Nice — I’ll give you a single, self-contained deep dive that starts at the basics and goes all the way to advanced CUDA topics (device assignment, memory allocation/transfer, streams/events, multi-GPU, pinned vs unified memory, examples and install commands). I’ll include runnable C/CUDA examples and concrete install commands for Linux (Ubuntu and RHEL/CentOS/DNF-based). I used the official CUDA docs and NVIDIA guide pages for the most important technical details. ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-c-programming-guide/?utm_source=chatgpt.com), [olcf.ornl.gov](https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf?utm_source=chatgpt.com), [events.prace-ri.eu](https://events.prace-ri.eu/event/989/sessions/3096/attachments/1196/2029/cuda-multiple-gpus.pdf?utm_source=chatgpt.com))

# **1 — Quick overview (what CUDA is)**

CUDA is NVIDIA’s parallel-computing platform + programming model that lets you run parts of your program (kernels) on the GPU (device) while the rest runs on the CPU (host). A kernel is executed by many lightweight threads organized into blocks and grids. The host launches kernels, allocates device memory, and copies data to/from the device. ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-c-programming-guide/?utm_source=chatgpt.com))

# **2 — CUDA execution model (threads, blocks, grids)**

* **Thread**: smallest execution unit; runs kernel code.
* **Block**: a group of threads that can cooperate via **shared memory** and sync with \_\_syncthreads(). Blocks are independent — they can be scheduled in any order.
* **Grid**: collection of blocks launched for one kernel call.  
   Typical kernel launch syntax (C/C++):

myKernel<<<numBlocks, threadsPerBlock, sharedMemBytes, stream>>>(args...);

* numBlocks and threadsPerBlock can be 1D/2D/3D (dim3).  
   This model provides massive parallelism; occupancy and mapping of blocks to SMs matter for performance. ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-c-programming-guide/?utm_source=chatgpt.com))

# **3 — CUDA memory types (summary and when to use)**

1. **Global memory (device DRAM)** — large, accessible by all threads; high latency; use for major arrays.  
    Allocated with cudaMalloc() / freed with cudaFree(). Transfers via cudaMemcpy() (or async variants).
2. **Shared memory** — per-block, on-chip, low latency, used for thread cooperation and reducing global memory accesses.
3. **Registers** — per-thread, fastest, auto-managed by compiler.
4. **Constant memory** — small, cached, read-only from kernels; \_\_constant\_\_ declarations.
5. **Texture memory** — cached read-only with specialized filtering/addressing (useful for image-like access patterns).
6. **Host (pageable) memory** — ordinary CPU memory. Copies to/from device require page migration and kernel stalls unless pinned.
7. **Pinned (page-locked) host memory** — allocated with cudaHostAlloc() or cudaHostRegister(); enables faster DMA and asynchronous transfers (zero-copy or faster host→device throughput).
8. **Unified Memory (Managed)** — one pointer usable on host and device; CUDA runtime/page migration does the transfers automatically (cudaMallocManaged()); simplifies coding, but performance characteristics depend on access patterns. ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html?utm_source=chatgpt.com), [olcf.ornl.gov](https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf?utm_source=chatgpt.com))

# **4 — Key CUDA runtime API functions (memory & device)**

* cudaGetDeviceCount(int\* count) — how many GPUs.
* cudaSetDevice(int dev) / cudaGetDevice() — select GPU for subsequent calls. Use per-thread device selection for multi-threaded host apps.
* cudaMalloc(void\*\* devPtr, size\_t size) / cudaFree(void\* devPtr).
* cudaMemcpy(void\* dst, const void\* src, size\_t count, cudaMemcpyKind kind) — kind is cudaMemcpyHostToDevice, etc.
* cudaMemcpyAsync(...) — async transfers tied to a stream (host memory must be pinned for overlapping).
* cudaMallocManaged() — unified memory allocation.
* cudaHostAlloc() — pinned host allocation (with flags like cudaHostAllocDefault or cudaHostAllocMapped).
* cudaStreamCreate() / cudaStreamDestroy() / cudaEventRecord() / cudaEventSynchronize() — streams and events for concurrency/timing. ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html?utm_source=chatgpt.com))

# **5 — How memory transfer works (host↔device)**

* By default, host pointers are pageable. cudaMemcpy() copies through the driver and kernel pages; this involves CPU-side staging and is synchronous.
* **Pinned host memory** avoids page faults and lets the GPU DMA directly into host memory — this enables true overlapping of kernel execution and transfers when using streams + cudaMemcpyAsync().
* **Unified Memory** (cudaMallocManaged) creates a managed allocation that the driver migrates between CPU and GPU as pages are accessed; simplifies coding but you must be aware of page faults and migration costs; cudaMemPrefetchAsync() helps performance on newer hardware by prefetching to a device or host. ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html?utm_source=chatgpt.com), [olcf.ornl.gov](https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf?utm_source=chatgpt.com))

# **6 — Device assignment and multi-GPU basics**

* Query devices with cudaGetDeviceCount(&n).
* Choose GPU with cudaSetDevice(dev\_id) on the host thread — all subsequent runtime calls on that host thread affect that device.
* For multi-GPU work: create separate CPU threads each with its own cudaSetDevice() and perform compute & transfers pinned to that thread/device.
* **Peer-to-peer** (P2P) access: if GPUs are P2P-capable (same PCIe root complex or NVLink), you can enable/perform cudaDeviceEnablePeerAccess() and use cudaMemcpyPeerAsync() to copy directly GPU→GPU without staging through host. This gives much lower latency/better bandwidth for GPU-to-GPU comms. ([events.prace-ri.eu](https://events.prace-ri.eu/event/989/sessions/3096/attachments/1196/2029/cuda-multiple-gpus.pdf?utm_source=chatgpt.com), [NVIDIA Developer Forums](https://forums.developer.nvidia.com/t/how-does-cudamemcpypeer-async-work-with-streams/267521?utm_source=chatgpt.com))

# **7 — Streams and concurrency**

* A **stream** is a sequence of operations (memcpy, kernel calls, events) that execute in order on the device. Different streams can run concurrently if there are hardware resources available.
* Use cudaStreamCreate() and pass the stream to kernel launches or cudaMemcpyAsync() to enable overlap of compute & comms.
* For overlap: host must use pinned memory for async memcpys and ensure kernels/transfer use different streams appropriately. Use cudaEventRecord()/cudaEventSynchronize() for precise ordering and timing.

# **8 — Performance tips (rules of thumb)**

* Minimize host↔device transfers; transfer in large chunks, not tiny ones.
* Use pinned host memory when overlapping transfers with compute.
* Use streams to overlap work, but keep an eye on contention (PCIe bandwidth, device concurrency).
* Use shared memory to reduce global memory traffic and coalesce global loads/stores for bandwidth efficiency.
* Keep threads per block as multiples of warp size (32) and tune occupancy (blocks per SM).
* Profile (nsight, nvprof/NVIDIA Nsight Systems/Compute) — algorithmic changes often more impactful than micro-optimizations.  
   (Official programming guide describes memory fences, ordering, and detailed model behavior.) ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-c-programming-guide/?utm_source=chatgpt.com))

# **9 — Concrete commands: Installing CUDA (Ubuntu 22.04 example)**

*Note: Always check NVIDIA’s CUDA download page for the CUDA version matching your driver/kernel. The commands below are representative; use the NVIDIA installer page to select your OS + version for the exact package and instructions.* ([NVIDIA Developer](https://developer.nvidia.com/cuda-downloads?Distribution=Ubuntu&target_arch=x86_64&target_os=Linux&target_type=deb_local&target_version=22.04&utm_source=chatgpt.com), [NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-installation-guide-linux/?utm_source=chatgpt.com))

Ubuntu (official .deb local installer flow — example for CUDA 13.x on Ubuntu 22.04):

# 1) Remove old drivers (careful on production systems)

sudo apt purge 'nvidia-\*' -y

sudo rm /etc/apt/sources.list.d/cuda\*

# 2) Add CUDA repository (example — get the correct URL from NVIDIA site for your CUDA version)

# (this is shown as an example; use the URL/commands on developer.nvidia.com/cuda-downloads)

wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86\_64/cuda-ubuntu2204.pin

sudo mv cuda-ubuntu2204.pin /etc/apt/preferences.d/cuda-repository-pin-600

sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86\_64/3bf863cc.pub

sudo add-apt-repository "deb https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86\_64/ /"

# 3) Install driver + toolkit (choose 'cuda-toolkit' or 'cuda' meta package)

sudo apt update

sudo apt -y install cuda-toolkit-13-0 # or: sudo apt -y install cuda

# 4) Reboot

sudo reboot

# 5) Verify

nvidia-smi

nvcc --version

If you prefer the .run installer, follow NVIDIA's detailed instructions — but many recommend using distro packages for easier driver/kernel integration. ([NVIDIA Developer](https://developer.nvidia.com/cuda-downloads?Distribution=Ubuntu&target_arch=x86_64&target_os=Linux&target_type=deb_local&target_version=22.04&utm_source=chatgpt.com), [NVIDIA Docs](https://docs.nvidia.com/cuda/archive/12.2.0/cuda-installation-guide-linux/index.html?utm_source=chatgpt.com))

RHEL / CentOS / Rocky / Alma (DNF/YUM flow — example):

# Add NVIDIA repo (example for RHEL8)

sudo dnf config-manager --add-repo \

http://developer.download.nvidia.com/compute/cuda/repos/rhel8/x86\_64/cuda-rhel8.repo

# Install driver and toolkit

sudo dnf clean all

sudo dnf -y module install nvidia-driver:latest-dkms

sudo dnf -y install cuda-toolkit

# Reboot

sudo reboot

# Verify

nvidia-smi

nvcc --version

Always consult the official CUDA Installation Guide for Linux for the exact repo URL and version-specific steps. ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-installation-guide-linux/?utm_source=chatgpt.com), [docs.heavy.ai](https://docs.heavy.ai/v6.4.3/installation-and-configuration/installation/installing-on-rhel/centos-yum-gpu-os?utm_source=chatgpt.com))

# **10 — Example 1 — Minimal host + kernel (C-style CUDA .cu)**

Save as vecAdd.cu and compile with nvcc vecAdd.cu -o vecAdd.

// vecAdd.cu

#include <stdio.h>

#include <cuda\_runtime.h>

\_\_global\_\_ void vecAdd(const float \*a, const float \*b, float \*c, int n) {

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

if (idx < n) c[idx] = a[idx] + b[idx];

}

int main() {

int n = 1<<20;

size\_t bytes = n \* sizeof(float);

// Host alloc

float \*h\_a = (float\*)malloc(bytes);

float \*h\_b = (float\*)malloc(bytes);

float \*h\_c = (float\*)malloc(bytes);

for (int i = 0; i < n; ++i) { h\_a[i] = i; h\_b[i] = 2\*i; }

// Device alloc

float \*d\_a, \*d\_b, \*d\_c;

cudaMalloc((void\*\*)&d\_a, bytes);

cudaMalloc((void\*\*)&d\_b, bytes);

cudaMalloc((void\*\*)&d\_c, bytes);

// Copy H->D

cudaMemcpy(d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

// Launch

int blockSize = 256;

int gridSize = (n + blockSize - 1) / blockSize;

vecAdd<<<gridSize, blockSize>>>(d\_a, d\_b, d\_c, n);

cudaDeviceSynchronize();

// Copy D->H

cudaMemcpy(h\_c, d\_c, bytes, cudaMemcpyDeviceToHost);

// Verify

for (int i = 0; i < 5; ++i) printf("%f ", h\_c[i]);

printf("\n");

// Cleanup

cudaFree(d\_a); cudaFree(d\_b); cudaFree(d\_c);

free(h\_a); free(h\_b); free(h\_c);

return 0;

}

This is the basic flow: allocate host, allocate device, copy, launch kernel, copy back, free.

# **11 — Example 2 — Using pinned memory and streams for overlap**

Key points: host buffers must be pinned (cudaHostAlloc) to overlap cudaMemcpyAsync with kernels.

// overlap\_example.cu (concept)

#include <cuda\_runtime.h>

#include <stdio.h>

\_\_global\_\_ void dummyKernel(float \*d, int n) {

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if (i < n) d[i] += 1.0f;

}

int main() {

int n = 1<<20;

size\_t bytes = n \* sizeof(float);

float \*h\_buf;

cudaHostAlloc((void\*\*)&h\_buf, bytes, cudaHostAllocDefault); // pinned

for (int i=0;i<n;i++) h\_buf[i]=i;

float \*d\_buf;

cudaMalloc((void\*\*)&d\_buf, bytes);

cudaStream\_t s1;

cudaStreamCreate(&s1);

// Async copy H->D in stream s1

cudaMemcpyAsync(d\_buf, h\_buf, bytes, cudaMemcpyHostToDevice, s1);

// Launch kernel in s1; will begin when H->D done

int block=256, grid=(n+block-1)/block;

dummyKernel<<<grid,block,0,s1>>>(d\_buf, n);

// Async copy D->H in same stream

cudaMemcpyAsync(h\_buf, d\_buf, bytes, cudaMemcpyDeviceToHost, s1);

// Wait

cudaStreamSynchronize(s1);

// cleanup...

cudaFree(d\_buf);

cudaFreeHost(h\_buf);

cudaStreamDestroy(s1);

return 0;

}

Use nvprof / Nsight to verify overlap.

# **12 — Example 3 — Unified Memory (very simple)**

// managed\_example.cu

#include <cuda\_runtime.h>

#include <stdio.h>

\_\_global\_\_ void add1(int \*x) { x[threadIdx.x] += 1; }

int main() {

int N = 32;

int \*x;

cudaMallocManaged(&x, N \* sizeof(int));

for (int i=0;i<N;i++) x[i]=i;

add1<<<1,N>>>(x);

cudaDeviceSynchronize();

for (int i=0;i<10;i++) printf("%d ", x[i]);

printf("\n");

cudaFree(x);

return 0;

}

Unified memory simplifies code — but for performance-critical kernels you’ll often prefer explicit cudaMalloc + cudaMemcpy and prefetch where supported using cudaMemPrefetchAsync().

Refer to the Unified Memory doc for behaviors and best practices. ([olcf.ornl.gov](https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf?utm_source=chatgpt.com))

# **13 — Multi-GPU example (sketch)**

* Query device count.
* Create host threads, each thread calls cudaSetDevice(myDev) and runs kernels/allocations for that GPU; or on single host thread, cudaSetDevice() before work for each device in sequence.
* If GPUs support P2P, enable via cudaDeviceEnablePeerAccess(peer, 0). Use cudaMemcpyPeerAsync() for GPU→GPU copies. See NVIDIA multi-GPU programming notes for details. ([events.prace-ri.eu](https://events.prace-ri.eu/event/989/sessions/3096/attachments/1196/2029/cuda-multiple-gpus.pdf?utm_source=chatgpt.com), [NVIDIA Developer Forums](https://forums.developer.nvidia.com/t/how-does-cudamemcpypeer-async-work-with-streams/267521?utm_source=chatgpt.com))

# **14 — Debugging & profiling tools**

* cuda-memcheck — detect out-of-bounds and API errors.
* nsys (Nsight Systems) and nv-nsight-cu/Nsight Compute — profiling and kernel analysis.
* nvidia-smi — driver, GPU usage, and process listing.  
   Use the NVIDIA developer tools suite for bottleneck analysis.

# **15 — Common pitfalls**

* Forgetting to check CUDA API return codes — always check cudaError\_t and call cudaGetLastError() after kernel launches.
* Using pageable host memory with cudaMemcpyAsync() — will not overlap; must use pinned.
* Ignoring stream/device contexts when using threads — record which host thread has set the device.
* Assuming unified memory is always as fast as explicit transfers — it simplifies code but can cause page migrations at runtime; prefetch or advise on access can help.

# **16 — Suggested next steps & resources**

If you want, I can:

* Provide a tailored optimization checklist for your radar signal processing code (focus: data layout, using shared memory, stream-based pipelining, multi-GPU tiling + P2P).
* Convert one of your specific C parts into a CUDA kernel and an optimized version with streaming/pinned memory.
* Give a reproducible step-by-step install for *your exact Linux distro and kernel* (I’ll need distro+version; I used general Ubuntu 22.04 and RHEL8 style commands above). ([NVIDIA Developer](https://developer.nvidia.com/cuda-downloads?Distribution=Ubuntu&target_arch=x86_64&target_os=Linux&target_type=deb_local&target_version=22.04&utm_source=chatgpt.com), [NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-installation-guide-linux/?utm_source=chatgpt.com))

# **Short bibliography (key pages I used)**

* CUDA C++ Programming Guide (memory model, execution model). ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-c-programming-guide/?utm_source=chatgpt.com))
* CUDA Runtime API — Memory Management (malloc/memcpy/managed/pinned). ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html?utm_source=chatgpt.com))
* CUDA Installation Guide for Linux (official install steps). ([NVIDIA Docs](https://docs.nvidia.com/cuda/cuda-installation-guide-linux/?utm_source=chatgpt.com))
* Unified Memory notes & examples (CUDA managed memory). ([olcf.ornl.gov](https://www.olcf.ornl.gov/wp-content/uploads/2019/06/06_Managed_Memory.pdf?utm_source=chatgpt.com))
* Multi-GPU / peer-to-peer guidance (P2P, cudaMemcpyPeer). ([events.prace-ri.eu](https://events.prace-ri.eu/event/989/sessions/3096/attachments/1196/2029/cuda-multiple-gpus.pdf?utm_source=chatgpt.com), [NVIDIA Developer Forums](https://forums.developer.nvidia.com/t/how-does-cudamemcpypeer-async-work-with-streams/267521?utm_source=chatgpt.com))

If you want, tell me:

1. Your OS (exact distro + version) and whether you have an NVIDIA GPU already (run nvidia-smi and paste output if it exists).
2. One concrete function from your radar code you want accelerated — paste the CPU C function and I’ll produce a CUDA port + optimization notes (shared memory, tiling, streams).

Which of those now — install steps or CUDA-ify a function?