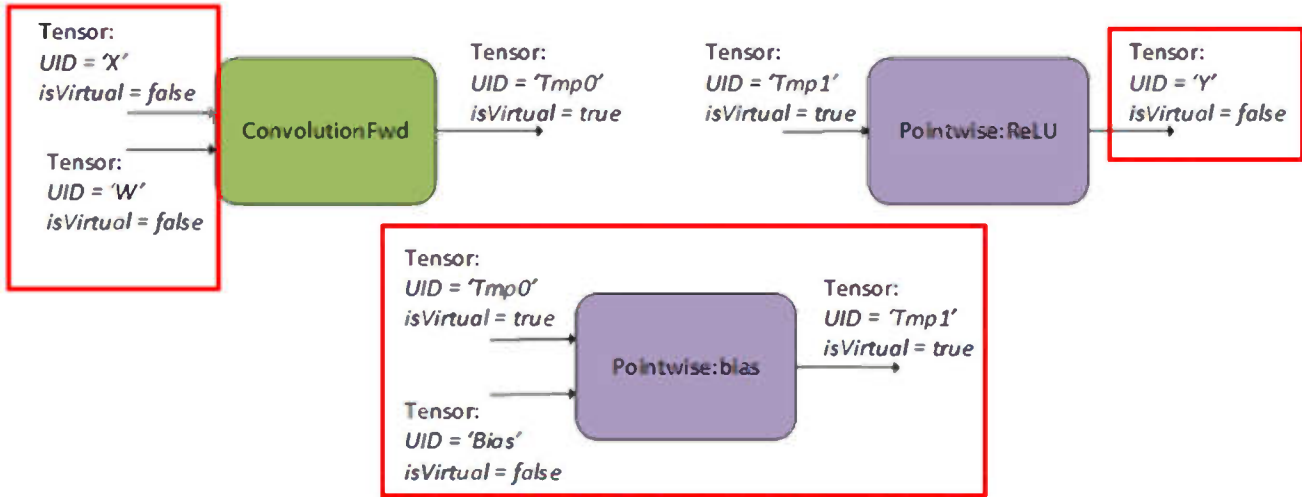
Key Features	Accused Products
	The diagram illustrates the memory access paths between two Grace Hopper Superchips connected via NVIDIA NVLink. Each superchip is composed of a Grace CPU and a Hopper GPU. The left superchip features a CPU LPDDR5X and GPU HBM3 or GPU HBM3e. The right superchip features a GPU HBM3 or GPU HBM3e and CPU LPDDR5X. Arrows indicate data flow: Local CPU <-> GPU (blue), GPU -> Peer GPU (orange), and GPU -> Peer CPU (blue). Figure 5. Memory Accesses across NVLink-connected Grace Hopper Superchips (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).) For instance, CUDA library cuDNN function “<code>cudaSetRNNDescriptor_v8</code>” “initializes a previously created RNN [recurrent neural network] descriptor object.” This function “store[s] all information needed to compute the total number of adjustable weights/biases in the RNN model” (e.g., <i>executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network</i>). In addition, the parameters “<code>dirMode</code>,” “<code>inputMode</code>,” and “<code>datatype</code>” confirm the exchange of calculations and values between the hidden layers of an RNN.

Key Features	Accused Products
	7.2.49. <code>cudaSetRNNDescriptor_v8()</code> This function initializes a previously created RNN descriptor object. The RNN descriptor configured by <code>cudaSetRNNDescriptor_v8()</code> was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model. <pre> cudaStatus_t cudaSetRNNDescriptor_v8(cudaRNNDescriptor_t rnnDesc, cudaRNNAlgo_t algo, cudaRNNMode_t cellMode, cudaRNNBiasMode_t biasMode, cudaRNNDirectionMode_t dirMode, cudaRNNInputMode_t inputMode, cudaDataType_t dataType, cudaDataType_t mathPrec, cudaMathType_t mathType, int32_t inputSize, int32_t hiddenSize, int32_t projSize, int32_t numLayers, cudaDropoutDescriptor_t dropoutDesc, uint32_t auxFlags); </pre> * * * * * dirMode <i>Input.</i> Specifies the recurrence pattern: CUDNN_UNIDIRECTIONAL or CUDNN_BIDIRECTIONAL. In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states. inputMode <i>Input.</i> Specifies how the input to the RNN model is processed by the first layer. When inputMode is CUDNN_LINEAR_INPUT, original input vectors of size inputSize are multiplied by the weight matrix to obtain vectors of hiddenSize. When inputMode is CUDNN_SKIP_INPUT, the original input vectors to the first layer are used as is without multiplying them by the weight matrix. dataType <i>Input.</i> Specifies data type for RNN weights/biases and input and output data.

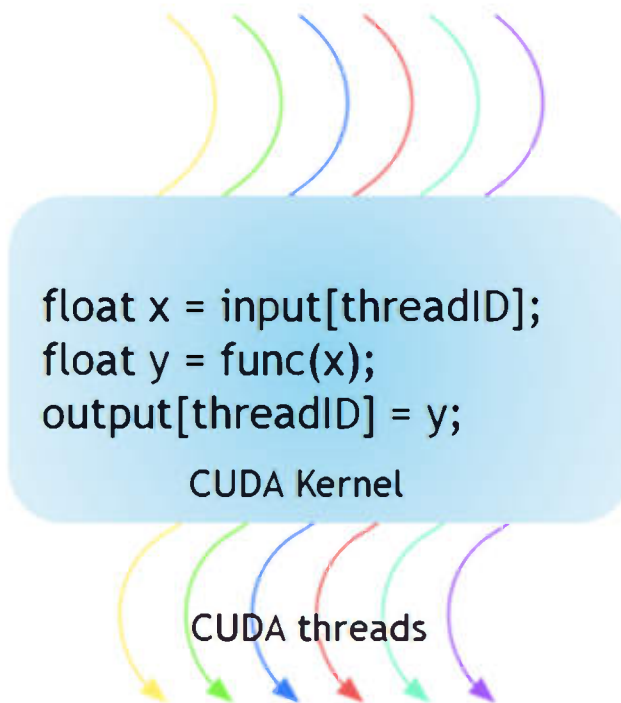
Key Features	Accused Products
	(See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-891/pdf/cuDNN-API.pdf (emphasis added).) In another cuDNN example, an excerpt from Nvidia’s cuDNN release notes exemplifies “numerical results” from RNN API calls as used for deep learning using recurrent neural networks. A “kernel selection,” which prepares memory space and resources for use, “may affect numerical results.” cuDNN Release Notes NVIDIA CUDA Deep Neural Network (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications. These release notes describe the key features, software enhancements and improvements, and known issues for the NVIDIA cuDNN 8.9.3 and earlier releases. * * * * * • RNN and multihead attention API calls may exhibit nondeterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in Results Reproducibility, <u>numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This happens when two buffer sizes (16 KB and 4 MB) are used in the default configuration.</u> When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. <u>The kernel selection may affect numerical results.</u> The user can eliminate the nondeterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the <code>CUBLAS_WORKSPACE_CONFIG</code> environmental variable, for example, <code>:16:8</code> or <code>:4096:2</code>. The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is <code>:16:8:4096:2</code>, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the <code>:16:8</code> non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

Key Features	Accused Products
	(See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-893/release-notes/index.html#abstract (emphasis added).) In a related example, cuDNN implements “tensor alignment checks to instance norm and layer norm engines.” In addition, cuDNN users do “not need to transfer [an] array [from RNN data descriptors] to device memory; the operation will be performed automatically by RNN APIs. This refinement simplifies the usage of cuDNN RNN APIs.” • <u>cuDNN 8.9.1 added tensor alignment checks to instance norm and layer norm engines to prevent IMA issues.</u> • Starting in cuDNN 8.9.1, the <code>const int32_t devSeqLengths[]</code> argument in <code>cudnnRNNForward()</code>, <code>cudnnRNNBackwardData_v8()</code>, and <code>cudnnRNNBackwardWeights_v8()</code> APIs will be ignored. <u>All three functions will source variable sequence length arrays from RNN data descriptors, configured through the <code>seqLengthArray</code> parameter of <code>cudnnSetRNNDataDescriptor()</code>. The user does not need to transfer this array to device memory; the operation will be performed automatically by RNN APIs.</u> This refinement simplifies the usage of cuDNN RNN APIs. It is also a workaround for random crashes in multi-GPU RNN training on TensorFlow. Replacing earlier versions of cuDNN 8.x shared libraries with cuDNN 8.9.1 will eliminate those crashes without forcing the user to switch the TensorFlow version. The cause of intermittent corruptions of <code>devSeqLengths[]</code>, fed to RNN APIs, is still being investigated. (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-893/release-notes/index.html#rel-891 (emphasis added).) In addition, cuDNN operations below exemplify “tak[ing] tensors as input and produc[ing] tensors as output.” As shown below for a “forward convolution operation” using cuDNN, “backend tensor <i>Tmp0</i> [is] both the output of the convolution operation and the input of the bias operation.” From this, “cuDNN infers that the dataflow runs from the convolution into the bias” (e.g., <i>the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network</i>).

Key Features	Accused Products
	2.2. Tensors and Layouts Whether using the graph API or the legacy API, <u>cuDNN operations take tensors as input and produce tensors as output.</u> * * * * * 3.2. Graph API Example with Operation Fusion The following example implements a fusion of convolution, bias, and activation. 3.2.1. Creating Operation and Tensor Descriptors to Specify the Graph Dataflow First, create three cuDNN backend operation descriptors. As can be seen in Figure 6, the user specified one forward convolution operation (using <code>CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR</code>), a pointwise operation for the bias addition (using <code>CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR</code> with mode <code>CUDNN_POINTWISE_ADD</code>), and a pointwise operation for the ReLU activation (using <code>CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR</code> with mode <code>CUDNN_POINTWISE_RELU_FWD</code>). Refer to the NVIDIA cuDNN Backend API for more details on setting the attributes of these descriptors. For an example of how a forward convolution can be set up, refer to the Setting Up An Operation Graph For A Grouped Convolution use case in the cuDNN backend API. You should also create tensor descriptors for the inputs and outputs of all of the operations in the graph. <u>The graph dataflow is implied by the assignment of tensors (refer to Figure 6), for example, by specifying the backend tensor <code>Tmp0</code> as both the output of the convolution operation and the input of the bias operation, cuDNN infers that the dataflow runs from the convolution into the bias. The same applies to tensor <code>Tmp1</code>. If the user doesn't need the intermediate results <code>Tmp0</code> and <code>Tmp1</code> for any other use, then the user can specify them to be virtual tensors, so the memory I/Os can later be optimized out.</u> * * * * *

Key Features	Accused Products
	Figure 6. A set of operation descriptors the user passes to the operation graph  (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (emphasis added).) As a further example, CUDA implements “kernels” running on streaming multiprocessors of Nvidia GPUs. CUDA kernels are “function[s] that get[] executed on GPU”; the “parallel portion of [] applications is executed K times in parallel by K different CUDA threads, as opposed to only one time like regular C/C++ functions” (e.g., <i>executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations</i>).

Key Features	Accused Products
	The CUDA programming model provides a heterogeneous environment where the host code is running the C/C++ program on the CPU and <u>the kernel runs on a physically separate GPU device</u>. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively. CUDA code also provides for data transfer between host and device memory, over the PCIe bus. * * * * * Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be <u>migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism)</u>. One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time. Figure 3 shows the kernel execution and mapping on hardware resources available in GPU. * * * * *

Key Features	Accused Products
	CUDA kernel and thread hierarchy Figure 1 shows that the CUDA kernel is a function that gets executed on GPU. The parallel portion of your applications is executed K times in parallel by K different CUDA threads, as opposed to only one time like regular C/C++ functions.  <pre data-bbox="873 813 1373 967">float x = input[threadID]; float y = func(x); output[threadID] = y;</pre> CUDA Kernel CUDA threads <i>Figure 1. The kernel is a function executed on the GPU.</i> *****

Key Features	Accused Products
	<i>Figure 3. Kernel execution on GPU.</i> (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).) As another example, controlling data execution between user interaction stream and computational stream during execution of the computations representing the neural network is facilitated by asynchronous execution features of the Accused Products.

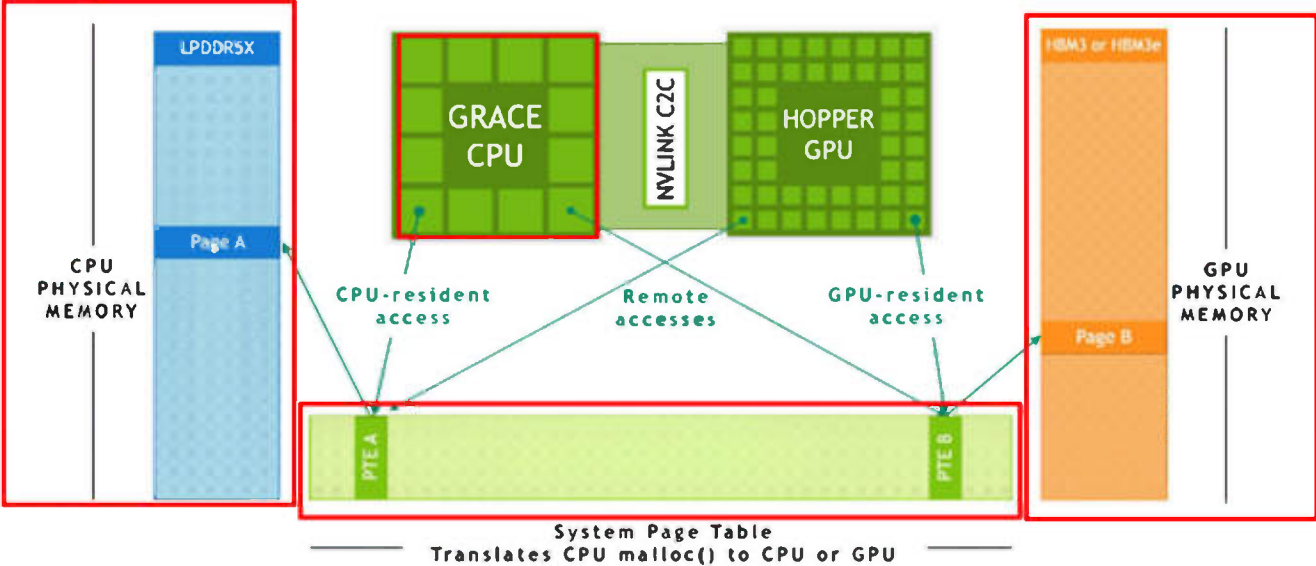
Key Features	Accused Products
	Asynchronous execution Each new generation of NVIDIA GPUs includes numerous architectural enhancements to improve performance, programmability, power efficiency, GPU utilization, and many other factors. Recent NVIDIA GPU generations have included <u>asynchronous execution capabilities to enable more overlap of data movement, computation, and synchronization.</u> <u>The NVIDIA Hopper Architecture provides new features that improve asynchronous execution and enable further overlap of memory copies with computation and other independent work, while also minimizing synchronization points. We describe the new async memory copy unit called the Tensor Memory Accelerator (TMA) and a new asynchronous transaction barrier.</u> (See https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth (emphasis added).)
[21.3] shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network;	The Accused Products perform a method that includes <i>shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network.</i> (See https://developer.nvidia.com/discover/artificial-neural-network (The Accused Products implement libraries and SDKs designed for neural networks that “are created from large numbers of identical neurons [that] are highly parallel by nature,” implementing cuDNN, a library “makes it easy to obtain state-of-the-art performance with Deep Neural Networks,” and TensorRT, a platform accelerator and runtime for optimizing, validating, and deploying neural networks for inference.) (e.g., <i>computations representing the artificial neural network</i>).) For instance, the “CUDA programming model” includes programmatic functions, primitives, and executable libraries for CPUs (host) and GPUs (device) that are used by the Accused Products to perform numerical simulations. As an example, the “host-to-device transfer” (CPU to GPU) first main step, the second main step is “[l]oad the GPU program and <i>execute, caching data on-chip for performance</i>” and

Key Features	Accused Products
	the third main step is “[c]opy the results from device [GPU] memory to host [CPU] memory, also known as device-to-host transfer” (GPU to CPU) (e.g., <i>shifting control of a data exchange between the user interaction stream and the computational stream to the computational stream in response to starting execution of the computations representing the artificial neural network</i>). Let me introduce two keywords widely used in CUDA programming model: <i>host</i> and <i>device</i>. The host is the CPU available in the system. The system memory associated with the CPU is called host memory. The GPU is called a device and GPU memory likewise called device memory. To execute any CUDA program, there are three main steps: • Copy the input data from host memory to device memory, also known as host-to-device transfer. • Load the GPU program and execute, caching data on-chip for performance. • Copy the results from device memory to host memory, also called device-to-host transfer. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).) As previously stated, the Grace Hopper Superchip “is designed to accelerate applications” using “Extended GPU Memory” and the GPU can access local/peer CPU and peer GPU memory via “NVLink.” The Grace Hopper Superchip’s Extended GPU Memory feature “enables GPUs to access all the system memory efficiently” and “physical memory in the system can be allocated to be accessible from any GPU thread.”

Key Features	Accused Products
	Accelerating Applications with <u>Extended GPU Memory</u> The NVIDIA GH200 is designed to accelerate applications with exceptionally large memory footprints, larger than the capacity of the HBM3 / HBM3e and LPDDR5X memory of a single superchip (see the NVIDIA GH200 Accelerated Applications section below). The Extended GPU Memory (EGM) feature over the high-bandwidth NVLink-C2C enables GPUs to access all the system memory efficiently. EGM provides up to 19.5TBs system memory in a multi-node NVSwitch-connected system. With EGM, physical memory in the system can be allocated to be accessible from any GPU thread. All GPUs can access EGM at the minimum of GPU-GPU NVLink or NVLink-C2C speed. Memory accesses within a Grace Hopper Superchip configuration go through the local high-bandwidth NVLink-C2C at 900GB/s total. Remote memory accesses are performed via GPU NVLink, and depending on the memory being accessed, also NVLink-C2C as shown in Figure 5. With EGM, GPU threads can now access all memory resources available over the NVSwitch fabric, both LPDDR5X and HBM3 or HBM3e, unidirectionally at 450GB/s. The diagram shows two identical Grace Hopper Superchips connected by an NVIDIA NVLink. Each superchip is composed of a Grace CPU and a Hopper GPU. The Grace CPU is connected to the Hopper GPU via an NVLink C2C interface. The Grace CPU contains CPU LPDDR5X memory, and the Hopper GPU contains GPU HBM3 or CPU HBM3e memory. Four types of memory access paths are illustrated: Local CPU ← GPU: A blue arrow pointing from the Hopper GPU to the Grace CPU within the same superchip. GPU → Peer GPU: An orange arrow pointing from the Hopper GPU of one superchip to the Hopper GPU of the other superchip via the NVIDIA NVLink. GPU → Peer CPU: A blue arrow pointing from the Hopper GPU of one superchip to the Grace CPU of the other superchip via the NVIDIA NVLink. Local GPU → CPU: A blue arrow pointing from the Hopper GPU to the Grace CPU within the same superchip. Figure 5. Memory Accesses across NVLink-connected Grace Hopper Superchips (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).)

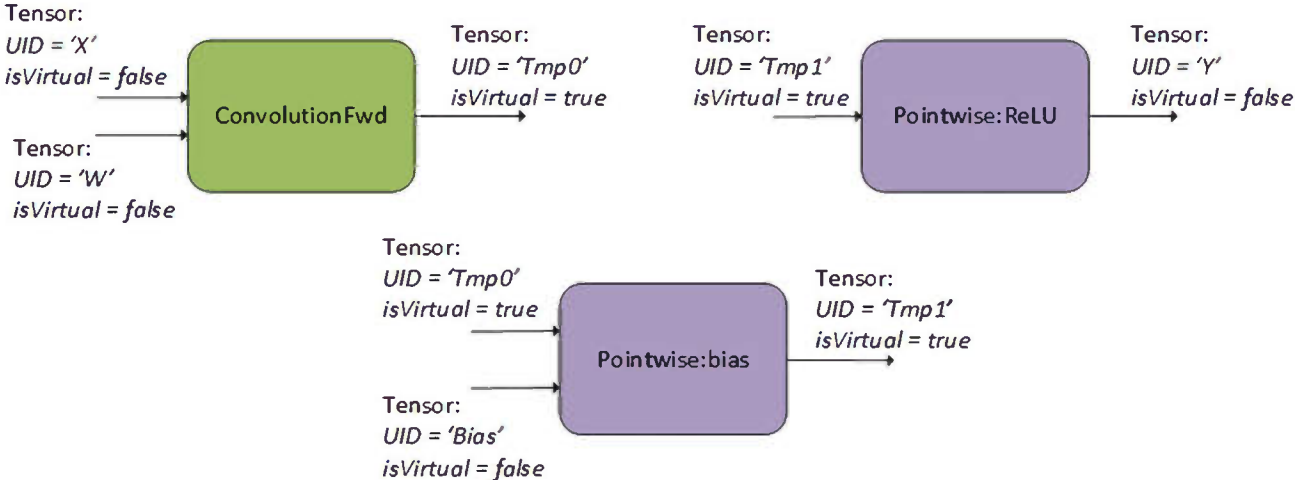
Key Features	Accused Products
[21.4] shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network;	The Accused Products perform a method that includes <i>shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network</i>. (See https://developer.nvidia.com/discover/artificial-neural-network (The Accused Products implement libraries and SDKs designed for neural networks that “are created from large numbers of identical neurons [that] are highly parallel by nature,” implementing cuDNN, a library “makes it easy to obtain state-of-the-art performance with Deep Neural Networks,” and TensorRT, a platform accelerator and runtime for optimizing, validating, and deploying neural networks for inference.) (e.g., <i>computations representing the artificial neural network</i>).) For instance, after the “CUDA programming model” “host-to-device transfer” (CPU to GPU) and GPU program load and execution steps, the third main step is “[c]opy the results from device [GPU] memory to host [CPU] memory, also known as <i>device-to-host transfer</i>” (GPU to CPU). The “<i>host-to-device transfer</i>” (CPU to GPU) first main step can be reintroduced for additional computations (e.g., <i>shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network</i>).

Key Features	Accused Products
	Let me introduce two keywords widely used in CUDA programming model: <i>host</i> and <i>device</i>. The host is the CPU available in the system. The system memory associated with the CPU is called host memory. The GPU is called a device and GPU memory likewise called device memory. To execute any CUDA program, there are three main steps: • Copy the input data from host memory to device memory, also known as <u>host-to-device transfer</u>. • Load the GPU program and execute, caching data on-chip for performance. • Copy the results from device memory to host memory, also called <u>device-to-host transfer</u>. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added.)) As a further example shown in the Grace Hopper Superchip architecture diagram below, the CPU (“GRACE CPU”) “share[s] a single per-process page table” with a GPU (“Hopper GPU”), “enabling all CPU and GPU threads to access all system-allocated memory.” “The CPU heap, CPU thread stack, global variables memory-mapped files, and inter-process memory are accessible to all CPU and GPU threads.” In NVIDIA Grace Hopper Superchip-based systems, Address Translation Service (ATS) enables the CPU and GPU to <u>share a single per-process page table, enabling all CPU and GPU threads to access all system-allocated memory (Figure 8), which can reside on physical CPU or GPU memory. The CPU heap, CPU thread stack, global variables, memory-mapped files, and inter-process memory are accessible to all CPU and GPU threads.</u>

Key Features	Accused Products
	 The diagram illustrates the Address Translation Service (ATS) in an NVIDIA Grace Hopper Superchip System. It shows the interaction between CPU and GPU memory spaces. On the left, the CPU Physical Memory contains LPDDR5X and Page A. On the right, the GPU Physical Memory contains HBM3 or HBM3e and Page B. In the center, the GRACE CPU and HOPPER GPU are connected via NVLINK C2C. A System Page Table at the bottom translates CPU malloc() to CPU or GPU. Arrows indicate CPU-resident access, GPU-resident access, and Remote accesses between the CPU, GPU, and the System Page Table.
	Figure 8. ATS in an NVIDIA Grace Hopper Superchip System (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).) As a further example, Nvidia CUDA programming model and extensions to CUDA enable concurrent execution, queuing by asynchronous calls, and dynamic parallelism of device operations. For instance, “[c]oncurrent host execution” through CUDA “is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver when appropriate device resources are available.” Concurrent operations include “[c]omputations on the device [GPU]” and “host [CPU]” and “[m]emory transfers” between both and “within the memory of a given device” and “among devices” (e.g., <i>shifting control of the data exchange between the user interaction stream and the computational stream to the user interaction stream in response to completion or interruption of the computations representing the artificial neural network</i>). Asynchronous CUDA operations include “Kernel launches,” “Memory copies performed by functions that are suffixed with Async,” and “Memory set function calls.”

Key Features	Accused Products
	3.2.8. Asynchronous Concurrent Execution CUDA exposes the following operations as independent tasks that can operate concurrently with one another: ➤ Computation on the host; ➤ Computation on the device; ➤ Memory transfers from the host to the device; ➤ Memory transfers from the device to the host; ➤ Memory transfers within the memory of a given device; ➤ Memory transfers among devices. The level of concurrency achieved between these operations will depend on the feature set and compute capability of the device as described below. 3.2.8.1. Concurrent Execution between Host and Device Concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver <u>when appropriate device resources are available. This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks.</u> The following device operations are asynchronous with respect to the host: ➤ Kernel launches; ➤ Memory copies within a single device's memory; ➤ Memory copies from host to device of a memory block of 64 KB or less; ➤ Memory copies performed by functions that are suffixed with <code>Async</code> ; ➤ Memory set function calls. (See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=cudaMemcpy#asynchronous-concurrent-execution (emphasis added).) The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as <u>launch configuration decisions can now be made at runtime by threads executing on the device.</u> Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads. Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed.

Key Features	Accused Products
	(See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism .)
[21.5] queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network; and	The Accused Products perform a method that includes <i>queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network</i>. (See https://developer.nvidia.com/discover/artificial-neural-network (The Accused Products implement libraries and SDKs designed for neural networks that “are created from large numbers of identical neurons [that] are highly parallel by nature,” implementing cuDNN, a library “makes it easy to obtain state-of-the-art performance with Deep Neural Networks,” and TensorRT, a platform accelerator and runtime for optimizing, validating, and deploying neural networks for inference.) (e.g., <i>computations representing the artificial neural network</i>).) For instance, as shown by publicly available CUDA toolkit, CUDA implements exemplary “memory management functions” that “[c]op[y] data between host [CPU] and device [GPU].” This includes CUDA functions “cudaMemcpy” and “cudaMemcpyAsync” (e.g., <i>queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network</i>). 6.11. Memory Management This section describes the memory management functions of the CUDA runtime application programming interface. Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module. Functions <pre> * * * * * __host__ cudaError_t cudaMemcpy (void* dst , const void* src <u>Copies data between host and device.</u> * * * * * __host__ __device__ cudaError_t cudaMemcpyAsync (void* dst , const void <u>Copies data between host and device.</u> </pre> (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).)

Key Features	Accused Products
	In a related example, a “user passes [operation descriptors] to the operation graph” for tensor operations as shown below (e.g., <i>user command received by the user interaction stream</i>). <u>Figure 6. A set of operation descriptors the user passes to the operation graph</u>  <pre> graph LR X["Tensor: UID = 'X' isVirtual = false"] --> Conv["ConvolutionFwd"] W["Tensor: UID = 'W' isVirtual = false"] --> Conv Conv --> Tmp0_1["Tensor: UID = 'Tmp0' isVirtual = true"] Tmp0_1 --> ReLU["Pointwise:ReLU"] ReLU --> Y["Tensor: UID = 'Y' isVirtual = false"] Tmp0_2["Tensor: UID = 'Tmp0' isVirtual = true"] --> Bias["Pointwise:bias"] Bias_["Tensor: UID = 'Bias' isVirtual = false"] --> Bias Bias --> Tmp1["Tensor: UID = 'Tmp1' isVirtual = true"] </pre> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (emphasis added).) As an example, exemplary CUDA memory management function “cudaMemcpyAsync” “[c]opies count bytes [data] from the memory area pointed to by src [source memory address pointer] to the memory area pointed to by dst [destination memory address pointer], where kind [type of transfer] specifies the direction of the copy.” Destinations includes “cudaMemcpyHostToDevice [CPU to device GPU], cudaMemcpyDeviceToHost [GPU to CPU], cudaMemcpyDeviceToDevice [GPU to GPU]. Because the function “cudaMemcpyAsync() is asynchronous with respect to the host, [] the call may return before the copy is complete. The copy can optionally be associated to a stream [identified stream] by passing a non-zero stream argument” (e.g., <i>queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network</i>).

Key Features	Accused Products
	<pre data-bbox="573 240 1669 289">__host__ __device__ cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)</pre> <u>Copies data between host and device.</u> Parameters dst - Destination memory address src - Source memory address count - Size in bytes to copy kind - Type of transfer stream - Stream identifier Returns cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemoryDirection Description <u>Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.</u> The memory areas may not overlap. Calling cudaMemcpyAsync() with dst and src pointers that do not match the direction of the copy results in an undefined behavior. <u>cudaMemcpyAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and the stream is non-zero, the copy may overlap with operations in other streams.</u> The device version of this function only handles device to device copies and cannot be given local or shared pointers. (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As a further example, Nvidia CUDA programming model and extensions to CUDA enable concurrent execution, queuing by asynchronous calls and dynamic parallelism of device operations.

Key Features	Accused Products
	3.2.8.1. Concurrent Execution between Host and Device Concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver <u>when appropriate device resources are available. This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks.</u> The following device operations are asynchronous with respect to the host: ➤ Kernel launches; ➤ Memory copies within a single device's memory; ➤ Memory copies from host to device of a memory block of 64 KB or less; ➤ Memory copies performed by functions that are suffixed with <code>Async</code> ; ➤ Memory set function calls. (See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=cudaMemcpy#asynchronous-concurrent-execution (emphasis added).)

Key Features	Accused Products
	9.1. Introduction % 9.1.1. Overview <i>Dynamic Parallelism</i> is an extension to the CUDA programming model enabling a CUDA kernel to create and synchronize with new work directly on the GPU. The creation of parallelism dynamically at whichever point in a program that it is needed offers exciting capabilities. The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can now be made at runtime by threads executing on the device. <u>Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads.</u> Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed. 9.2.1. Execution Environment % The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. <u>Support for dynamic parallelism in CUDA extends the ability to configure, launch, and implicitly synchronize upon new grids to threads that are running on the device.</u> (See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism (emphasis added).)
[21.6] executing the user command during execution of the computations representing the artificial	The Accused Products perform a method that includes <i>executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream.</i>

Key Features	Accused Products
neural network at times determined by the computational stream.	(See https://developer.nvidia.com/discover/artificial-neural-network (The Accused Products implement libraries and SDKs designed for neural networks that “are created from large numbers of identical neurons [that] are highly parallel by nature,” implementing cuDNN, a library “makes it easy to obtain state-of-the-art performance with Deep Neural Networks,” and TensorRT, a platform accelerator and runtime for optimizing, validating, and deploying neural networks for inference.) (e.g., <i>computations representing the artificial neural network</i>).) For instance, as shown by exemplary and publicly available CUDA toolkit documentation, CUDA implements “memory management functions” that “[c]op[y] data between host [CPU] and device [GPU].” 6.11. Memory Management This section describes the memory management functions of the CUDA runtime application programming interface. Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module. Functions <pre> * * * * * __host__ <u>cudaError_t</u> <u>cudaMemcpy</u> (void* dst , const void* src <u>Copies data between host and device.</u> * * * * * __host__ __device__ <u>cudaError_t</u> <u>cudaMemcpyAsync</u> (void* dst , const void <u>Copies data between host and device.</u> </pre> (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As an example, exemplary CUDA memory management function “cudaMemcpyAsync” “[c]opies count bytes [data] from the memory area pointed to by src [source memory address pointer] to the memory area pointed to by dst [destination memory address pointer], where kind [type of transfer] specifies the direction of the copy.” Because the function “cudaMemcpyAsync() is asynchronous with respect to the host, [] the call may return before the copy is complete. The copy can optionally be associated to a stream

Key Features	Accused Products
	[identified stream] by passing a non-zero stream argument” (e.g., <i>executing the user command during execution of the computations representing the artificial neural network at times determined by the computational stream</i>). <pre data-bbox="573 386 1694 440">__host__ __device__ cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)</pre> <u>Copies data between host and device.</u> Parameters dst - Destination memory address src - Source memory address count - Size in bytes to copy kind - Type of transfer stream - Stream identifier Returns cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemoryDirection Description Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. The memory areas may not overlap. Calling cudaMemcpyAsync() with dst and src pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpyAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and the stream is non-zero, the copy may overlap with operations in other streams. The device version of this function only handles device to device copies and cannot be given local or shared pointers. (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).)

Key Features	Accused Products
	In another example, CUDA implements “CUDA-specific memory APIs [that] provide users with guarantees about where the memory resides, which threads can access it, whether it is migratable, and many other features that enable users to extract all the performance the hardware has to offer.” <u>CUDA-specific memory APIs provide users with guarantees about where the memory resides, which threads can access it, whether it is migratable, and many other features that enable users to extract all the performance the hardware has to offer.</u> Applications can hint the system about their memory access patterns, for example, using <u>CUDA</u> and/or <u>NUMA</u> APIs, to enable the users to perform application-specific optimizations. NUMA memory hints enable applications to inform the runtime about their memory access patterns. (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).)
[22] The method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network.	The Accused Products perform <i>the method of claim 21, wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network.</i> See, e.g., [21.5] (“<i>queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network</i>”), <i>supra</i>. (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>). For instance, as shown by publicly available CUDA toolkit, CUDA implements exemplary “memory management functions” that “[c]op[y] data between host [CPU] and device [GPU].” This includes CUDA functions “cudaMemcpy” and “cudaMemcpyAsync” (e.g., <i>wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network</i>).

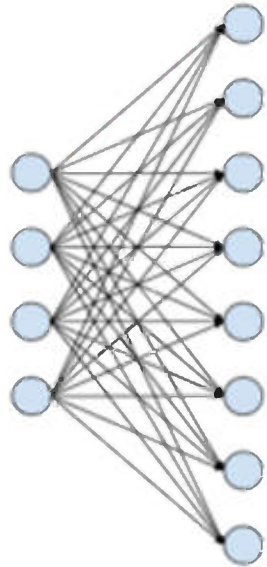
Key Features	Accused Products
	<u>6.11. Memory Management</u> This section describes the memory management functions of the CUDA runtime application programming interface. Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module. Functions <pre> *** __host__ cudaError_t cudaMemcpy (void* dst , const void* src Copies data between host and device. *** __host__ __device__ cudaError_t cudaMemcpyAsync (void* dst , const void Copies data between host and device. </pre> (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As an example, exemplary CUDA memory management function “cudaMemcpyAsync” “[c]opies count bytes [data] from the memory area pointed to by src [source memory address pointer] to the memory area pointed to by dst [destination memory address pointer], where kind [type of transfer] specifies the direction of the copy.” Destinations includes “cudaMemcpyHostToDevice [CPU to device GPU], cudaMemcpyDeviceToHost [GPU to CPU], cudaMemcpyDeviceToDevice [GPU to GPU]. Because the function “cudaMemcpyAsync() is asynchronous with respect to the host, [] the call may return before the copy is complete. The copy can optionally be associated to a stream [identified stream] by passing a non-zero stream argument” (e.g., <i>wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network</i>).

Key Features	Accused Products
	<pre data-bbox="573 240 1692 289">__host__ __device__ cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)</pre> <u>Copies data between host and device.</u> Parameters dst - Destination memory address src - Source memory address count - Size in bytes to copy kind - Type of transfer stream - Stream identifier Returns cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemoryDirection Description <u>Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing.</u> The memory areas may not overlap. Calling cudaMemcpyAsync() with dst and src pointers that do not match the direction of the copy results in an undefined behavior. <u>cudaMemcpyAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and the stream is non-zero, the copy may overlap with operations in other streams.</u> The device version of this function only handles device to device copies and cannot be given local or shared pointers. (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As a further example, Nvidia CUDA programming model and extensions to CUDA enable concurrent execution, queuing by asynchronous calls, and dynamic parallelism of device operations. For instance, “[c]oncurrent host execution” through CUDA “is facilitated through asynchronous library functions that

Key Features	Accused Products
	return control to the host thread before the device completes the requested task. Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver when appropriate device resources are available. <i>This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks.</i> Asynchronous CUDA operations include “Kernel launches,” “Memory copies performed by functions that are suffixed with Async,” and “Memory set function calls” (e.g., <i>wherein the user interaction stream controls the data exchange between the user interaction stream and the computational stream outside of execution of the computations representing the artificial neural network</i>). 3.2.8.1. Concurrent Execution between Host and Device Concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. <u>Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver when appropriate device resources are available. This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks.</u> The following device operations are asynchronous with respect to the host: > Kernel launches; > Memory copies within a single device's memory; > Memory copies from host to device of a memory block of 64 KB or less; > Memory copies performed by functions that are suffixed with <code>Async</code> ; > Memory set function calls. 3.2.8.1. Concurrent Execution between Host and Device Concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. <u>Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver when appropriate device resources are available. This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks.</u> The following device operations are asynchronous with respect to the host: > Kernel launches; > Memory copies within a single device's memory; > Memory copies from host to device of a memory block of 64 KB or less; > Memory copies performed by functions that are suffixed with <code>Async</code> ; > Memory set function calls. (See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=cudaMemcpy#asynchronous-concurrent-execution (emphasis added).)

Key Features	Accused Products
	9.1. Introduction % 9.1.1. Overview <i>Dynamic Parallelism</i> is an extension to the CUDA programming model enabling a CUDA kernel to create and synchronize with new work directly on the GPU. The creation of parallelism dynamically at whichever point in a program that it is needed offers exciting capabilities. The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can now be made at runtime by threads executing on the device. <u>Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads.</u> Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed. 9.2.1. Execution Environment % The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. <u>Support for dynamic parallelism in CUDA extends the ability to configure, launch, and implicitly synchronize upon new grids to threads that are running on the device.</u> (See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism (emphasis added).)
[23] The method of claim 21, wherein executing the user interaction stream comprises: controlling setting the and editing of computational	The Accused Products perform <i>the method of claim 21, wherein executing the user interaction stream comprises: controlling setting the and editing of computational elements of the computations representing the artificial neural network.</i>

Key Features	Accused Products
elements of the computations representing the artificial neural network.	<i>See, e.g., [21.2] (“executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network”), supra.</i> (<i>See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>).</i> For instance, the below illustrates an exemplary neural network the Accused Products are designed to accelerate using parallel computations. “Input” (four) and “Output” (eight) neurons are depicted below in a full-connected or linear layer structure in which all of the input neurons depicted in a first layer are connected to all of the output neurons depicted in a second layer. Computations for the neural network are performed using, for example, “NVIDIA Matrix Multiplication.” Examples of inputs and outputs for forward propagation, activation gradient computation, and weight gradient computation (as matrix by matrix multiplications) for GEMMs (General Matrix Multiplications) are shown below (e.g., <i>controlling setting the and editing of computational elements of the computations representing the artificial neural network</i>). For example, for “(a) forward propagation . . . of a fully-connected layer” (the process of feeding input data through a neural network to generate an output), “K = # of inputs” and “M = # of outputs” and in “N = batch size” with results for “Input Activations,” “Output Activations,” and “Weights.” As a rough guideline, choose batch sizes and neuron counts greater than 128 to avoid being limited by memory bandwidth (NVIDIA® A100-SXM4-80GB; this threshold is similar for other A100 and V100 GPUs); see <u>Batch Size</u>.

Key Features	Accused Products
	<u>2. Fully-Connected Layer</u> Fully-connected layers, also known as <u>linear layers</u>, connect every input neuron to every output neuron and are commonly used in neural networks. <i>Figure 1. Example of a small fully-connected layer with <u>four input and eight output neurons</u>.</i>  The diagram illustrates a fully-connected layer within a neural network. It consists of two vertical columns of light blue circular nodes. The left column, labeled 'Input Neurons', contains four nodes. The right column, labeled 'Output Neurons', contains eight nodes. Every node in the input column is connected to every node in the output column by a thin black line, representing a complete bipartite graph. The entire diagram is enclosed in a red rectangular border.

Key Features	Accused Products																
	Three parameters define a fully-connected layer: batch size, number of inputs, and number of outputs. <u>Forward propagation, activation gradient computation, and weight gradient computation are directly expressed as matrix-matrix multiplications.</u> How the three parameters map to GEMM dimensions (General Matrix Multiplication, background in the <u>NVIDIA Matrix Multiplication Background User's Guide</u>) varies among frameworks, but the underlying principles are the same. For the purposes of the discussion, we adopt the convention used by PyTorch and Caffe where A contains the weights and B the activations. In TensorFlow, matrices take the opposite roles, but the performance principles are the same. Table 1. <u>Mapping of inputs, outputs, and batch size to GEMM parameters M, N, K.</u> <table><tr><th>Computation Phase</th><th>M</th><th>N</th><th>K</th></tr><tr><td>Forward Propagation</td><td>Number of outputs</td><td>Batch size</td><td>Number of inputs</td></tr><tr><td>Activation Gradient</td><td>Number of inputs</td><td>Batch size</td><td>Number of outputs</td></tr><tr><td>Weight Gradient</td><td>Number of inputs</td><td>Number of outputs</td><td>Batch size</td></tr></table>	Computation Phase	M	N	K	Forward Propagation	Number of outputs	Batch size	Number of inputs	Activation Gradient	Number of inputs	Batch size	Number of outputs	Weight Gradient	Number of inputs	Number of outputs	Batch size
Computation Phase	M	N	K														
Forward Propagation	Number of outputs	Batch size	Number of inputs														
Activation Gradient	Number of inputs	Batch size	Number of outputs														
Weight Gradient	Number of inputs	Number of outputs	Batch size														

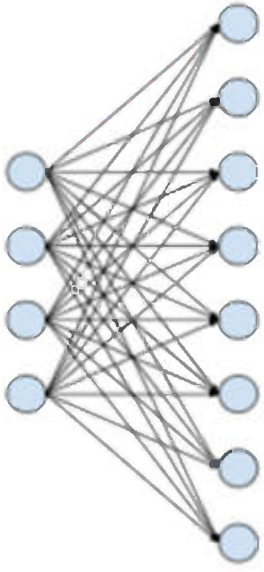
Key Features	Accused Products
	* * * * * <i>Figure 2. Dimensions of equivalent GEMMs for (a) forward propagation, (b) activation gradient, and (c) weight gradient computations of a fully-connected layer.</i> Figure 2 illustrates the dimensions of equivalent GEMMs for three types of computations in a fully-connected layer: (a) Forward propagation: A matrix of M rows and K columns, labeled Weights^T (where $M = \# \text{ of outputs}$ and $K = \# \text{ of inputs}$), is multiplied by a matrix of K rows and N columns, labeled Input Activations (where $N = \text{batch size}$). The result is a matrix of M rows and N columns, labeled $\text{Output Activations}$. (b) Activation gradient: A matrix of M rows and K columns, labeled Weights (where $M = \# \text{ of inputs}$ and $K = \# \text{ of outputs}$), is multiplied by a matrix of K rows and N columns, labeled $\text{Gradients (Output)}$ (where $N = \text{batch size}$). The result is a matrix of M rows and N columns, labeled Gradients (Input). (c) Weight gradient: A matrix of M rows and K columns, labeled Input Activations (where $M = \# \text{ of inputs}$ and $K = \text{batch size}$), is multiplied by a matrix of K rows and N columns, labeled $\text{Gradient (Weights)}$ (where $N = \# \text{ of outputs}$). The result is a matrix of M rows and N columns, labeled Gradient (Output). (See https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#performance (emphasis added).) In another example, CUDA library cuDNN function “cudnnSetRNNDescrptor_v8” “initializes a previously created RNN [recurrent neural network] descriptor object.” This function “store[s] all information needed to compute the total number of adjustable weights/biases in the RNN model.” In addition, the parameters “dirMode,” “inputMode,” and “datatype” confirm the exchange of calculations

Key Features	Accused Products
	and values between the hidden layers of an RNN (e.g., <i>wherein executing the user interaction stream comprises: controlling setting the and editing of computational elements of the computations representing the artificial neural network</i>). 7.2.49. <code>cudaSetRNNDescriptor_v8()</code> This function initializes a previously created RNN descriptor object. The RNN descriptor configured by <code>cudaSetRNNDescriptor_v8()</code> was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model. <pre> cudaStatus_t cudaSetRNNDescriptor_v8(cudaRNNDescriptor_t rnnDesc, cudaRNNAlgo_t algo, cudaRNNMode_t cellMode, cudaRNNBiasMode_t biasMode, cudaDirectionMode_t dirMode, cudaRNNInputMode_t inputMode, cudaDataType_t dataType, cudaDataType_t mathPrec, cudaMathType_t mathType, int32_t inputSize, int32_t hiddenSize, int32_t projSize, int32_t numLayers, cudaDropoutDescriptor_t dropoutDesc, uint32_t auxFlags); </pre> *****

Key Features	Accused Products
	dirMode <i>Input.</i> Specifies the recurrence pattern: CUDNN_UNIDIRECTIONAL or CUDNN_BIDIRECTIONAL. <u>In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states.</u> inputMode <i>Input.</i> <u>Specifies how the input to the RNN model is processed by the first layer.</u> When inputMode is CUDNN_LINEAR_INPUT, original input vectors of size inputSize are multiplied by the weight matrix to obtain vectors of hiddenSize. When inputMode is CUDNN_SKIP_INPUT, the original input vectors to the first layer are used as is without multiplying them by the weight matrix. dataType <i>Input.</i> <u>Specifies data type for RNN weights/biases and input and output data.</u> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-891/pdf/cuDNN-API.pdf (emphasis added).) In a related example, cuDNN implements “tensor alignment checks to instance norm and layer norm engines.” In addition, cuDNN users do “not need to transfer [an] array [from RNN data descriptors] to device memory; the operation will be performed automatically by RNN APIs (e.g., <i>controlling setting the and editing of computational elements of the computations representing the artificial neural network</i>). This refinement simplifies the usage of cuDNN RNN APIs.”

Key Features	Accused Products
	• <u>cuDNN 8.9.1 added tensor alignment checks to instance norm and layer norm engines to prevent IMA issues.</u> • Starting in cuDNN 8.9.1, the <code>const int32_t devSeqLengths[]</code> argument in <code>cudaNNRNNForward()</code>, <code>cudaNNRNNBackwardData_v8()</code>, and <code>cudaNNRNNBackwardWeights_v8()</code> APIs will be ignored. <u>All three functions will source variable sequence length arrays from RNN data descriptors, configured through the <code>seqLengthArray</code> parameter of <code>cudaNNSetRNNDataDescriptor()</code>. The user does not need to transfer this array to device memory; the operation will be performed automatically by RNN APIs.</u> This refinement simplifies the usage of cuDNN RNN APIs. It is also a workaround for random crashes in multi-GPU RNN training on TensorFlow. Replacing earlier versions of cuDNN 8.x shared libraries with cuDNN 8.9.1 will eliminate those crashes without forcing the user to switch the TensorFlow version. The cause of intermittent corruptions of <code>devSeqLengths[]</code>, fed to RNN APIs, is still being investigated. (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-893/release-notes/index.html#rel-891 (emphasis added).)
[24] The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the computations representing the artificial neural network.	The Accused Products perform <i>the method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the computations representing the artificial neural network.</i> See, e.g., [21.2] (“<i>executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network</i>”), <i>supra</i>. (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>). For instance, the below illustrates an exemplary neural network the Accused Products are designed to accelerate using parallel computations. “Input” (four) and “Output” (eight) neurons are depicted below

Key Features	Accused Products
	in a full-connected or linear layer structure in which all of the input neurons depicted in a first layer are connected to all of the output neurons depicted in a second layer. Computations for the neural network are performed using, for example, “NVIDIA Matrix Multiplication.” Examples of inputs and outputs for forward propagation, activation gradient computation, and weight gradient computation (as matrix by matrix multiplications) for GEMMs (General Matrix Multiplications) are shown below. For example, for “(a) forward propagation . . . of a fully-connected layer” (the process of feeding input data through a neural network to generate an output), “K = # of inputs” and “M = # of outputs” and in “N = batch size” with results for “Input Activations,” “Output Activations,” and “Weights” (e.g., <i>controlling setting and editing of parameters of the computations representing the artificial neural network</i>). <u>2. Fully-Connected Layer</u> <u>Fully-connected layers, also known as linear layers, connect every input neuron to every output neuron and are commonly used in neural networks.</u> <i>Figure 1. Example of a small fully-connected layer with <u>four input and eight output neurons.</u></i>

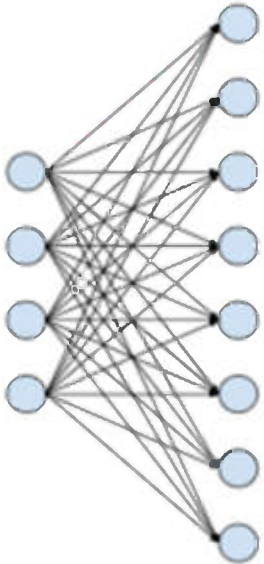
Key Features	Accused Products
	 Input Neurons Output Neurons * * * * *

Key Features	Accused Products
	<i>Figure 2. Dimensions of equivalent GEMMs for (a) forward propagation, (b) activation gradient, and (c) weight gradient computations of a fully-connected layer.</i> Figure 2 consists of three diagrams labeled (a), (b), and (c), each showing the dimensions of matrices involved in GEMM operations for a fully-connected layer. (a) Forward propagation: Shows a matrix multiplication between Weights^T and Input Activations to produce Output Activations. The dimensions are: M = # of outputs (height of Weights^T), K = # of inputs (width of Weights^T and height of Input Activations), and N = batch size (width of Input Activations and height of Output Activations). (b) Activation gradient: Shows a matrix multiplication between Weights and Gradients (Input) to produce Gradients (Output). The dimensions are: M = # of inputs (height of Weights), K = # of outputs (width of Weights and height of Gradients (Input)), and N = batch size (width of Gradients (Input) and height of Gradients (Output)). (c) Weight gradient: Shows a matrix multiplication between Input Activations and Gradient (Weights) to produce Gradient^T (Output). The dimensions are: M = # of inputs (height of Input Activations), K = batch size (width of Input Activations and height of Gradient (Weights)), and N = # of outputs (width of Gradient (Weights) and height of Gradient^T (Output)). (See https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#performance (emphasis added).) In addition, CUDA library cuDNN function “<code>cudaSetRNNDescriptor_v8</code>” “initializes a previously created RNN [recurrent neural network] descriptor object.” This function “store[s] all information needed to compute the total number of adjustable weights/biases in the RNN model.” In addition, the parameters “<code>dirMode</code>,” “<code>inputMode</code>,” and “<code>datatype</code>” confirm the exchange of calculations and values between the

Key Features	Accused Products
	hidden layers of an RNN (e.g., wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the computations representing the artificial neural network). 7.2.49. cudnnSetRNNDescrptor_v8() This function initializes a previously created RNN descriptor object. The RNN descriptor configured by <u>cudnnSetRNNDescrptor_v8()</u> was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model. <pre> cudnnStatus_t cudnnSetRNNDescrptor_v8(cudnnRNNDescrptor_t rnnDesc, cudnnRNNAlgo_t algo, cudnnRNNMode_t cellMode, cudnnRNNBiasMode_t biasMode, cudnnDirectionMode_t dirMode, cudnnRNNInputMode_t inputMode, cudnnDataType_t dataType, cudnnDataType_t mathPrec, cudnnMathType_t mathType, int32_t inputSize, int32_t hiddenSize, int32_t projSize, int32_t numLayers, cudnnDropoutDescrptor_t dropoutDesc, uint32_t auxFlags); </pre> * * * * *

Key Features	Accused Products
	dirMode <i>Input.</i> Specifies the recurrence pattern: CUDNN_UNIDIRECTIONAL or CUDNN_BIDIRECTIONAL. <u>In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states.</u> inputMode <i>Input.</i> <u>Specifies how the input to the RNN model is processed by the first layer.</u> When inputMode is CUDNN_LINEAR_INPUT, original input vectors of size inputSize are multiplied by the weight matrix to obtain vectors of hiddenSize. When inputMode is CUDNN_SKIP_INPUT, the original input vectors to the first layer are used as is without multiplying them by the weight matrix. dataType <i>Input.</i> <u>Specifies data type for RNN weights/biases and input and output data.</u> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-891/pdf/cuDNN-API.pdf (emphasis added).)
[25] The method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the inputs to the artificial neural network.	The Accused Products perform <i>the method of claim 21, wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the inputs to the artificial neural network.</i> <i>See, e.g., [21.2], (“executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network”), supra.</i> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>). For instance, the below illustrates an exemplary neural network the Accused Products are designed to accelerate using parallel computations. “Input” (four) and “Output” (eight) neurons are depicted below in a full-connected or linear layer structure in which all of the input neurons depicted in a first layer are connected to all of the output neurons depicted in a second layer. Computations for the neural network are performed using, for example, “NVIDIA Matrix Multiplication.” Examples of inputs and outputs for

Key Features	Accused Products
	forward propagation, activation gradient computation, and weight gradient computation (as matrix by matrix multiplications) for GEMMs (General Matrix Multiplications) are shown below. For example, for “(a) forward propagation . . . of a fully-connected layer” (the process of feeding input data through a neural network to generate an output), “K = # of inputs” and “M = # of outputs” and in “N = batch size” with results for “Input Activations,” “Output Activations,” and “Weights” (e.g., <i>controlling setting and editing of parameters of the inputs to the artificial neural network</i>). <u>2. Fully-Connected Layer</u> <u>Fully-connected layers, also known as linear layers, connect every input neuron to every output neuron and are commonly used in neural networks.</u> <i>Figure 1. Example of a small fully-connected layer with <u>four input and eight output neurons.</u></i>

Key Features	Accused Products
	 Input Neurons Output Neurons * * * * *

Key Features	Accused Products
	<i>Figure 2. Dimensions of equivalent GEMMs for (a) forward propagation, (b) activation gradient, and (c) weight gradient computations of a fully-connected layer.</i> (a) (b) (c) (See https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#performance (emphasis added).) In addition, CUDA library cuDNN function “<code>cudaSetRNNDescriptor_v8</code>” “initializes a previously created RNN [recurrent neural network] descriptor object.” This function “store[s] all information needed to compute the total number of adjustable weights/biases in the RNN model.” In addition, the parameters “<code>dirMode</code>,” “<code>inputMode</code>,” and “<code>datatype</code>” confirm the exchange of calculations and values between the

Key Features	Accused Products
	hidden layers of an RNN (e.g., wherein executing the user interaction stream comprises: controlling setting and editing of parameters of the inputs to the artificial neural network). 7.2.49. <code>cudaSetRNNDescriptor_v8()</code> This function initializes a previously created RNN descriptor object. The RNN descriptor configured by <u><code>cudaSetRNNDescriptor_v8()</code> was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model.</u> <pre> cudaStatus_t cudaSetRNNDescriptor_v8(cudaRNNDescriptor_t rnnDesc, cudaRNNAlgo_t algo, cudaRNNMode_t cellMode, cudaRNNBiasMode_t biasMode, cudaDirectionMode_t dirMode, cudaRNNInputMode_t inputMode, cudaDataType_t dataType, cudaDataType_t mathPrec, cudaMathType_t mathType, int32_t inputSize, int32_t hiddenSize, int32_t projSize, int32_t numLayers, cudaDropoutDescriptor_t dropoutDesc, uint32_t auxFlags); </pre> *****

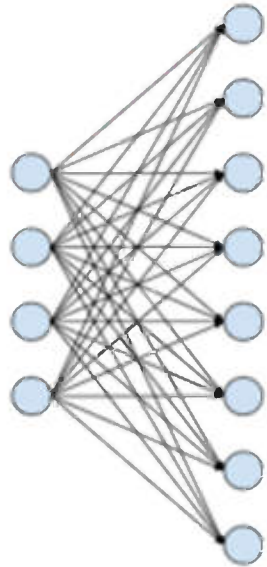
Key Features	Accused Products
	dirMode <i>Input.</i> Specifies the recurrence pattern: CUDNN_UNIDIRECTIONAL or CUDNN_BIDIRECTIONAL. <u>In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states.</u> inputMode <i>Input.</i> <u>Specifies how the input to the RNN model is processed by the first layer.</u> When inputMode is CUDNN_LINEAR_INPUT, original input vectors of size inputSize are multiplied by the weight matrix to obtain vectors of hiddenSize. When inputMode is CUDNN_SKIP_INPUT, the original input vectors to the first layer are used as is without multiplying them by the weight matrix. dataType <i>Input.</i> <u>Specifies data type for RNN weights/biases and input and output data.</u> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-891/pdf/cuDNN-API.pdf (emphasis added).)
[26] The method of claim 21, wherein executing the user interaction stream comprises: specifying an output to be saved to disk and/or displayed on a screen.	The Accused Products perform <i>the method of claim 21, wherein executing the user interaction stream comprises: specifying an output to be saved to disk and/or displayed on a screen.</i> <i>See, e.g., [21.5] (“queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network”), supra.</i> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>)). For instance, as shown by publicly available CUDA toolkit, CUDA implements exemplary “memory management functions” that “[c]op[y] data between host [CPU] and device [GPU].” This includes CUDA functions “cudaMemcpy” and “cudaMemcpyAsync” (e.g., <i>specifying an output to be saved to disk and/or displayed on a screen</i>).

Key Features	Accused Products
	<u>6.11. Memory Management</u> This section describes the memory management functions of the CUDA runtime application programming interface. Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module. Functions <pre> *** __host__ cudaError_t cudaMemcpy (void* dst , const void* src Copies data between host and device. *** __host__ __device__ cudaError_t cudaMemcpyAsync (void* dst , const void Copies data between host and device. </pre> (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As an example, exemplary CUDA memory management function “cudaMemcpyAsync” “[c]opies count bytes [data] from the memory area pointed to by src [source memory address pointer] to the memory area pointed to by dst [destination memory address pointer], where kind [type of transfer] specifies the direction of the copy.” Destinations includes “cudaMemcpyHostToDevice [CPU to device GPU], cudaMemcpyDeviceToHost [GPU to CPU], cudaMemcpyDeviceToDevice [GPU to GPU] (e.g., <i>wherein executing the user interaction stream comprises: specifying an output to be saved to disk and/or displayed on a screen</i>). Because the function “cudaMemcpyAsync() is asynchronous with respect to the host, [] the call may return before the copy is complete. The copy can optionally be associated to a stream [identified stream] by passing a non-zero stream argument.”

Key Features	Accused Products
	<pre data-bbox="577 240 1690 292">__host__ __device__ cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)</pre> <u>Copies data between host and device.</u> Parameters dst - Destination memory address src - Source memory address count - Size in bytes to copy kind - Type of transfer stream - Stream identifier Returns cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemoryDirection Description Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. The memory areas may not overlap. Calling cudaMemcpyAsync() with dst and src pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpyAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero stream argument. If kind is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and the stream is non-zero, the copy may overlap with operations in other streams. The device version of this function only handles device to device copies and cannot be given local or shared pointers. (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As a further example, Nvidia CUDA programming model and extensions to CUDA enable concurrent execution, queuing by asynchronous calls and dynamic parallelism of device operations (e.g., <i>specifying an output</i>).

Key Features	Accused Products
	3.2.8.1. Concurrent Execution between Host and Device Concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. Using asynchronous calls, many device operations can be queued up together to be executed by the CUDA driver <u>when appropriate device resources are available. This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks.</u> The following device operations are asynchronous with respect to the host: ➤ Kernel launches; ➤ Memory copies within a single device's memory; ➤ Memory copies from host to device of a memory block of 64 KB or less; ➤ Memory copies performed by functions that are suffixed with <code>Async</code> ; ➤ Memory set function calls. (See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=cudaMemcpy#asynchronous-concurrent-execution (emphasis added).) 9.1. Introduction % 9.1.1. Overview <i>Dynamic Parallelism</i> is an extension to the CUDA programming model enabling a CUDA kernel to create and synchronize with new work directly on the GPU. The creation of parallelism dynamically at whichever point in a program that it is needed offers exciting capabilities. The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can now be made at runtime by threads executing on the device. <u>Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads.</u> Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed. 9.2.1. Execution Environment % The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. <u>Support for dynamic parallelism in CUDA extends the ability to configure, launch, and implicitly synchronize upon new grids to threads that are running on the device.</u> (See https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism (emphasis added).)

Key Features	Accused Products
[27] The method of claim 21, wherein executing the user interaction stream comprises: parsing elements to be used in the computations representing the artificial neural network.	The Accused Products perform <i>the method of claim 21, wherein executing the user interaction stream comprises: parsing elements to be used in the computations representing the artificial neural network.</i> <i>See, e.g., [21.2], (“executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network”), supra.</i> (<i>See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>).</i> For instance, the below illustrates an exemplary neural network the Accused Products are designed to accelerate using parallel computations. “Input” (four) and “Output” (eight) neurons are depicted below in a full-connected or linear layer structure in which all of the input neurons depicted in a first layer are connected to all of the output neurons depicted in a second layer. Computations for the neural network are performed using, for example, “NVIDIA Matrix Multiplication.” Examples of inputs and outputs for forward propagation, activation gradient computation, and weight gradient computation (as matrix by matrix multiplications) for GEMMs (General Matrix Multiplications) are shown below. For example, for “(a) forward propagation . . . of a fully-connected layer” (the process of feeding input data through a neural network to generate an output), “K = # of inputs” and “M = # of outputs” and in “N = batch size” with results for “Input Activations,” “Output Activations,” and “Weights” (e.g., <i>parsing elements to be used in the computations representing the artificial neural network</i>).

Key Features	Accused Products
	<u>2. Fully-Connected Layer</u> <u>Fully-connected layers, also known as linear layers, connect every input neuron to every output neuron and are commonly used in neural networks.</u> <i>Figure 1. Example of a small fully-connected layer with <u>four input and eight output neurons.</u></i>  The diagram illustrates a fully-connected layer within a neural network. It consists of two vertical columns of light blue circular nodes. The left column, labeled 'Input Neurons', contains four nodes. The right column, labeled 'Output Neurons', contains eight nodes. Every node in the input column is connected to every node in the output column by a thin black line, representing a complete bipartite graph. The entire diagram is enclosed in a red rectangular border. *****

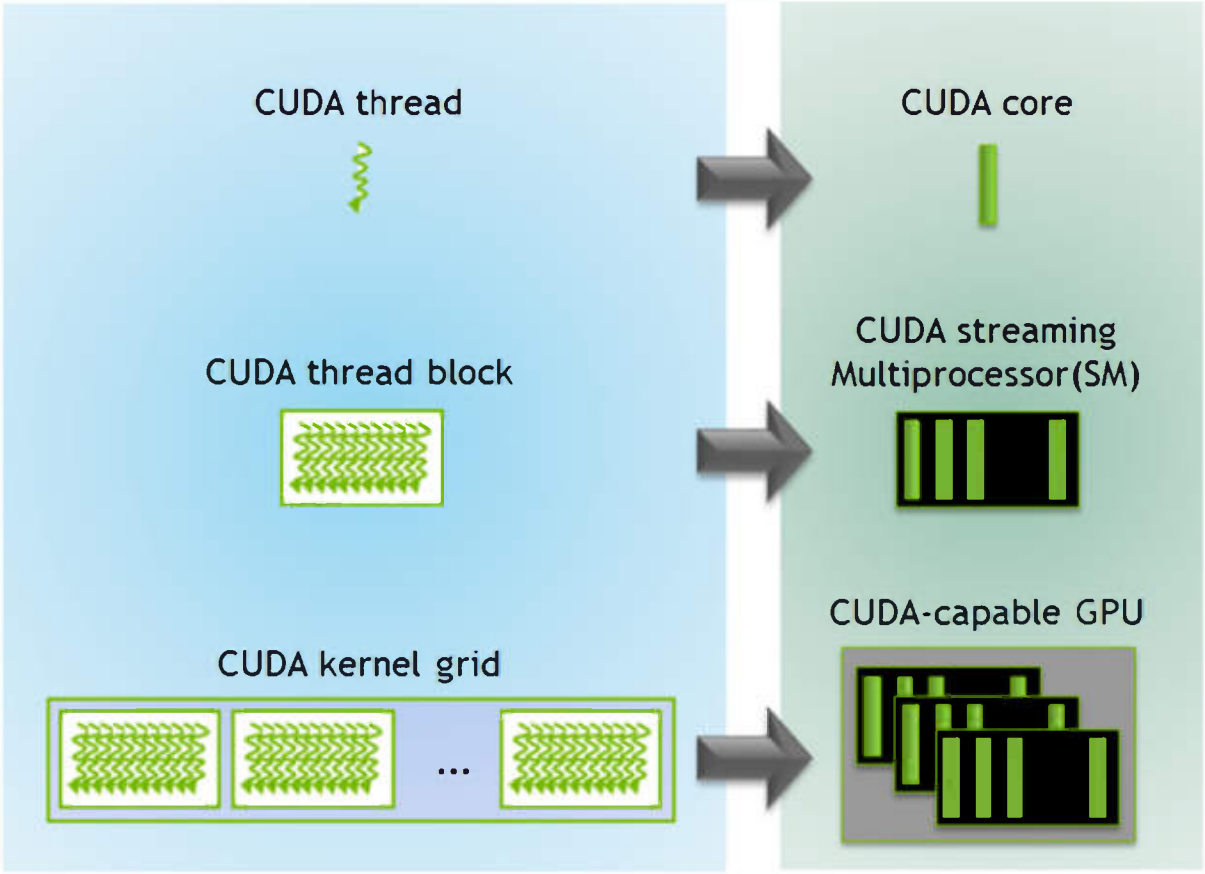
Key Features	Accused Products
	<i>Figure 2. Dimensions of equivalent GEMMs for (a) forward propagation, (b) activation gradient, and (c) weight gradient computations of a fully-connected layer.</i> Figure 2 illustrates the dimensions of equivalent GEMMs for three types of computations in a fully-connected layer: (a) Forward propagation: The input is a matrix of Input Activations with dimensions $K \times N$, where K is the number of inputs and N is the batch size. The weights are represented as Weights^T with dimensions $M \times K$, where M is the number of outputs. The output is a matrix of Output Activations with dimensions $M \times N$. (b) Activation gradient: The input is a matrix of Gradients (Input) with dimensions $M \times K$, where M is the number of inputs and K is the number of outputs. The other input is a matrix of Gradients (Output) with dimensions $K \times N$, where N is the batch size. The output is a matrix of Gradients (Weights) with dimensions $M \times N$. (c) Weight gradient: The input is a matrix of Input Activations with dimensions $M \times K$, where M is the number of inputs and K is the batch size. The other input is a matrix of Gradient (Output) with dimensions $K \times N$, where N is the number of outputs. The output is a matrix of Gradient (Weights) with dimensions $M \times N$. (See https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#performance (emphasis added).) CUDA library cuDNN function “cudnnSetRNNDescrptor_v8” “initializes a previously created RNN [recurrent neural network] descriptor object.” This function “store[s] all information needed to compute the total number of adjustable weights/biases in the RNN model.” In addition, the parameters “dirMode,” “inputMode,” and “datatype” confirm the exchange of calculations and values between the hidden layers

Key Features	Accused Products
	of an RNN (e.g., wherein executing the user interaction stream comprises: parsing elements to be used in the computations representing the artificial neural network). 7.2.49. <code>cudaSetRNNDescriptor_v8()</code> This function initializes a previously created RNN descriptor object. The RNN descriptor configured by <u><code>cudaSetRNNDescriptor_v8()</code> was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model.</u> <pre> cudaStatus_t cudaSetRNNDescriptor_v8(cudaRNNDescriptor_t rnnDesc, cudaRNNAlgo_t algo, cudaRNNMode_t cellMode, cudaRNNBiasMode_t biasMode, cudaDirectionMode_t dirMode, cudaRNNInputMode_t inputMode, cudaDataType_t dataType, cudaDataType_t mathPrec, cudaMathType_t mathType, int32_t inputSize, int32_t hiddenSize, int32_t projSize, int32_t numLayers, cudaDropoutDescriptor_t dropoutDesc, uint32_t auxFlags); </pre> *****

Key Features	Accused Products
	dirMode <i>Input.</i> Specifies the recurrence pattern: CUDNN_UNIDIRECTIONAL or CUDNN_BIDIRECTIONAL. <u>In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states.</u> inputMode <i>Input.</i> <u>Specifies how the input to the RNN model is processed by the first layer.</u> When inputMode is CUDNN_LINEAR_INPUT, original input vectors of size inputSize are multiplied by the weight matrix to obtain vectors of hiddenSize. When inputMode is CUDNN_SKIP_INPUT, the original input vectors to the first layer are used as is without multiplying them by the weight matrix. dataType <i>Input.</i> <u>Specifies data type for RNN weights/biases and input and output data.</u> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-891/pdf/cuDNN-API.pdf (emphasis added).)
[28] The method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises: converting the elements into GPU programs.	The Accused Products perform <i>the method of claim 27, wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises: converting the elements into GPU programs.</i> <i>See, e.g., [21.2], (“executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network”), supra.</i> (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>). For instance, the “CUDA programming model” includes programmatic functions, primitives, and executable libraries for CPUs and GPUs that are used by the Accused Products to perform numerical simulations. As previously stated, the host is the CPU and the device is the GPU. After “[c]opy[ing] the input data from host [CPU] memory to device [GPU] (e.g., <i>wherein the processing unit comprises a</i>

Key Features	Accused Products
	<i>graphics processing unit (GPU))]</i> memory,” the second main CUDA program execution step is “[<i>load[ing] the GPU program and execut[ing]</i>]” (e.g., <i>executing the user interaction stream further comprises: converting the elements into GPU programs</i>). Let me introduce two keywords widely used in CUDA programming model: <i>host</i> and <i>device</i>. The host is the CPU available in the system. The system memory associated with the CPU is called <i>host memory</i>. The GPU is called a <i>device</i> and GPU memory likewise called <i>device memory</i>. To execute any CUDA program, there are three main steps: • Copy the input data from host memory to device memory, also known as host-to-device transfer. • <u>Load the GPU program and execute</u>, caching data on-chip for performance. • <u>Copy the results from device memory</u> to host memory, also called device-to-host transfer. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).) As a further example, CUDA implements “kernels” running on streaming multiprocessors of Nvidia GPUs (e.g., <i>wherein the processing unit comprises a graphics processing unit (GPU) and executing the user interaction stream further comprises: converting the elements into GPU programs</i>). The CUDA programming model provides a heterogeneous environment where the host code is running the C/C++ program on the CPU and <u>the kernel runs on a physically separate GPU device</u>. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively. CUDA code also provides for data transfer between host and device memory, over the PCIe bus. * * * * *

Key Features	Accused Products
	Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be <u>migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism)</u>. One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time. Figure 3 shows the kernel execution and mapping on hardware resources available in GPU.

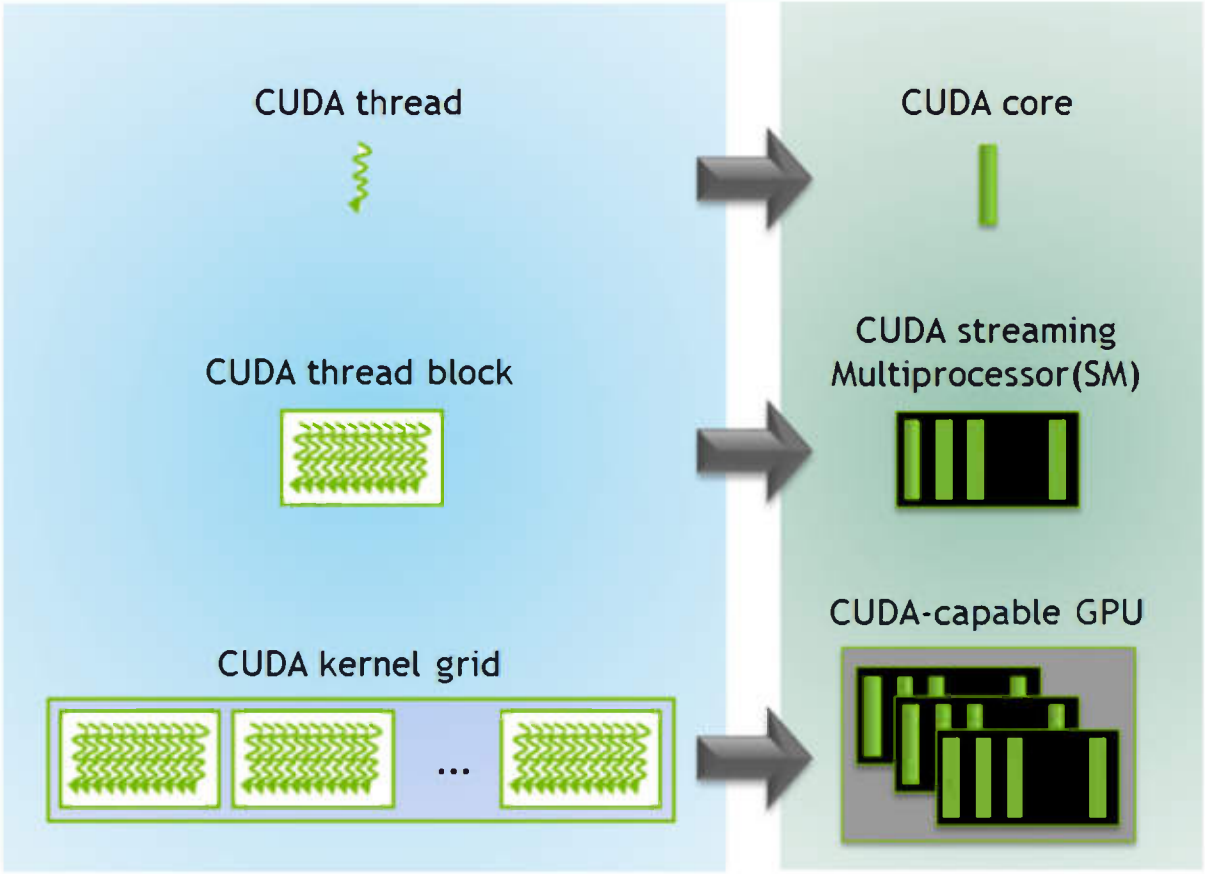
Key Features	Accused Products
	 <i>Figure 3. Kernel execution on GPU.</i> (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).)
[29] The method of claim 28, wherein executing the user interaction stream comprises:	The Accused Products perform <i>the method of claim 27, method of claim 28, wherein executing the user interaction stream comprises: compiling the GPU programs.</i>

Key Features	Accused Products
compiling the GPU programs.	See, e.g., [21.2], (“executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network”), <i>supra</i>. (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>)). For instance, the “CUDA programming model” includes programmatic functions, primitives, and executable libraries for CPUs and GPUs that are used by the Accused Products to perform numerical simulations. As previously stated, the host is the CPU and the device is the GPU. After “[c]opy[ing] the input data from host [CPU] memory to device [GPU] memory,” the second main CUDA program execution step is “[l]oad[ing] the GPU program and execut[ing]” (e.g., <i>wherein executing the user interaction stream comprises: compiling the GPU programs</i>). Let me introduce two keywords widely used in CUDA programming model: <i>host</i> and <i>device</i>. The host is the CPU available in the system. The system memory associated with the CPU is called host memory. The GPU is called a device and GPU memory likewise called device memory. To execute any CUDA program, there are three main steps: • Copy the input data from host memory to device memory, also known as host-to-device transfer. • <u>Load the GPU program and execute</u>, caching data on-chip for performance. • Copy the results from device memory to host memory, also called device-to-host transfer. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).) As an example, the Grace Hopper Superchip “is designed to accelerate applications” using “Extended GPU Memory.” As depicted in the gram of the Grace Hopper architecture below, a GPU (“HOPPER GPU”) can access “Local CPU,” “Peer CPU,” and “Peer GPU” memory via “NVLink” (e.g., <i>wherein executing the user interaction stream comprises: compiling the GPU programs</i>).

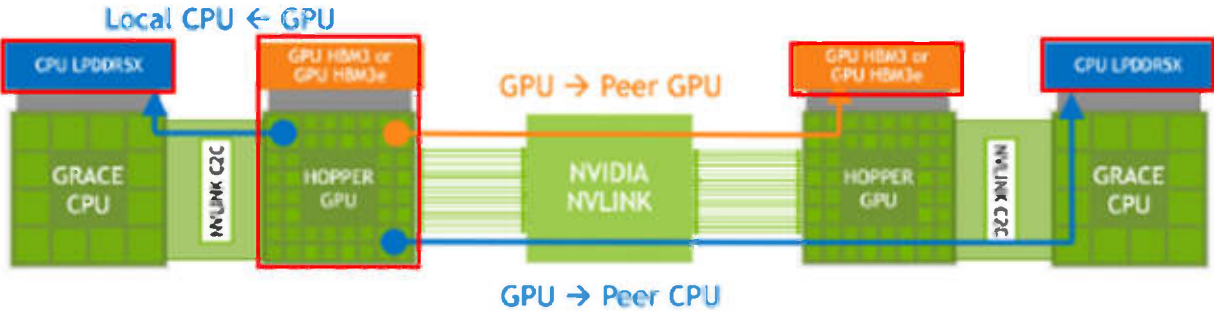
Key Features	Accused Products
	The diagram illustrates the memory access paths in a system with two Grace Hopper Superchips connected via an NVIDIA NVLink. Each Superchip is composed of a Grace CPU and a Hopper GPU. The left Superchip features a CPU LPDDR5X and GPU HBM3 or GPU HBM3e. The right Superchip features a GPU HBM3 or GPU HBM3e and CPU LPDDR5X. Arrows indicate data flow: Local CPU <-> GPU (blue), GPU -> Peer GPU (orange), and GPU -> Peer CPU (blue). Figure 5. Memory Accesses across NVLink-connected Grace Hopper Superchips (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).) For instance, CUDA library cuDNN function “<code>cudaSetRNNDescriptor_v8</code>” “initializes a previously created RNN [recurrent neural network] descriptor object” (e.g., <i>wherein executing the user interaction stream comprises: compiling the GPU programs</i>). This function “store[s] all information needed to compute the total number of adjustable weights/biases in the RNN model.” In addition, the parameters “<code>dirMode</code>,” “<code>inputMode</code>,” and “<code>datatype</code>” confirm the exchange of calculations and values between the hidden layers of an RNN.

Key Features	Accused Products
	7.2.49. <code>cudaSetRNNDescriptor_v8()</code> This function initializes a previously created RNN descriptor object. The RNN descriptor configured by <u><code>cudaSetRNNDescriptor_v8()</code></u> was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model. <pre> cudaStatus_t cudaSetRNNDescriptor_v8(cudaRNNDescriptor_t rnnDesc, cudaRNNAlgo_t algo, cudaRNNMode_t cellMode, cudaRNNBiasMode_t biasMode, cudaRNNDirectionMode_t dirMode, cudaRNNInputMode_t inputMode, cudaDataType_t dataType, cudaDataType_t mathPrec, cudaMathType_t mathType, int32_t inputSize, int32_t hiddenSize, int32_t projSize, int32_t numLayers, cudaDropoutDescriptor_t dropoutDesc, uint32_t auxFlags); </pre> * * * * * dirMode <i>Input.</i> Specifies the recurrence pattern: CUDNN_UNIDIRECTIONAL or CUDNN_BIDIRECTIONAL. <u>In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states.</u> inputMode <i>Input.</i> <u>Specifies how the input to the RNN model is processed by the first layer.</u> When inputMode is CUDNN_LINEAR_INPUT, original input vectors of size inputSize are multiplied by the weight matrix to obtain vectors of hiddenSize. When inputMode is CUDNN_SKIP_INPUT, the original input vectors to the first layer are used as is without multiplying them by the weight matrix. dataType <i>Input.</i> <u>Specifies data type for RNN weights/biases and input and output data.</u>

Key Features	Accused Products
	(See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-891/pdf/cuDNN-API.pdf (emphasis added).) As a further example, CUDA implements “kernels” running on streaming multiprocessors of Nvidia GPUs (e.g., <i>wherein executing the user interaction stream comprises: compiling the GPU programs</i>). The CUDA programming model provides a heterogeneous environment where the host code is running the C/C++ program on the CPU and the kernel runs on a <u>physically separate GPU device</u>. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively. CUDA code also provides for data transfer between host and device memory, over the PCIe bus. * * * * * Each <u>CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism)</u>. One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time. Figure 3 shows the kernel execution and mapping on hardware resources available in GPU.

Key Features	Accused Products
	 <i>Figure 3. Kernel execution on GPU.</i> (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).)
[30] The method of claim 29, wherein executing the user interaction stream comprises:	The Accused Products perform the <i>method of claim 29</i>, wherein <i>executing the user interaction stream comprises: transferring the GPU programs to the second memory partition.</i>

Key Features	Accused Products
transferring the GPU programs to the second memory partition.	See, e.g., [21.2], (“executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network”), <i>supra</i>. (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (A “user passes [operation descriptors] to the operation graph” for tensor operations) (e.g., <i>the user interaction stream</i>). For instance, the “CUDA programming model” includes programmatic functions, primitives, and executable libraries for CPUs and GPUs that are used by the Accused Products to perform numerical simulations. As previously stated, the host is the CPU and the device is the GPU. After “[c]opy[ing] the input data from host [CPU] memory to device [GPU] memory,” the second main CUDA program execution step is “[l]oad[ing] the GPU program and execut[ing], caching data on-chip for performance” (e.g., <i>wherein executing the user interaction stream comprises: transferring the GPU programs to the second memory partition</i>). Let me introduce two keywords widely used in CUDA programming model: <u>host and device</u>. <u>The host is the CPU available in the system.</u> The system memory associated with the CPU is called host memory. <u>The GPU is called a device</u> and GPU memory likewise called device memory. To execute any CUDA program, there are three main steps: • <u>Copy the input data from host memory to device memory, also known as host-to-device transfer.</u> • <u>Load the GPU program and execute, caching data on-chip for performance.</u> • Copy the results from device memory to host memory, also called device-to-host transfer. (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).)

Key Features	Accused Products
	As an example, the Grace Hopper Superchip “is designed to accelerate applications” using “Extended GPU Memory.” As depicted in the gram of the Grace Hopper architecture below, a GPU (“HOPPER GPU”) can access “Local CPU,” “Peer CPU,” and “Peer GPU” memory via “NVLink” (e.g., <i>wherein executing the user interaction stream comprises: transferring the GPU programs to the second memory partition</i>).  Figure 5. Memory Accesses across NVLink-connected Grace Hopper Superchips (See https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper (emphasis added).) In another example, the Accused Products implement the CUDA platform and its GPU-acceleration libraries. CUDA implements numerous “memory management functions” that “[c]op[y] data between host [CPU] and device [GPU]” (e.g., <i>transferring the GPU programs to the second memory partition</i>). This includes CUDA functions “cudaMemcpy,” “cudaMemcpy2D,” “cudaMemcpy2DArrayToArray,” “cudaMemcpy2DAsync,” “cudaMemcpy2DFromArray,” “cudaMemcpy2DFromArrayAsync,” “cudaMemcpy2DToArray,” “cudaMemcpy2DToArrayAsync,” and “cudaMemcpyAsync.”

Key Features	Accused Products
	6.11. Memory Management This section describes the memory management functions of the CUDA runtime application programming interface. Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module. Functions <pre> * * * * * __host__ cudaError_t cudaMemcpy (void* dst , const void* src Copies data between host and device. __host__ cudaError_t cudaMemcpy2D (void* dst , size_t dpitch Copies data between host and device. __host__ cudaError_t cudaMemcpy2DFromArray (cudaArray_t d Copies data between host and device. __host__ __device__ cudaError_t cudaMemcpy2DAsync (void* dst , size_t d Copies data between host and device. __host__ cudaError_t cudaMemcpy2DFromArray (void* dst , size Copies data between host and device. __host__ cudaError_t cudaMemcpy2DFromArrayAsync (void* dst , Copies data between host and device. __host__ cudaError_t cudaMemcpy2DToArray (cudaArray_t dst , Copies data between host and device. __host__ cudaError_t cudaMemcpy2DToArrayAsync (cudaArray_t d Copies data between host and device. __host__ cudaError_t cudaMemcpy3D (const cudaMemcpy3DParms* Copies data between 3D objects. __host__ __device__ cudaError_t cudaMemcpy3DAsync (const cudaMemcpy3DPa Copies data between 3D objects. __host__ cudaError_t cudaMemcpy3DPeer (const cudaMemcpy3DPe Copies memory between devices. __host__ cudaError_t cudaMemcpy3DPeerAsync (const cudaMemcpy Copies memory between devices asynchronously __host__ __device__ cudaError_t cudaMemcpyAsync (void* dst , const void Copies data between host and device. </pre>

Key Features	Accused Products
	(See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).)
[31] The method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk.	The Accused Products perform <i>the method of claim 21, further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk.</i> <i>See, e.g., [21.2] (“executing, by the processing unit, a computational stream, the computational stream controlling data exchange between the user interaction stream and the computational stream during execution of the computations representing the artificial neural network”) and [21.5] (“queueing a user command received by the user interaction stream during execution of the computations representing the artificial neural network”), supra.</i> For example, after the “host-to-device transfer” (CPU to GPU) and GPU program load and execution steps, the third main step is “[c]opy[ing] the results from device [GPU] memory to host [CPU] memory, also known as device-to-host transfer” (GPU to CPU) (e.g., <i>the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk</i>).

Key Features	Accused Products
	Let me introduce two keywords widely used in CUDA programming model: <i>host</i> and <i>device</i>. The host is the CPU available in the system. The system memory associated with the CPU is called host memory. The GPU is called a device and GPU memory likewise called device memory. To execute any CUDA program, there are three main steps: • Copy the input data from host memory to device memory, also known as host-to-device transfer. • Load the GPU program and execute, caching data on-chip for performance. • <u>Copy the results from device memory to host memory, also called device-to-host transfer.</u> (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).) For instance, CUDA library cuDNN function “<code>cudaSetRNNDescriptor_v8</code>” “initializes a previously created RNN [recurrent neural network] descriptor object.” This function “store[s] all information needed to compute the total number of adjustable weights/biases in the RNN model.” In addition, the parameters “<code>dirMode</code>,” “<code>inputMode</code>,” and “<code>datatype</code>” confirm the exchange of calculations and values between the hidden layers of an RNN (e.g., <i>further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk</i>).

Key Features	Accused Products
	7.2.49. <code>cudaSetRNNDescriptor_v8()</code> This function initializes a previously created RNN descriptor object. The RNN descriptor configured by <u><code>cudaSetRNNDescriptor_v8()</code></u> was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model. <pre data-bbox="575 427 1176 841"> cudaStatus_t cudaSetRNNDescriptor_v8(cudaRNNDescriptor_t rnnDesc, cudaRNNAlgo_t algo, cudaRNNMode_t cellMode, cudaRNNBiasMode_t biasMode, cudaRNNDirectionMode_t dirMode, cudaRNNInputMode_t inputMode, cudaDataType_t dataType, cudaDataType_t mathPrec, cudaMathType_t mathType, int32_t inputSize, int32_t hiddenSize, int32_t projSize, int32_t numLayers, cudaDropoutDescriptor_t dropoutDesc, uint32_t auxFlags); </pre> ***** <code>dirMode</code> <i>Input.</i> Specifies the recurrence pattern: <code>CUDNN_UNIDIRECTIONAL</code> or <code>CUDNN_BIDIRECTIONAL</code>. In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states. <code>inputMode</code> <i>Input.</i> Specifies how the input to the RNN model is processed by the first layer. When <code>inputMode</code> is <code>CUDNN_LINEAR_INPUT</code>, original input vectors of size <code>inputSize</code> are multiplied by the weight matrix to obtain vectors of <code>hiddenSize</code>. When <code>inputMode</code> is <code>CUDNN_SKIP_INPUT</code>, the original input vectors to the first layer are used as is without multiplying them by the weight matrix. <code>dataType</code> <i>Input.</i> Specifies data type for RNN weights/biases and input and output data.

Key Features	Accused Products
	(See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-891/pdf/cuDNN-API.pdf (emphasis added).) In another cuDNN example, an excerpt from Nvidia’s cuDNN release notes exemplifies “numerical results” from RNN API calls as used for deep learning using recurrent neural networks. A “kernel selection,” which prepares memory space and resources for use, “may affect numerical results.” cuDNN Release Notes NVIDIA CUDA Deep Neural Network (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications. These release notes describe the key features, software enhancements and improvements, and known issues for the NVIDIA cuDNN 8.9.3 and earlier releases. * * * * * • RNN and multihead attention API calls may exhibit nondeterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in Results Reproducibility, <u>numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This happens when two buffer sizes (16 KB and 4 MB) are used in the default configuration.</u> When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. <u>The kernel selection may affect numerical results.</u> The user can eliminate the nondeterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the <code>CUBLAS_WORKSPACE_CONFIG</code> environmental variable, for example, <code>:16:8</code> or <code>:4096:2</code>. The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is <code>:16:8:4096:2</code>, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the <code>:16:8</code> non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

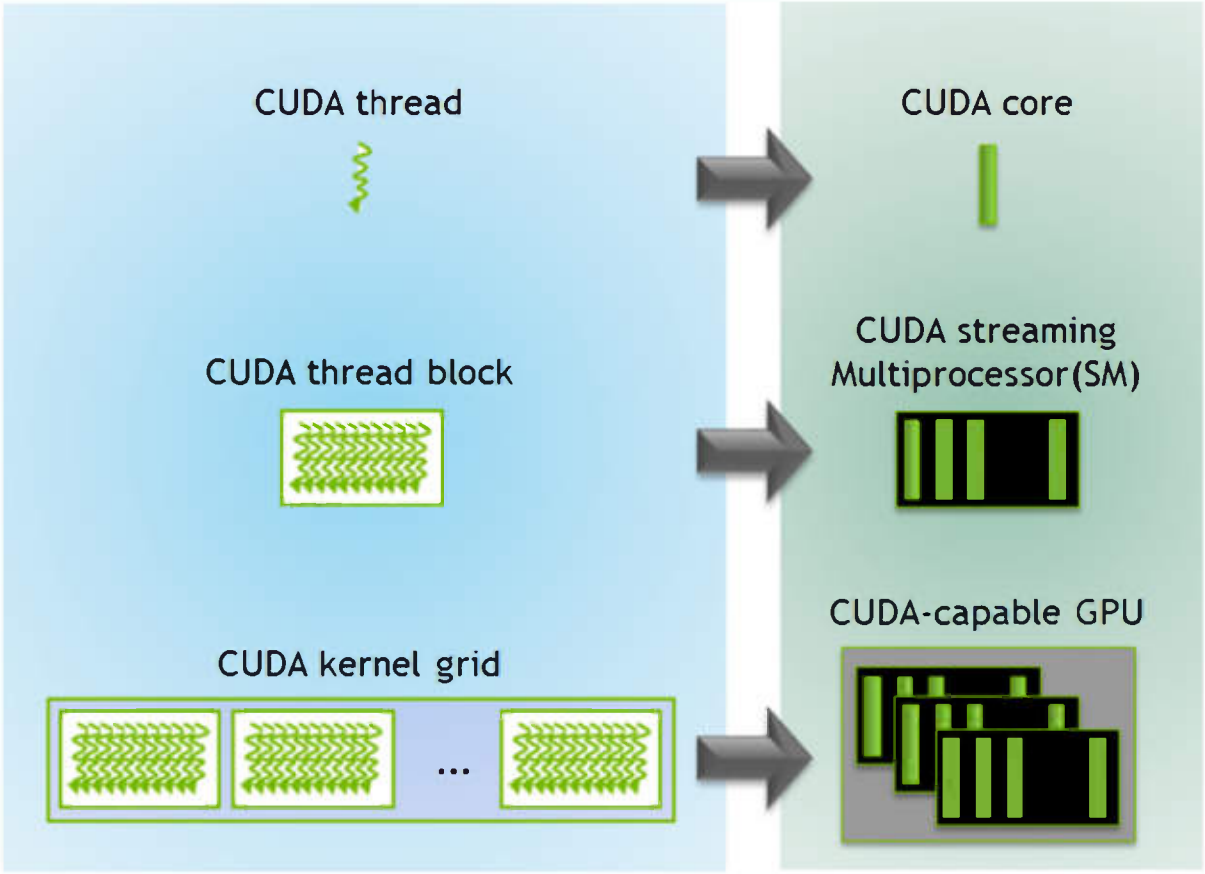
Key Features	Accused Products
	(See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-893/release-notes/index.html#abstract (emphasis added).) As shown by publicly available CUDA toolkit, CUDA implements exemplary “memory management functions” that “[c]op[y] data between host [CPU] and device [GPU]” (e.g., <i>the data output stream controlling transfer of outputs</i>). This includes CUDA functions “cudaMemcpy” and “cudaMemcpyAsync.” 6.11. Memory Management This section describes the memory management functions of the CUDA runtime application programming interface. Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module. Functions <pre> * * * * * __host__ <u>cudaError_t</u> <u>cudaMemcpy</u> (void* dst , const void* src <u>Copies data between host and device.</u> * * * * * __host__ __device__ <u>cudaError_t</u> <u>cudaMemcpyAsync</u> (void* dst , const void <u>Copies data between host and device.</u> </pre> (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).) As an example, exemplary CUDA memory management function “cudaMemcpyAsync” “[c]opies count bytes [data] from the memory area pointed to by src [source memory address pointer] to the memory area pointed to by dst [destination memory address pointer], where kind [type of transfer] specifies the direction of the copy.” Destinations includes “cudaMemcpyHostToDevice [CPU to device GPU], cudaMemcpyDeviceToHost [GPU to CPU], cudaMemcpyDeviceToDevice [GPU to GPU] (e.g., <i>executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk</i>). Because the function “cudaMemcpyAsync() is asynchronous with respect to the host, [] the call may return before the copy is

Key Features	Accused Products
	complete. The copy can optionally be associated to a stream [identified stream] by passing a non-zero stream argument.” <pre data-bbox="573 349 1694 402">__host__ __device__ cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)</pre> <u>Copies data between host and device.</u> Parameters dst - Destination memory address src - Source memory address count - Size in bytes to copy kind - Type of transfer stream - Stream identifier Returns cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemoryDirection Description Copies <code>count</code> bytes from the memory area pointed to by <code>src</code> to the memory area pointed to by <code>dst</code>, where <code>kind</code> specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. The memory areas may not overlap. Calling cudaMemcpyAsync() with <code>dst</code> and <code>src</code> pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpyAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero <code>stream</code> argument. If <code>kind</code> is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and the <code>stream</code> is non-zero, the copy may overlap with operations in other streams. The device version of this function only handles device to device copies and cannot be given local or shared pointers. (See https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (emphasis added).)

Key Features	Accused Products
	In addition, cuDNN operations below exemplify “tak[ing] tensors as input and produc[ing] tensors as output” (e.g., <i>the data output stream controlling transfer of outputs of the computations representing the artificial neural network</i>). As shown below for a “forward convolution operation” using cuDNN, “backend tensor <i>Tmp0</i> [is] both the output of the convolution operation and the input of the bias operation.” From this, “cuDNN infers that the dataflow runs from the convolution into the bias.” 2.2. Tensors and Layouts Whether using the graph API or the legacy API, <u>cuDNN operations take tensors as input and produce tensors as output.</u> * * * * * 3.2. Graph API Example with Operation Fusion The following example implements a fusion of convolution, bias, and activation. 3.2.1. Creating Operation and Tensor Descriptors to Specify the Graph Dataflow First, create three cuDNN backend operation descriptors. As can be seen in Figure 6, the user specified one forward convolution operation (using <code>CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR</code>), a pointwise operation for the bias addition (using <code>CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR</code> with mode <code>CUDNN_POINTWISE_ADD</code>), and a pointwise operation for the ReLU activation (using <code>CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR</code> with mode <code>CUDNN_POINTWISE_RELU_FWD</code>). Refer to the NVIDIA cuDNN Backend API for more details on setting the attributes of these descriptors. For an example of how a forward convolution can be set up, refer to the Setting Up An Operation Graph For A Grouped Convolution use case in the cuDNN backend API. You should also create tensor descriptors for the inputs and outputs of all of the operations in the graph. <u>The graph dataflow is implied by the assignment of tensors (refer to Figure 6), for example, by specifying the backend tensor <i>Tmp0</i> as both the output of the convolution operation and the input of the bias operation, cuDNN infers that the dataflow runs from the convolution into the bias.</u> The same applies to tensor <i>Tmp1</i>. If the user doesn’t need the intermediate results <i>Tmp0</i> and <i>Tmp1</i> for any other use, then the user can specify them to be virtual tensors, so the memory I/Os can later be optimized out. * * * * *

Key Features	Accused Products
	Figure 6. A set of operation descriptors the user passes to the operation graph (See https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-881/developer-guide/index.html#tensors-layouts (emphasis added).) As a further example, CUDA implements “kernels” running on streaming multiprocessors of Nvidia GPUs. “CUDA code [] provides for data transfer between host and device memory.” CUDA provides a “heterogenous environment” where the host code is running the C/C++ program on the CPU” (e.g., <i>further comprising: executing, by the at least one CPU, a data output stream, the data output stream controlling transfer of outputs of the computations representing the artificial neural network to disk</i>)).

Key Features	Accused Products
	The CUDA programming model provides a heterogeneous environment where the host code is running the C/C++ program on the CPU and <u>the kernel runs on a physically separate GPU device</u>. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces, referred to as host memory and device memory, respectively. CUDA code also provides for data transfer between host and device memory, over the PCIe bus. * * * * * Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be <u>migrated to other SMs in GPU (except during preemption, debugging, or CUDA dynamic parallelism)</u>. One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks. Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time. Figure 3 shows the kernel execution and mapping on hardware resources available in GPU.

Key Features	Accused Products
	 <i>Figure 3. Kernel execution on GPU.</i> (See https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (emphasis added).)