ChatGPT

Optimizing AI Model Performance on NVIDIA GPUs (Windows Guide)

Writing and Compiling Custom CUDA Kernels on Windows

Developing custom CUDA kernels on Windows involves writing GPU-accelerated C/C++ code (`.cu` files) and compiling it with NVIDIA's CUDA Toolkit. The typical workflow is: (1) **Write** CUDA C/C++ code with `__global__` functions (GPU kernels) and host functions to launch them; (2) **Integrate** the `.cu` file into a Visual Studio project or build system; and (3) **Compile** using *NVCC* (the CUDA compiler) which wraps the Microsoft Visual C++ compiler for host code.

In Visual Studio, after installing the CUDA Toolkit, you can start a new project using the CUDA template or add CUDA support to an existing project. For new projects, Visual Studio's CUDA templates (e.g. "CUDA 13.0 Runtime") will preconfigure the build with NVIDIA's custom build rules [1]. For existing projects, you must enable CUDA file compilation by ticking the CUDA build customizations: *Project* → *Build Dependencies* → *Build Customizations...* and select the appropriate CUDA Toolkit version [2] [3]. This ensures `.cu` files are recognized and compiled with NVCC. When adding a new source, mark it as a *CUDA C/C++ file* so Visual Studio knows to use NVCC [4]. NVCC will generate GPU device code and host CPU code, linking against the NVIDIA runtime. You can then build and run the program like a normal C++ application (ensuring the NVIDIA driver is installed).

Alternatively, CUDA code can be compiled via command-line. For example, to compile `mykernel.cu` to an executable, one might run: `nvcc mykernel.cu -o myprog.exe` (with flags for architecture, etc.). NVCC will invoke MSVC internally on Windows (so a compatible MSVC version must be installed) [5]. The CUDA Toolkit installer sets up environment variables like `%CUDA_PATH%` and includes Visual Studio integration files, making it straightforward to compile CUDA code inside the IDE or via MSBuild [6] [7].

As a simple example, consider a vector addition kernel:

```
// CUDA kernel for vector addition
__global__ void vecAdd(const float* A, const float* B, float* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N) {
        C[i] = A[i] + B[i];
    }
}
```

This kernel adds two arrays `A` and `B` into `C`. We could launch it from host code with something like: `int threads=256; int blocks=(N+255)/256; vecAdd<<<blocks, threads>>>(A, B, C, N);`. After writing such a kernel, we compile with NVCC. On Windows, the CUDA Toolkit provides Visual Studio project templates which include sample code; for instance, building the `bandwidthTest` or `deviceQuery` samples verifies the toolchain is set up correctly [8] [9]. Once compiled, you can run the program and the CUDA kernels will execute on the NVIDIA GPU.

Note: Ensure that your GPU is CUDA-compatible and the correct NVIDIA driver is installed. In Windows, 64-bit development is standard for CUDA (32-bit support is dropped in recent CUDA versions). Also, be mindful of the MSVC toolset version supported by your CUDA Toolkit (e.g., CUDA 12 supports VS2019/2022). Visual Studio integration simplifies editing and debugging: you can set breakpoints in device code using Nsight (discussed later). If you prefer not to use Visual Studio, you can use Visual Studio Code or other editors with CMake: just call NVCC in your build scripts. (Visual Studio Code with the CUDA Toolkit is supported via the Nsight plugin [10].) The main goal is to successfully compile the `.cu` file into an application or library that can run on the GPU.

Key Limitations and Constraints in GPU Kernel Development

GPU programming introduces several hardware and software constraints that developers must consider. **Hardware-wise**, NVIDIA GPUs execute kernels in a SIMT (single-instruction, multiple-thread) fashion with a hierarchy of threads, thread blocks, and grids. Each thread block can contain up to 1024 threads (this is an architectural limit) [11], and threads within a block can cooperate via fast shared memory and synchronization. Threads are grouped into warps of 32 that execute in lockstep, so divergent execution (threads taking different branches) can reduce efficiency. Each Streaming Multiprocessor (SM) on the GPU executes many threads in parallel, but a given block is always scheduled entirely on one SM and cannot migrate [12]. This means the block size and grid size impact occupancy (how fully the GPU is utilized). There are also limits on resource usage: for example, each block has a limited amount of **shared memory** (on-chip scratchpad memory) and registers available, and using too many registers per thread can limit how many threads run concurrently.

Memory hierarchy of an NVIDIA GPU (example from Ampere architecture). GPU threads have access to various memory levels: each thread has its own registers, each thread block has shared memory (L1 cache), and all threads access global (device) memory through L2 cache. [13]

As shown above, GPU memory is hierarchical. **Registers** are private to each thread (fastest access, but very limited in total) [13]. Each SM has a **shared memory** (L1 cache) that is accessible by all threads in a block for explicit cooperation – this memory is much faster than global memory but is limited (tens of KB per block). There are also read-only caches for constant and texture memory. All blocks and threads can access the large **global (device) memory** on the GPU, but accesses to it have higher latency and throughput limits (mitigated by caches and memory coalescing techniques). Effective kernel implementations try to maximize use of registers and shared memory and minimize global memory traffic (or access it in a coalesced pattern) to achieve high performance. In practice, this means you may need to use tiling techniques: load chunks of data into shared memory, reuse it for multiple computations, and thereby reduce expensive global memory accesses.

Beyond hardware threading and memory limits, **software constraints** in CUDA include the lack of certain C++ features on device code (no dynamic memory allocation with `new/delete` on device, no full C++ standard library, limited recursion or function pointers, etc. on device). Kernels must be written to avoid system calls or blocking I/O – they run on the GPU device with no OS services. Each kernel should also execute within a watchdog time limit when running on a display-attached GPU in Windows. Notably, Windows imposes a **Timeout Detection and Recovery (TDR)** for GPUs: if a kernel runs longer than ~2 seconds without yielding, the OS will consider the driver unresponsive and reset the GPU driver [14] [15]. This means long-running kernels can be aborted on Windows, which is a constraint not present (by default) on Linux. Developers must split large workloads into smaller kernel launches or disable the timeout (not recommended unless absolutely necessary, as it requires registry tweaks and can affect system stability) [15]. A better solution is to use a GPU that is not the primary display or to use Tesla/Workstation GPUs that support **TCC mode**.

Under Tesla Compute Cluster (TCC) mode (available on NVIDIA Tesla and some Quadro/Titan GPUs), the GPU operates as a pure compute device without a desktop timeout [16] [17]. This eliminates the TDR issue and also bypasses some Windows Display Driver Model (WDDM) overhead. However, GeForce gaming GPUs on Windows must run in WDDM (they cannot use TCC except the Titan class) [18]. As a result, there is an inherent driver overhead on Windows for consumer GPUs: kernel launch batching and scheduling under WDDM can introduce additional latency. A known consequence is that back-to-back kernel launches might be **batched** by the Windows driver to amortize overhead, but this can add unpredictability. NVIDIA experts note that under WDDM, GPU commands may incur a significant overhead and variable latency, whereas switching to TCC (on supported GPUs) avoids this batching [19] [20]. In summary, on Windows with a GeForce GPU, you should keep kernel execution times short and frequent (to avoid TDR and pipeline stalls), and if possible insert `cudaDeviceSynchronize()` or use `cudaStreamQuery` to flush work if you suspect excessive batching [21] [22].

Another constraint is **memory capacity**: GPUs have limited VRAM, and running out of memory will cause failures. Unlike CPU memory, GPU memory is not automatically paged to system RAM (unless using Unified Memory oversubscription, which has severe performance penalties). This means models and data must be carefully sized to fit in GPU memory. Tools like NVIDIA's *Compute Sanitizer* can help detect out-of-bounds or illegal memory accesses on GPU, but debugging memory issues is generally harder than on CPU. It's important to use APIs like `cudaMallocManaged` (Unified Memory) or host-pinned memory to optimize transfers (discussed next) if applicable, but also to free GPU memory when not needed to avoid leaks.

In summary, writing efficient CUDA kernels on Windows requires awareness of GPU hardware limits (threads, memory hierarchy) and OS-related constraints (the WDDM driver model's watchdog and overhead). With these in mind, developers can craft kernels that play to the GPU's strengths: massive parallelism with thousands of threads, high memory bandwidth (if accesses are coalesced), and overlapping computation with data movement.

Essential Tools and Libraries for GPU Development

To effectively develop and optimize GPU applications, NVIDIA provides a rich ecosystem of tools and libraries. On Windows, you will typically install the **CUDA Toolkit**, which includes the GPU compiler, drivers, and fundamental libraries. For deep learning, you will also use libraries like **cuDNN** for neural network primitives and **TensorRT** for inference optimization. Profiling and debugging rely on **Nsight** tools. Each of these components plays a specific role:

CUDA Toolkit (Compiler and Core Libraries)

The NVIDIA CUDA Toolkit is the fundamental software development kit for GPU computing. It includes the NVCC compiler, the CUDA runtime, and numerous optimized libraries (for BLAS, FFT, etc.) and utilities. The toolkit enables integration of GPU computing into C++ projects and provides the necessary headers and link libraries. On Windows, the CUDA Toolkit installer will also setup Visual Studio integration and environment variables [6] [7]. When you install CUDA (e.g., CUDA 12 or 13), you choose the Visual Studio version integration, and the toolkit places **Build Customization files** (`CUDA *.props` and `.rules`) so that Visual Studio knows how to compile `.cu` sources [23] [24]. It also automatically installs the **NVIDIA drivers** (if not already present) and **Nsight Visual Studio Edition** for debugging/profiling [25].

Crucially, the CUDA Toolkit comes with *thousands of code samples and documentation*. For example, it provides the CUDA **Programming Guide** and **Best Practices Guide** (which are must-reads for

performance tuning). It also includes command-line tools like `nvprof` / `nvsys` (on older versions) or the new Nsight CLI tools for profiling, and `cuda-memcheck` (now part of Compute Sanitizer) for memory checking. The core libraries packaged with the toolkit cover many HPC and AI needs: **cuBLAS** (dense linear algebra), **cuFFT** (Fast Fourier Transforms), **cuSPARSE** (sparse matrices), **cuRAND** (random number generation), **Thrust** (C++ parallel algorithms on GPU), etc. Using these libraries is often preferable to writing custom kernels, as they are highly optimized. For instance, multiplying matrices or performing convolution via cuBLAS or cuDNN (next section) will typically achieve near-peak hardware throughput.

In summary, the CUDA Toolkit provides the base infrastructure to write and compile GPU code. On Windows, ensure you have a toolkit version compatible with your GPU and Visual Studio. The toolkit's installer also sets environment variables like `CUDA_PATH`, which Visual Studio uses to locate compiler and libraries [26] [27]. Whenever possible, leverage the Toolkit's optimized libraries rather than reinventing the wheel, as this will save time and yield better performance.

NVIDIA cuDNN (CUDA Deep Neural Network library)

For anyone working on deep learning models (CNNs, RNNs, etc.), **cuDNN** is an indispensable library. The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks [28]. It provides highly optimized implementations for standard neural network operations such as **forward and backward convolution**, pooling, activation functions, normalization (batchnorm), recurrent neural network cells (LSTM/GRU), and more [28]. Essentially, cuDNN serves as the backend for many deep learning frameworks on NVIDIA GPUs – when you call a high-level API like TensorFlow's `tf.nn.conv2d` or PyTorch's `torch.nn.Conv2d`, under the hood it will often use cuDNN to leverage GPU acceleration.

cuDNN is optimized by NVIDIA engineers to exploit each GPU architecture's features (like using Tensor Cores on Volta/Ampere for FP16 matrix ops, using Winograd algorithms for conv, etc.). Using cuDNN can dramatically speed up model training and inference. For example, when cuDNN added optimized support for LSTM RNNs, it delivered up to **6× speedup** over previous GPU implementations of LSTMs [29]. These gains come from kernel fusion and low-level optimizations that would be very complex to reproduce manually. cuDNN also often handles details like workspace memory (for certain algorithms) and chooses the fastest algorithm for a given operation (many cuDNN functions allow an "auto-tune" or "find" mode to select the best algorithm for your tensor dimensions).

From a developer perspective, you typically don't call cuDNN directly unless you are writing custom GPU code that needs NN operations. Instead, your chosen deep learning framework will link against cuDNN. On Windows, cuDNN is distributed as a separate download (due to licensing) – you would download the cuDNN zip for Windows (matching your CUDA version), and copy the `cudnn64_<ver>.dll` into your CUDA Toolkit folder (`cuda\bin`) and the headers into `cuda\include`. After that, frameworks like TensorFlow or PyTorch will automatically use it if installed. Always ensure the cuDNN version is compatible with your framework version (their documentation will specify required versions).

In short, **cuDNN accelerates neural network layers** and is crucial for training high-performance models on NVIDIA GPUs. It contributes both to training and inference: e.g., it speeds up convolution and pooling in CNN training, and also provides optimized inference for those layers (including support for half-precision and INT8 in some cases to boost inference). Whenever you write custom GPU code for deep learning, consider using cuDNN primitives rather than low-level CUDA C if possible – you'll benefit from the years of optimization already done.

NVIDIA Nsight Tools (Profiling and Debugging)

Optimizing GPU programs requires visibility into their behavior. NVIDIA's **Nsight** tool suite provides this visibility, with specialized tools for debugging and profiling GPU-accelerated applications:

- **Nsight Visual Studio Edition**: Integrates into Visual Studio on Windows, allowing you to debug GPU kernels similar to CPU debugging. You can set breakpoints in CUDA C code, step through kernel execution, inspect GPU memory, and see race conditions or assertions. This is extremely helpful for ensuring your kernel logic is correct. Nsight VS Edition also includes a basic profiler and timeline viewer inside Visual Studio.

- **Nsight Systems**: A system-wide profiler that captures *timelines* of CPU and GPU events. It helps answer questions like: Is my GPU being fully utilized? Is my CPU preparing work fast enough? Where are the stalls? It visualizes events such as kernel launches, memory copies, CUDA API calls, and CPU functions on a timeline. This is great for finding *bottlenecks at a high level*, e.g., if your GPU is idle because the data loader is slow or if there are serialization points between CPU-GPU.

- **Nsight Compute**: A deep dive *kernel profiler*. This tool analyzes individual GPU kernel executions in detail, providing metrics like achieved occupancy, memory throughput, instruction throughput, warp divergence, and more. Nsight Compute collects hardware counter metrics to show how your kernel utilized the GPU (for example, how many DRAM bytes were read, how effective the L1 cache was, the arithmetic intensity, etc.) [30]. It also can provide guidance by highlighting likely bottlenecks (memory-bound vs compute-bound) and suggesting optimizations. Nsight Compute is an interactive GUI and also has a command-line version. It's essential when you need to optimize a custom kernel – you can see if, say, 50% of time is spent waiting on memory, or if half the threads are inactive due to branch divergence.

- **Compute Sanitizer (cuda-memcheck)**: This is not branded "Nsight" but is another toolkit tool – it can detect memory errors (out-of-bounds, illegal accesses) and race conditions in GPU code, analogous to Valgrind or AddressSanitizer for CPU. Use it if you suspect memory corruption in kernels.

Using these tools, you can iteratively improve performance. For example, you might run Nsight Systems to see that GPU kernels are short but there's a long gap between them – indicating CPU overhead or synchronization issues. Then you could inspect CPU code or batching logic to fix that. Or, you might see GPU is busy 100% but overall throughput is low; then Nsight Compute could reveal the kernel is memory-bandwidth bound (e.g., lots of cache misses). You could then try optimizing memory access patterns or use shared memory to alleviate this. In sum, Nsight tools are critical for performance tuning: they let you **measure and visualize** what's happening, which guides your optimization efforts. NVIDIA includes these tools with the CUDA Toolkit (Nsight VS Edition is installed by default on Windows [25], and Nsight Systems/Compute can be downloaded from NVIDIA or via the NVIDIA Nsight Compute CLI in the toolkit).

NVIDIA TensorRT (High-Performance Inference Optimizer)

TensorRT is an SDK and runtime focused on optimizing trained neural network models for **inference** deployment. After you've trained a model (in PyTorch, TensorFlow, etc.), TensorRT can take the model and **convert it into a highly optimized engine** that runs faster on NVIDIA GPUs. According to NVIDIA's documentation, *"NVIDIA TensorRT is an SDK for optimizing trained deep-learning models to enable high-performance inference. TensorRT contains a deep learning inference optimizer and a runtime for*

execution.” [31]. In practice, TensorRT performs a variety of optimizations: it fuses layers/kernels, optimizes memory usage, selects efficient kernel implementations for the target GPU, and can quantize weights and activations to lower precision (FP16 or INT8) for speedup. The output is a binary **engine** that can be loaded and executed with the TensorRT runtime API for fast inference.

How does TensorRT benefit model performance? It can dramatically reduce latency and increase throughput. Many real-time services (e.g., video analytics, speech recognition, recommender systems) see 2×–5× inference speedups using TensorRT optimizations [32]. For example, TensorRT can take a ResNet50 model and combine convolution, batch norm, and ReLU layers into one fused kernel, saving memory bandwidth and launch overhead. It will also exploit Tensor Cores by using FP16 or INT8 arithmetic, which can significantly boost throughput on Turing, Ampere, etc., without much loss in accuracy (INT8 requires calibration or quantization-aware training to maintain accuracy). In one case, integrating TensorRT with PyTorch (via Torch-TensorRT) achieved up to **6× faster inference** on GPUs by using mixed precision and engine optimizations [33].

To use TensorRT on Windows, you would typically either use it through a framework integration or via the C++/Python API. For instance, TensorFlow has **TF-TRT**, an integration that automatically converts parts of a TensorFlow graph into TensorRT engines. This hybrid approach means you can get TensorRT performance gains *with only a few lines of code inside TensorFlow* [34]. The framework will partition the model: subgraphs that are supported by TensorRT will run as TensorRT optimized engines, while the rest runs as normal in TensorFlow [35]. This gives the “best of both” – flexibility of the framework and speed of TensorRT. The user guide notes: *“TF-TRT partitions the TensorFlow graph and delegates execution of compatible subgraphs to TensorRT, while the rest run in TensorFlow – providing both flexibility and performance.”* [35]. Importantly, you can develop and test your model in TensorFlow or PyTorch, then **deploy** with TensorRT without rewriting the model in C++. NVIDIA highlights that you get the performance boost *“with just a few additional lines of code, without having to develop in C++ using TensorRT directly”* [34].

Outside of TensorFlow, you can also use TensorRT with PyTorch via the *Torch-TensorRT* integration. Torch-TensorRT allows you to feed a `torch.nn.Module` (or a TorchScript model) into TensorRT and get back an accelerated module. It will compile the model (or parts of it) to a TensorRT engine and then you can run inference as normal in PyTorch. As mentioned, this can give multi-x speedups and integrates seamlessly – for example, `torch_tensorrt.compile(module, enabled_precisions={torch.float16})` might return a module that is much faster on GPU for inference. Under the hood, Torch-TensorRT will perform layer fusion and use TensorRT’s tactics to optimize the graph, while any unsupported operations fall back to regular PyTorch, ensuring correctness [36] [37]. This hybrid execution is similar to TF-TRT’s approach. The bottom line is that **TensorRT is used to squeeze maximum inference performance** out of NVIDIA GPUs by using lower precision and optimized kernels. It’s particularly beneficial for production environments where latency per inference or throughput is critical (e.g., serving many queries per second).

Keep in mind that TensorRT is primarily for **inference** (it does not help with training, which is where cuDNN and the frameworks’ own optimizers focus). Also, using INT8 in TensorRT requires calibration with sample data to ensure minimal accuracy loss. The tooling provides calibrators for this. Lastly, TensorRT engines are specific to a model and GPU model (they incorporate kernel selection optimized for a specific GPU architecture), so if you upgrade GPUs or change model topology, you often need to rebuild the TensorRT engine. Despite these considerations, TensorRT is a powerful tool to achieve production-grade performance on NVIDIA hardware.

Techniques and Best Practices for Optimizing AI Model Performance

Optimizing performance on NVIDIA GPUs involves addressing both **training throughput** (for faster model training) and **inference latency/throughput** (for deployment). The following best practices and techniques apply broadly to AI workloads on GPUs:

Memory Management and Data Transfers

Efficient memory management is crucial for performance. GPU global memory (VRAM) is much faster than CPU RAM, but still an order of magnitude slower than on-chip cache. To maximize speed:

- **Minimize CPU-GPU transfers**: Data should reside on the GPU as much as possible. Transferring data across PCIe is relatively slow (a few GB/s) compared to GPU's internal bandwidth (hundreds of GB/s). For training, load entire batches or datasets onto GPU if memory allows, rather than moving data each iteration. Use **batching** to amortize transfer costs. When transfers are needed, use asynchronous copies (`cudaMemcpyAsync`) on a non-default stream to overlap communication with computation.

- **Use pinned (page-locked) host memory** for transfers: When copying data from host to device (or vice versa), allocate the host memory as page-locked if possible (e.g., `cudaHostAlloc` or setting PyTorch DataLoader with `pin_memory=True`). Pinned memory transfers are much faster because they avoid an extra copy via pageable memory. This can significantly speed up input pipeline in training – e.g., a DataLoader producing batches in pinned memory can copy to GPU ~2–3× faster than from normal pageable memory.

- **Overlap data transfers with computation**: GPUs support concurrency via streams. You can launch a kernel on one stream while copying data on another. For example, while the GPU is processing batch *n*, you can asynchronously copy batch *n+1* into GPU memory on a separate stream, so it's ready by the time the current batch finishes. This *pipeline parallelism* hides communication latency. Frameworks often do this under the hood (TensorFlow's runtime and PyTorch's asynchronous execution both try to overlap device computation with host preprocessing when possible).

- **Reuse GPU memory (memory pooling)**: Frequent allocations/deallocations of GPU memory (via `cudaMalloc` / `cudaFree`) can be slow. Deep learning frameworks use memory pools (allocators) to avoid returning memory to the OS frequently – instead they keep a cache of GPU memory to service future requests. As a developer, you should also aim to allocate once and reuse buffers if writing custom code. If using frameworks, prefer in-place operations or pre-allocated tensors to prevent unnecessary new allocations. Also be mindful of memory fragmentation (sometimes long-running processes fragment the GPU memory – frameworks' allocators mitigate this, but if writing custom CUDA code, consider using something like the *cub* caching allocator or similar).

- **Consider Unified Memory for simplicity, but prefer explicit memory management for performance**: Unified Memory (using `cudaMallocManaged`) allows the CUDA system to manage moving data between CPU/GPU for you. While convenient (especially for simpler programs), it might introduce unpredictable performance (e.g., page faults at runtime when data migrates). In ML workloads where data access patterns are well-known, it's usually better to manually control data placement (i.e., explicitly copy what you need to the GPU at the right time)

rather than rely on Unified Memory's on-demand paging. Unified Memory is more useful when porting algorithms that have complex memory access patterns or when you need memory oversubscription (which is generally to be avoided in deep learning due to severe slowdowns).

- **Memory coalescing and access patterns**: In custom GPU kernels, arrange memory accesses so that threads access contiguous memory addresses where possible. Coalesced accesses (adjacent threads accessing adjacent addresses) maximize global memory throughput. Strided or scattered accesses can cause serialization and cache thrashing. Sometimes changing data layout (e.g., from AoS to SoA – Array of Structures to Structure of Arrays) yields big performance improvements because it aligns memory access with how threads are organized.

In summary, treat GPU memory as a precious resource. Keep data on device, move it efficiently if needed (asynchronously and in large chunks), and use the fastest memory available (shared memory, registers) for intermediate calculations. This reduces the time threads spend waiting on memory and increases overall utilization of the GPU's computing units.

Maximizing Parallelism and GPU Utilization

NVIDIA GPUs achieve performance through massive parallelism – tens of thousands of threads can run concurrently. To utilize the GPU fully:

- **Use large batch sizes (for training)** when memory allows. Larger batches mean more data parallel work that the GPU can perform in one go, which often leads to better occupancy and higher ALU utilization. There is a limit to this (too large a batch can exhaust memory or hit diminishing returns), but in general batch size is a key knob. For example, training CNNs often scales well with batch size until the GPU is saturated. However, for very large models (like certain transformers), batch size might be limited by memory – in such cases, consider gradient accumulation as an alternative to simulate larger batch.

- **Identify parallelizable work and offload it to the GPU**. This sounds obvious, but it's worth analyzing your application with Amdahl's Law in mind. If your CPU code is doing a lot of preprocessing or other computations per batch, that can become a bottleneck. Try to push computable work into CUDA kernels – e.g., if you are augmenting images for training, libraries like DALI (NVIDIA's Data Loading Library) can perform those augmentations on GPU. Or if you have custom loss functions or metrics, implement them with CUDA so they run in parallel with the rest of the model. The goal is to *keep the GPU busy*. A profile might show that after each batch's forward/backward pass, the GPU sits idle while the next batch is prepared – overlapping and moving that work to GPU (if possible) will improve overall throughput.

- **Use multiple streams for independent work**. In scenarios such as inference serving, you might have multiple independent inputs that can be processed in parallel. Using CUDA streams, you can launch multiple kernels that run concurrently on the GPU (the GPU will time-slice or spatially share SMs among them if resources allow). This is advanced usage, but for example, two inference requests could be executed in parallel on different streams to reduce latency (provided the GPU has enough resources). For training, most frameworks use a single stream for the main work by default, but sometimes overlap of compute and communication (e.g., in distributed training, one stream for all-reduce of gradients while another continues computation) is used.

- **Multi-GPU parallelism**: While a single GPU can handle a lot, many training tasks use multiple GPUs to further scale. On Windows, multi-GPU training (data parallelism) is supported, though

frameworks historically had better support on Linux. PyTorch, for instance, can use NCCL or other backends on Windows for multi-GPU, but older versions fell back to slower backends. Ensure you use *Distributed Data Parallel* (DDP) or an equivalent approach to scale out training across GPUs efficiently. If doing so, also ensure the inter-GPU links (like NVLink or PCIe) are utilized – e.g., enable peer-to-peer access if available (`torch.cuda.nccl.all_reduce` will do this internally). Keep in mind that Windows does not support NVIDIA's NVSwitch or complex GPU topologies as commonly as Linux on servers, so large multi-GPU clusters are usually on Linux, but for 2–4 GPUs in a workstation, Windows can still be used effectively.

- **GPU Scheduling and concurrency**: Modern NVIDIA GPUs can execute multiple kernels at once (as long as resources (SMs, registers, etc.) are available). This means if you launch a kernel that uses, say, 50% of the GPU's resources, another kernel can potentially run concurrently to use the rest. However, if one kernel uses almost all SMs, others will wait. If you have many tiny kernels, they may not fully utilize the GPU individually; in such cases, launching some of them in parallel streams could improve overall utilization. The Nsight Systems timeline can reveal if your GPU has bubbles of idle time between kernels – if those come from synchronous CPU behavior, fix that (e.g., remove `cudaDeviceSynchronize()` calls in inner loops, they force CPU-GPU sync). The rule of thumb is to launch as much parallel work as the problem permits and let the GPU schedule it. Thousands of threads *per SM* are often needed to hide latency (switching between warps when one is waiting on memory). Therefore, *ensure your kernel launch dimensions are large enough*: e.g., launching only 100 threads total on a GPU that can run 50,000 concurrently will underutilize it. Launch at least as many threads as there are data points or more, and use grid-stride loops in kernels if necessary to allow a single kernel to handle very large N by having each thread loop over multiple elements.

- **Occupancy vs. ILP**: GPU tuning often involves occupancy (having enough active warps to hide latency). Sometimes using slightly fewer threads per block but more blocks can yield better overall usage if it allows more warps to be scheduled. NVIDIA provides an *Occupancy Calculator* (in Visual Profiler or Nsight Compute) that given your register and shared memory usage, tells how many warps/blocks can reside per SM. High occupancy is good but not the sole goal – sometimes a kernel with lower occupancy but heavy instruction-level parallelism (ILP) can still fully utilize the GPU. Still, as a guideline, try not to use excessive registers per thread or large shared memory per block unless needed, as that can cap occupancy. Balance these based on profiler feedback.

Kernel Launch Configuration and Tuning

Choosing the right kernel launch parameters (grid and block dimensions) and optimizing kernel code can have a big impact:

- **Grid and block sizing**: Generally, use a number of threads per block that is a multiple of the warp size (32). Common choices are 128, 256, or 512 threads per block. 256 is a safe starting point for many kernels. The grid size (number of blocks) should be high enough so that all SMs get work. Typically, you'd use something like `blocks = ceil(N / threads_per_block)` for one-dimensional kernels. If N (problem size) is very large, you might also use 2D grids, but 1D is often fine. If your kernel is not meeting expected performance, experiment with block size – too small a block (e.g. 32 threads) might not utilize some hardware features well, but too large (1024) might cause fewer blocks to run concurrently. Tools: Nsight Compute can show achieved occupancy [30], and the CUDA Occupancy API can suggest an optimal block size.

- **Shared memory usage**: Using shared memory can greatly speed up algorithms like matrix multiplication (by tiling inputs into shared memory to reuse). However, shared memory is a limited resource; using it per block means fewer blocks might fit on an SM at once. Optimize shared memory usage by only storing what is needed and consider using *bank conflict*-free data layouts (split data such that threads access different memory banks in shared memory to avoid serialization). For instance, for a convolution kernel, using shared memory to hold the tile of an image patch can reduce global memory reads, but ensure threads read from shared in a conflict-free manner (pad the shared mem if needed to avoid multiple threads hitting the same bank).

- **Instruction bottlenecks**: If a kernel is compute-heavy (lots of math per data element), ensure you are taking advantage of fast math operations. For example, use the **CUDA** *fast math* intrinsics if reduced precision is acceptable (`__fmul_rn` etc., or in high-level code use `-use_fast_math` compile flag) – this can allow the compiler to use faster approximations for transcendentals etc. Also, try to unroll small loops (the compiler often does this automatically or you can use `#pragma unroll`) to expose ILP. Conversely, avoid *unnecessary* work in a kernel: e.g., don't recompute the same value in a loop if it can be hoisted out.

- **Branching and divergence**: Try to structure your code so that threads within the same warp follow the same execution path as much as possible. Divergent branches (e.g., an `if` where some threads take it and some don't) cause the warp to serially execute each path, hurting performance. Sometimes this is unavoidable (like in ReLU activation, different threads might be active or not), but you can often design algorithms to minimize divergence. For instance, use predicated assignments instead of branches for simple conditions, or segregate data so that warps handle homogeneous cases if possible. The compiler will often convert minor `if` conditions into predicated instructions, which is fine. But heavy divergence (like one thread doing a lot of work while others idle in a warp) is to be avoided. If you have a scenario with very divergent work per element, consider reorganizing the kernel or using separate kernels for different cases.

- **Loop efficiency and memory access in kernel**: If a kernel uses loops internally, make sure each thread does a significant amount of work to amortize the kernel launch overhead. It's often better to launch fewer kernels that each do more work (provided they still have enough parallel threads) than launching a kernel for every tiny operation. For example, it's more efficient to have one kernel that computes both a forward and backward result in one pass (if possible) than two separate kernels with a global memory round-trip in between – this concept is called *kernel fusion* and is essentially what libraries like cuDNN do internally. You as a developer can sometimes manually fuse operations in a custom kernel.

- **Use of Tensor Cores**: Modern NVIDIA GPUs (Volta, Turing, Ampere and beyond) have specialized Tensor Core units for mixed-precision matrix math (FP16/FP32 or INT8/FP16 accumulations, etc.). To use them in your kernels, you can use WMMA (Warp Matrix Multiply and Accumulate) APIs or rely on libraries (cuBLAS, cuDNN, etc.) which automatically use them. If you're writing custom GEMM or convolution, it might be worth studying how to leverage tensor cores via WMMA instructions for significant speedups (8× throughput for matrix math operations in FP16 versus FP32). Keep in mind Tensor Cores usually require using FP16/BF16 data or specific sizes (multiples of 8/16, etc.). This is a more advanced optimization but very powerful.

In summary, **tuning GPU kernels** is an iterative process: configure launch dimensions, test performance, and adjust. Use profiling tools to see if your kernel is bound by memory, compute, or latency, and address the bottleneck accordingly (e.g., increase parallelism or use faster memory if

memory-bound, or reduce instruction count if compute-bound). Over time, you'll develop an intuition for how many threads and what patterns work best for a given problem (for example, reduction operations benefit from a hierarchical approach using shared memory and warp shuffle instructions, which is a known optimized pattern). Nvidia's Best Practices Guide and sample code are excellent references for such patterns.

Mixed Precision Training (FP16/BF16) and Tensor Cores

One of the most important techniques in modern deep learning is **mixed precision**, which means using lower-precision data types (FP16 or BF16) for most computations while keeping certain parts in FP32 for stability. Mixed precision can dramatically accelerate training by utilizing Tensor Cores on GPUs and reducing memory traffic. NVIDIA reports that switching to mixed precision (FP16) training can yield **up to 3× overall speedup** on compatible models, thanks to faster math and reduced bandwidth usage [38].

The idea is that half-precision (16-bit) floats take half the memory of 32-bit floats, so **memory usage** is halved (allowing larger batch sizes or models) and **memory bandwidth** requirements are halved (since you move half the bytes) [39] [40]. Furthermore, Tensor Cores can perform 16-bit matrix multiply-adds at extremely high throughput – on the order of 8–16× the throughput of 32-bit units, depending on the GPU generation. For example, on Volta and Turing, Tensor Cores provided 8× the FP16 throughput, and on Ampere, TF32 (a specialized 19-bit format for FP32 math) also boosted certain ops. By using FP16 for most layers, you exploit this hardware. The challenge is that FP16 has a narrower range and precision, which can cause underflow/overflow or numerical issues for gradients. Mixed precision training addresses this by keeping a master copy of weights in FP32 and using **loss scaling** to prevent gradient underflow [41] [42]. Loss scaling multiplies the loss (and hence gradients) by a scale factor during backprop and divides out later, effectively increasing the precision headroom.

From a user standpoint, frameworks have made mixed precision easy. In PyTorch, you can wrap your training loop with `torch.cuda.amp.autocast()` and scale gradients with `GradScaler` – or simply use `model.half()` and be mindful to convert back to float for certain operations (PyTorch's `amp` handles this automatically). In TensorFlow, you can enable the `mixed_float16` policy (Keras mixed precision API) and it will do the rest, or use `tf.function(jit_compile=True)` combined with mixed precision. Empirically, one can often see 2×–3× speedups on convolutional models or transformers by using FP16, with negligible accuracy loss if done correctly [38]. NVIDIA's automatic mixed precision (AMP) approach has proven its worth; for instance, an automatic mixed precision run of a speech model saw a 50% training speedup with no accuracy drop [43].

Another variant is **BFloat16 (BF16)**, supported on NVIDIA Ampere GPUs and beyond. BF16 has a wider exponent range than FP16 (same as FP32's range, but with 16-bit precision), which makes it more robust numerically (it often doesn't require loss scaling). Ampere GPUs can do BF16 compute at near FP16 speeds as well. Many frameworks (TF, PyTorch) support BF16 training on Ampere+ when available. BF16 is particularly popular for very large models where the range of values is huge (e.g., large language models) because it simplifies mixed precision (no need for dynamic loss scaling). You might choose BF16 as the compute dtype if your framework and GPU support it, for stability and ease of use, while still getting performance gains.

For **inference**, you can push precision even lower: INT8 inference is widely used to get 2–4× speedups. Quantization techniques allow converting weights (and even activations) to 8-bit integers. TensorRT and PyTorch's quantization toolkit can take a trained model and calibrate or quantize it. If done well, INT8 inference can be nearly as accurate as FP32 but run much faster, especially on devices with INT8 tensor core support (Turing, Ampere). This is beyond mixed *floating-point* precision, but worth noting as an

optimization: e.g., using TensorRT INT8, one can often double inference throughput again over FP16 [36] [44].

In summary, **use mixed precision for training**: it's one of the highest ROI optimizations. Modern frameworks have automated it, so it often requires just a few lines of code. Always test that the accuracy remains close (usually it does, maybe requiring some hyperparameter tuning or simply using provided utilities). Mixed precision gives you not only speed but also the ability to train bigger models or bigger batches within the same memory footprint. Combine that with Tensor Cores and you're harnessing the full power of the GPU. NVIDIA's own guidance is that mixed precision training with Tensor Cores should be a default for modern networks – the speedups are significant and well-documented [39] [40].

Profiling and Performance Analysis

"**Premature optimization is the root of all evil**," the saying goes – meaning one should first identify *what* needs optimizing. This is where profiling comes in. On GPUs, performance can be non-intuitive, so it's vital to measure and locate bottlenecks:

- **Use Profiler tools**: We discussed Nsight Systems and Nsight Compute earlier. In practice, a good approach is to start with a timeline/profile of the overall training or inference run. For example, use Nsight Systems (or even the built-in profiler in PyTorch or TensorFlow – e.g., PyTorch's `torch.profiler.profile` or TensorFlow's TensorBoard profiler) to record a few iterations of your training loop. This will show you how much time is spent in various phases: data loading on CPU, forward pass kernels on GPU, backward pass kernels, memory copies, etc. Identify if the GPU is waiting at any stage. If you see gaps where GPU is idle, focus on that – e.g., if there's a long gap after each batch while the next batch loads, then your bottleneck is data input (solution: more DataLoader workers, use faster storage, or move preprocessing to GPU). If the GPU is constantly busy but overall training is slow, then the limiter might be GPU kernel efficiency.

- **Examine GPU kernel efficiency**: Using Nsight Compute (or PyTorch's profiler with CUDA metrics) on critical kernels (like the matrix multiply or convolution kernel taking the most time) can tell if they are hitting expected performance. Fortunately, if you're using cuDNN and cuBLAS, those kernels are usually near optimal. But if you wrote a custom CUDA kernel (say, a custom activation or a novel layer), profile it. You might discover it only achieves, for example, 30% memory bandwidth and has low occupancy – then you know to optimize memory access or increase parallelism. Or you might find it achieves good occupancy but is still slow, possibly due to memory divergence or bank conflicts. The profiler's guided analysis will point out issues (for instance, "Memory throughput not saturated; likely memory-bound" or "Large amount of divergent branching detected").

- **Iterative tuning**: After identifying a bottleneck kernel or step, try changes and measure again. For instance, if your training is augmentation-heavy on CPU, try turning off augmentations to see the speed difference – if it becomes much faster, consider GPU augmentation or more CPU workers. If your custom kernel is slow, try a simpler baseline (maybe copy data without processing) to see the overhead, then build up complexity. This method of isolating components helps determine where the slowness comes from.

- **Utilization monitoring**: Keep an eye on GPU utilization and memory usage in a coarse way too. Tools like `nvidia-smi` (on Windows, you can run it in PowerShell or use NVIDIA System Management Interface tool) show GPU % utilization and memory usage. During a training run, if you see the GPU utilization frequently drop to 0% or very low in between spikes, that indicates a

pipeline stall – again pointing to likely input or coordination bottlenecks. Ideally, the GPU utilization should be high (close to 100%) most of the time during a training epoch. Memory usage should also be near what you expect (if it's very low, you might not be using the GPU fully, e.g., batch size too small, or some layers not on GPU by mistake).

- **Framework-specific profilers**: Both PyTorch and TensorFlow have their own profilers that integrate with their execution model. TensorFlow's TensorBoard profiler can give nice breakdowns of time per operation, and highlight the slowest ops or steps. PyTorch's profiler can output traces that you can view in e.g. TensorBoard or Chrome tracing tools, showing each operation's duration on CPU and GPU. Use these to find "hotspots" – e.g., if you see a particular custom op taking 50% of the step time, that's a candidate to optimize or replace with an optimized library version.

- **Benchmark and optimize inference separately**: If you are concerned with inference performance, benchmark the model in inference mode (no grad, model.eval()) separately from training. Different issues might surface – e.g., maybe training was fine but inference is memory-bound due to different batch sizes or sequence lengths. Use TensorRT or TorchScript in those cases and measure the latency. Often for inference, you want to measure latency distribution (p99 latency etc. if it's a service). Tools like TRT's trtexec or custom scripts can help. Optimize things like batching (throughput vs latency trade-off) and CPU-GPU overlap for inference pipelines as well (e.g., decoding output on CPU while next inference runs on GPU).

- **Platform-specific quirks**: Since this is Windows-focused, also be mindful of background processes or GPU driver behavior that might affect profiling. For instance, Windows might be running other graphics tasks if it's the display GPU, which can introduce jitter. When profiling, close unnecessary programs and maybe set Windows to *Graphics Performance* "High Performance" mode for that app (there is a Graphics Settings in Windows to bias scheduling). Also, the first kernel launch in a CUDA program does extra initialization (context establishment, etc.) – so it's normal that the first iteration is slower; focus on steady-state iterations for profiling.

To summarize, **profiling is an iterative guide**: measure -> identify bottleneck -> apply optimization -> measure again. This loop should continue until either the performance meets requirements or you hit diminishing returns. Modern GPUs and frameworks are complex, so data-driven optimization is key. Often, a small tweak (like enabling mixed precision or increasing a buffer size) identified via profiling can yield huge gains. Conversely, trying random optimizations without profiling might waste time on parts of the code that were not bottlenecks. Thus, use the tools at your disposal to shine light on what the GPU and CPU are doing during your workload.

Framework-Specific GPU Optimization (PyTorch & TensorFlow)

High-level frameworks like **PyTorch** and **TensorFlow** handle many GPU optimization details for you, but they also offer additional features to maximize performance on NVIDIA GPUs. We'll look at some framework-specific options:

PyTorch Optimizations (JIT, Graph Mode, TensorRT integration)

PyTorch is known for its eager execution, but to boost performance, PyTorch provides mechanisms to leverage static graph optimizations. The primary tool is **TorchScript** and the newer `torch.compile` in PyTorch 2.x.

- **TorchScript (JIT)**: TorchScript allows you to convert PyTorch models or functions into a serializable, **optimized graph representation** that can run independent of Python [45]. By scripting or tracing your model, you eliminate Python's runtime overhead and enable compiler optimizations like operator fusion. For example, TorchScript might fuse consecutive elementwise ops into one GPU kernel. It also makes deployment easier (you can load the model in C++ without the Python interpreter) [46]. In practice, you would use `torch.jit.trace` or `torch.jit.script` on your model. The performance gain from TorchScript alone varies – for models heavy on Python control flow or small ops, it can be significant; for large matrix-heavy models it may be modest since the heavy ops are already in optimized C++ (cuDNN). Still, it often helps, and it's a prerequisite for certain integrations (like Torch-TensorRT expects a TorchScript module as input, which it then compiles further [37]).

- `torch.compile` **(PyTorch 2.x)**: Introduced in PyTorch 2.0, `torch.compile` is a higher-level interface that **automatically applies graph capture and optimization** to your model. It uses components like TorchDynamo and TorchInductor under the hood to dynamically compile your model to efficient code. Notably, TorchInductor can generate fused CUDA kernels, using the OpenAI Triton compiler for custom code generation [47]. The result is that you can often get significant speedups with minimal code changes: you wrap your model or training step as `compiled_model = torch.compile(model)`, then use it normally. On supported GPUs (Volta and newer) and with most ops supported, users have seen substantial improvements. NVIDIA reported that across a suite of 163 models, `torch.compile` provided an average **43% speedup in training throughput** on an A100 GPU (51% faster when using AMP mixed precision) [48]. This is a huge gain achieved by more aggressive graph-level optimizations and kernel fusion that weren't present in eager mode. Essentially, `torch.compile` can take sequences of ops that the PyTorch execution would normally run as separate kernels and fuse them, similar in spirit to XLA or TensorRT but within PyTorch's ecosystem. This is cutting-edge (PyTorch 2.0+), so some models might run into unsupported ops or lower gains, but it's improving rapidly.

- **Distributed Data Parallel (DDP)**: For multi-GPU training, PyTorch's `DistributedDataParallel` is the go-to. On Windows, as of PyTorch 1.10+, DDP is supported but uses the Gloo backend by default (which is CPU-based communication). NCCL backend (which is faster and GPU-native) had limited Windows support historically, but experimental NCCL support on Windows has been appearing. If you have multiple GPUs on one machine, ensure you initialize DDP correctly and that your batch is split across GPUs. DDP overlaps gradient all-reduce with backward computation by default, which is an optimization. There's also an `bucket_cap_mb` setting to control how much to all-reduce at once; tuning that might improve throughput. Although Windows is not the typical OS for multi-node GPU training, for single-node multi-GPU it works and you can achieve near-linear speedups in many cases.

- **Memory format and channels-last**: In PyTorch, one specific optimization for CNNs is using `channels_last` memory format for tensors (especially on Volta/Ampere GPUs). This can improve memory coalescing in convolution operations. E.g., converting your model and inputs to `memory_format=torch.channels_last` can yield 5-10% speedups for conv-heavy models

when using FP16, as cuDNN is optimized for NHWC (channels-last) in some cases. It's worth trying if you have a lot of Conv2d.

- **Torch-TensorRT integration**: As discussed, if you need even more inference speed, you can integrate TensorRT with PyTorch. Using Torch-TensorRT, you can compile submodules of your model to TensorRT engines, yielding big wins. For example, a ResNet or BERT model compiled with Torch-TensorRT might achieve 2-3× the throughput of the regular PyTorch model (especially if using INT8 or FP16). The process is straightforward: install the `torch-tensorrt` package, then do something like `trt_model = torch_tensorrt.compile(model, inputs=[torch_tensorrt.Input(...)], enabled_precisions={torch.float16})`. The result is that `trt_model` runs the model using TensorRT under the hood, but you can call it like a normal PyTorch module. NVIDIA's blog demonstrated up to **6× faster** inference on some models using this integration [33]. Keep in mind this is mainly for deployment (it's not something you'd use during training).

- **Other PyTorch tips**: Use the latest CUDA and PyTorch versions if possible; each release often brings performance improvements and new kernels. For example, PyTorch added support for convolution via `cudnn64` heuristics, improved its pooling and normalization implementations, etc. Also, monitor GPU memory – PyTorch by default retains the computational graph for backward until you `loss.backward()`, so if you do multiple forwards before backward, you could be using extra memory. Use `with torch.no_grad()` for inference or parts where grad isn't needed to save memory. Freeing unused memory (`torch.cuda.empty_cache()`) doesn't improve performance per se, but it can help fragmentation in long runs, though the PyTorch allocator usually handles this.

In essence, **PyTorch's philosophy** is to give you dynamic flexibility with the ability to opt into graph optimizations when needed. Leverage TorchScript/compile to get graph-level speedups, use mixed precision and the latest libraries for kernel-level speedups, and keep your data on GPU as much as possible. The combination of these can bring PyTorch training speed on par with or even exceeding some static graph frameworks, while retaining the ease of use.

TensorFlow Optimizations (XLA, Graph Mode, TensorRT integration)

TensorFlow (TF2, in particular, since TF1 always had static graphs) has its own strategies for optimization:

- **XLA (Accelerated Linear Algebra)**: XLA is a JIT compiler for TensorFlow graphs that can optimize and fuse ops. By enabling XLA, TensorFlow will JIT-compile subgraphs of your computation into efficient machine code (much like what we described for PyTorch compile). This can lead to significant speedups. Google's team reported that using XLA on a ResNet-50 model improved training throughput from ~6.8k images/sec to ~10.5k images/sec on 8 GPUs (a ~1.5× speedup) [49]. Gains between 1.1× and 3× have been observed on various internal models with XLA [49]. XLA works by fusing chains of operations into single kernels and optimizing memory access. For example, without XLA a TensorFlow graph might do a multiplication, then an addition, then a reduction in three separate GPU kernels. XLA can **fuse** these into one kernel – it "knows" the whole computation and can generate one GPU kernel that performs multiply-add-reduce all in one go [50] [51]. This not only reduces kernel launch overhead but also saves memory bandwidth (intermediate results are kept in registers on-chip instead of written to global memory) [50] [51]. Fusion is XLA's most important optimization [52], as noted. To use XLA in TF2, you can either decorate functions with `@tf.function(jit_compile=True)` or set the environment variable

`TF_XLA_FLAGS=--tf_xla_auto_jit=2` for global graph compilation. Keras also has a `experimental_compile=True` flag for model `fit`. Be aware that XLA sometimes has compatibility issues or compilation overhead, so it's best used for long-running jobs where the upfront compile time is amortized. But it generally boosts GPU utilization by reducing those tiny GPU kernel launches that often plague TensorFlow graphs.

- **tf.function and Graph mode**: Even without XLA, using `@tf.function` in TensorFlow 2 to create a static graph helps a lot. Eager execution in TF (like PyTorch eager) is great for debugging but slower for training because each op is executed one by one. Wrapping your training step in `tf.function` will let TensorFlow execute it as a single graph (with possibly many ops fused by its Grappler optimizer). Grappler (the default graph optimizer) will perform optimizations like constant folding, common subexpression elimination, and some operator fusion on its own. Always ensure your performance-critical code is inside a `tf.function` – otherwise you're likely underutilizing the GPU.

- **Data Input pipeline (tf.data)**: Using the `tf.data` API to create efficient input pipelines can hugely affect performance. For example, use `.prefetch()` to overlap data preprocessing on CPU with GPU training. Use `.map(num_parallel_calls=tf.data.AUTOTUNE)` to use multiple threads for parsing/augmenting data. And if you have a beefy CPU, consider `.map(..., deterministic=False)` for even faster throughput with mixed order (if order doesn't matter). In short, the data input pipeline in TF should be optimized so that the GPU never waits for data. If using TensorFlow's `Model.fit`, monitor the throughput and ensure the `tf.data` pipeline is not the bottleneck.

- **cuDNN and oneDNN**: TensorFlow will automatically utilize cuDNN for operations like Conv2D, LSTM, etc., when available (assuming you installed the GPU version of TF). There's not much you need to do to enable this – just ensure you've installed `tensorflow-gpu` (for TF1.x) or for TF2.x, the standard `tensorflow>=2.x` pip package will automatically use GPU if found. One optimization to be aware of: in TensorFlow 2.4+, the integration with oneDNN (for CPU) sometimes caused GPU ops to be replaced with CPU ops if it thought it was faster. If you ever observe TensorFlow not using GPU for something, you can force placement or check if some weird autotuning happened. But generally, TF will use cuDNN for CNNs and RNNs. For RNNs, TF's Keras LSTM layer will use the cuDNN implementation if you set `LSTM(..., use_cudnn_kernel=True)` or if certain conditions are met (like activation is tanh/relu, etc., which is by default). In TF2, the `tf.keras.layers.LSTM` automatically tries to use cuDNN when running on GPU unless you disable it. This yields huge speedups (the 6× LSTM speedup from cuDNN applies here as well).

- **TensorRT integration (TF-TRT)**: For inference, TF-TRT can be a game-changer. If you have a SavedModel or Keras model, you can convert it via TF-TRT to an optimized graph where parts are executed by TensorRT. This is done through `tf.experimental.tensorrt.Converter` or via `SavedModel` conversion utilities. As mentioned, it will replace subgraphs with a single op (`TRTEngineOp`) that calls TensorRT [53]. The result retains the same TensorFlow interface but internally uses TensorRT for those parts. Users have reported significant latency reductions and throughput gains for models like object detectors and classifiers using TF-TRT – often 2× or more, especially when using INT8 precision. The advantage is you can still use TensorFlow Serving or the TF runtime to execute the model; it's just faster. If full TensorRT is more complicated to deploy, TF-TRT is a nice middle ground.

- **Distributed training (MirroredStrategy etc.)**: On Windows, TensorFlow's MultiWorkerMirroredStrategy or even single-machine multi-GPU MirroredStrategy should work (though multi-worker is less common on Windows). Ensure to use these high-level strategies to simplify multi-GPU training. They will handle splitting batches and all-reducing gradients via NCCL. As of TF2, multi-GPU is fairly easy: `strategy = tf.distribute.MirroredStrategy()` then do your training inside `strategy.scope()`. Under the hood, it will use NCCL if available to synchronize updates. This will give near-linear scaling for many models.

- **Memory growth option**: By default, TensorFlow grabs all GPU memory at once. If you run multiple processes or want to be nice in a shared environment, you can enable `tf.config.experimental.set_memory_growth(True)` to make it allocate on demand. This doesn't directly affect speed, but can prevent OS paging if multiple processes contend. Usually for dedicated training, leaving it to allocate all is fine (perhaps even beneficial to avoid fragmentation).

In conclusion, **TensorFlow** performance is maximized by using static graphs with XLA where possible, ensuring you use the GPU-optimized ops (which TF does by default), and optimizing the input pipeline and multi-GPU setups. XLA can be seen as the TensorFlow equivalent to PyTorch's `torch.compile` – it's not on by default (as of TF2.x, it's optional), but it can give big wins by fusing ops and reducing overhead. The principle we saw in the XLA example is exactly that: three kernels become one, eliminating intermediate memory writes [50] [51] – this can drastically reduce the strain on memory bandwidth, often the bottleneck in deep learning training. So it's worth experimenting with XLA on your training jobs. For inference, take advantage of TF-TRT or export to ONNX and use TensorRT directly if needed. Each bit of optimization (graph fusions, memory overlap, quantization) adds up.

Windows-Specific Considerations and Bottlenecks

While the core optimization techniques are similar across operating systems, there are a few **Windows-specific quirks** to keep in mind:

- **GPU Driver Mode (WDDM vs TCC)**: As discussed earlier, consumer GPUs on Windows run with the WDDM driver which can introduce overhead and the TDR timeout. If you are using a GeForce card on Windows, you may notice somewhat lower GPU throughput compared to the same card on Linux in some scenarios [54]. This is partly due to WDDM scheduling and the fact that Windows might batch GPU commands or insert waits to keep the display responsive [55]. If you have a workstation GPU (like Quadro RTX, or a Tesla) that supports TCC mode, switching it to TCC can improve performance and remove the 2-second timeout [20] [56]. In fact, running a GPU in TCC on Windows typically yields performance on par with Linux [54]. But since most deep learning users on Windows have GeForce cards (which can't use TCC), be mindful of the watchdog: avoid extremely long kernel executions. If you absolutely must run a very long kernel (e.g., a huge custom operation that takes >2s), you could increase the TDR delay in the registry or use CUDA's multi-device contexts to dedicate one GPU purely to compute (so Windows might not reset it), but these are advanced steps. A simpler approach is to break the work into smaller kernels or insert periodic __syncthreads and yields (though those don't truly avoid TDR if the GPU is busy). The recommendation: keep kernels reasonably sized and let your program return control to the OS now and then.

- **Performance differences**: Historically, there have been reports and bug filings that TensorFlow and PyTorch can be **slower on Windows than Linux** (on the same hardware) by a noticeable factor [57] [58]. For example, one user found 2–5× slower YOLOv3 inference on Windows vs Linux

with a 2080Ti [57]. The primary culprit was thought to be the WDDM overhead and possibly differences in how system memory is allocated or how threading works on Windows. Microsoft introduced a feature called *Hardware-Accelerated GPU Scheduling* (WDDM 2.7 in Windows 10) which can reduce latency by cutting some overhead – make sure to enable this in Graphics Settings if available, as it could help (though the benefit is small, a few percent perhaps). The consensus from experts is: "For serious compute work, a WDDM GPU on Windows is not ideal" [59], meaning if maximum performance is crucial, Linux or a TCC-mode GPU is preferred. That said, Windows can absolutely be used for development and moderate workloads, and the gap has narrowed with improvements in drivers. Some optimizations like GPU Hardware scheduling and Windows Subsystem for Linux 2 (WSL2) with GPU support provide more options. WSL2, for instance, allows you to run Linux environment on Windows with GPU passthrough – some users find training in a WSL2 Ubuntu environment yields performance closer to native Linux, while still using Windows as the host OS. This might be an option if one wants the best of both (though it adds complexity).

- **Multi-GPU and networking**: If you're doing multi-node training (rare on Windows), the lack of NCCL support historically could be an issue. But for single node with multiple GPUs, PyTorch and TensorFlow both now support NCCL on Windows (PyTorch added experimental NCCL in 1.8+, TensorFlow's NCCL should work as long as the toolkit is present). Ensure you have the Microsoft MPI or OpenMPI if you try multi-node. But frankly, large-scale training is typically on Linux clusters.

- **CPU differences**: This is not GPU-specific, but note that the default math libraries on Windows (like MKL for PyTorch) might have minor differences. Usually not an issue unless you compare training reproducibility across OS. But one interesting note: PyTorch on Windows by default uses the *THC* allocator and a different threading backend (native threads vs pthreads on Linux). These could cause slight performance differences in DataLoader or CPU ops. If your pipeline involves a lot of CPU preprocessing, consider that on Linux one might use forked workers (Python multiproc) whereas on Windows all DataLoader workers spawn fresh processes (no fork). Sometimes Windows can't fork efficiently, leading to slower data loading if lots of workers spawn heavy processes each epoch. A solution can be to reduce worker count but do more work per worker, or move the workload onto the GPU (as mentioned). Also, make sure you enable asynchronous disk prefetch (the `prefetch` in tf.data or PyTorch DataLoader's prefetch factor) to overlap disk I/O with compute.

- **Monitoring and system**: Use Windows Performance Monitor or tools like GPU-Z to monitor if something unexpected is eating GPU cycles. Sometimes, Windows might have background tasks (like an antivirus doing GPU compute maybe for some reason, or the Windows Display itself). It's rare but keep the system lean while training. If the display is driven by the same GPU doing compute, avoid moving windows or doing graphics-intensive tasks simultaneously, as that can steal GPU time for rendering. In a multi-GPU system, consider using one GPU as dedicated to driving the display and the others for compute (you can select which GPU is primary in BIOS or Windows settings). Or, if you have just one GPU, you might see some stutter in UI while training – that's normal, but also indicates your training might be interrupted momentarily for display refreshes.

To conclude, Windows is fully capable for machine learning on NVIDIA GPUs, but you must be aware of the WDDM overhead. The difference might be, say, 5-10% performance loss versus Linux for identical code (sometimes more in edge cases) [54]. If you need every ounce of performance and stability for multi-GPU scaling, Linux is typically the go-to in industry. However, for development convenience, using Windows with careful optimizations (and maybe WSL2 for a Linux-like environment) is completely fine

and nowadays quite performant. The key Windows-specific advice is: try to bypass or mitigate the GPU scheduler limitations (use TCC if possible, enable GPU hardware scheduling, keep kernels short), and ensure your workflow doesn't trigger the TDR. Many researchers train on Windows machines successfully by following these practices.

Model-Specific Considerations: CNNs, RNNs, and Transformers

Different types of AI models have different computational characteristics. Optimizing a **Convolutional Neural Network (CNN)** versus a **Recurrent Neural Network (RNN)** versus a **Transformer** can entail some unique considerations. Here's a breakdown of these model types and optimization tips for each:

Convolutional Neural Networks (CNNs)

CNNs, commonly used in image processing and vision (e.g., ResNet, VGG, EfficientNet), heavily rely on convolution and matrix multiplication operations. These are **highly parallel and compute-intensive**, which makes them very well-suited to GPUs. Key points for CNN performance:

- **Convolution algorithms**: Modern CNN layers use batch convolution, which under the hood is often implemented via GEMM (matrix multiplications) or FFT-based algorithms. Libraries like cuDNN handle this: they will pick the fastest convolution algorithm for your GPU (such as using Winograd minimal filtering or FFT for large kernels, etc.). As a user, ensure you enable cuDNN tuning (in PyTorch, `torch.backends.cudnn.benchmark = True` will make cuDNN profile your first few batches to choose the best algorithm for the layer dimensions). This can significantly improve throughput for fixed input sizes, as cuDNN will e.g. decide to use Winograd convolution which can be 2–3× faster than naive conv for medium filter sizes [60] [29].

- **Data layout**: As mentioned, using NHWC (`channels_last`) can optimize memory access for conv layers on some backends. This layout means each pixel's channels are contiguous in memory, which can improve coalescing when threads load channel data. If your framework supports it (TensorFlow uses NHWC by default on GPU for convs, PyTorch can with channels_last), it's worth using – especially when combined with Tensor Cores (which prefer FP16 NHWC data for convolutions in some cases).

- **Batch size and throughput**: CNN training often achieves higher efficiency with larger batch sizes. This is because large batches keep the GPU's math units busy with big matrix multiplies. If you have the memory, increase batch size until you see diminishing returns or until the GPU is nearly 100% utilized. For inference, batching multiple images can also improve throughput (process e.g. 32 images in one go rather than 32 separate calls). However, note that batching increases latency for a single item, so in real-time systems there's a trade-off. Tools like TensorRT allow you to create an engine that is optimized for a range of batch sizes or even dynamic shapes.

- **Precision**: CNNs are generally very amenable to reduced precision. They tend to be resilient to FP16 and even INT8 quantization (especially classification models). Using FP16 training is almost a no-brainer for CNNs on modern GPUs – you get Tensor Core acceleration on convolutions, which are the bulk of the work. For instance, an Ampere A100 can deliver over **600 TFLOPS** of FP16 Tensor Core performance [61] (as a reference, FP32 is about 19.5 TFLOPS on an A100). That enormous compute is accessible if you use mixed precision. Many CNNs have been successfully quantized to INT8 for inference with minimal accuracy loss (through calibration). So, CNNs benefit a lot from the whole spectrum of precision optimization.

- **Memory bandwidth vs compute**: Conv layers in CNNs, especially large ones, can be compute-bound (lots of multiplications relative to memory accessed). This is good because it means they scale well with more ALUs/Tensor Cores. However, some layers (like depthwise conv or small 1x1 conv layers in e.g. MobileNet) are more memory-bound. Profiling can show if your model is compute-limited or memory-limited. If memory-limited, try to fuse layers (e.g., fuse Conv+Bias+ReLU in one kernel, which frameworks often do) to reduce memory traffic. If compute-limited (which is often the case on smaller GPUs or high-res images), then using Tensor Cores/mixed precision is the primary way to speed up.

- **Parallelism**: CNNs parallelize extremely well across both the spatial domain and batch. So utilize multiple GPUs if needed via data parallelism (each GPU handles a portion of the batch). CNNs also benefit from pipeline parallelism if you split the model (e.g., model parallel across GPUs layerwise), but that's more complex and usually not necessary unless model is huge (like very deep nets or 3D-Unets etc.). In general, a well-optimized CNN can achieve high GPU utilization.

In practice, if you follow best practices (cuDNN, large batch, mixed precision), CNN training on GPUs is very efficient. We've seen, for example, ResNet-50 training scaling to near hardware limits – images/second throughput on a single GPU often is mostly constrained by memory bandwidth and ultimately the data pipeline after those optimizations. For inference, CNNs can often saturate even lower-end GPUs with moderate batch sizes, meaning you can get excellent throughput per dollar on something like a T4 or 3080 for ResNet inference (especially with TensorRT INT8).

Recurrent Neural Networks (RNNs and LSTMs)

RNNs, including LSTMs and GRUs, are used for sequence data like text, speech, or time-series. They pose different challenges because of their **sequential nature**:

- **Sequential dependency**: RNNs process sequence steps one after another (each step uses the previous hidden state). This inherently limits parallelism along the time dimension – you can't compute time step *t+1* until *t* is done (unlike CNN layers which can be done in parallel across spatial positions). This means that if you have a single long sequence, the GPU will be underutilized as it works through the sequence one step at a time (each step being a relatively small matrix multiply for the hidden state). The way to counteract this is **batching** multiple sequences together, or using short sequences (truncated backprop through time).

- **Batching and unrolling**: By feeding multiple sequences (or multiple segments of one long sequence) in parallel, you increase parallel work. Frameworks like cuDNN's RNN API effectively "unroll" the RNN for a given sequence length and then perform a fused operation that computes all timesteps in one kernel per gate. cuDNN can take a batch of sequences (padded to equal length) and process them with highly optimized kernels. This is why using Keras's LSTM layer with `return_sequences=True` and a given sequence length will be fast – it delegates to cuDNN which might use a single kernel to do all timesteps' matrix multiplies in a fused manner [62] [63]. If you were to write an RNN in pure PyTorch Python loop (one step at a time), it would be very slow, as it launches tiny kernel for each time step.

- **Use cuDNN RNNs**: Always prefer using the built-in RNN layers that exploit cuDNN (or oneDNN on CPU if needed). In PyTorch, `nn.LSTM` will use cuDNN by default if the configuration is supported (e.g., no weird activations). In TensorFlow, as noted, `tf.keras.layers.LSTM` will by default use the fast path when GPU is available. The speed difference is dramatic: one benchmark showed cuDNN LSTM was **5–6× faster** than a non-optimized LSTM for large hidden

sizes [29]. These cuDNN kernels achieve this by fusing the four gate computations and using batched GEMMs effectively. They also can run multiple layers of LSTM stacked in one go, which improves data locality and caching.

- **Parallelize where possible**: If you have very long sequences, consider **bucketing** or **truncating**. Bucketing means group sequences of similar lengths and pad them, so you don't waste compute on padding but still can batch. Truncating (TBPTT) means instead of one sequence of length 1000, process it in chunks of, say, 100 (with carrying hidden state between chunks) – this lets you have more batches and possibly overlap computation. Also, if model allows, use **bidirectional RNNs** or other architectures that might give you more parallel paths (though bidirectional still has two passes sequentially, one forward, one backward in time).

- **Alternative architectures**: It's worth noting that *transformers* have largely overtaken RNNs for many applications because they parallelize better (self-attention can be done in parallel across sequence). But if you are sticking with RNNs, you may consider using techniques like unrolling multiple time steps into one CUDA kernel (if writing custom), or even model parallel if one time step is extremely large (not usually the case; usually the bottleneck is the sequential aspect not the per-step compute).

- **Memory and precision**: RNNs (especially LSTMs) can also benefit from FP16, but they can be a bit more finicky because of small gradients or the state carrying. cuDNN's FP16 RNN works, but you may need to use loss scaling to avoid gradient underflow. BF16 on newer GPUs is a good option too. The savings in memory bandwidth using FP16 helps RNN as well, though if your RNN is small, the overhead might be minimal. Also, if the sequence lengths vary a lot each batch, there might be some wasted computation on padding – some frameworks allow *dynamic sequence length* input so the padded timesteps are skipped internally for efficiency.

In short, RNNs are **less GPU-efficient** than CNNs or Transformers because of their sequential nature. The key optimization is to use the optimized libraries (cuDNN RNN) to handle the gates efficiently and to batch as much as possible. If your sequence processing still isn't fast enough, you might consider whether a transformer model could replace the RNN, since transformers can utilize GPUs far more effectively (at the cost of memory, as attention is O(n^2) in sequence length). In some cases, people also use **TCNs (temporal conv networks)** or WaveNet-style dilated convs to avoid sequential steps. But that goes into architecture changes beyond pure optimization.

Transformers and Attention-based Models

Transformers, the foundation of models like BERT, GPT, and Vision Transformers (ViT), present yet another profile. They are extremely compute-intensive due to matrix multiplications, but also **memory-intensive** due to the self-attention mechanism:

- **Matrix Multiplications**: Transformers use large GEMMs for the QKV projections and for the feed-forward networks (which are basically two linear layers with a GELU in between). These are very friendly to GPUs – large matrix multiplies can achieve high FLOPs. So for those parts, using cuBLAS (which frameworks do) and Tensor Cores (via mixed precision) gives huge speed. E.g., BERT-large has dense layers that will fly on an Ampere GPU with FP16 – these should be near theoretical TFLOPs if everything is optimized.

- **Self-Attention**: The self-attention mechanism, which computes attention scores as Q * K^T (and then a softmax and weighting by V), is both compute and memory heavy. The Q*K^T operation is*

*an O(n^2 * d) operation (n = sequence length, d = hidden dim per head). The subsequent scaling and softmax and weighted sum are also O(n^2). For large sequence lengths, the memory usage** of storing the attention matrix (n x n) becomes a bottleneck (both in terms of capacity and bandwidth to read/write it). For example, a sequence length of 1024 has a million elements in the attention matrix for one head, and if you have 16 heads, that's 16 million floats just for one layer's attention matrix. Compute-wise, it's also a lot of multiply-adds. This is why transformers can actually be limited by memory and memory bandwidth for long sequences.

- **Optimization for attention**: Recently, there have been breakthroughs like **FlashAttention** [64], which is an algorithm that reorders attention computation to use **tiling in shared memory** and avoid explicitly materializing the full n×n matrix. FlashAttention v2/v3 can significantly reduce memory usage and also increase math utilization – one report shows it reaching 75% of H100's peak in attention, up from ~35% before [65] [66]. If you are implementing transformers, using an optimized kernel for attention (like NVIDIA's CUTLASS-based fused attention or FlashAttention from the academic community) can dramatically speed up the attention part. These kernels do the QK^T, softmax, and QK^T*V in a fused manner, keeping data in high-speed on-chip memory as much as possible [64]. For instance, FlashAttention can provide 1.5–2× speedup for the attention block and reduce memory by not storing the large intermediate [61]. In practice, libraries like PyTorch and TensorFlow are integrating such optimizations. PyTorch 2.0's scaled_dot_product_attention uses a faster path if available. If you use HuggingFace Transformers, check if they have a flag to enable FlashAttention or use xFormers library which has efficient attention.

- **Parallelism**: Transformers parallelize in multiple dimensions: across batch, across sequence (for the matrix multiplies), and across heads. They are generally very GPU-friendly. The only non-parallel part is maybe layer normalization and some elementwise ops, which are trivial overhead. So the main challenge is ensuring you have enough memory to run them. Model parallelism is common for huge models (like GPT-3 scale) – slicing the model across GPUs (tensor parallelism for different parts of matrices, or pipeline parallel for different layers). On Windows, you're less likely to run those enormous models (those are on multi-GPU servers with Linux). But if you do have, say, 2 GPUs and a very large model, PyTorch's `tensor_parallel` or Megatron-LM style parallelism could be used (though support on Windows might be hit-or-miss).

- **Memory optimization**: For training large transformers, techniques like **gradient checkpointing** (also called activation checkpointing) are important. This trades computation for memory by not storing all intermediate activations and recomputing some during backprop. Libraries like PyTorch's checkpoint or TensorFlow's recompute_grad do this. It can cut memory usage significantly (often enabling 2× longer sequences or bigger batch). The cost is extra compute (some layers computed twice). Given transformers are often memory-bound at high sequence lengths, this trade-off is usually worth it to enable either bigger models or longer sequences.

- **Mixed precision**: Transformers are very amenable to mixed precision – in fact, they *need* it to run huge models on current hardware. Almost all transformer training nowadays uses FP16 or BF16. One thing to watch is the **layer norm** and some softmax, which can sometimes be sensitive in FP16 (risk of overflow in sum). The standard solutions (like doing softmax in FP32 or maintaining an FP32 copy of certain variances) are usually employed by frameworks, so you typically don't have to worry. BF16 is great for transformers due to its range – a lot of LLM training uses BF16 because it avoids some of the tricky loss scaling logic. So definitely use mixed precision for transformers; it gives large speedups thanks to Tensor Cores.

- **INT8 and quantization**: For inference, transformers can be quantized to INT8 (and even 4-bit in bleeding edge research) to reduce memory and improve speed. TensorRT, for instance, can run BERT in INT8 and double the throughput over FP16. One must provide calibration for the attention softmax and such, but it's doable. If deploying on GPU, consider using TensorRT's optimization profiles for transformer models – they will fuse the attention, use quantization if enabled, etc. The NVIDIA Transformer Engine (open-sourced) is another library that helps with mixed and lower precision for training and inference of transformers.

In summary, **transformers are extremely powerful but resource-hungry models**. Optimizing them means dealing with the attention's quadratic cost – using advanced fused kernels like FlashAttention or reducing sequence lengths (or using sparse attention patterns if applicable to your model). It also means taking advantage of all GPU features: Tensor Cores via mixed precision, large memory (use high-end GPUs with more VRAM if possible, or utilize multi-GPU parallelism). Because transformers have become so central, NVIDIA and others have put a lot of work into optimizing them: for example, frameworks now automatically fuse dropout + bias + gelu in the transformer MLP block into one kernel. Ensuring you're on recent library versions will give you these benefits.

To illustrate the progress: a transformer-based GPT-2 model that might have taken X time in 2019 can now run several times faster on the same GPU in 2025 due to software optimizations alone (not to mention newer GPUs). So keep your software updated (e.g., use PyTorch 2.x or TF 2.10+, and latest CUDA/cuDNN) to utilize these optimizations.

In conclusion, optimizing AI models on NVIDIA GPUs (in Windows or otherwise) is about leveraging the full stack: the **GPU hardware capabilities** (parallel threads, fast memory, tensor cores), the **software libraries** (CUDA, cuDNN, TensorRT, etc.), and **algorithmic adjustments** (like mixed precision or efficient architectures). By writing efficient kernels or using the optimized ones, managing memory smartly, and utilizing framework features like XLA or TorchScript, one can often achieve substantial speedups. We've covered how to set up and compile custom CUDA code, the constraints to respect, the essential tools for development, best practices for performance tuning, and looked at how different model types exploit the GPU. With these guidelines and up-to-date sources, practitioners can navigate the specifics of Windows GPU programming and get the most out of their NVIDIA hardware for training and deploying AI models.

Sources: Connected inline throughout the guide for reference to official documentation, technical blogs, and forums that substantiate the strategies and claims made [1] [16] [33] [50] [54] [29] [48], among others. Each citation corresponds to material that further explains or provides evidence for the point in context.

---

1 3 4 5 6 7 8 9 16 17 18 23 24 26 27 CUDA Installation Guide for Microsoft Windows — Installation Guide Windows 13.0 documentation
https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html

2 10 25 CUDA Development + Setup for Visual Studio - August 2025
https://www.wholetomato.com/blog/intro-to-cuda-and-visual-studio-installation/

11 12 13 CUDA Refresher: The CUDA Programming Model | NVIDIA Technical Blog
https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

14 15 21 22 c++ - Long-running cuda kernel stops when TDR kicks in - Stack Overflow
https://stackoverflow.com/questions/20254652/long-running-cuda-kernel-stops-when-tdr-kicks-in

19 20 55 linux - CUDA performance penalty when running in Windows - Stack Overflow
https://stackoverflow.com/questions/19944429/cuda-performance-penalty-when-running-in-windows

28 NVIDIA cuDNN - NVIDIA Docs
https://docs.nvidia.com/cudnn/index.html

29 60 62 63 Optimizing Recurrent Neural Networks in cuDNN 5 | NVIDIA Technical Blog
https://developer.nvidia.com/blog/optimizing-recurrent-neural-networks-cudnn-5/

30 Performance Analysis of Your GPU CUDA Kernels with Nsight ...
https://www.nas.nasa.gov/hecc/support/kb/performance-analysis-of-your-gpu-cuda-kernels-with-nsight-compute-cli_706.html

31 Quick Start Guide — NVIDIA TensorRT Documentation
https://docs.nvidia.com/deeplearning/tensorrt/latest/getting-started/quick-start-guide.html

32 Understanding Nvidia TensorRT for deep learning model optimization
https://medium.com/@abhaychaturvedi_72055/understanding-nvidias-tensorrt-for-deep-learning-model-optimization-dad3eb6b26d9

33 36 37 44 Accelerating Inference Up to 6x Faster in PyTorch with Torch-TensorRT | NVIDIA Technical Blog
https://developer.nvidia.com/blog/accelerating-inference-up-to-6x-faster-in-pytorch-with-torch-tensorrt/

34 35 53 Accelerating Inference in TensorFlow with TensorRT User Guide - NVIDIA Docs
https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html

38 39 40 41 42 Train With Mixed Precision - NVIDIA Docs
https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html

43 Automatic Mixed Precision for Deep Learning | NVIDIA Developer
https://developer.nvidia.com/automatic-mixed-precision

45 46 TorchScript — PyTorch 2.8 documentation
https://docs.pytorch.org/docs/stable/jit.html

47 48 PyTorch 2.x
https://pytorch.org/get-started/pytorch-2-x/

49 50 51 52 Pushing the limits of GPU performance with XLA — The TensorFlow Blog
https://blog.tensorflow.org/2018/11/pushing-limits-of-gpu-performance-with-xla.html

54 56 58 59 TITAN RTX windows 10 vs ubuntu - CUDA Programming and Performance - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/titan-rtx-windows-10-vs-ubuntu/69367

57 TF and Pytorch are slower on Windows than on linux - CUDA Programming and Performance - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/tf-and-pytorch-are-slower-on-windows-than-on-linux/77064

61 Next Generation of FlashAttention | NVIDIA Technical Blog
https://developer.nvidia.com/blog/next-generation-of-flashattention/

64 The Evolution of Flash Attention: Revolutionizing Transformer ...
https://medium.com/@sailakkshmiallada/the-evolution-of-flash-attention-revolutionizing-transformer-efficiency-8a039918d507

65 66 FlashAttention-3: Fast and Accurate Attention with Asynchrony and ...
https://tridao.me/blog/2024/flash3/