O'REILLY®

NVIDIA Open Source Ecosystem

A Practitioner's Guide to CUDA Libraries, RAPIDS, and GPU-Accelerated Computing



Max Buckley

Nvidia: Open Source Ecosystem

A Practitioner's Guide to CUDA Libraries, RAPIDS, and GPU-Accelerated Computing

With Early Release ebooks, you get books in their earliest form — the author's raw and unedited content as they write — so you can take advantage of these technologies long before the official release of these titles.

Max Buckley



Nvidia: Open Source Ecosystem

by Max Buckley

Copyright © 2028 O'Reilly Media, Inc. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (https://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield

Development Editor: Sara Hunter

Production Editor: Clare Laylock

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

February 2028: First Edition

Revision History for the Early Release

- 2026-10-27: First Release

See https://oreilly.com/catalog/errata.csp?isbn=9798295901928 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Nvidia: Open Source Ecosystem*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have

used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-295-90188-1

[LSI]

Brief Table of Contents (*Not Yet Final*)

Preface (unavailable)

Part I: Foundations

Chapter 1: The GPU Computing Landscape and NVIDIA's Open-Source Bet

Chapter 2: GPU Architecture from the Programmer's Perspective (unavailable)

Chapter 3: The CUDA Programming Model (unavailable)

Chapter 4: The Standard Library for CUDA C++ (unavailable)

Part II: GPU-Accelerated Data Science with RAPIDS

Chapter 5: RMM and the RAPIDS Memory Model (unavailable)

Chapter 6: GPU DataFrames and the Pandas Revolution (unavailable)

Chapter 7: cuML — Machine Learning at GPU Speed (unavailable)

Chapter 8: cuGraph — Graph Analytics and the NetworkX Backend (unavailable)

Chapter 9: cuVS — Search for the Era of Unstructured Data

Part III: Deep Learning Infrastructure

Chapter 10: CUTLASS and CuTe — Open-Source Matrix Multiplication (unavailable)

Chapter 11: DALI and CV-CUDA — GPU-Accelerated Data Pipelines (unavailable)

Chapter 12: NCCL and Multi-GPU Communication (unavailable)

Chapter 13: TensorRT-LLM and Triton — Inference at Scale (unavailable)

Part IV: The Broader Ecosystem and Practice

Chapter 14: Profiling and Debugging GPU Applications (unavailable)

Chapter 15: The Wider NVIDIA Open-Source World (unavailable)

Chapter 16: Contributing to the NVIDIA Open-Source Ecosystem (unavailable)

Chapter 1. The GPU Computing Landscape and NVIDIA's Open-Source Bet

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form — the author's raw and unedited content as they write — so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at shunter@oreilly.com.

Nvidia has changed the world. If you are reading this book, that idea hardly comes as news to you. Through reading this book you will become more familiar with the company, with their core products, and most importantly, intimately familiar with the vast array of software they make freely available. This knowledge will help you make the most of whatever Nvidia hardware you have access to, and help you expedite any software development tasks that benefit from extensive amounts of numerical computation.

This introductory chapter is unique within this book, focusing on the broader context of the company, of its history, and the economics behind it. We aim to in part answer the question of, why, why Nvidia invests so much in open source that we have a book to write in the first place. Subsequent chapters focus more on the individual repositories, the problems they solve

and how the technologies enable it. So if this introductory chapter's style isn't for you, stick with it, the other chapters will be much more technology, much less narrative.

This chapter starts by outlining the backstory and recent history of the company; telling the story of how Nvidia went from a maker of PC accessories to becoming the world's largest company by market capitalization. A huge driver of this success story is CUDA, their accelerated computing platform. Later in this chapter we will cover what CUDA actually is, which is not as clear cut as many assume. This deeper understanding of CUDA will enable readers to better understand the space and disambiguate what people mean when they talk about CUDA. Finally this chapter gives a high level map of the content to come giving an overview of what to expect.

The backstory outlines how a company went from the production of accelerator cards and the rendering of video game graphics to making a bet on repurposing these same graphics cards for general purpose and scientific computing. That bet in particular took several years to find its killer app, but when it did it showed the world that Nvidia graphics cards (later rebranded as GPUs) paired with deep learning could produce astounding results, results that would soon spill over to all sorts of deep learning applications which rapidly took over most areas of what was then mostly referred to as machine learning (ML).

While the growth of deep learning rapidly accelerated in 2012, the stock market growth for Nvidia has really exploded since the end of 2022 with the release of OpenAI's ChatGPT. ChatGPT's release marked the start of the current AI boom. Where the word AI was resuscitated, and even normal everyday nontechnical people became very aware of the phenomenon and of Nvidia as a company.

As this book is being written (July 2026) Nvidia sits at a market cap of $5.4 Trillion dollars and is the world's most valuable company by market capitalization. But how did we end up here?

NVIDIA Corp

NASDAQ: NVDA

223.96 USD ↑ 559,800.00% +223.92 all time + Follow

Closed: 7 Aug, 19:59 GMT-4 • Disclaimer

After hours 223.80 −0.16 (0.071%)

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max



Open	221.54	Mkt cap	5.42T	Dividend	0.45%
High	224.76	P/E ratio	34.30	Qtrly Div Amt	0.25
Low	220.66	52-wk high	236.54	52-wk low	164.07

Figure 1-1. Nvidia share price

Why the Road from Computer Graphics Led to Scientific Computing

For much of its history Nvidia was a hardware company, its products selling almost exclusively to gamers for use in high end gaming PCs, naturally Nvidia also provided hardware for some games consoles (notably the original Xbox, PS3, and Nintendo Switch), though far fewer games consoles than their competitor AMD.

While in recent years there have been only two companies competing in the dedicated computer graphics card space (Nvidia and AMD), this was not always the case. In the 90s when Nvidia started out there were several companies competing in the space, the winners, Nvidia acquired the early leader 3dfx and ATI was later acquired by AMD. A major market consolidation.

Fortunately for Nvidia, gamers were a demanding bunch. Humans are visual creatures, they can instantly and accurately assess graphical fidelity. As a result of this, game marketing often focused on graphical fidelity, game studios were driven to produce as high quality, and hardware manufacturers invested enormous sums in the hardware to power those ambitions.

The combination of intense competition and strong demand in the graphics market forced constant, rapid innovation. Without this innovation games would be unable to deliver the increasingly complex 3d worlds in real time at both high resolutions, and high framerates that customers demanded. Because rendering is inherently parallel, millions of pixels computed independently, each hardware generation exploited more parallelism than the last; and as developers demanded richer effects, the fixed-function rendering pipelines gave way to programmable hardware.

The quest for photorealistic graphics is one that fundamentally requires effectively unbounded computation. 2d gave way to basic 3d, 3d became

progressively more complex with tricks like precomputed lighting giving way to real time light and shadows and eventually ray tracing, path tracing, and most recently neural rendering.

By the mid-2000s NVIDIA was shipping massively parallel, increasingly general-purpose processors, built for graphics but no longer limited to it. The sheer compute and potential of this hardware was not a secret and some companies went as far as hiring game programmers to wring performance out of GPUs by leveraging graphics APIs in unconventional ways.

The CUDA Inflection Point

Note: the word 'device' is used throughout the CUDA and Nvidia literature to refer to the GPU: the coprocessor that executes kernels which usually has its own dedicated memory (VRAM), as distinct from the 'host', refers to the CPU and main system memory (RAM). In some more modern unified memory architecture systems (UMA) there is a single pool of RAM shared by the CPU and GPU.

CUDA (Compute Unified Device Architecture) is Nvidia's general purpose GPU (GPGPU) programming model. The name is quite a mouthful and luckily is rarely expanded anymore.

The history of CUDA traces back to the early 2000s when a computer science PhD student named Ian Buck was exploring making GPUs available for general purpose computing as part of his PhD at Stanford. In 2004, as part of said PhD work, Ian released "Brook", an early and influential example of GPGPU programming.

After his PhD in 2004 Ian immediately joined Nvidia. There he started working alongside John Nickolls developing what would go on to become CUDA. This work took influence from his prior work on Brook. CUDA was publicly announced in late 2006 and the first major version was released in early 2007.

The core idea behind CUDA was to allow developers to write C, with modest extensions, and have it execute on the GPU: to be able to take

existing C programs, identify the data-parallel parts, and move those parts to the GPU while leaving the serial parts on the CPU. The term "heterogeneous computing" came to describe this arrangement of different processor types working in unison.

From when it was first announced in late 2006 and made available on that year's newly released GPU (the GeForce 8800). NVIDIA built CUDA support into every subsequent GPU it shipped, including its cheapest consumer gaming cards. This was a strange decision on its face: it added cost to every chip, confused investors, and served a market that did not yet exist.

To make matters worse, uptake by industry and research was initially limited, and the investment did not end at the hardware. Rather than ship and forget, NVIDIA began laying the groundwork for much of what this book covers: high-performance libraries, tooling, and the surrounding ecosystem. This was a long term bet and something that required both strategic vision and strong leadership on the part of Jensen Huang.

For the first six years following CUDA's release, the uptake was not great. From 2006 CUDA grew slowly but steadily. Scientists in physics, chemistry, and finance adopted it for simulations. A small community of parallel computing enthusiasts built libraries and tools. But the broader software industry remained skeptical. GPUs were fast for certain problems, the argument went, but the programming model was too alien and the use cases too narrow.

Industry was not idle in those years either; it was solving the problem of scaling in a different way. Google's MapReduce paper (2004) and its open-source descendant Hadoop (2006) taught a generation of engineers that the answer to a large computation was a large number of cheap CPU machines and a program written as maps and reduces. That model fit the workloads of the day, which were dominated by reading and shuffling text rather than by arithmetic, and it came with tooling that hid the parallelism from the programmer. CUDA asked for the same discipline, independent work followed by a reduction, but exposed it on a single chip in a language that

looked like C with the safety removed. For the web-scale companies that set the agenda, the GPU was answering a question they weren't asking. That would change when a workload arrived that scale-out model handled poorly.

The AlexNet Moment

In late 2012, a team from the University of Toronto: Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, won the ImageNet computer-vision competition. Their success reoriented the field of machine learning around deep learning, and, crucially for NVIDIA, around its GPUs and CUDA. This victory sparked the modern deep learning revolution and marked the first major inflection point in Nvidia's ascent to world dominance.

AlexNet's result was significant for two reasons. Firstly, the margin: AlexNet achieved a top-5 error rate of 15.3 percent against 26.2 percent for the 2nd place runner-up, a gap so large that it cemented neural networks, now rebranded as deep learning, as the new method of choice for computer vision. Secondly, the hardware: AlexNet was trained on just two consumer NVIDIA gaming cards in a single desktop machine, using custom CUDA code Krizhevsky had written himself, at a time when serious computation meant CPU-based supercomputing clusters or distributed systems. Two gamer GPUs had delivered what was, for the era, an enormous amount of training computation.

It is worth reiterating that in 2012 NVIDIA did not have a monopoly on the GPU market; AMD, which had acquired ATI in 2006, held roughly 40% percent of the discrete GPU market. What NVIDIA had that AMD did not was CUDA: a stable, (relatively) easy-to-use programming model, in every card it had shipped for five years, with a budding software ecosystem built around it, much of which we will discuss throughout this book. This CUDA based software ecosystem is a huge part of Nvidia's competitive advantage and why it is the largest company in the world today.

So What Exactly Is CUDA?

> **NOTE**
>
> How to pronounce "cu": The "cu" prefix in NVIDIA library names stands for CUDA. Most developers pronounce it "coo": so cuDF is "coo-D-F," cuML is "coo-M-L," cuVS is "coo-V-S," and cuGraph is "coo-graph." You'll occasionally hear "C-U-D-F" spelled out, but the "coo" pronunciation dominates at GTC talks and throughout the Nvidia developer community.

Before we can say anything particularly useful about NVIDIA's open-source ecosystem, we have to disambiguate a word that will appear on nearly every page of this book: CUDA. This term has already appeared in this book dozens of times so we'd best define it more accurately. This disambiguation is useful beyond these pages, the word CUDA is widely used in technology and more recently even appears regularly in business and finance discourse.

Beyond CUDA being one of the most successful brand names in computing, it is also one of the most misunderstood. If you ask five practitioners what CUDA is, you will likely get back five different, yet reasonable answers: a programming language, a compiler, an API, a set of libraries, and a platform.

All five of these answers are in common circulation, often within a single conversation, and the ambiguity is a major source of confusion. Many public arguments about the "CUDA moat", about whether some new language or framework will "break CUDA's lock-in", or about whether a project "depends on CUDA" turn out, on inspection, to be arguments between people using the word for different things at different layers of the stack. Here are the five, in descending order, from what you type to what the hardware actually executes.

1. A programming language

When many people think of CUDA, they picture the C (now C++) extension language, with its functions decorated as `__host__` and `__device__` and its `<<>>` launch syntax. This makes sense,

because it is how they first encountered GPU programming, and it is the dialect in which the canonical tutorials and textbooks are written.

2. A compiler

The toolchain that turns CUDA C++ into something the CUDA platform can run. nvcc is the visible front door, but the real contract is PTX, the platform's virtual instruction set: nvcc compiles your code down to PTX, and there the toolchain's job ends. Turning PTX into what the silicon actually executes is the platform's business. When people say a language "targets CUDA," they often mean it emits PTX directly: OpenAI's Triton and Numba (a Python JIT compiler) skip CUDA C++ and this front door, emitting PTX directly.

3. An API

Or rather two APIs for controlling the GPU: the low-level driver API (cu* functions: load this module, allocate this memory, launch this kernel on this stream), exposed directly by the driver, and the higher-level runtime API (cuda* functions) layered above it. This is how a CPU program, in any language with a C foreign function interface, talks to the platform.

4. A set of libraries.

cuBLAS, cuDNN, cuFFT, cuSPARSE, Thrust, CUB, NCCL, and dozens more: precompiled or header-only, heavily tuned implementations of the operations that dominate real workloads. For a large fraction of CUDA users, especially in deep learning, "using CUDA" means calling these libraries and never having to write a kernel at all.

5. A platform

The original meaning of the word, dating to CUDA's 2007 launch as the answer to a hardware question: now that GPUs are massively parallel processors, how does anyone program them for something other than graphics? The platform is an execution model (kernels launched over

grids of thread blocks), the PTX virtual instruction set, and the driver that lowers PTX to SASS, the GPU's native code, and manages the hardware. Everything else called "CUDA" sits on top of this.

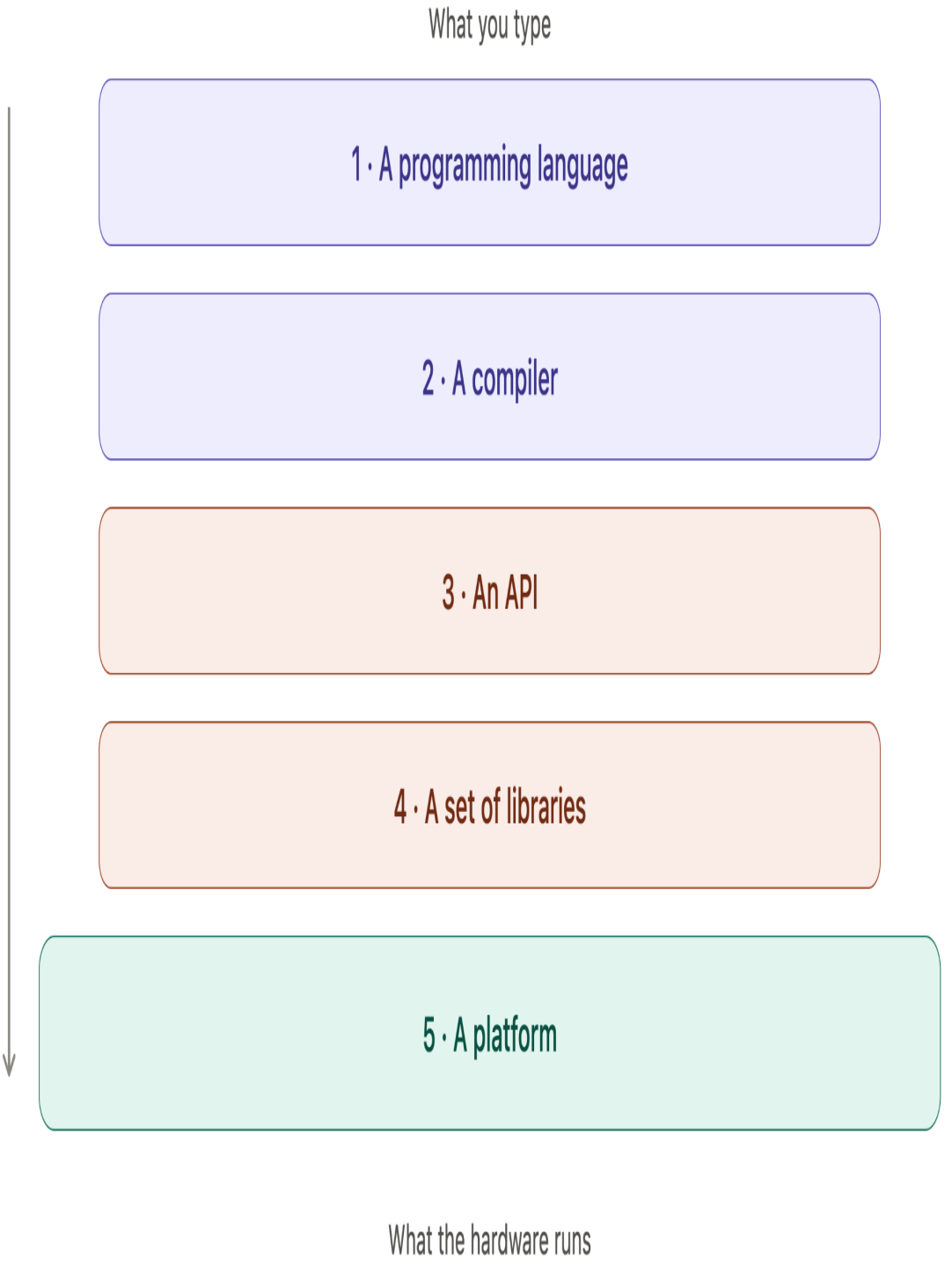

What you type
1 · A programming language
2 · A compiler
3 · An API
4 · A set of libraries
5 · A platform
What the hardware runs

Figure 1-2. The layers of CUDA

Many systems described as 'CUDA competitors' actually only target the top levels of, which were never the CUDA moat: PTX is a publicly documented target, and many open source compilers already emit it. The APIs are clonable too; The libraries resist harder, because matching a decade of tuned kernels is not a porting project but a performance race against a moving target. Yet even they are downstream of the real answer. NVIDIA can regenerate the libraries at will because they control the platform: the driver, the ISA (Instruction Set Architecture), the hardware they're tuned against. The libraries are where the lock-in is thickest today. The platform is why it regenerates.

CUDA Is a Platform, Not a Programming Language

CUDA is NVIDIA's brand for GPGPU computing: an umbrella term covering **everything that runs on NVIDIA GPUs** without going through the graphics APIs. CUDA sits on top of the NVIDIA device driver, and all other GPGPU workloads route through it.

This platform view also brings clarity to OpenCL's real position on NVIDIA hardware. OpenCL is often described as an alternative or competitor to CUDA, and as a programming language (OpenCL C) and an API it is one. But NVIDIA's OpenCL implementation is not an independent stack: it is a sibling library beside the CUDA driver API, and behind both sits shared machinery. OpenCL kernels are compiled by the same NVVM compiler to the same PTX, lowered by the same JIT, and dispatched through the same kernel-mode driver. The competition was only ever at the front door.

The stable contract at the heart of CUDA is PTX (Parallel Thread Execution), the virtual instruction set that nvcc and other tools emit, not CUDA C++. The programming language is one client of the platform among many. Anything that can produce PTX and launch the result through the driver API is using CUDA in the sense that matters, irrespective of whatever the original source code looked like.

Why PTX?

Each NVIDIA GPU generation has a different physical architecture, and PTX gives compilers a common virtual target while NVIDIA maintains responsibility and control over the mapping from PTX to each of their real physical hardware architectures. This intermediate representation also gives Nvidia the flexibility to significantly change the physical architecture from each generation to the next.

If every compiler had to target every architecture directly, supporting n languages on k architectures would require O(n·k) compiler backends; with PTX as an intermediate representation, it requires only O(n+k). Because the mapping happens at load time, binaries shipping PTX can run on architectures that didn't exist when they were compiled. This is what gives PTX its forward-compatibility guarantee: code written for one GPU generation will run on later ones.

It is important to reiterate that PTX has always been publicly documented, though proprietary. It was always Nvidia's intention that other compilers would be able to generate PTX, even before there were multiple languages and compilers that actually created it. The landmark for open source was 2015, when Google's gpucc work landed in Clang and gave the world the first CUDA C++ compiler that NVIDIA didn't own.

The code that actually executes on Nvidia GPUs is called SASS, the native machine code of each architecture. PTX is lowered to SASS either ahead of time by a closed source binary called `ptxas`, or JIT by the driver. That second path is how the forward-compatibility promise is maintained, since the driver translates old PTX for new silicon.

In 2007, the C extension language was the only production-ready way to write CUDA GPU code, and nvcc was the only compiler that emitted PTX. It took years for support to appear elsewhere, but today PTX is a target for many compilers, most of them built on top of LLVM's NVPTX backend. So you can be writing Python with OpenAI's Triton or Numba, Julia, or C++ with OpenCL. Regardless of what you write, your code is compiled to PTX,

converted to SASS, and run through the CUDA platform. You are using CUDA without ever writing a line of CUDA C++.

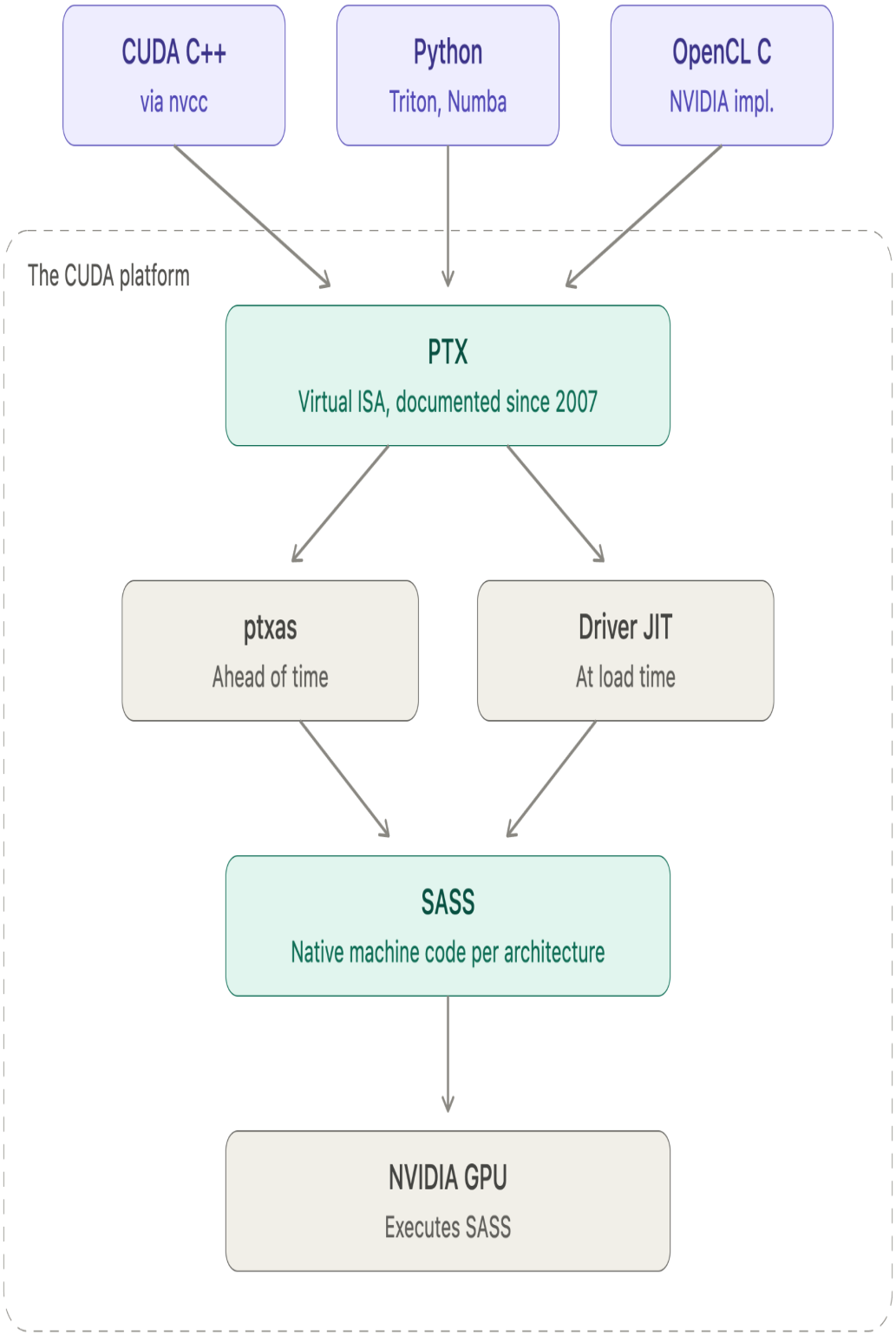

CUDA C++
via nvcc
Python
Triton, Numba
OpenCL C
NVIDIA impl.
The CUDA platform
PTX
Virtual ISA, documented since 2007
ptxas
Ahead of time
Driver JIT
At load time
SASS
Native machine code per architecture
NVIDIA GPU
Executes SASS

Figure 1-3. All roads lead to CUDA

Why This Book Uses the Platform Definition

Throughout this book, when we refer to "CUDA", we mean the platform unless otherwise qualified; when the language is meant, it will be called CUDA C or CUDA C++. Without this framing NVIDIA's open-source strategy looks incoherent. Everything in this book relies on CUDA in various ways even if it is not exactly CUDA C.

Consider what NVIDIA has opened up, and will fill the chapters ahead: language frontends and domain specific languages (DSLs), kernel libraries like CUTLASS, entire frameworks in simulation and robotics, reference implementations across the AI stack. If CUDA were only the programming language, this behavior would be baffling: a company giving away its moat. Under the platform definition, the pattern snaps into focus: none of those things are the moat. The layers that remain closed: the driver, the ptxas and libnvvm optimizers, the SASS instruction set and its undocumented and mysterious scheduling details all sit beneath every frontend, open or closed, first-party or third-party. Each new language that targets PTX widens the top of the funnel; the neck stays exactly where it was. NVIDIA can be genuinely generous at the layers where openness recruits developers, precisely because the layer where the leverage lives is not up for grabs.

That tension of a sprawling, fast-growing open ecosystem funneling into a closed and singular core is the central subject of this book. How the programming model at the top of that funnel actually works threads, blocks, memory hierarchy, and the newer tile-based abstractions is the subject of Chapter 3.

SIDEBAR: IS OPENCL ON AN NVIDIA GPU USING "CUDA"?

OpenCL is the vendor-neutral standard, the historical rival to CUDA C, the language of choice for anyone who wanted GPU compute without vendor lock-in. Yet NVIDIA's OpenCL implementation compiles OpenCL C kernels to PTX and runs them on the same kernel driver, compiler infrastructure, and PTX JIT as everything else. Ask it for your compiled program and the "binary" it returns is PTX, in plain text. By the language definition of CUDA, an OpenCL program is CUDA-free. By the platform definition, it never left.

Why did OpenCL and AMD fall behind?

OpenCL, specified in late 2008, was often pitched as an open competitor to CUDA: a multivendor framework for programming heterogeneous systems composed of CPUs, GPUs, and other accelerators. Being a multivendor solution, it supported not only NVIDIA GPUs but AMD's and others'. It was also governed like a multivendor standard, designed by committee toward the lowest common denominator, while CUDA had a single owner shipping libraries against its own hardware roadmap.

By the time AlexNet appeared in 2012, CUDA already had a multi-year head start on the things that really mattered: libraries and tools, a part of what would later be called the CUDA moat. CUDA had already shipped cuBLAS (linear algebra), cuFFT (Fast Fourier Transforms), Thrust (data structures and algorithms, later folded into the CUDA C++ Core Libraries CCCL (the subject of Chapter 4)), and reasonably mature profiling and debugging tools (Nsight, cuda-gdb, covered in Chapter 14) .

In 2012 Alex Krizhevsky had written complex custom CUDA kernels to train his convolutional network across two GPUs. In the years that followed, NVIDIA and the community codified that work into general-purpose frameworks and libraries, letting far less specialized engineers and

researchers move a model from CPU to GPU with a single line of code. That ease of use helped facilitate massive adoption.

The arrival of cuDNN in 2014 cemented the pattern: deep learning framework owners (Caffe, Torch, Theano, later TensorFlow and PyTorch) stopped writing their own convolution kernels and called cuDNN instead. AMD had no equivalent. OpenCL gave you a compiler and a runtime; clBLAS existed but wasn't competitive, and there was no DNN primitives library until MIOpen in 2017, which arrived on HIP (a C++ runtime API and kernel language for AMD GPUs) rather than OpenCL. So the actual decision facing a framework developer in 2014-2016 was not "CUDA vs OpenCL" but "use cuDNN for free vs write and maintain your own kernel zoo for a second vendor." Nobody had the staffing for the latter.

The growth of CUDA

Table 1-1. CUDA usage by year

Year	CUDA Developers	Key Milestone
2006	1	Released
2007	Thousands	CUDA 1.0 released
2012	~100,000 (est.)	AlexNet wins ImageNet
2020	1 million	Deep learning mainstream
2022	2 million	13 years to reach 1 million registered developers, and less than two more to reach 2 million!
2025	4+ million	3,000+ GPU-accelerated apps, 40,000+ companies
2026	6 million	233% growth in 6 years

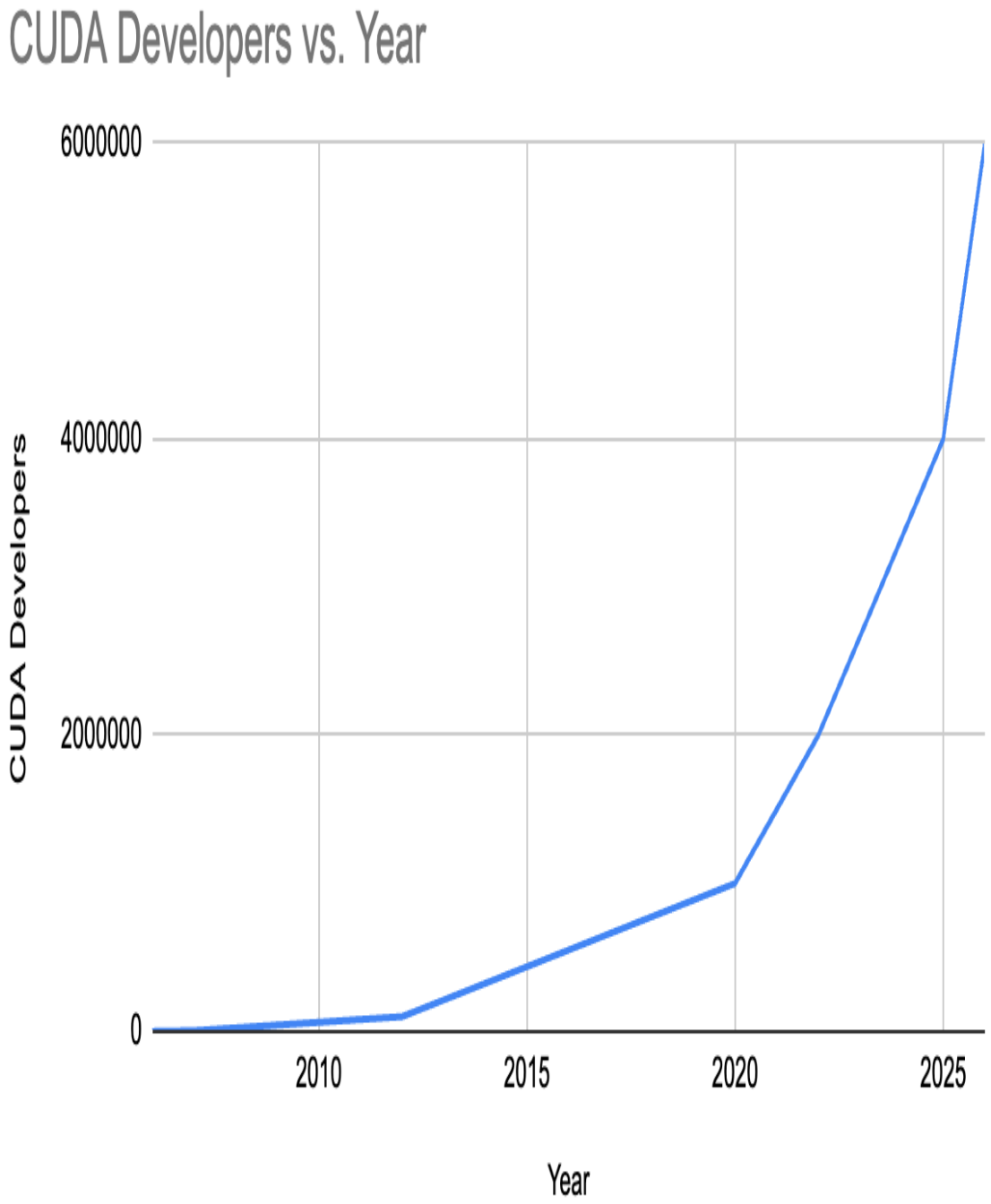


Figure 1-4. CUDA usage by year

The crucial detail that this growth curve obscures: the vast majority of those 6 million developers have never written CUDA C. They are likely not intimately familiar with the CUDA APIs, or low level libraries. For many

they call model.cuda() in PyTorch, they run import cudf in a Jupyter notebook, or they deploy models on the Triton Inference Server. They are consumers of the NVIDIA open-source ecosystem, not necessarily authors of raw CUDA C code.

In spite of this, no book to date explains that ecosystem: what it contains, how the pieces fit together, and when to use which library to unlock the huge potential of your GPU without the complexity of writing low level parallel code. This book fills that gap. To tell you what you should use and how you should use it to build on top of all the amazing work Nvidia has published. How by integrating their packages you can accelerate the data heavy parts of your applications and unlock potentially orders of magnitude speedups.

Why NVIDIA Went Open Source

Nvidia is a hardware company. It makes most of its money selling GPUs. So why does it maintain over a thousand open-source repositories, employ hundreds of engineers to write software that it gives away for free?

The answer is strategic, and Jensen Huang has articulated it clearly. When asked about NVIDIA's competitive moat, he has said that "the single most important property of NVIDIA is the installed base of CUDA". Not the GPUs themselves, the software ecosystem that runs on them. CUDA's 6 million developers, 3,000+ optimized applications, and two decades of accumulated libraries represent switching costs that no competitor can replicate by simply shipping faster hardware.

But what is this strategy and where did it originate?

Commoditizing Your Complement

In microeconomics a complement is "a good whose appeal increases with the popularity of another good". In Nvidia's case a GPU is only really useful if you have software that runs on it. The more software that exists. the more useful and hence valuable a GPU is. By making sure a lot of software is

available and ideally for free, Nvidia makes their GPUs as valuable as possible.

This strategy was explained well by Joel Spolsky in a 2002 essay. The logic is that rational firms try to commoditize (drive toward free/ubiquitous) the complements of their goods. By commoditizing their compliments, the firm can capture the maximum profit from their own products.

Early on, other GPU vendors did not prioritize GPGPU software development and hence left their hardware focused on graphics rendering. Nvidia by contrast, proactively had its engineers build and deploy an ecosystem for the future. In 2007 Nvidia released the first major version of CUDA and then spent ~5 years building out its ecosystem in advance of the AlexNet moment.

It is worth noting that neither Nvidia, nor CUDA is entirely dedicated to open source, but they do open source a huge amount of software. As of the time of writing, Nvidia owns and maintains over 1000 repos on GitHub and are the owner of HuggingFace (Purchasing it in August of 2026 after becoming the number one company on it in February of that same year). The lowest levels of CUDA and several of its key libraries (cuBLAS, cuDNN) are closed source, proprietary code, but most of the layers above them and what integrates with much of the outside world is open source or public in various guises.

Interestingly, this strategic pattern is repeating itself. NVIDIA is currently running the same play in other areas of AI, most visibly in physical AI, the artist formerly known as robotics. This time NVIDIA performed the renaming itself, which tells you something about their intention to own the category. Nvidia has spent years publishing the research, open-sourcing the models, and building the hardware, and now stands ready for what Jensen Huang describes as the next trillion-dollar opportunity.

For many of the readers of this book, you may have an Nvidia GPU in your machine, you likely already benefit from GPU acceleration in various applications and you probably achieve this without worrying about writing CUDA. More specifically, Nvidia's success has been in making Nvidia

GPU support and acceleration a first class citizen in commonly used libraries and frameworks like TensorFlow and PyTorch.

As the story above may already have made clear, this ecosystem makes using GPUs easier and enables their use for a broader set of problems and by a broader set of users. This drives the sales of more GPUs and hence Nvidia's bottom line. Had Nvidia shipped CUDA and never added the libraries on top, OpenCL and AMD may have caught up. Had Nvidia not built cuDNN, it would have been a lot harder and more work for all the downstream ML framework providers to support Nvidia GPUs.

There is a pattern in NVIDIA's open sourcing strategy, and once you see it you cannot unsee it. They publish the specification, allowing many providers to build to that spec, while keeping the lowering proprietary, the conversion of this intermediate representation to machine code is private while multiple providers can write the intermediate representation.

Publishing the interface grows the number of producers. Keeping the lowering proprietary means every producer terminates in their binary.

A Brief Timeline

The following milestones chart the evolution from CUDA's creation to the mature ecosystem we explore in subsequent chapters:

Year	Milestone
2004	Ian Buck joins NVIDIA; CUDA development begins
2006	CUDA announced alongside the GeForce 8800 GTX
2007	CUDA 1.0 released
2012	AlexNet wins ImageNet on two GTX 580 GPUs, the deep learning revolution begins
2017	GOAI (GPU Open Analytics Initiative) formed; Volta introduces Tensor Cores
2018	CUTLASS open-sourced (May); RAPIDS announced at GTC Europe (October)
2020	Ampere architecture; CUDA developer count reaches 1.8 million
2022	Hopper architecture; open-source GPU kernel modules released (May)
2024	Blackwell architecture; cudf.pandas GA; RAPIDS on PyPI; open kernel modules go production
2025	NVIDIA Dynamo open-sourced; cuML on PyPI; pip-installable RAPIDS
2026	CUDA 20th anniversary, 6 million developers, Dynamo 1.0, Vera Rubin

The Ecosystem Architecture

Onions Have Layers

Software engineering has always made great use of layering and abstraction. Most software developers rely on libraries and frameworks written and tested by other developers. As a concrete example most numerical computing relies on highly performant low level math libraries written and reused across most applications. These developers may want to complete some high level task like training a machine learning model or computing a route over a set of cities by means of some library. This library, which was written by someone else, breaks the problem into a sequence of tasks, and each of these tasks in turn is likely computed by, at least in part by a lower level library.

In Nvidia's ecosystem they have built and continue to maintain many of these layers, much of what is sometimes referred to as the broader CUDA ecosystem.

When you want to use your GPU to accelerate a data science workload you can substitute cuML instead of Scikit Learn and that in turn relies both on lower level libraries within RAPIDS (cuVS, RAFT, RMM) and on even lower level closed source GPU libraries like cuSOVER, cuRAND and cuBLAS.You have changed a single import and now have GPU acceleration for large parts of your machine learning application. Similarly if you want to do operations research you can replace your solver with cuOPT and that in turn gives you a deeply optimized solver that leverages other RAPIDS layer (RMM, RAFT) and lower level closed source parallel GPU libraries (cuBLAS, cuSPARSE, etc).

CUDA architect Stephen Jones goes as far as saying "avoid parallel programming wherever possible" as not only is it difficult and error prone, there is a lot of room for subtle optimizations. To make the most out of modern hardware requires a deep knowledge and understanding of not only the programming language and its architecture, but when to drop below the abstractions, when to write PTX or use very specific hardware level instructions.

This means that you, a user of CUDA, do not have to, and in fact should aim not to hand roll most of the parts of your GPU accelerated computing system, you just need to know which parts already exist and where to find them. This book will take you through the layered ecosystem so that you know what exists, where to find it, and how to use it. By using Nvidia's own libraries like cuML, cuVS, cuOpt, you are benefiting from cuBLAS or cuDNN under the hood and you are getting algorithms that have been deeply tested and optimized by expert kernel engineers at Nvidia over decades. These kernels are also optimized for each different generations of Nvidia hardware.

This is a high level overview of the layers of the Nvidia OS ecosystem. In subsequent chapters we will go into these layers in much greater detail.

Layer 0: The Core CUDA Platform

The foundational layer is the platform itself. What you get when you install the CUDA Toolkit and NVIDIA Driver for your GPU. This corresponds to our broader definition of CUDA from above. All of the platform, the two APIs, the compiler (nvcc), and the CUDA C language, the programming model, the runtime, and the driver that let software execute on NVIDIA GPUs. All of this and how code is compiled and executed on Nvidia GPUs will be discussed in more detail in chapters 2 and 3.

The foundational layer is what you get when you install the CUDA Toolkit and an NVIDIA driver for your GPU.

What is the CUDA Toolkit?

The CUDA Toolkit is what a developer installs to build GPU-accelerated applications, from embedded systems, desktop machines up through enterprise, data center, and even at hyperscale deployments. It contains the compiler toolchain, the CUDA runtime (cudart), and a range of libraries and tools. Building an application requires only the CUDA Toolkit plus whichever libraries you link against.

The driver and the Toolkit are separate artifacts on separate release schedules, and this seam is visible from your first day of GPU work. The driver ships `libcuda.so`, one copy per system, serving every GPU in the box. The Toolkit ships `libcudart.so`, which is statically linked into your binary by default. That is why five conda environments on one machine can each carry their own cudart while the system has exactly one libcuda. Chapter 3 returns to why the split exists and what each layer is responsible for.

CUDA's core is not open source (though some of the kernel modules now are), but it is the bedrock that everything else depends on. As such we need to mention some of these parts here so we can call back to them later

Layer 1: CUDA-X Math Libraries

> **NOTE**
>
> *What does "CUDA-X" mean as a brand?* When this book started being written, the term covered the core math libraries (CUDA-X Math). In August 2026 RAPIDS was rebranded as CUDA-X Data Science, and recent marketing uses CUDA-X as an umbrella for all of NVIDIA's libraries across every domain, with a claimed count of 1000 specialized libraries as of September 2027. Expect the term to keep widening.

The CUDA-X Math libraries build on the base CUDA platform by providing high-performance GPU implementations of core mathematical primitives. These primitives are reused by most downstream computational libraries, powering fields like physics, chemistry, deep learning, finance and more.

> **NOTE**
>
> As will become a recurring theme, the term is used differently in different quarters. Sometimes it means the narrower set that ships with the CUDA Toolkit; sometimes it means a broader set including libraries that require a separate download. And some, like cuBLAS, have parts bundled with the toolkit and parts that are external downloads (cuBLASDx, the device-side variant in the MathDx package).

The stable core is cuBLAS, cuSPARSE, cuSOLVER, cuFFT and cuRAND. These ship inside the CUDA Toolkit with no additional download, are released alongside major toolkit versions, and are closed source, as is everything else in this layer. Everything above this layer is open source. Newer or faster-moving libraries (cuTENSOR, cuDSS, cuDNN) are released on their own cadence outside the toolkit and are covered under "Additional downloads" below.

The dense linear algebra core: cuBLAS and cuSOLVER

BLAS has been the lingua franca of linear algebra for forty years, and cuBLAS is its CUDA-accelerated version. cuSOLVER is the GPU analogue of LAPACK, the layer that sits above BLAS. It builds on both cuBLAS and cuSPARSE to provide the operations you reach for when you need to solve something rather than just multiply: factorizations, triangular solves, least squares, eigenvalues. It is what cuML and other libraries lean on when they need, say, a singular value decomposition inside a principal component analysis or a least-squares fit inside a linear regression.

The cuSOLVER name covers three fairly different things. cuSolverDN is the dense half, offering LU, QR, SVD and LDL^T, and it is where most users end up. cuSolverSP handles sparse systems via a sparse QR, and ships with a CPU fallback because some sparsity patterns simply do not expose enough parallelism to be worth putting on a GPU. cuSolverRF is a refactorization library for re-solving a sparse system whose values have changed but whose sparsity pattern has not. The sparse components are now legacy: cuSolverSP and cuSolverRF are deprecated, and cuDSS (below) is the replacement.

Sparse and structured: cuSPARSE

cuSPARSE is the low-level library for sparse matrices. It handles the standard storage formats (coordinate (COO), compressed sparse row (CSR), compressed sparse column (CSC) and others) and the operations between sparse and dense matrices.

Unlike dense linear algebra, sparse linear algebra is memory bound rather than compute bound. In the dense case you load contiguous blocks and reuse each element many times, so you get a high ratio of flops (compute) to bytes (memory bandwidth). In the sparse case each stored nonzero participates in exactly one multiply, and you move indices alongside values. There is no reuse to exploit, so performance is usually set by how fast you can stream the matrix, not by how fast the GPU can multiply.

cuSPARSE underpins other CUDA-X Math libraries and parts of cuGraph, which is covered in Chapter 8.

Signals and randomness: cuFFT and cuRAND

cuFFT does Fast Fourier Transforms, which Gilbert Strang called "the most important numerical algorithm of our lifetime" and which made the Top 10 Algorithms of the Century list in Computing in Science & Engineering.

cuFFT computes discrete Fourier transforms on the GPU. It handles 1D, 2D and 3D transforms, complex-to-complex, real-to-complex and complex-to-real, in half, single and double precision, and it batches: you can issue thousands of small independent transforms in a single call, which is the common case in signal processing and deep learning. Sizes that factor into small primes run fastest, and cuFFT falls back to a slower general algorithm for awkward ones.

The cuFFT API is plan-based and borrowed from FFTW (Fastest Fourier Transform in the West), close enough that cuFFT ships a `cufftw.h` shim for porting existing code. You build a plan once, then execute it repeatedly, which amortizes the cost of choosing a radix decomposition for your size.

cuRAND generates random numbers, of the pseudo and quasi kind. The first is what most people mean by random: a deterministic algorithm

producing a sequence that passes the statistical tests a truly random one would. The second is different in intent. A quasirandom sequence deliberately fills an n-dimensional space evenly rather than independently, which converges faster for numerical integration precisely because it avoids the clumping that genuine randomness produces.

Generating random numbers is easy. Generating them on a GPU is not, and the difficulty is not throughput. If ten thousand threads each need their own stream, those streams must be uncorrelated, and you have to be able to hand each thread its starting point without generating everything before it. That is the skip-ahead problem, and it is what separates a parallel RNG from a serial one. It also determines whether your results are reproducible, since a run that depends on thread scheduling is a run you cannot debug.

Additional downloads

The libraries below are sometimes described as part of the CUDA-X Math libraries but are separate downloads. They are functionally similar: closed source, and providing low-level functionality reused by many higher-level libraries.

Direct sparse solving: cuDSS

cuDSS, the CUDA Direct Sparse Solver, is a newer and faster-evolving library that solves systems of linear equations with very sparse matrices on the GPU. It is the recommended replacement for the deprecated cuSolverSP and cuSolverRF. Direct sparse solvers matter in real-time numerical computing, in applications like autonomous driving and process simulation, where growing model complexity and throughput demands need a solver that keeps up.

Tensor contractions: cuTENSOR

cuTENSOR handles the einsum class of operations: contractions, reductions and elementwise work over tensors of arbitrary dimensionality. Where cuBLAS gives you matrix multiply, cuTENSOR lets you specify which indices to sum over and works out how to map that onto Tensor Cores. Like

cuFFT it is plan-based. It ships a large set of precompiled kernels and can optionally JIT-compile a kernel for a specific contraction, which is what makes unusual index patterns tractable. The API also fuses elementwise operations into the contraction, so scaling and activation don't cost an extra pass over memory.

The core user base is quantum chemistry and quantum circuit simulation, where contracting large tensor networks is the whole computation. That workload drives the feature list: block-sparse contractions, 64-bit extents for tensors that overflow 32-bit indexing, and 3xTF32, which recovers close to FP32 accuracy from three TF32 passes. cuTENSORMp extends contractions across multiple GPUs and nodes, which matters when a single tensor no longer fits in one GPU's memory; NVIDIA reports near-linear scaling on GB200 NVL72.

The deep learning primitives: cuDNN

cuDNN was created at NVIDIA Research in response to AlexNet, with Bryan Catanzaro leading the initial work, and shipped in September 2014. It provides the building blocks of neural networks: forward and backward convolution, matrix multiplication, pooling, normalization and, since version 8, fused attention. Strictly it belongs to the deep learning branch of CUDA-X rather than the math libraries, and it ships separately from the CUDA Toolkit, but readers may look for it here.

The interesting shift in cuDNN is how you call it. The original API was a catalogue of fixed functions: you picked an algorithm from an enum and cuDNN ran it. Since cuDNN 8 (2020) the primary interface is a graph API. You describe an operation graph, cuDNN returns an execution plan, and its runtime fusion engine generates a fused kernel on the fly rather than selecting a precompiled one. That moved fusion out of the frameworks and into the library, which is why a faster fused attention kernel now arrives with a cuDNN update rather than a PyTorch release.

The other thing worth saying is that almost nobody calls cuDNN directly. PyTorch and JAX invoke it constantly on every training run, and most of the people running those jobs have never typed its name.

Layer 2: CCCL — The Standard Library for CUDA C++

The CUDA Core Compute Libraries (CCCL) acts as the standard library for CUDA, analogous to the libc in C or the Standard Library (STL) in C++. CCCL provides a range of standard computer science algorithms optimized for use on GPUs. Operations like reductions, scans, sorts, histograms, compaction, memory loads and stores with a specific layout, synchronisation primitives.

CCCL is composed of three preexisting libraries: Thrust, CUB, and libcudacxx. CCCL also comes bundled inside the CUDA Toolkit. CCCL will be covered in much greater detail in chapter 4, but here is a brief overview.

Thrust

Thrust is a high-level C++ library of parallel algorithms. It helped motivate adding parallel algorithms to the C++ Standard Library. Its abstract interface boosts developer productivity and lets the same code run efficiently on GPUs or multicore CPUs by swapping the backends (CUDA, TBB, OpenMP, and others).

CUB

CUB sits lower in the stack and is CUDA-exclusive. It aims for near-peak performance of parallel algorithms on every GPU architecture. Besides providing device only algorithms, it also supplies cooperative primitives such as block-wide reductions and warp-wide scans so kernel authors can assemble custom, high-performance kernels from proven building blocks.

libcudacxx

libcudacxx (libcu++) provides C++ Standard Library features usable in both host and device code. It provides extensions to Standard Library facilities, and it also exposes CUDA-specific hardware features — synchronization primitives, cache control, atomics, and similar abstractions.

The major distinction between Thrust and CUB comes down to two questions: 1) who writes the kernel, and 2) who manages the memory. With Thrust, the answer to both questions is "not you": you call an algorithm from host code and Thrust handles the kernels and the scratch memory behind it. With CUB, the memory is your problem, and the kernel usually is too. "Usually," because CUB operates at two levels: device-wide algorithms you call from the host, much like Thrust, and block- and warp-level building blocks that live inside kernels you write yourself. The two are not separate products; the device-wide algorithms are themselves constructed from the block-level pieces, a layering we return to in Chapter 4.

Parallel algorithms

: Host-launched algorithms including sorting, scanning and reductions that remove the need to write custom kernels for commonly used operations

Cooperative algorithms

: Gpu-side algorithms such as block-wide or warp-wide reductions or scans that further simplify custom kernel development

Language idiomatic CUDA abstractions

: Fundamental abstractions for CUDA-specific operations including memory allocation, resource management, and specific hardware features

While CCCL is a C++ library, Nvidia also ships official Python bindings (cuda-cccl), which JIT-compile the underlying C++ at runtime rather than wrapping precompiled binaries.

Layer 3: CUDA-X Data Science — GPU-Accelerated Data Science

NVIDIA's CUDA-X Data Science (formerly RAPIDS) is a collection of libraries for accelerating data science workflows on NVIDIA GPUs. The

common design goal is that working with them feels like working in the existing Python data stack, with optimized CUDA underneath. Beyond that, the libraries under the umbrella follow three distinct strategies, and it helps to know which one you're looking at.

A high-level taxonomy of CUDA-X Data Science

1. Clone an incumbent's API. This works when there is a canonical CPU library with a large installed base and NVIDIA wants migration to be as close to free as possible. cuDF (pandas), cuML (scikit-learn) and cuGraph (NetworkX) are the clearest cases.
2. Borrow the concept, build an original API. Used where no single Python API dominates the problem space, so there is nothing to clone. cuVS's native interface is one example.
3. Be the engine inside other people's products. Here the integration point is inside another system, not in a notebook. cuDF powering the Polars GPU engine and the RAPIDS Accelerator for Apache Spark, and cuVS acting as the backend for FAISS and Milvus, are this kind.

Some libraries span more than one bucket, and later chapters will say which. The point of the taxonomy is that it explains the design choices: why one library slavishly matches an existing signature and another invents its own.

The progenitor: CuPy

> **NOTE**
>
> The curious case of CuPy: CuPy is the natural starting point for most of the strategy 1 work, but it is not an NVIDIA project, despite sometimes appearing in NVIDIA marketing and being relied on throughout the stack. A small tell: CuPy is written with a capital C, while NVIDIA's own libraries use a lower-case c.

CuPy is an open-source Python library that reimplements the NumPy and SciPy array interfaces on top of NVIDIA GPUs. Rather than writing its own kernels for everything, it delegates to the CUDA Toolkit's math libraries where one exists for the job: cuBLAS for linear algebra, cuFFT for transforms, cuSPARSE for sparse matrices, cuSOLVER for decompositions, cuRAND for random number generation, cuDNN for neural network primitives, and NCCL for multi-GPU communication. The API compatibility is close enough that for a large fraction of NumPy and SciPy code, porting to the GPU means changing import numpy as np to import cupy as np and scipy to cupyx.scipy, and nothing else.

CuPy was born at Preferred Networks, a Tokyo-based deep learning company, and it wasn't originally a standalone project at all. In 2017 they extracted the array library from their deep learning framework and released it as CuPy v1.0, on the theory that a NumPy-compatible GPU array was useful far beyond that framework. Turns out they were right. Preferred Networks has since stopped developing Chainer, which is mostly remembered for its define-by-run design and its direct influence on PyTorch. CuPy, the piece they extracted, continues to go from strength to strength, and its design has heavily influenced NVIDIA's strategy in this space.

By 2018 CuPy had demonstrated that serious API mimicry of a beloved Python library was a viable product strategy, not just a demo. RAPIDS launched at GTC Europe in October 2018 explicitly marketing "pandas-like," "scikit-learn-like," and "NetworkX-like" APIs, and that framing owes a lot to CuPy having normalized the pattern. NVIDIA people have been open that matching existing PyData APIs was the adoption thesis: data scientists won't rewrite code, so meet them where they are.

Back to CUDA-X Data Science

There are a range of libraries here aimed at different use cases, and we will go through them in much greater detail later in this book. CUDA-X Data Science includes cuDF (a DataFrame library modeled on pandas) for extract, transform, load (ETL) and data processing, cuML for traditional

machine learning workflows (think scikit-learn), cuGraph for graph analytics (modeled on NetworkX), cuSpatial for spatial analytics (GeoPandas) and cuCIM for computer vision (scikit-image).

- cuDF: Chapter 6
- cuML: Chapter 7
- cuGraph: Chapter 8
- cuVS (CUDA Vector Search): Chapter 2

Two further libraries don't fit the "clone of a Python library" mold and won't get their own chapters, but they matter for understanding the rest.

RAFT

RAFT (Reusable Accelerated Functions and Tools) is the shared C++/CUDA primitives layer underneath the RAPIDS family: sparse and dense linear algebra, clustering, distance computation, random sampling and the like, consumed by cuML, cuGraph and others. It is not modeled after any user-facing Python library. Portions of it have since been spun out into their own libraries, most notably the vector search components, which became cuVS.

cuCIM

cuCIM (CUDA Clara IMage) is an accelerated image processing library for multidimensional images, aimed at biomedical, geospatial, materials and remote sensing workloads. Its Python API is a clone of scikit-image, so it belongs in the strategy 1 bucket, though its user base is narrow enough that it won't feature much in this book.

Zero-code-change

A transformative development in recent releases is zero-code-change acceleration. With `cudf.pandas`, any pandas workflow can be GPU-accelerated by adding a single import, with automatic CPU fallback for unsupported operations; NVIDIA claims speedups of up to 150× on suitable

workloads. The `cuml.accel` module does the same for scikit-learn, and the `nx-cugraph` backend accelerates NetworkX. These tools expand the audience from "CUDA kernel authors" to "anyone who uses Python for data."

The barrier to adoption dropped further when RAPIDS libraries became pip-installable directly from PyPI in late 2024, with cuML following in 2025. A `pip install cudf-cu12` is all it takes.

What is PyPI? The Python Package Index (pypi.org) is the official repository for Python packages, and the place `pip install` downloads from by default. Before RAPIDS landed on PyPI, installing GPU-accelerated libraries required conda or Docker containers, which was a significant barrier for many Python developers and data scientists. PyPI availability means RAPIDS can now be installed in any standard Python environment with a single command.

Layer 4: Deep Learning Infrastructure

NVIDIA's rise has had two inflection points. The first was deep learning, which arrived with AlexNet in 2012 and made the GPU the default training hardware. The second was the mass-market adoption of large language models that followed ChatGPT in late 2022, which made the GPU the default inference hardware too. Both have shaped the hardware roadmap and the software that sits on top of it.

NVIDIA's response to the first inflection point was cuDNN, introduced in Layer 1: a closed-source library of tuned primitives for the operations that dominate neural networks, including forward and backward convolution, matrix multiplication, pooling, normalization and, more recently, fused attention. By shipping these primitives, NVIDIA let framework authors get strong GPU support without writing kernels themselves, which is why TensorFlow (2015), PyTorch (2017) and JAX (2018) all launched with NVIDIA GPUs as a first-class target. cuDNN remains closed, however. The open-source part of this layer starts one level down, with the library that lets you write those primitives yourself.

CUTLASS

CUTLASS is NVIDIA's open-source CUDA C++ template library for high-performance matrix multiplication and the operations built on it, including convolution and attention. It is roughly the open counterpart to cuBLAS: a hierarchy of reusable tiles and layout abstractions that let you write Tensor Core kernels reaching cuBLAS-level performance, with the ability to fuse your own work into them. Its history tracks the history of Tensor Cores, from the first Volta-era WMMA kernels in 2017 through CuTe in CUTLASS 3.0 and, most recently, the Python-based CuTe DSL. Today it underpins the attention and GEMM implementations in FlashAttention, TensorRT-LLM and vLLM, among many others. Chapter 10 covers CUTLASS and CuTe DSL in much more depth.

NCCL

The NVIDIA Collective Communications Library is what makes multi-GPU and multi-node work possible. It detects the network topology, both within a node (NVLink, PCIe) and across nodes (InfiniBand, Ethernet), and uses that map to choose efficient algorithms for the standard collective operations: AllReduce, AllGather, ReduceScatter and their relatives. These are the primitives that PyTorch DDP and FSDP, DeepSpeed and Megatron-Core all depend on for training, and that tensor-parallel inference depends on too. Chapter 12 covers NCCL in detail.

TensorRT

TensorRT is NVIDIA's family of tools for high-performance deep learning inference. At its core is a compiler that takes a trained model, typically via ONNX, and produces an engine tuned for one specific GPU SKU: kernel selection, precision, and layer fusion are all decided against the actual target hardware. Around the compiler sit TensorRT Model Optimizer (quantization, pruning, sparsity, distillation and speculative decoding support), TensorRT for RTX (a lighter build for consumer GPUs), and TensorRT-LLM, covered next. TensorRT will be covered in Chapter 13

TensorRT-LLM

TensorRT-LLM extends TensorRT's ideas specifically for large language models, and is NVIDIA's LLM inference engine. It supplies the features that LLM serving now takes for granted: FP8 and FP4 quantization, in-flight batching, paged attention, disaggregated prefill and decode, the full set of parallelism strategies including wide expert parallelism for mixture-of-experts models, and speculative decoding. Chapter 13 covers it.

Triton Inference Server (now Dynamo-Triton)

Note: not to be confused with Triton, OpenAI's GPU kernel language Triton.

Triton Inference Server was for years NVIDIA's general-purpose model server: a single process that loads models from any major framework (TensorRT, PyTorch, ONNX, FIL for tree models, and custom backends), exposes them over HTTP and gRPC using the KServe protocol, and handles dynamic batching, concurrent execution, model ensembles and metrics. It runs on GPUs and CPUs, in the datacenter and at the edge, with C and Java APIs for in-process use. In March 2025 NVIDIA folded it into the Dynamo platform as Dynamo-Triton. It remains open source and actively released, and it is still the right tool for serving a mixed repository of non-LLM models, but new inference investment has moved to Dynamo. Chapter 13 covers both.

NVIDIA Dynamo

Dynamo is Triton's successor: a datacenter-scale inference orchestration framework, open-sourced in March 2025 and reaching 1.0 at GTC 2026. Written in Rust for the performance-critical paths and Python for extensibility, it coordinates TensorRT-LLM, SGLang and vLLM workers into a multi-node system with disaggregated prefill and decode, KV-cache-aware routing, and offloading of KV cache to cheaper memory tiers. NVIDIA claims throughput gains of up to 30× on Blackwell for reasoning models, relative to Hopper. Chapter 13.

NeMo

NVIDIA NeMo began as an end-to-end framework for LLM training and fine-tuning, and its scope has widened over time to cover the full model lifecycle: data curation, pretraining, fine-tuning and alignment, retrieval components for RAG pipelines (NeMo Retriever), safety filtering (NeMo Guardrails), and agent tooling. NVIDIA now positions it as an agent-first suite of libraries for specializing, optimizing and governing AI agents. NeMo is covered in Chapter 15.

Layer 5: The Wider World

Beyond the math, data science and deep learning libraries, NVIDIA maintains hundreds of further open-source projects. There is no space to cover them all, so this section picks out the ones that matter most, and Chapter 15 surveys them in more depth. Two adjacent topics get their own chapters: Chapter 14 covers profiling and debugging (Nsight Systems, Nsight Compute, compute-sanitizer), and Chapter 16 covers the practice of contributing to the ecosystem.

GPU-accelerated data pipelines

As models got faster, the CPU-side work of loading, decoding and augmenting training data became the bottleneck. Two libraries move that work onto the GPU.

DALI

DALI (Data Loading Library) is a GPU-resident input pipeline for images, video and audio. You describe the pipeline in Python (read from disk with sharding and shuffling, decode, crop, resize, augment) and DALI executes it on the GPU and hands tensors directly to PyTorch, JAX or TensorFlow through framework iterators. Chapter 11.

CV-CUDA

CV-CUDA is the complementary piece for inference: a stateless library of computer vision kernels (resize, warp, normalize, colour conversion, morphology, non-maximum suppression and the like) with C, C++ and

Python bindings and zero-copy interop with the frameworks. It targets the pre- and post-processing around a vision model in a cloud service, which is often where the latency and cost actually go. Chapter 11.

Recommender systems

Merlin

Merlin is NVIDIA's umbrella for recommender systems: a set of libraries covering preprocessing and feature engineering at the hundreds-of-terabytes scale these systems run at (NVTabular), training (Merlin Models, HugeCTR for very large embedding tables), and serving. Recommenders are a distinctive workload, dominated by sparse embedding lookups rather than dense compute, and Merlin exists because the general deep learning stack handles them badly. Chapter 15.

Simulation and numerical computing

Warp

Warp is a Python framework for writing simulation and graphics kernels. You write ordinary Python functions with type annotations and Warp JIT-compiles them to CUDA (or CPU code), so you get kernel-level performance without leaving Python. The kernels are differentiable, which makes Warp useful inside PyTorch and JAX training loops, and it ships with primitives for physics, geometry and robotics. Chapter 15.

MatX

MatX is the C++ analogue: a header-only C++20 library that lets you write NumPy- or MATLAB-style array expressions and compiles them into fused GPU kernels via expression templates. It exposes one tensor type across its whole API and wraps the standard CUDA math libraries (and CPU equivalents such as FFTW and OpenBLAS) behind it. Chapter 15.

Robotics

Isaac

Isaac is NVIDIA's robotics platform, a collection of projects rather than a single library: Isaac Sim for simulation, Isaac Lab for robot learning, Isaac ROS for GPU-accelerated ROS 2 nodes, teleoperation tooling, and the GR00T foundation models for humanoids. Chapter 15 covers which parts are open, which are not, and how they fit together.

Operations research

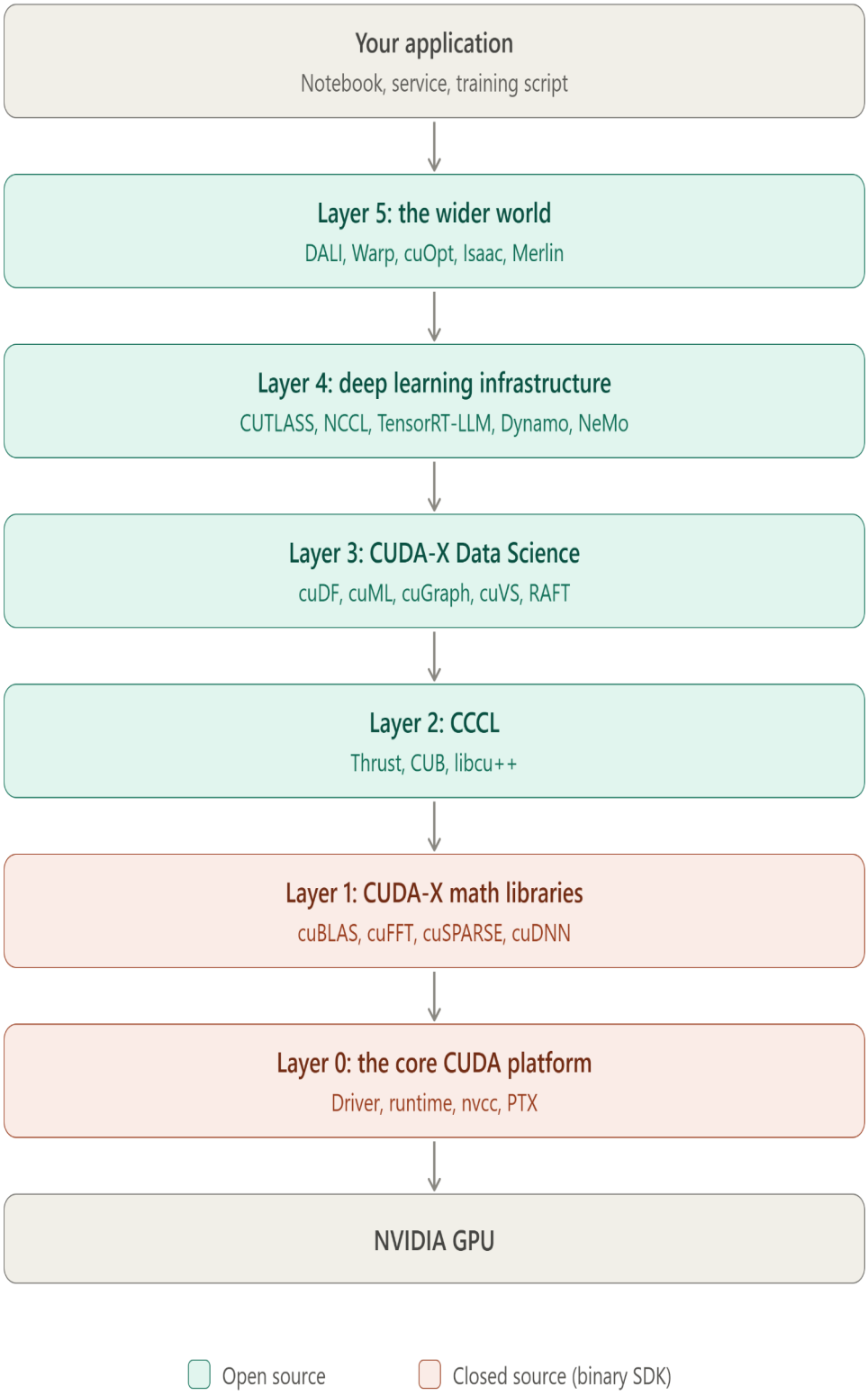
cuOpt

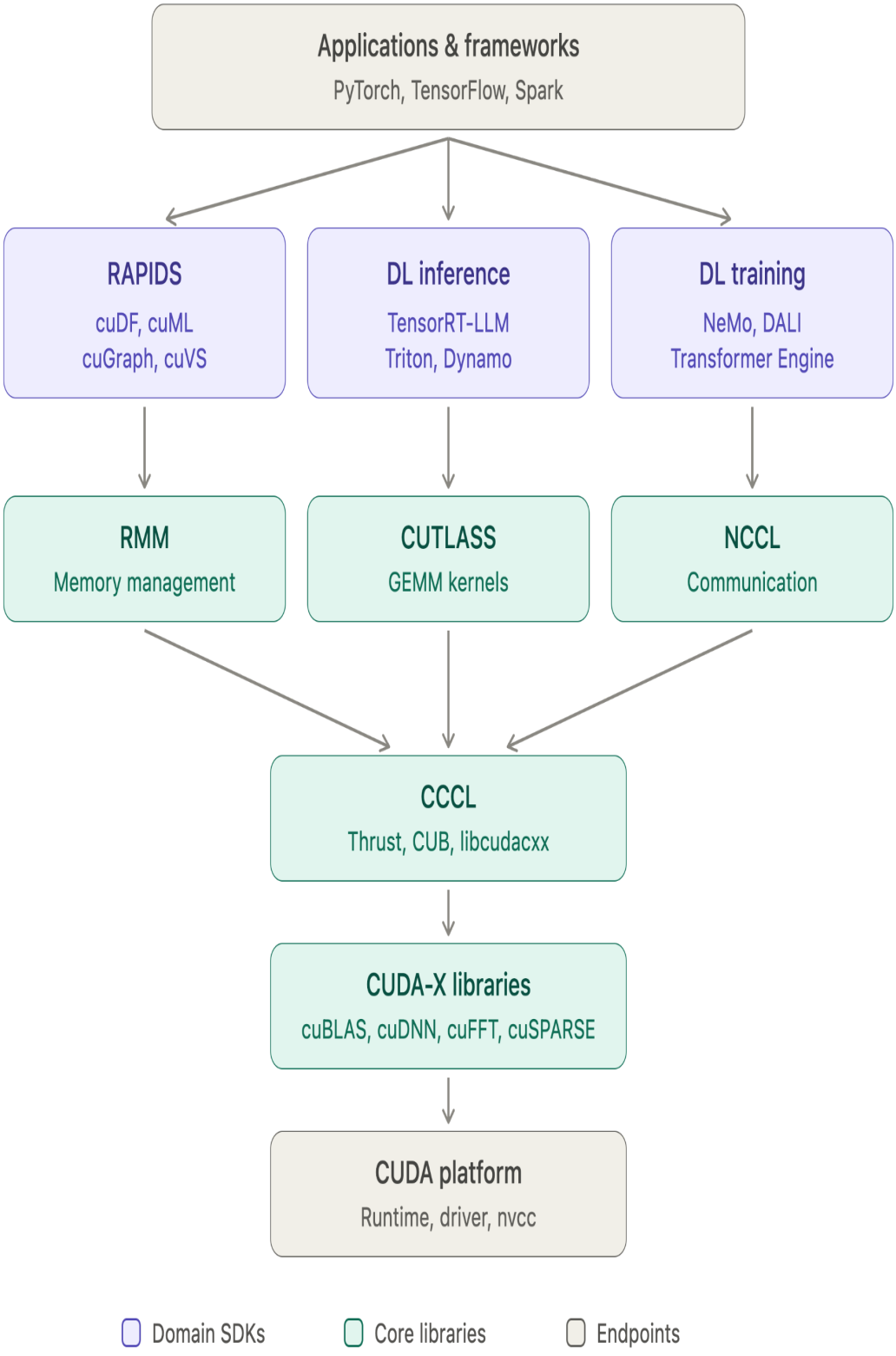
cuOpt is NVIDIA's GPU-accelerated solver for optimization problems, open-sourced in 2025. It handles linear, quadratic and mixed-integer programming and vehicle routing at scales of millions of variables and constraints, and the case for the GPU is the same as elsewhere in this book: the inner loops of the solvers parallelize well. NVIDIA has also begun packaging cuOpt as a skill that AI agents can call to formulate and solve optimization problems. Chapter 15.

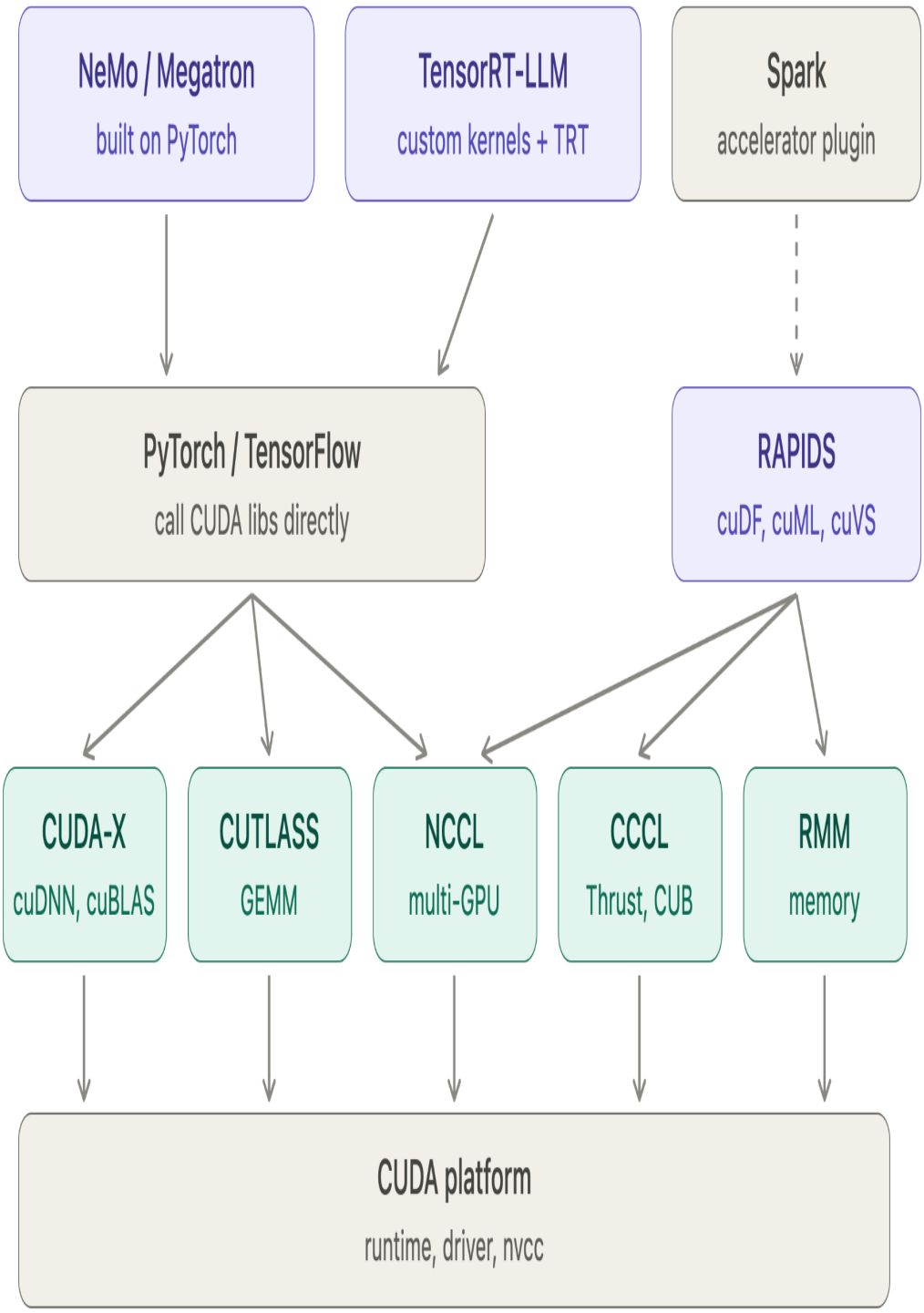
And much, much, more

As mentioned above Nvidia has over 1000 open source repositories so may well have acceleration for your specific domain. The vast majority live in what I called Layer 5 and in turn depend heavily on the lower layers

A Visual Dependency Map


Your application
Notebook, service, training script
Layer 5: the wider world
DALI, Warp, cuOpt, Isaac, Merlin
Layer 4: deep learning infrastructure
CUTLASS, NCCL, TensorRT-LLM, Dynamo, NeMo
Layer 3: CUDA-X Data Science
cuDF, cuML, cuGraph, cuVS, RAFT
Layer 2: CCCL
Thrust, CUB, libcu++
Layer 1: CUDA-X math libraries
cuBLAS, cuFFT, cuSPARSE, cuDNN
Layer 0: the core CUDA platform
Driver, runtime, nvcc, PTX
NVIDIA GPU
Open source
Closed source (binary SDK)


Applications & frameworks
PyTorch, TensorFlow, Spark
RAPIDS
cuDF, cuML
cuGraph, cuVS
DL inference
TensorRT-LLM
Triton, Dynamo
DL training
NeMo, DALI
Transformer Engine
RMM
Memory management
CUTLASS
GEMM kernels
NCCL
Communication
CCCL
Thrust, CUB, libcudacxx
CUDA-X libraries
cuBLAS, cuDNN, cuFFT, cuSPARSE
CUDA platform
Runtime, driver, nvcc
Domain SDKs
Core libraries
Endpoints


NeMo / Megatron
built on PyTorch
TensorRT-LLM
custom kernels + TRT
Spark
accelerator plugin
PyTorch / TensorFlow
call CUDA libs directly
RAPIDS
cuDF, cuML, cuVS
CUDA-X
cuDNN, cuBLAS
CUTLASS
GEMM
NCCL
multi-GPU
CCCL
Thrust, CUB
RMM
memory
CUDA platform
runtime, driver, nvcc

The State of the Ecosystem in 2026

Thriving! As mentioned above in February of 2026 Nvidia took the crown as the number one company on Hugging Face. On August 27th Nvidia agreed to purchase HuggingFace for $12.9 Billion dollars. They are actively pushing new research, new models, and even new GitHub repos. So much so this book can't possibly cover them all. There is a constant stream of new hardware, new software, and new ideas.

If you are interested in a particular domain (AI, Robotics, Autonomous Vehicles, etc.), follow the social media profiles of Nvidia for that domain or sign up for their newsletters to keep track of new releases and news.

As you have hopefully seen above, we have a lot to cover so we had best get started!

Chapter 2. cuVS — Search for the Era of Unstructured Data

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form — the author's raw and unedited content as they write — so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at shunter@oreilly.com.

Most of the data produced today is unstructured: text, images, audio, video, code. The way modern systems make this data searchable is to embed it, representing each item as a dense vector such that similar items land near each other. Once everything is a vector, the core operation becomes nearest-neighbour search: given a query vector, find the most similar vectors in a collection that may hold billions of them. Over the past decade this "vector search" has become a foundational primitive, underpinning search engines, recommender systems, retrieval-augmented generation, deduplication, anomaly detection, and more.

cuVS (pronounced "coo-V-S") is NVIDIA's library for doing this on GPUs. It provides GPU-accelerated implementations of vector search, both exact and approximate, along with the supporting operations those workloads need: clustering, pairwise distance computation, and quantization. Clustering earns its place in that list twice over. Beyond its standalone uses in data analysis, k-means is how the inverted-file (IVF) indexes we meet

later in this chapter partition their vectors, and index construction is where GPUs deliver their largest speedups over CPUs.

Chapter 1 grouped the CUDA-X Data Science libraries into three strategies: clone an incumbent's API, borrow the concept but build an original API, or be the engine inside other people's products. cuVS is mostly an example of the third. FAISS (Facebook AI Similarity Search) is the obvious incumbent, and cuVS was inspired by it, but NVIDIA chose not to mirror the FAISS API. Instead cuVS has its own native API, callable from C, C++, Python, Rust, Java, and Go, and is designed to be embedded as a GPU backend inside other systems: Meta's FAISS, Milvus, Elasticsearch, OpenSearch, Apache Lucene, and NVIDIA's own cuML library (Chapter 7). The consequence is that many cuVS users never have to import it directly. They see faster index builds in a vector database they already run, or a faster UMAP and HDBSCAN when using cuML, where the nearest-neighbour graphs cuVS builds do the lion's share of the computation. Those same graphs can also be handed on to cuGraph (Chapter 8) for community detection.

Two Properties of Embeddings

An embedding model is a neural network trained to map each input, whether text, image, audio, or code, to a point in a space of fixed dimension, typically 768 to 4,096 for current models, such that proximity in that space encodes similarity of meaning. Two properties of these models matter for the rest of the chapter.

The first is what the model was trained to bring together. In the symmetric case it puts similar things together, a sentence with its paraphrase or an image with its near-duplicate, which is what deduplication, clustering, and "more like this" recommendations rely on; Sentence-BERT is an example. In the asymmetric case it puts different but related things together, queries with the documents that answer them or captions with their images, which is what search and retrieval-augmented generation rely on; Dense Passage Retrieval and CLIP are examples. The distinction matters beyond training.

In the asymmetric case the queries come from a different distribution than the vectors in the index, and we will see in the tuning section that this affects how an index should be built and searched.

The second is dimensionality, which multiplies the memory of every vector and the work of every distance computation, and is ordinarily fixed when the model is trained. Matryoshka representation learning (MRL), introduced by Kusupati and colleagues in 2022, loosens that constraint. The model is trained so that the first 64 dimensions of each embedding form a usable embedding on their own, as do the first 128, the first 256, and so on, like the nested dolls the technique is named after. Such an embedding can simply be truncated to the size a system can afford, at a modest and predictable cost in quality, and several widely used models and APIs now expose this as a dimension parameter. We will see MRL again as one of the cheapest ways to shrink an index.

The consequence is that retrieval now has a single bottleneck shape: find the nearest neighbours of a query among billions of candidates in a space of hundreds or thousands of dimensions. Why that shape suits GPUs is the subject of a later section, but first we need to state the problem precisely and see why it is harder than it looks.

The Problem of Vector Search

Stated formally, the problem is simple. Given a query vector of dimension d and a set V of vectors of the same dimension, find the k vectors in V that are closest to the query. "Closest" is defined by a metric, usually Euclidean (L2) distance, inner product, or cosine similarity, where closest means smallest for a distance and largest for a similarity. For vectors normalised to unit length (that is, with an L2 norm of 1), all three produce the same ranking, since the squared Euclidean distance between two unit vectors is just 2 minus twice their inner product. Most embedding pipelines normalise for exactly this reason, and unless specified otherwise this chapter assumes they have been, so that comparing two vectors means taking their inner product. The equivalence holds only between unit vectors, though. The

centroids and residuals that indexes compute internally are not unit vectors, which is one reason every index in cuVS is built for a specific choice of metric.

The direct approach is to compute the distance from the query vector to every vector in V and select the top k. That costs O(d|V|) per query, linear in the size of the collection, and at scale it is limited by memory bandwidth as much as by arithmetic, because every vector must be read for every query. A billion 1,024-dimensional vectors in 32-bit floats is about 4 TB, all of which has to be read to answer a single query. As |V| grows from millions to billions, serving real-time queries this way needs a great deal of computational power and hardware.

The alternative is approximate nearest neighbour (ANN) search. ANN algorithms trade a small amount of accuracy for a query cost that grows far more slowly than |V|. They do this by building an index: a data structure over V, whether a set of partitions, a graph, or a tree, that lets each query examine only a promising fraction of the collection. The price is that the k results returned may not be exactly the k closest.

The loss is measured as recall: the fraction of the true k nearest neighbours, as found by an exact search, that the approximate search returns. A recall of 0.9 at k = 10, written recall@10 = 0.9, means that on average over an evaluation set 9 of the 10 true neighbours were found, and most production systems target somewhere between 0.9 and 0.99. (The theoretical literature works instead with the (1+ε)-approximate nearest neighbour, any vector within a factor of (1+ε) of the true nearest distance, a definition we will meet again in the next section.)

NOTE

Multi-vector retrieval, begun by ColBERT, represents queries and documents as sets of vectors and scores them by "late interaction", summing over each query vector its maximum similarity to any document vector. It typically improves retrieval quality at the cost of much larger storage and compute. cuVS does not support it at the time of writing, so this chapter focuses on the single-vector case.

When Is "Nearest Neighbour" Meaningful?

An index can only help if a promising fraction exists to be found, and whether it exists depends on the data.

In 1999 Beyer et al. asked a blunt question: when is the concept of a "nearest neighbour" meaningful? Their answer was that under surprisingly broad conditions, as dimensionality grows, the ratio between the distance to a query's farthest neighbour and the distance to its nearest converges to one. Every point becomes roughly equidistant, the nearest neighbour is nearest by a negligible margin, and a small perturbation of the query returns a different answer. They called such queries "unstable". Notice what this does to the formal definition above: when most of the collection lies within a factor of $(1+\varepsilon)$ of the nearest distance, almost any vector is a valid $(1+\varepsilon)$-approximate answer, and the guarantee says nothing.

They showed empirically that the effect can set in with as few as 10 to 15 dimensions: for uniformly random data at 20 dimensions, the farthest point is only about four times as far as the nearest. They also observed that in this regime the index structures being published at the time were consistently beaten by a simple linear scan, the brute-force approach, which few of those papers had bothered to compare against.

The embeddings we use today are in the range of 768 to 4,096 dimensions, so this should be alarming. The reason it isn't is the escape hatch the same paper identified: their result requires that the data genuinely fill the space. Data with low intrinsic dimensionality, points lying on or near a lower-dimensional manifold embedded within the high-dimensional space, keep their nearest neighbours well separated no matter how many nominal dimensions are added. Embedding models are trained precisely to produce such structure, and real-world datasets, images, text, code, are nowhere close to uniformly random.

Both halves of the story depend on this. Nearest neighbour is a meaningful, well defined concept because of the manifold, and ANN indexes work because that manifold gives them something to exploit. Where the structure is weak, recall and meaning degrade together, and brute force remains the

honest baseline. That is why cuVS ships an exact search alongside its approximate ones, and why the algorithms section covers it first.

There is far more to say about the theory of vector search, and whole books have been written on it. Sebastian Bruch's *Foundations of Vector Retrieval* is the one to reach for, and it is freely available online.

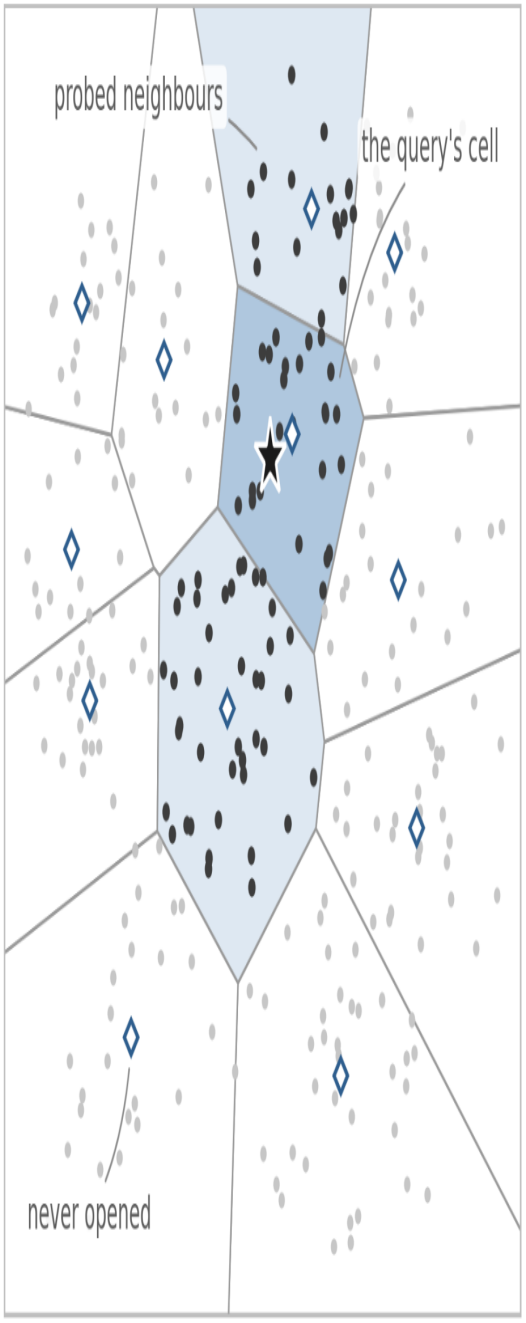
How an Index Skips Most of the Data

The simplest way to see how an index allows you to skip most of the data is to imagine dividing the space into regions. If the query lands in one region, its nearest neighbours are almost certainly within that region or its immediate neighbours, and every other region can be ignored. The quality of the search then depends entirely on how well the regions reflect the actual structure of the data. This is the partition family, and in cuVS it is represented by the IVF indexes.

The other main family does not partition at all. A graph index connects every vector to a few dozen of its near neighbours, and a query is answered by walking: start somewhere, look at the neighbours of the current best candidates, move towards whichever are closest to the query, and stop when nothing closer turns up. The walk only ever touches the vectors along its path. HNSW on CPUs and CAGRA in cuVS are both of this kind. The same dependence on structure applies, because the walk converges only if a near neighbour of a near neighbour itself tends to be near, which is the manifold assumption in another form. Trees and hashing schemes exist too (see Bruch for more), but at the dimensionality of modern embeddings these two families dominate, and they are the two cuVS implements.

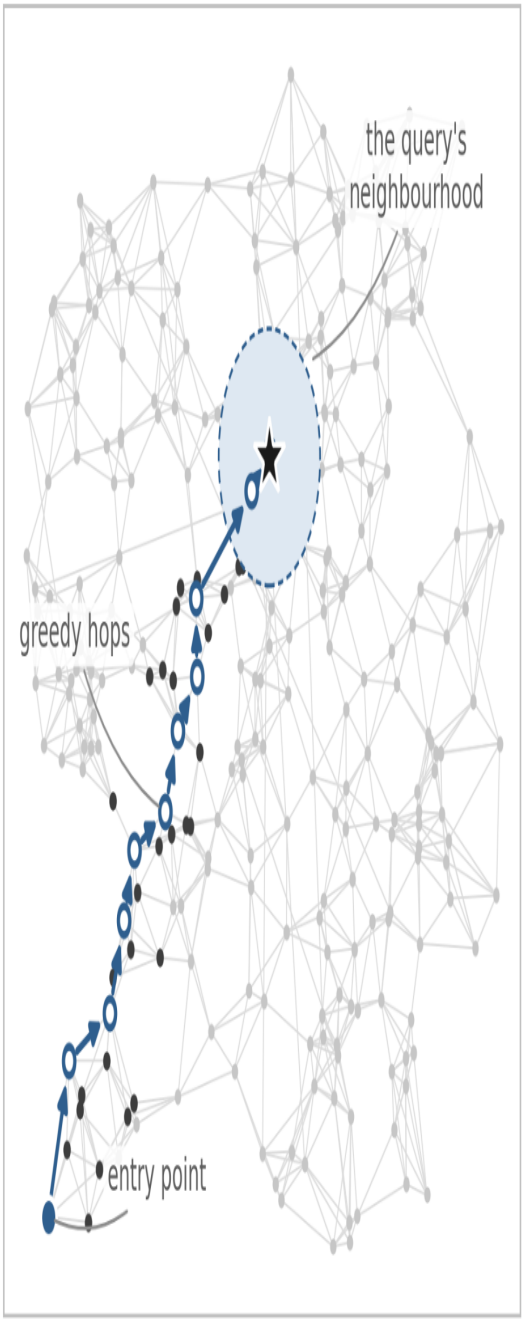
Partitioned index

probe 3 of 12 cells, scan all 85 of 287 vectors inside



Graph index

10 hops downhill, 48 of 287 vectors touched



query · vector examined · vector skipped · cell centroid · entry point

Figure 2-1. Two ways to skip most of the data. Left: a partitioned index, with the query's region and its probed neighbours shaded. Right: a graph index, with the path of a greedy walk from an entry point to the query's neighbourhood.

Drawing the regions for the first family is a clustering problem. Less obviously, the second family also leans on clustering, since one of the standard ways to build a search graph on a GPU begins life as a clustered index. Clustering is also a workload in its own right, and it turns out to be the same computation as search viewed from the other side. So before the indexes, let's talk a little more about clustering.

The Clustering Problem

Clustering is the task of partitioning a set of unlabelled points into groups such that points within a group are more alike than points across groups. It is a family of problems more than a single one, since each notion of a good group (compact balls around centroids, dense regions, connected components of a graph) defines a different objective with its own algorithms, and cuML in Chapter 7 covers that landscape. What matters here is narrower: what clustering computes, and why a vector search library ends up containing it.

Start with k-means, which chooses k centroids so as to minimize the sum of squared distances from each point to its nearest centroid. (This k is the number of clusters, and has nothing to do with the k of top-k search). Lloyd's algorithm, which is what almost everyone means by "k-means", repeats two steps until convergence. In the assignment step, every point is assigned to its nearest centroid. In the update step, each centroid is replaced with the mean of the points assigned to it. Look closely at the assignment step and it is exactly the brute-force nearest-neighbour search of the previous section, with the k centroids forming the corpus and the n points as queries. The same distance-and-argmin kernel that answers one query against a billion vectors answers a billion queries against a thousand centroids.

Put numbers on it. A billion 1,024-dimensional embeddings clustered into 100,000 groups is 10^{17} multiply-adds per Lloyd iteration, and convergence usually takes ten or twenty iterations. Nobody runs that twenty times. In practice the centroids are trained on a sample, often a few hundred thousand to a few million points, and the full dataset is then assigned to them in a single pass. That pass costs one full iteration, it is again a brute-force search, and it is the step that dominates build time for the IVF indexes below.

Those indexes also need something plain k-means does not provide. It makes no promises about cluster sizes, and on real embeddings a few centroids often end up owning a large share of the points. Every query that lands in an oversized partition scans far more vectors than one that lands in a small one, and on a GPU, where work is dispatched in uniform batches, a handful of giant lists throttles the whole search. Balanced k-means accepts slightly larger distances between points and their centroids in exchange for partitions of roughly equal size, and it is what cuVS's IVF indexes train with by default.

K-means is the cheap case. Most other clustering algorithms start from all-pairs information: agglomerative clustering from the full $n \times n$ distance matrix, DBSCAN from every point's neighbours within a radius. For a million points that matrix has 10^{12} entries, which is not a data structure you can build. The modern stack (UMAP followed by HDBSCAN, spectral methods, graph community detection) works instead from a k-nearest-neighbour graph, which is both affordable and more robust in high dimensions, where relative rank survives better than absolute distance. Building that graph is an approximate nearest-neighbour problem.

So the two problems use each other. An IVF index is a k-means partition of the corpus, with the centroids used to prune the search. Clustering at scale begins by building an approximate kNN graph, using the same machinery as ANN search. They share the same kernels, pairwise distance, fused distance-and-top-k, batched selection, and a library that does vector search well already contains the bones of a clustering library. That is the pattern the rest of this chapter follows: a handful of primitives at the bottom, and

everything above them, the indexes, the graph builders, and the clustering algorithms that cuML draws on, assembled by composing them in different orders. Figure 2-5, once all the pieces have been introduced, shows the dependencies.

Search and clustering, then, come down to the same inner loop. The next section is about why that loop belongs on a GPU.

Why GPUs?

Both the vector search problem and the clustering problem grow with |V|. Exact search reads every vector for every query; clustering reads every vector on every iteration; and the approximate methods that avoid this at query time pay for it in advance, at build time instead. GPUs are this chapter's answer to that growth, and the reason is not just that they have more arithmetic. It is that the shared inner loop, compute a great many distances between vectors and then reduce them, either to a top-k or to an argmin over centroids, has three properties that make it close to ideal as a GPU workload.

The work is independent.

> A brute-force search over a batch of queries is a matrix multiply, queries times the transpose of the corpus, followed by a per-row top-k. Nothing about query i depends on query j. K-means assignment is the same shape with centroids in place of the corpus. Graph construction, as we will see, processes every point's candidate list in parallel and synchronises only at the end of each round. Where a CPU has to serialise or thread carefully, the GPU offers tens of thousands of independent lanes for free. And because the arithmetic is a general matrix multiply (GEMM), the most heavily optimised kernel on the device, Tensor Cores apply whenever the vectors are stored in supported precisions (see Chapter 2). The pairwise distance kernels in cuVS are written as a matrix multiply with an epilogue that finishes the distance and starts the reduction in the same pass, which is why they run within a modest factor of peak GEMM throughput.

The data is bandwidth-bound, and GPUs have the bandwidth.

Beyond the dense arithmetic, these algorithms spend the majority of their time streaming vectors through memory: scanning the lists of a partitioned index, gathering neighbours' neighbours during a graph walk, reading compressed codes. These are memory-bound operations, and the GPU's advantage shows at every tier: a consumer card like the RTX 5090 offers about 1.8 TB/s against roughly 100 GB/s for a desktop CPU, and a data-centre GPU delivers several terabytes per second against a few hundred gigabytes per second per server CPU socket.. A large fraction of the GPU speedup on vector search is simply that ratio.

Batching turns latency into throughput.

A single query against a graph index is a sequential walk and parallelises poorly on its own. On a CPU with a warm cache it takes a few milliseconds, and a naive port of that walk to a GPU gains little. The GPU wins outright when there are many queries at once, because it can run thousands of walks concurrently and amortise memory latency across them. Serving systems that batch requests, and offline workloads that have all their queries in advance, get the full benefit. The single-query case is not hopeless either: CAGRA's search is designed around exactly this problem, spreading one query's candidate expansion across many GPU threads, and we will see in the CAGRA section what that enables.

Those three properties explain why the GPU is fast. A fourth explains where it actually gets used.

The Expensive Part Is Offline.

This is the strongest part of the argument in practice. Query-time HNSW search on a CPU is already fast enough for most applications; the bottleneck is in building the index. Constructing an HNSW graph over a hundred million vectors takes hours on a large CPU box, and training a k-means model with a hundred thousand centroids is a similar order of work. Every such job is a throughput problem with no latency constraint, which is exactly the shape the GPU is best at, and the reported speedups on index build run to an order of magnitude or more. That changes what an index is. An index that takes a night to build is rebuilt weekly and serves increasingly stale results in between; one that takes minutes to build can be rebuilt continuously, and freshness stops being an architectural compromise. The same applies to clustering, which is offline almost by definition: a UMAP-plus-HDBSCAN pass over a million embeddings that took an hour becomes an interactive step.

From RAFT to cuVS

cuVS began life as a part of RAFT (Reusable Accelerated Functions and Tools), the RAPIDS library of CUDA primitives. RAFT had itself been carved out of cuML around 2020 so that cuML (Chapter 7), cuGraph (Chapter 8), and the other RAPIDS projects could share one implementation of the routines they all needed: a resources handle, device array views, linear algebra, pairwise distances, nearest-neighbour search, and clustering.

By 2023 the nearest-neighbour corner of RAFT had become the most active part of the library, and the CAGRA paper that year made it the part that external projects wanted. FAISS, Milvus, and Apache Lucene all had reason to build on it, and none of them had any interest in the rest of RAFT. They wanted a library with a stable C ABI, a small binary, its own release cadence, and bindings in languages other than Python. RAFT, a large and mostly header-only C++ template library with a thin Python layer, was the wrong vehicle for that.

So RAFT was split in two. The cuVS repository was created in early 2024 and shipped its first release in the 24.04 cycle. RAFT kept the primitives, the resources handle, array views, linear algebra, and random number generation, and cuVS took everything built on a distance between vectors. The code carried over with new namespaces, raft::neighbors becoming cuvs::neighbors, and only minor API changes; the old copies lingered in RAFT's headers until RAFT 26.02, so code still written against the RAFT namespaces needs porting. cuVS still builds on RAFT, and its C++ APIs still take raft::resources and raft::device_matrix_view. RAFT became the foundation and stopped being the product.

Architecture and Language Bindings

cuVS has three layers, and the split between them is what gave the external projects the stable ABI and bindings that RAFT could not.

At the bottom is the C++ core, libcuvs, where every algorithm is implemented. It sits on RAFT for its handle and array types and on RMM for device memory management. Each algorithm lives in its own namespace and exposes stateless free functions that take a resources handle, a parameter struct, and non-owning views of the caller's data. The handle carries the CUDA stream, the cuBLAS and other library handles, and the memory resources. Nothing global hides inside the library, and two threads holding two handles do not interfere with each other.

The index types also share a vocabulary. Each has a set of index parameters and a set of search parameters, a build function that turns a dataset into an index, a search function that answers a whole batch of queries at once, and serialize and deserialize functions for persistence. Several indexes can also be extended with new vectors after they are built. Learn the shape once with any index and the rest read the same way; only the parameters differ, and those are the subject of most of the rest of this chapter.

The middle layer is what the projects that wanted to repurpose RAFT's vector search were missing. A C++ template library is a poor thing to bind to from another language: templates have no ABI until they are instantiated,

errors arrive as exceptions that cannot safely cross a language boundary, and the arguments are types no other language can construct. The C API, libcuvs_c, answers each of those problems. The templates are instantiated inside the library and hidden behind opaque handles, every function returns a status code instead of throwing exceptions, and arrays cross the boundary as DLPack tensors. DLPack is a small, widely adopted C struct that describes a tensor someone else owns: a pointer, a device, a dtype, a shape. Because it is only a description, nothing is copied to make a call, and because it names the device, cuVS can tell whether the caller's vectors are in host or GPU memory. PyTorch, CuPy, JAX, and most other array libraries can produce one.

With that layer in place, a language binding is mostly mechanical, and none of them contain algorithms. Python wraps the C API with Cython and, as elsewhere in the NVIDIA ecosystem, is the first-class citizen with the broadest coverage. The Rust bindings are generated from the C headers with a safe crate on top. The Java API uses Panama, the foreign function interface that arrived in JDK 22, and the Go module uses cgo. The choice of languages follows from the strategy described at the start of this chapter. To be the engine inside other people's products, the engine has to be callable from the languages those products are written in, and search infrastructure is written in Java (Lucene and everything built on it), Go, Rust, and C++. Coverage outside Python and C++ is uneven, so check the API reference for your language before designing around a particular index type. The Go bindings, for example, currently accept only float32 datasets as of the time of writing.

The C layer is also where compatibility is promised: it is documented as the library's ABI-stable boundary, which is what lets a database vendor ship against cuVS without rebuilding for every release.

The namespaces are the map of the library, and they are the same in every language. The neighbors namespace holds the indexes, the graph builders, and the filtering and refinement machinery around them. The cluster namespace holds k-means, its balanced variant, single-linkage, and spectral clustering. Around those two sit the supporting pieces: distance for pairwise

metrics, preprocessing for the quantisers, PCA, and spectral embedding, selection for batched top-k, and stats for the silhouette and trustworthiness scores. Nearly everything in the supporting namespaces exists to serve the first two, and the rest of this chapter is mostly about the first, taken in order of increasing sophistication.

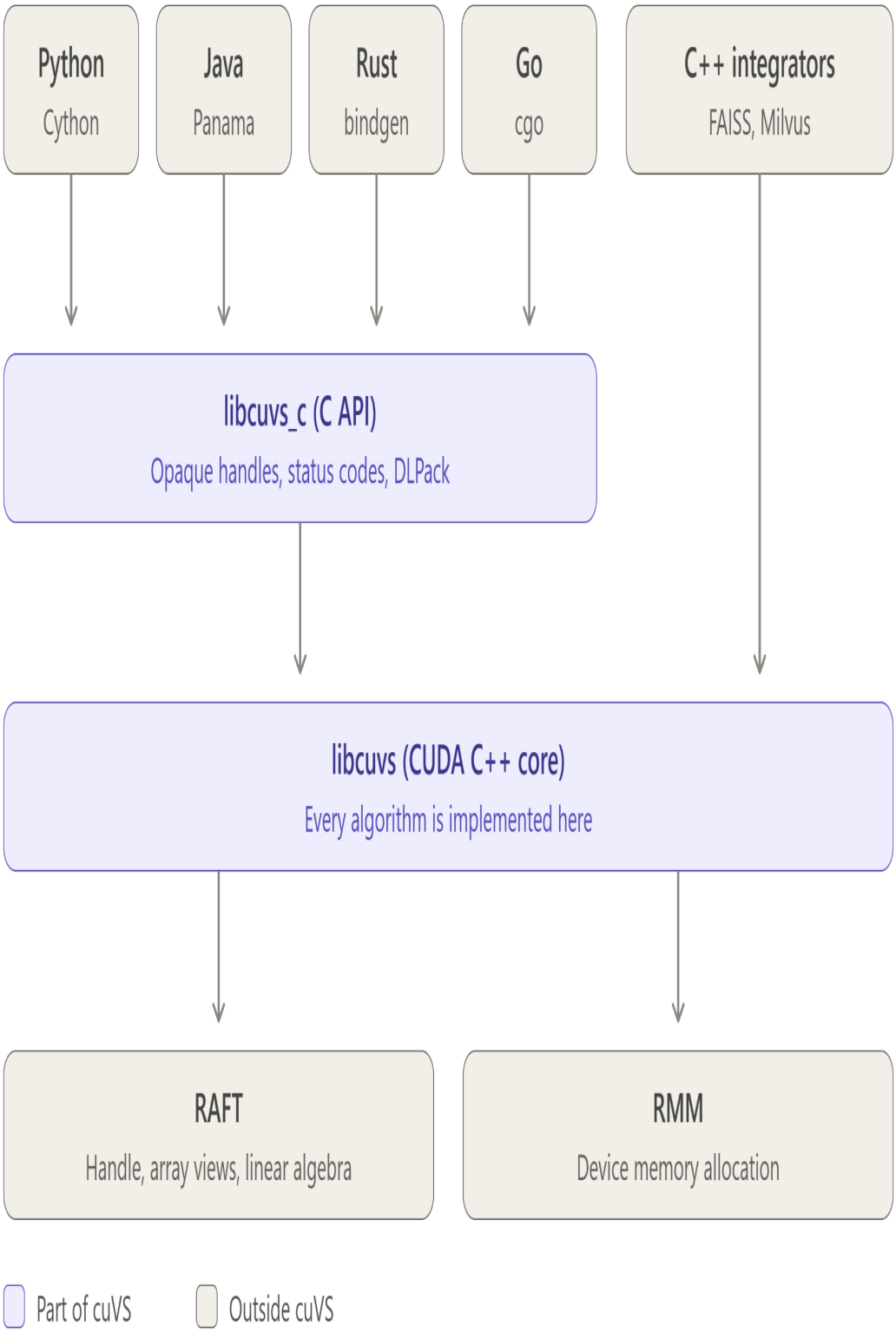

Python
Cython
Java
Panama
Rust
bindgen
Go
cgo
C++ integrators
FAISS, Milvus
libcuvs_c (C API)
Opaque handles, status codes, DLPack
libcuvs (CUDA C++ core)
Every algorithm is implemented here
RAFT
Handle, array views, linear algebra
RMM
Device memory allocation
Part of cuVS
Outside cuVS

Figure 2-2. The layers of cuVS. Bottom: RMM and RAFT. Middle: the libcuvs C++ core, then the libcuvs_c C API with DLPack at its boundary. Top: the Python, Java, Rust, and Go bindings, and beside them the C++ integrators (FAISS, Milvus) that link against the core directly.

The Algorithms

The indexes in cuVS are presented here in order of increasing sophistication, each one being motivated by the limits of the previous. Brute force comes first, because it is exact and because it is the one everything else is measured against. The IVF family follows: IVF-Flat, which partitions the data so a query can skip most of it, and IVF-PQ, which also compresses what remains. Then come the graph indexes: what makes a graph searchable, how an approximate neighbour graph is built in the first place, and CAGRA, which puts the two together. A short tour of the rest of the catalogue, the clustering and preprocessing APIs, and a guide to choosing between all of these close the section. Every index follows the build-and-search shape described in the previous section, so what follows concentrates on how each one works, what it costs, and which parameters matter.

Brute Force: The Exact Baseline

Brute-force search does exactly what it says on the tin. For every query vector, compute the distance to every vector in V and return the k closest. There is no approximation and no recall to measure, which gives brute force its first job: producing the ground truth that every approximate index in this chapter is scored against.

The computation has two stages, a distance stage and a selection stage, and the distance stage is a matrix multiply: the inner products of every query with every corpus vector are one product of two matrices. For data that is not normalised, squared Euclidean distance expands to $\|q\|^2 + \|x\|^2 - 2q \cdot x$, the same multiply plus a correction per row and per column. The corpus norms never change, so they are computed once on index build, and that is essentially all that building a brute-force index does.

The one complication is size. Ten thousand queries against ten million vectors is a distance matrix of 10^{11} entries, about 400 GB in 32-bit floats, and it cannot be materialised. cuVS computes it in tiles, a block of queries against a block of the corpus at a time, selects the best k from each tile, and merges the partial results as it goes. Only the running top-k per query is persisted.

Selection is the less obvious half. A CPU heap scans each row sequentially with data-dependent branches, which suits a GPU badly, while a full sort wastes work when k is 10 and the row has ten million entries. cuVS offers two GPU approaches behind one select-k routine, choosing per call. Sorting-network methods, descended from WarpSelect in the 2017 FAISS GPU paper, are fastest for small k but top out around a thousand. Radix methods narrow in on the k-th value a few bits at a time and cost roughly the same for any k. Every index in this chapter ends with this step.

Beyond ground truth, there are three situations in which brute force is the right production index. The first is a corpus of modest size. A modern GPU can search tens of millions of vectors exactly with sub-second latency, and in that range the recall lost to an approximate index buys very little. The second is a corpus that changes constantly. With no index to maintain, adding or removing vectors costs nothing beyond updating the norms, and there is never a rebuild to schedule. The third is heavily filtered search. When a filter leaves only a small fraction of the corpus eligible, scanning the survivors exactly is both faster and more reliable than asking an approximate index to find them, a point we return to in the section on filtered search below.

The limit is memory, not speed. Ten million 1,024-dimensional vectors occupy about 41 GB in 32-bit floats, which is most of a data-centre GPU, and the cost of every query still grows linearly with the corpus. Storing the vectors in half precision doubles the ceiling. Beyond that the corpus shards across GPUs, which for an exact search is painless, since merging per-shard top-k lists gives exactly the single-scan answer. But a linear cost per query eventually stops being affordable, however it is divided, and the way out is to stop looking at most of the data.

IVF-Flat: When You Need High Recall on a Budget

IVF-Flat is the partition idea we discussed earlier made concrete. IVF stands for inverted file, a term borrowed from text retrieval, where an inverted index maps each word to the list of documents containing it. Here the role of the word is played by a cluster centroid, and each centroid maps to the list of vectors that are closer to it than to any other. "Flat" means those vectors are stored as they are, uncompressed, which is the difference between this index and the one in the next section.

Building the index is the clustering problem we have already met. cuVS trains a balanced k-means model, usually on a sample of the data, then assigns every vector in the corpus to its nearest centroid and stores each resulting list contiguously in GPU memory. The number of lists is the main build parameter. As the clustering section noted, the training is cheap because it runs on a sample, and the single assignment pass over the full corpus dominates the build time.

Search happens in two steps, and both are operations we have already seen. The coarse step compares the query with every centroid and selects the nearest few, the number being the search parameter known as the number of probes. The fine step scans the lists belonging to those centroids, computing exact distances to every vector in them, and selects the best k. Each step is a brute-force search followed by a top-k selection. The coarse step runs it over the centroids, and the fine step runs it over a small slice of the corpus.

The savings here are easy to quantify. Take a hundred million vectors, divide them into 10,000 lists of about 10,000 vectors each. A query that probes 100 lists computes 10,000 distances in the coarse step and a million in the fine step, which is about 1% of the work of a full scan. More generally, with L lists and p probes over a corpus of size $|V|$, the cost per query is proportional to $L + p|V|/L$. The first term grows with the number of lists and the second shrinks, which is the reasoning behind the common rule of thumb of setting the number of lists near the square root of the corpus size. The tuning section gives more practical guidance, but the shape of the

trade is already visible: more lists make each one cheaper to scan and the coarse step more expensive.

What is given up for this saving is the guarantee. A query that lands near a cell boundary may have true neighbours on the other side, in a list that was never probed, and those neighbours are simply missed. Recall is therefore governed almost entirely by the number of probes. Probing a handful of lists is fast and lossy, probing more recovers the neighbours near the boundaries, and probing every list is brute force again with some overhead. The property that gives this section its title is that everything inside the probed lists is scored exactly. The only error IVF-Flat can make is failing to look in the right place, so with enough probes its recall can be pushed as close to 1 as the latency budget allows.

It is tempting to think that probing the ten nearest cells must catch the true neighbour. In two dimensions a cell has about six neighbours, so ten probes feel generous. But the count grows exponentially with dimension, and in thousands of dimensions a cell can border almost every other cell. More importantly, distances concentrate: the query is nearly as close to its tenth-nearest centroid as to its first, so it sits near many boundaries at once. IVF works because of that low "intrinsic dimensionality" we discussed above, which is also why the right nprobe can only be found by measuring recall.

Two properties of the GPU shape the implementation. The unit of work is a query paired with a probed list, and a batch of queries produces thousands of such pairs to be scanned in parallel. That only works well if the lists are of similar length, which is why cuVS trains these indexes with balanced k-means: one oversized list would leave most of the device waiting for a single scan to finish. Like brute force, IVF-Flat is also at its best with large query batches, since a lone query probing a few lists cannot keep a GPU busy.

New vectors can be appended to their nearest list without retraining, but the centroids were fitted to the data at build time, and as the data drifts recall and latency degrade until the index is rebuilt, a subject we return to under Keeping an index current.

IVF-PQ: Compressing Vectors to Enable Billion-Scale Search

IVF-PQ keeps the partitioning of IVF-Flat, but changes what is stored in the lists. In place of each full-precision vector it instead stores a short code produced by product quantization (PQ), a lossy compression scheme designed so that distances can be estimated directly from the codes without needing to decompress them. The result is an index one or two orders of magnitude smaller than the data it covers, which is what brings a billion vectors within reach of a single GPU.

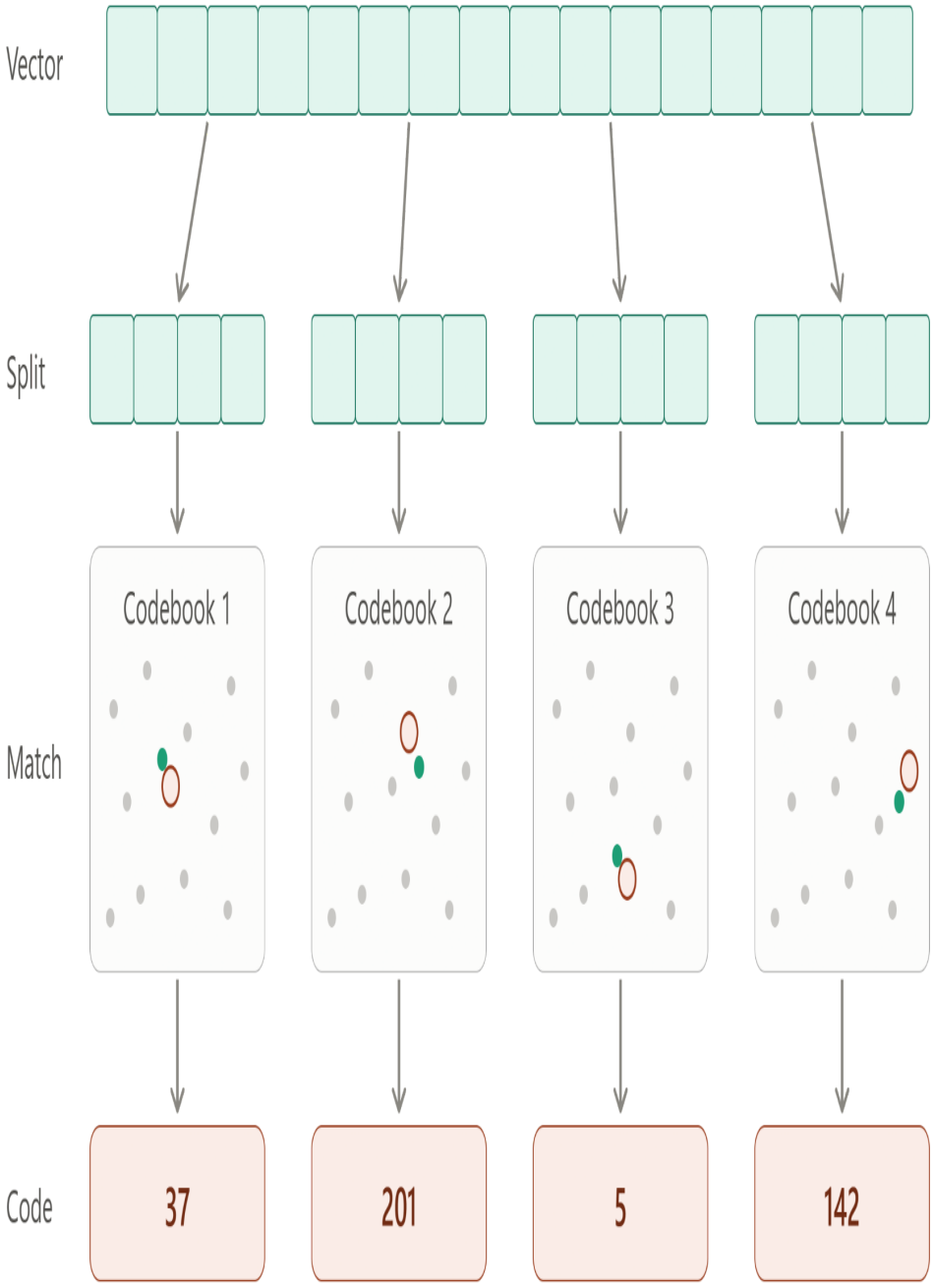
Product Quantization (PQ)

The idea, from Jégou et al.'s 2011 paper, is to quantise a vector in pieces. Each vector of dimension d is split into m subvectors of dimension d/m. For each of the m positions a separate codebook of 256 centroids is trained with k-means, on the subvectors that occupy that position across a sample of the data. A subvector is then replaced by the index of its nearest centroid in its codebook, which fits in one byte (a uint8). The original vector of d 32-bit floats becomes m bytes, a compression factor of 4d/m.

The name comes from what this does to the set of representable points. A single codebook with 256 entries could only distinguish 256 different vectors, which is useless. But m independent codebooks can be combined in 256^m ways, so the effective codebook is the Cartesian product of the small ones. With m = 64 that is 2^512 distinct reconstructions, from only 64 × 256 stored centroids. This is how PQ gets a fine-grained quantiser for the price of training a handful of very small ones.

The choice of m is the main knob. Recall the billion 1,024-dimensional vectors from earlier in the chapter, which occupied 4 TB. At m = 256, a quarter of the dimension and a common starting point, the codes take 256 GB, sixteen times smaller but still several GPUs' worth. At m = 64 they take 64 GB, which a single large GPU can hold. Each step down costs accuracy, because fewer centroids are being asked to describe more dimensions. cuVS also allows fewer than 8 bits per code for harder compression still.

One refinement to this picture matters in practice. IVF-PQ does not quantise the vectors themselves but their residuals, the difference between each vector and the centroid of the list it was assigned to. Residuals are smaller and more uniformly distributed than the raw vectors, so the same codebooks describe them more accurately. The coarse partition and the fine quantiser do complementary work.



16 floats (64 bytes) stored as 4 one-byte centroid IDs

Figure 2-3. Product quantizing a vector

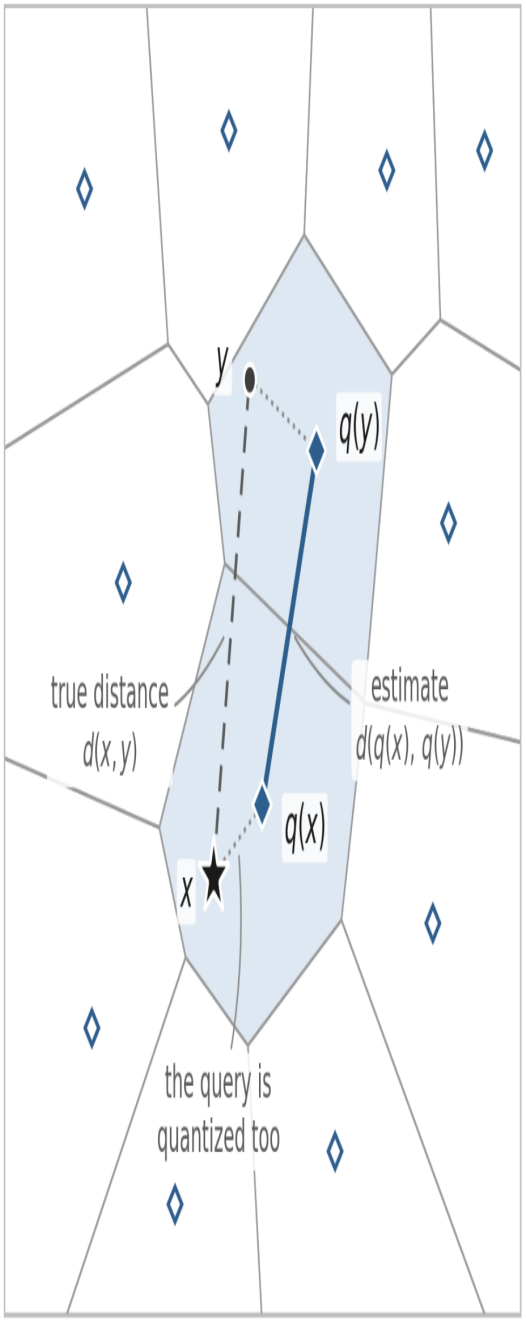
Searching compressed vectors

The coarse step is the same as in IVF-Flat: compare the query with the list centroids and select which lists to probe. The fine step is where PQ goes to work. For a given query, cuVS first computes the distance from each of the query's m subvectors to all 256 centroids in the corresponding codebook and stores the results in a lookup table of m × 256 entries. The estimated distance to any stored vector is then the sum of m table entries, one per byte of its code. A distance that cost d multiply-adds against a full vector now costs m lookups and additions, and the memory read per vector falls by the compression factor as well, which really matters on hardware where scans are bandwidth-bound. Because residuals are encoded, the table has to be built per probed list and not only per query, which is why cuVS offers reduced-precision tables as a tuning option.

This scheme is called asymmetric distance computation (ADC), because the stored vector is quantised while the query is not. The alternative is to quantise the query too, which gives symmetric distance computation (SDC). The one real advantage of SDC is that the lookup table becomes centroid-to-centroid, so it can be computed once offline and never depends on the query. But recall is worse, since error is now introduced on both sides, and the cost ADC pays for its per-query table is negligible next to scanning millions of codes. In practice ADC is what everyone uses.

Symmetric (SDC)

compare two codes: both vectors become centroids



Asymmetric (ADC)

compare the query to a code: only y becomes a centroid

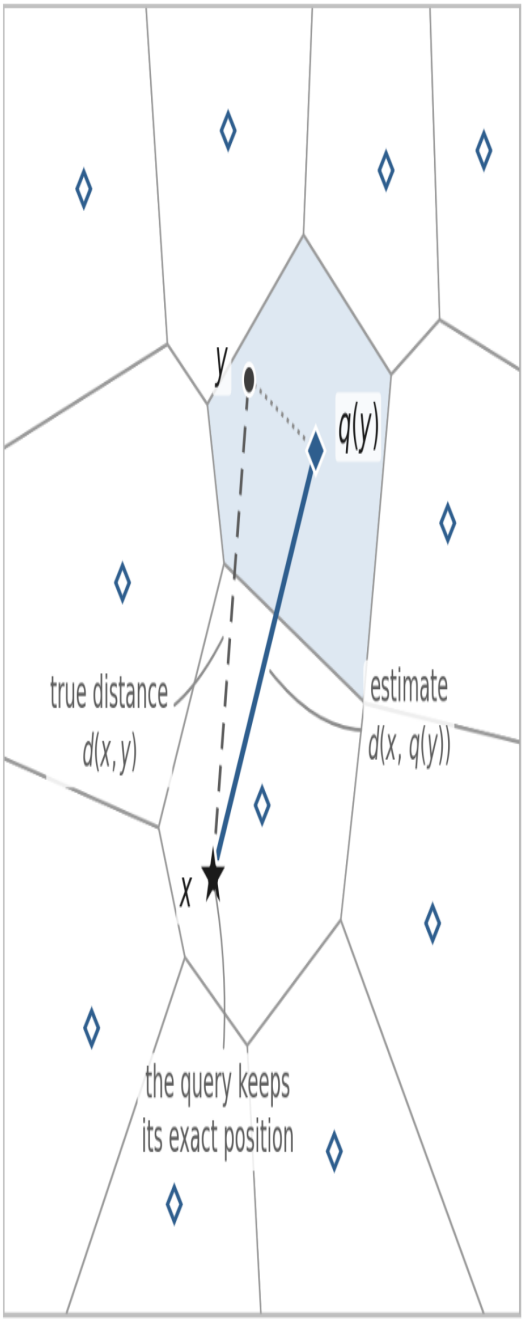




Figure 2-4. Symmetric vs Asymmetric distance computation for product quantization.

IVF-PQ therefore has two sources of error where IVF-Flat has one. It can fail to probe the list that holds a true neighbour, exactly as before, and it can also misrank the vectors it does look at, because their distances are estimates. Raising the number of probes repairs the first kind of error and does nothing for the second. Even probing every list does not bring the recall to 1.

Refinement

The repair for the second kind of error is a pattern that recurs throughout this chapter, so it is worth stating once in general form. Ask the compressed index for more candidates than are needed, say 4k in place of k. Then recompute the distance from the query to each of those candidates using the original, uncompressed vectors, and keep the best k. cuVS calls this step refinement, and this chapter uses the word only in that sense.

Refinement works because PQ's errors are small perturbations of the true ranking. A true top-10 neighbour might be estimated as the 25th closest, but it is very unlikely to be estimated as the 500th, so a modest over-fetch captures nearly all of them, and the exact pass then puts them in the right order. It is cheap because it touches only a few dozen vectors per query. And the full-precision vectors it needs do not have to be in GPU memory. They can stay in host memory or on disk, since so few of them are read. The combination is what makes aggressive compression practical: the GPU holds a small, fast, lossy index, and a small exact step at the end restores most of the recall the compression gave away.

Both IVF indexes answer a query by scanning lists, and at high recall targets the number of lists to scan grows quickly. The family that currently wins the trade between speed and recall at the high end takes a different approach altogether.

Graph-Based Indexes

A graph-based index does not partition the data. Instead, every vector becomes a node, each node keeps edges to a few dozen other nodes, most of them near neighbours, and a query is answered by walking the graph towards itself. The best-known example is HNSW (Hierarchical Navigable Small World), the default index used in most CPU vector databases and the standard cuVS's own graph index is measured against.

Searching a graph

A purely greedy walk, always moving to whichever neighbour is closest to the query, is fast, but gets stuck at the first node closer than all of its own neighbours, which need not be the nearest node overall. Every practical graph index therefore uses a beam search. It keeps a bounded list of the best candidates found so far, repeatedly expands the closest one not yet expanded by computing distances to its neighbours and merging them into the list, and stops when every candidate has been expanded; the top k of the list are the result. The list length is the main search-time knob in every graph index, ef_search in HNSW and itopk_size in CAGRA, and a longer list means more backtracking, more distance computations, and higher recall.

The appeal is how little of the data each query touches. The number of distance computations is roughly hops times degree, typically a few hundred to a few thousand, and it grows very slowly with the size of the corpus, whereas an IVF index chasing the same recall has to keep adding probes. This is why graphs win the trade between speed and recall at the high end. They pay for it in build time and in memory, since the index stores the edges as well as the full vectors.

What makes a graph searchable

The natural first guess, the k-nearest-neighbour graph in which every node points to its k closest nodes, is actually a poor search graph. Many of its edges are redundant, since a node's nearest neighbours lie in the same direction and lead to the same places; plus it has no long-range edges, so crossing the dataset takes a great many short hops; on clustered data it can

fall apart into disconnected components; and some nodes have no incoming edges at all, so no walk can ever reach them. A good search graph keeps most edges short for precision, spends a minority of its bounded degree on longer edges in diverse directions so the walk can cross the dataset in a few hops, and makes every node reachable.

The tool that produces such a graph is a pruning rule, and every index's version shares one idea: an edge to a neighbour is redundant if another, closer neighbour already offers a route to the same place. Drop those and what remains is sparser, points in different directions, and leaves room in the degree budget for longer links. HNSW's neighbour-selection heuristic, the pruning rules of NSG (Navigating Spreading-out Graph) and DiskANN's Vamana, and CAGRA's removal of what it calls detourable edges are all variations on this theme.

Two ways to build

HNSW, introduced by Malkov and Yashunin in 2016, builds its graph incrementally. Nodes are inserted one at a time, and each insertion is itself a search: find the new node's nearest neighbours in the graph as it stands, connect to a pruned selection of them, and update their edge lists in turn. Its build parameters are M, the edges per node, and ef_construction, the beam width used during insertion. The hierarchy in its name is a stack of sparser layers above the base graph, descended greedily to find an entry point; the base layer, which holds every node, is where the beam search described above happens.

Incremental construction means the index is usable from the first insertion and accepts new vectors at any time, which suits a database that ingests continuously. Its cost is that a build is one search per vector, each depending on the graph left by the insertions before it. That parallelises across CPU threads with careful locking, but it cannot be restructured into the large independent batches a GPU needs, and hours-long builds are the result. Deletion is awkward too, since removing a node can cut the paths that ran through it, so implementations mark deleted nodes and rebuild once enough accumulate.

The other strategy works in two stages: build an approximate k-nearest-neighbour graph over the whole dataset, then turn it into a search graph by pruning redundant edges and adding the ones needed for reachability. NSG and several related indexes are built this way, and so is CAGRA. A 2021 survey by Wang and colleagues calls this the refinement strategy; we will call it build-then-optimise, following the CAGRA paper, because refinement already means something else in this chapter. It needs the whole dataset up front and has no natural way to absorb a stream of insertions, but every stage is a pass over all the nodes at once with each node's work independent, which is exactly the shape a GPU is good at. It also starts from something HNSW never needs: a k-nearest-neighbour graph over the entire dataset, which computed exactly is the all-pairs problem, quadratic in the corpus. Building one approximately and cheaply is the subject of the next section.

Building Approximate k-NN Graphs

cuVS's approach is a pattern this chapter returns to several times. Produce an approximate k-NN graph quickly, using a method that is allowed to make mistakes, then hand it to a consumer that can tolerate them. The graph only has to be good, not perfect, because it is raw material. Optimised, it becomes a search index, which is how CAGRA is built. Used as it stands, it is the input to a whole family of clustering and dimensionality-reduction algorithms. cuVS has two ways of building it.

Building from an IVF-PQ index

The first builder uses nothing that has not already been introduced. Train an IVF-PQ index over the dataset, then use the dataset as its own query set: search every vector against the index in large batches, asking for more neighbours than the graph needs, and apply refinement against the original vectors to keep the best. The quantization error is tolerable because refinement corrects most of the misranking, and because the consumer will process the graph further in any case. The advantage of this route is memory. Only the compressed codes have to be on the GPU, and the

original vectors can stay in host memory for the refinement step, so it works for datasets much larger than the device.

NN-Descent

The second builder is an algorithm designed for exactly this problem, published by Dong et al. in 2011, and it fits in a paragraph. Start with a random graph: give every point k arbitrary neighbours. Then iterate. For each point, look at its neighbours' neighbours, including the points that list it as a neighbour, compute the distance to each, and if any is closer than the worst entry in the point's current list, swap it in. Repeat until an iteration produces almost no updates.

The intuition is that a neighbour of a neighbour is probably a neighbour. The random starting graph is terrible, but even a terrible graph contains a few lucky edges, and those propagate. The authors' argument for why requires the space to be growth-restricted: the number of points within radius 2r of any point is at most a constant multiple of the number within r. Under that condition each iteration roughly halves the radius within which a point's approximate neighbours lie. Growth-restricted is another way of saying that the intrinsic dimensionality is bounded, the same assumption that makes nearest-neighbour search meaningful at all.

In practice recall passes 90 percent within five or six iterations and the algorithm converges in about a dozen, having compared only a few percent of all possible pairs. A handful of engineering measures keep each iteration cheap: comparing two entries only if at least one is new since the last iteration, sampling a fraction of each list, capping the otherwise unbounded reverse-neighbour lists, and stopping once updates become rare.

The paper is also explicit about where the method fails: on synthetic data with k = 20, recall holds above 95 percent up to an intrinsic dimensionality of about 20 and collapses beyond 50, and raising k helps only up to a point. NN-Descent works on real embeddings because their intrinsic dimensionality is far below their nominal one.

NN-Descent on the GPU

The cuVS implementation descends from GNND, published by Wang et al. in 2021. The paper is a case study in what porting an algorithm to a GPU involves, because the algorithm barely changes and yet almost every implementation decision does. On a CPU, NN-Descent spends its time computing distances. On a GPU, distances are nearly free and the cost moves to the graph update. Inserting a candidate into a neighbour list is a read-modify-write on global memory under a lock, and when thousands of threads attempt it at once the memory system sets the pace.

GNND responds in two ways. First, everything becomes fixed-size. It takes a fixed number of new and old samples from each list and caps each reverse list, so the working set is a pair of contiguous fixed-degree arrays with no dynamic allocation. Second, and responsible for most of the speedup, is selective update. Classic NN-Descent inserts every improving pair it finds. GNND inserts almost none of them: for each sample, only its single nearest new candidate and its single nearest old one. Finding a nearest object is a min-reduction, the cheapest thing a GPU can do, and the number of writes to global memory drops to about three per node per iteration. Convergence is barely affected, because the discarded candidates were mostly ones that would have been squeezed out of the list a few iterations later anyway.

On an RTX 3090, GNND built a graph with recall@10 of 0.99 for SIFT1M in under four seconds. A naive port was already ten times faster than the CPU; the remaining factor of more than twenty came entirely from managing memory traffic.

Choosing a builder

The choice is mostly a question of memory. NN-Descent is generally the faster of the two, but it needs the vectors on the device, where cuVS holds them in half precision. The IVF-PQ route keeps only compressed codes on the GPU, so it is the one to use when the dataset is larger than GPU memory allows. By default cuVS chooses for you. One loss of generality is worth knowing. The 2011 algorithm works with any similarity measure, since it only ever asks for a way to evaluate one. The cuVS kernels support Euclidean, inner product, cosine, and bitwise Hamming distance.

The graph as an end in itself

An approximate k-NN graph is also the first step of a long list of other algorithms that have nothing to do with search: UMAP and t-SNE, HDBSCAN, spectral clustering, Leiden community detection, label propagation, and near-duplicate detection. cuVS exposes the pattern directly as its all-neighbors API, which builds the graph with either builder and returns it as it stands. cuML (Chapter 7) uses it for both UMAP and HDBSCAN, and the result can be passed on to cuGraph (Chapter 8) for community detection.

The other kind of consumer does not use the graph as it stands. It prunes it, repairs it, and searches it, and it is the subject of the next section.

CAGRA: The GPU-Native Graph Index

CAGRA (CUDA ANN GRAph-based) is cuVS's flagship index, introduced by Ootomo et al. in 2023. It is a build-then-optimise graph index in which both the build and the search were designed for the GPU from the ground up. Where HNSW resists the GPU with sequential insertions, variable node degree, and a per-query priority queue descending a hierarchy, CAGRA has a batch build, a fixed degree so every node costs the same to expand, and no hierarchy.

Building

The first stage is the subject of the previous section: construct an approximate k-NN graph over the whole dataset with NN-Descent or an IVF-PQ index. Its degree is a build parameter, the intermediate graph degree, defaulting to 128, twice the final degree; it is simply the k that NN-Descent is run with.

The second stage, which the paper calls graph optimisation, is CAGRA's version of the pruning rule. An edge from a node to a neighbour is detourable if another of the node's neighbours offers a two-hop route to the same place with both hops shorter than the direct edge; the edges with the most detours are dropped until each node is down to the final degree. CAGRA's twist is to judge "shorter" by rank rather than distance, so the

position of an edge in a node's sorted neighbour list stands in for its length and the whole stage runs on the graph alone, without touching the vectors. Reverse edges are then added, so that if A points to B, B points back to A, which rescues nodes that few others point to. The output is a flat fixed-degree graph stored as one row per node at four bytes per edge. Because pruning discards half the edges it is given, a mediocre initial graph still contains enough good edges to choose from, which is why an approximate builder is acceptable and why NN-Descent can be stopped after a few iterations.

Searching

CAGRA's search is the beam search described earlier with fixed sizes everywhere, work arranged to fit in on-chip memory, and no locks. With no hierarchy to descend, it seeds the candidate list with a set of random nodes, since on a GPU a few dozen extra distance computations cost almost nothing. Each iteration then does the same fixed amount of work: take the best few unexpanded candidates, gather their neighbours, which is always the same number because the degree is fixed, discard those already visited, compute the remaining distances in parallel, and merge them into the candidate list. That list has a fixed length, itopk_size, and is kept in order with a bitonic sorting network rather than a heap, the same exchange of a branching structure for a fixed comparison pattern seen in top-k selection. Visited nodes go in a small hash table owned by the query, so there is nothing to contend for.

For large batches each query gets a thread block of its own and the GPU runs thousands of searches side by side. For small batches or single queries, CAGRA can instead spread one search across several blocks to minimise latency, and cuVS chooses between the two modes automatically.

Performance

The paper's headline results against HNSW on a contemporary server CPU: graph construction 2.2 to 27 times faster, large-batch search throughput at 90 to 95 percent recall 33 to 77 times higher, and single-query search at 95

percent recall 3.4 to 8.5 times faster. The ranges are wide because the datasets differ and any such comparison depends on the hardware on each side, so treat the figures as indicative. The durable pattern is that the gap is widest at build time, where HNSW's sequential insertions hurt it most.

HNSW interoperability

That asymmetry suggests building on the GPU and serving wherever is convenient, and cuVS supports it directly by converting a CAGRA index into an HNSW index searched on CPU threads. The conversion works because the base layer of an HNSW index is a bounded-degree graph over all the nodes, which is exactly what a CAGRA graph is. What CAGRA lacks is the upper layers, and cuVS offers three options. It can leave them out, giving a base-layer-only index that is immutable and searchable only through cuVS's own wrapper around hnswlib. It can build them on the CPU, giving a complete index that can be extended by ordinary HNSW insertion and serialises to the standard hnswlib format, so any system that reads hnswlib files can serve it. Or it can build them on the GPU. The build can also be specified with HNSW's own M and ef_construction arguments, from which cuVS derives the CAGRA settings. This conversion is what lets a GPU be rented for the build and released while the serving path stays as it was, and most of the database integrations at the end of this chapter are built on it.

The Rest of the Catalogue

Brute force, the two IVF indexes, and CAGRA are the core of cuVS and the indexes most users will ever touch. Around them the library has accumulated a second tier, most of it recent and some of it experimental. There is no need to treat each one at length, because none introduces a new idea. Each extends one of two themes this chapter has already developed: compressing vectors, and building on the GPU for someone else to serve. A third group, the indexes that address updates after the build, is covered under Keeping an index current below.

More ways to compress

Product quantization is one point on a spectrum of compression schemes, and cuVS now covers more of that spectrum.

IVF-SQ is the gentlest option. Scalar quantization treats every dimension independently and maps each 32-bit float to an 8-bit integer, using a range learned from the data. That is a fixed compression factor of four, far less than PQ achieves, but also less lossy, and there are no codebooks to train. It fills the gap between IVF-Flat, which may not fit in memory, and IVF-PQ, which may cost more recall than the application can afford.

IVF-RaBitQ sits at the other extreme. RaBitQ, published by Gao and Long in 2024, applies a random rotation to each vector and then keeps a single bit per dimension, its sign, for a compression factor of 32. What distinguishes it from earlier binary schemes is that it comes with an unbiased estimator of the true distance and a provable bound on the estimator's error, so a search can tell which candidates are close enough to the cut-off to be worth refining. There is again nothing to train, which makes the build fast, and distances between bit strings are computed with bitwise operations that GPUs execute very efficiently. The cuVS implementation is described in one of the papers listed in the references. It arrived in release 26.08 and is, at the time of writing, available from C++ only.

CAGRA-Q applies the same thinking to the graph index. CAGRA's real memory cost is not its graph but the full-precision vectors that every hop of a search has to read. CAGRA-Q replaces them with a compressed copy, so that the graph and the codes together fit on a device that the original vectors would not. The walk is then guided by approximate distances, and as with IVF-PQ the natural companion is a refinement step against the original vectors held in host memory.

Building for other engines

The conversion from CAGRA to HNSW is one case of a more general pattern, in which the GPU builds an index in a format that some other system searches, and cuVS has two more.

Vamana is the graph at the heart of DiskANN, Microsoft's index for datasets too large for memory: compressed vectors in RAM steer the search, the graph and full vectors sit on SSD, and each hop reads only the few nodes it needs, which is how it serves a billion vectors from one machine. cuVS builds the Vamana graph on the GPU and writes it in DiskANN's own file format; searching means loading the files with DiskANN. The build is worth a glance because it is incremental insertion restructured for the hardware: vectors are inserted by searching the graph built so far, as in HNSW, but in batches that grow as the graph does, so that each batch's searches run in parallel.

ScaNN is Google's partition-and-quantise index, and cuVS has an experimental builder for its open-source format, from C++ only and again with no search of its own. Its components are familiar: a partition of the dataset, product quantization of the residuals, and an optional reduced-precision copy of the vectors for reranking. It adds two ideas. The first is an anisotropic loss: for inner-product search, quantization error parallel to a vector distorts its score more than error orthogonal to it, because the queries that score highly against a vector point roughly the same way, so ScaNN penalizes the parallel error more heavily. The second is SOAR, which gives each vector a backup assignment to a second partition. A query misses a vector's partition when it aligns with the vector's residual, so SOAR chooses the backup whose residual is closest to orthogonal to the first, and the two assignments fail on different queries.

These second-tier components vary in maturity and in how many of the language bindings expose them; some build but do not search, and some exist in C++ alone. Check the current documentation before designing a system around any of them.

Clustering, Distances, and Preprocessing

The indexes are assembled from smaller parts, and the library exposes those parts as APIs in their own right. Figure 2-5 shows the dependencies: two primitives at the bottom, and everything above them built by composing what sits below.

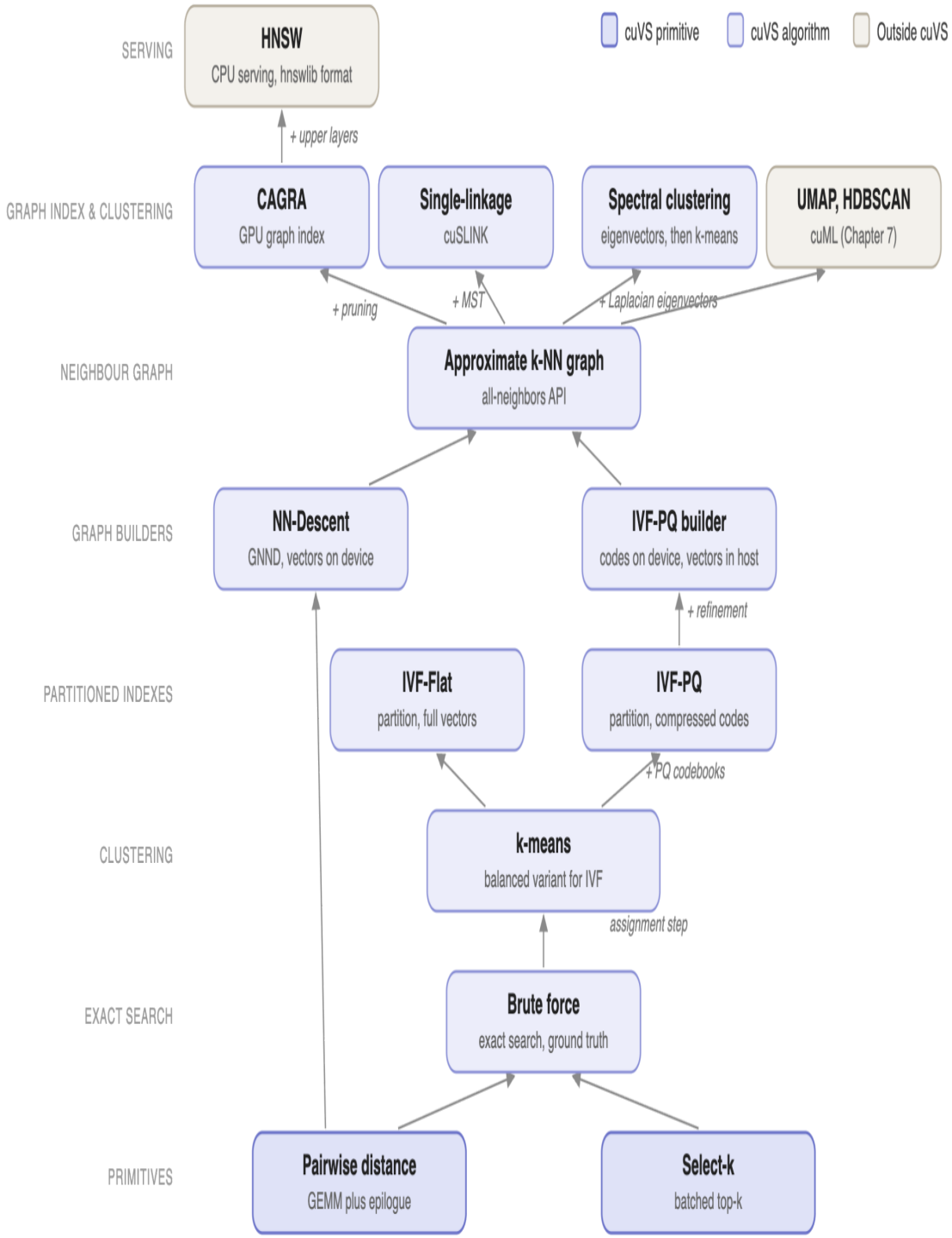

cuVS primitive
cuVS algorithm
Outside cuVS
SERVING
HNSW
CPU serving, hnswlib format
+ upper layers
GRAPH INDEX & CLUSTERING
CAGRA
GPU graph index
Single-linkage
cuSLINK
Spectral clustering
eigenvectors, then k-means
UMAP, HDBSCAN
cuML (Chapter 7)
+ pruning
+ MST
+ Laplacian eigenvectors
NEIGHBOUR GRAPH
Approximate k-NN graph
all-neighbors API
GRAPH BUILDERS
NN-Descent
GNND, vectors on device
IVF-PQ builder
codes on device, vectors in host
+ refinement
PARTITIONED INDEXES
IVF-Flat
partition, full vectors
IVF-PQ
partition, compressed codes
+ PQ codebooks
CLUSTERING
k-means
balanced variant for IVF
assignment step
EXACT SEARCH
Brute force
exact search, ground truth
PRIMITIVES
Pairwise distance
GEMM plus epilogue
Select-k
batched top-k

Figure 2-5. How cuVS composes. Bottom: pairwise distance and select-k. Above them brute force; k-means, built from brute force over centroids; IVF-Flat and IVF-PQ, built from k-means; the k-NN graph builders, one built from IVF-PQ plus refinement; CAGRA, built from a k-NN graph plus pruning, and HNSW from CAGRA plus upper layers; and the clustering algorithms, single-linkage and spectral from the k-NN graph, with UMAP and HDBSCAN in cuML consuming the same graph.

K-means comes in several forms: Lloyd's algorithm with a k-means++ style initialisation, the balanced variant the IVF indexes train with, and multi-GPU and batched variants for datasets that do not fit on one device. The other clustering algorithms are compositions of pieces the library already has. Single-linkage agglomerative clustering, described in the cuSLINK paper in the references, builds a k-nearest-neighbour graph, computes a minimum spanning tree of it on the GPU, reconnects any components the graph left disconnected, and reads the cluster hierarchy off the tree. Spectral clustering follows the same recipe with different middle steps: neighbour graph, leading eigenvectors of its Laplacian, k-means on the result. A Gaussian mixture model is the most recent addition. Two scores judge the output: silhouette, computed in batches to avoid the full distance matrix, and trustworthiness, the standard check on how well a dimensionality reduction preserved each point's neighbourhood.

Underneath everything is the pairwise distance API, which supports a far longer list of metrics than the indexes do, among them L1, correlation, Hamming, Jaccard, Hellinger, and Kullback-Leibler divergence, and has a sparse counterpart for data such as TF-IDF vectors. Beside the indexes sit two neighbour searches for particular needs: the epsilon-neighbourhood search, which returns every point within a fixed radius and is the query DBSCAN is built on, and the ball cover index, an exact method for low-dimensional data such as geographic coordinates.

The scalar, binary, and product quantisers are available as standalone transforms, for when the consumer is not a cuVS index at all, such as a database with its own index. PCA and spectral embedding complete the set: reducing dimension before building an index shrinks the memory, the build, and the cost of every distance, and PCA is the routine way to do it for embeddings that were not trained to be truncated.

Choosing the Right Index

Choosing an index comes before tuning one, and it is mostly a matter of asking a few questions in the right order.

Do you need an index at all?

Start by assuming you do not. Brute force is exact, needs no build, and on a GPU stays fast up to tens of millions of vectors, particularly when queries arrive in large batches. It is also the right answer when the corpus changes constantly, since there is no structure to maintain. And it is the right answer when most queries carry a filter that excludes nearly all of the data. Move on only when you have measured it and found it too slow, or when the data no longer fits in GPU memory.

Does the full-precision data fit in GPU memory?

If it does not, the choice is made for you, and what remains is how hard to compress. IVF-PQ is the established answer, paired with refinement against the original vectors in host memory. IVF-SQ is worth considering when a factor of four is enough, and CAGRA-Q when you want a graph index over data that will not otherwise fit. If the data does fit, prefer an index that keeps it at full precision, because a source of error you do not introduce is one you never have to tune away.

What recall do you need, and how do queries arrive?

This separates the two full-precision indexes. CAGRA is the choice for high recall at low latency, and its advantage is greatest at small batch sizes, which is the online serving case. IVF-Flat is simpler, builds faster, and does well when queries come in large batches and the recall target is moderate, which describes much offline work. As the recall target rises, IVF-Flat has to probe more and more lists and the graph pulls ahead.

Where will the index be served?

If the answer is a fleet of CPU machines, build a CAGRA graph and convert it to HNSW, keeping the GPU only for the build. If the dataset

is too large even for host memory and will be served from SSD, build a Vamana graph for DiskANN. In both cases cuVS is a build tool, and the serving characteristics are those of the engine that loads the result.

How often does the data change?

Every approximate index here is built in batch, and they differ in how gracefully they absorb change afterwards. The IVF indexes and CAGRA can all be extended with new vectors, and this is fine for moderate growth, but all of them drift from the data they were fitted to and eventually want rebuilding. When rebuilds take minutes, the simplest policy is often to rebuild on a schedule and not to update at all. For a steady stream of insertions that must be searchable at once, use the tiered index (described later), which holds fresh vectors in a brute-force tier until they are folded in.

Do you want a search index, or the graph?

If the goal is clustering, dimensionality reduction, or deduplication, what you need is the all-neighbors API and not an index.

Table 2-1 summarises the four core indexes. The memory figures are for N vectors of dimension d stored as 32-bit floats. For IVF-PQ, m is the number of bytes per code, and for CAGRA, g is the degree of the graph.

Table 2-1. The core cuVS indexes compared

	Brute force	IVF-Flat	IVF-PQ	CAG
Recall	Exact	High, set by probes	Moderate, high with refinement	High amor appro index
GPU memory	4Nd	4Nd plus centroids	Nm plus codebooks	4Nd
Build cost	None	Low	Moderate	High four
Best query pattern	Large batches	Large batches	Large batches	Any, singl
Absorbing new data	Trivial	Extend, rebuild on drift	Extend, rebuild on drift	Exter merg on dr
Serve without a GPU	No	Centroids are portable	Centroids are portable	Yes,

These are starting points and not conclusions. The only reliable way to choose between two candidate indexes is to build both on a sample of your own data and measure recall against throughput with your own queries. cuVS ships a benchmarking tool for exactly that, and it is where the tuning section begins.

Filtered Vector Search

Real queries are rarely just "what is nearest to this vector?" They are "what is nearest among the documents this user may see, in this language, from the past year?" An approximate index is built to answer the first question, and adding the second makes its job considerably harder. There are three common ways of combining a filter with a vector search, and which one works best depends mostly on the filter's selectivity, by which we mean the fraction of the corpus that passes it. Terminology varies between systems, and the definitions below follow the most common convention.

Pre-filtering applies the filter first and then computes exact distances to the vectors that pass, bypassing the approximate index entirely. It is best when the filter is highly restrictive, passing perhaps 2% of the corpus or less. It needs an efficient way to enumerate the passing vectors, and it works only while there are few enough of them for a brute-force scan to stay cheap. An approximate index does badly in this regime. Asked for some multiple of k candidates, it would often return too few that pass the filter, and sometimes none.

Post-filtering queries the index for more candidates than are needed, expecting some to fail, and discards the failures afterwards. It is best when the filter is mild, passing perhaps 60% of the corpus. Computing exact distances to 60% of the corpus would be nearly as expensive as having no index at all, while over-fetching by a factor of two costs very little.

In-flight filtering covers the awkward middle. At 10% selectivity there are too many passing vectors to scan cheaply, but post-filtering would have to over-fetch by ten times or more to end up reliably with k results. The remedy is to make the search itself aware of the filter: it is passed in alongside the query and applied as candidates are considered, not before or after.

How cuVS represents a filter

cuVS takes the third approach throughout, although its API calls the argument a prefilter, which is worth knowing when reading its documentation. A filter reaches a search in one of three forms.

A bitset has one bit per vector in the index, and applies to every query in the batch. It suits filters shared across a batch, such as excluding deleted rows. A bitmap has one row of bits per query, so each query in a batch can carry a different filter. Both are compact per bit and costly in bulk. A bitset over a billion vectors is 125 MB, which is tolerable, but a bitmap for a batch of a thousand queries over the same index would be 125 GB, which is not. In either case something outside cuVS, normally the database, has to evaluate the predicate over all of the metadata and produce the bitsets before the search can begin.

The third form, introduced in release 26.08 and so far available for CAGRA from C++, removes that step. A filter UDF is a small piece of CUDA source, compiled at run time, that decides whether a given candidate is valid for a given query. It reads metadata already resident on the device, such as tenant identifiers, access control lists, or timestamps. The predicate then runs inside the search kernel, evaluated only for the candidates the search actually visits, and nothing has to be materialised for the whole index. It is deliberately limited. It can accept or reject candidates, but it cannot steer the traversal or alter the distance computation.

What each index does with it

Brute force treats filtering as a cost decision rather than a choice of semantics. cuVS counts the bits that pass and picks one of three strategies. When very few pairs pass, it pre-filters, computing only those distances with a masked matrix multiply at a cost proportional to the number of passing pairs. When a moderate fraction pass, it also pre-filters, gathering the passing rows into a smaller dense matrix and searching that. Otherwise it post-filters, running the usual tiled matrix multiply over everything and masking the results. Because the mask is applied before the top-k selection, this last path loses nothing, and all three return the same exact answer. These are the regimes discussed in the previous section, chosen automatically, and the first is why brute force is the right tool at the restrictive end. The gather path needs a single set of passing rows, so only a bitset can use it; a bitmap goes straight from the masked multiply to the full search.

The IVF indexes skip the distance computation for any vector in a probed list that fails the filter, but a passing vector in a list that was not probed is never seen, and the more restrictive the filter the more likely the probed lists hold too few passing vectors to fill k results. The remedy is to raise the number of probes as the filter tightens.

CAGRA excludes failing nodes from the results but still walks through them, because a walk that refused to enter filtered nodes would, under a restrictive filter, find the graph cut into disconnected fragments with most of the passing nodes unreachable. The price is that failing nodes occupy places in the candidate list that passing ones could have had, so a tighter filter calls for a longer list, and CAGRA's search parameters take a hint for the expected filtering rate so it can make that adjustment itself. The disconnection problem is what research indexes such as ACORN and Filtered-DiskANN address, and VecFlow, in the references, brings that line of work to the GPU on top of cuVS.

In practice, then, the three textbook strategies collapse into one rule. Filtering inside the search is cheap enough in cuVS that post-filtering is only worth it when the predicate cannot be brought to the GPU at all, because the metadata lives elsewhere and is known only once candidates come back. The working question is where to switch to brute force: at a few percent of the corpus passing or less, use the brute-force index with the filter; above that, use CAGRA or an IVF index with the same filter and give it more candidates or more probes as selectivity falls. The crossover depends on the data, and a benchmark run without filters says little about a workload that always has them.

Beyond One GPU, and After the Build

Everything so far has assumed a single GPU, a dataset that fits on it at least in compressed form, and an index that is built once and then searched. Production systems break each of those assumptions, and cuVS has an answer for each.

Searching across several GPUs

Multi-GPU support in cuVS is a coordination layer and not a new family of indexes. It wraps CAGRA, IVF-Flat, and IVF-PQ, and lets each be built, searched, extended, and saved across all the GPUs in a single machine. The index-specific parameters keep their meaning, and the layer adds one decision, which is how the index is distributed.

In replicated mode every GPU holds a complete copy of the index and incoming queries are spread across them. Nothing gets bigger, but throughput scales with the number of devices available, and any one of them can answer any query. Queries can be dealt out in strict rotation or by load, and balancing by load is the safer default when batch sizes vary.

In sharded mode the dataset is split and each GPU indexes and holds only its own part. This is the mode meant for increasing capacity. Every query now goes to every GPU, each searches its shard, and the per-shard results are merged into a final top k. As the brute force section noted, merging per-shard results is exact when the underlying search is exact. With an approximate index each shard loses a little recall independently, but each shard is also a smaller and easier search problem. The merge can be done on one GPU or as a tree across all of them, and the tree is preferable as the number of devices grows, because it avoids funnelling all the results through a single device.

The documentation's advice on order is sound: tune the index on one representative GPU first, and only then choose how to distribute it. Two things are worth knowing. The multi-GPU APIs expect datasets and queries to reside in host memory, because it is the layer itself that moves data to the devices. And the coordination underneath is NCCL, reached through RAFT's communicator abstraction: the "clique" object a multi-GPU build takes is an NCCL communicator over the participating devices, and the sharded merge is a collective reduction over it, either gathered on a root rank or reduced as a tree. Chapter 12 covers NCCL in its own right. cuVS can also attach an externally created NCCL communicator so that ranks span machines, but it does no cluster orchestration of its own; starting

workers, assigning ranks, and placing shards remain the job of the database that embeds it. That fits the library's place in the stack. A system like Milvus already shards collections across nodes, and needs cuVS only to make each node fast.

Building What Does Not Fit

A separate problem is a dataset too large to build an index over, even when the finished index would be small enough to serve. The IVF-PQ route to a k-NN graph keeps only compressed codes on the GPU and works in batches.

The current answer for graph indexes is a build method called ACE, for Augmented Core Extraction, which works for both CAGRA and HNSW output. ACE first partitions the dataset with balanced k-means. It then builds a sub-graph for each partition independently, each small enough to fit on the GPU, and finally concatenates the sub-graphs into one index. The obvious flaw in such a scheme is that independently built sub-graphs have no edges between them, and a walk that starts in one partition could never leave it. That is what the augmentation is for. Each partition is built not only from its own core vectors but also from nearby vectors that belong to neighbouring partitions, so that the edges formed during the build reach across the boundaries. The number of partitions can be derived automatically from the memory available, and the intermediate data can be kept on disk, so the size of the dataset is limited by storage and not by memory.

The same pressure has reached the supporting components. Recent releases added k-NN graph construction over data held in host memory and a streaming, out-of-core k-means, since an IVF index over a billion vectors has to train its centroids somehow.

Keeping an Index Current

The third assumption, that an index is built once, fails in any system that accepts writes. Batch-built indexes are awkward to update. For CAGRA the

reason is structural, since the fixed-degree, contiguous layout that makes the graph fast to read is exactly what makes it hard to modify. cuVS attacks the problem from several sides, and none of them is a complete answer by itself.

Adding vectors

The IVF indexes and CAGRA can all be extended in place. For an IVF index each new vector is appended to the list of its nearest centroid. For CAGRA each new vector is connected into the existing graph. Both are cheap, and both slowly degrade the index. The centroids were fitted to the old data, and the graph's new edges are not chosen with the same degree of care the original optimisation took.

Removing vectors

There is no physical deletion from any of these indexes. The practical mechanism is the filter from the previous section. Keep a bitset of live rows, clear a bit when a row is deleted, and pass the bitset alongside every search. The deleted vectors remain in the structure, and in a graph index remain useful for navigation, until the next rebuild removes them for good.

Combining indexes

Two CAGRA indexes can be merged physically into a single graph, or logically through the composite index, which presents several separate indexes as one, searches each, and merges the results. Merging suits systems that ingest data in segments and consolidate them later, which is how most databases built on log-structured storage already work.

Buffering fresh data

The tiered index packages the most common pattern of all. A small brute-force index sits in front of a large approximate one, at present always CAGRA. New vectors go into the brute-force tier, where they are searchable immediately and exactly; queries search both tiers and merge the results; and when the fresh tier has grown large enough its

contents are folded into the approximate index. A collection below a configurable minimum size gets no approximate index at all, which is this chapter's advice for small datasets made automatic. Most vector databases implement some version of this themselves, and the point of having it in the library is that they no longer need to.

Rebuilding

The last resort is also the cleanest. Extending, masking, and merging all postpone the rebuild, and none of them removes the need for it, since an index drifts away from its data as the data changes. What the GPU changes is the cost of that rebuild. When it takes minutes and not hours, rebuilding nightly, or hourly, stops being an emergency measure and becomes a reasonable design, and the more elaborate update machinery is needed only for the gap between one rebuild and the next.

Building a Semantic Search Pipeline

The canonical semantic search system has two halves: an offline workflow that embeds a corpus and builds an index over it, and an online workflow that embeds each incoming query and searches that index. This section builds both in a few dozen lines, then adds the third piece that turns a demo into something you can tune.

At indexing time

Each document is passed through an embedding model, and the resulting vectors are indexed. Retrieval is the asymmetric case described at the start of this chapter, so the model has to be told which side of the relationship it is looking at. Different models do this in different ways. The E5 family used here expects the text to be prefixed with "passage: " for documents and "query: " for queries. The embeddings are normalised to unit length, which, as we saw in the problem statement, makes inner product, cosine, and Euclidean distance rank identically, so we can build the index with the cheapest of them.

```
import json
import cupy as cp
from sentence_transformers import SentenceTransformer
from cuvs.neighbors import cagra
docs = load_documents()  # list[str]
# Embed every document, marked as a document ("passage: " is E5's
convention)
model = SentenceTransformer("intfloat/e5-base-v2")
embeddings = model.encode(docs, prompt="passage: ",
normalize_embeddings=True)
# Build the graph index on the GPU
params = cagra.IndexParams(metric="inner_product")
index = cagra.build(params, cp.asarray(embeddings))
# Persist the index, plus the documents that its row ids point to
cagra.save("docs.cagra", index)
with open("docs.json", "w") as f:
    json.dump(docs, f)
```

Two things about this code are worth pointing out. The index knows nothing about documents. It stores vectors and returns row numbers, and the mapping from row number back to document is the application's responsibility, handled here by saving the list in the same order. And the choice of index is a single line. For a corpus of a few hundred thousand documents, the advice of the previous sections is to use brute force, and swapping it in means changing the import and the build call and nothing else.

At query time

A query arrives, is embedded with the same model and the query prefix, and is looked up in the index. The top k row numbers come back and are mapped to documents.

```
import json
import cupy as cp
from sentence_transformers import SentenceTransformer
from cuvs.neighbors import cagra
# Done once, at server startup
model = SentenceTransformer("intfloat/e5-base-v2")
index = cagra.load("docs.cagra")
with open("docs.json") as f:
    docs = json.load(f)
```

```
def search(query: str, k: int = 5) -> list[str]:
    # Embed the query, marked as a query
    q = model.encode([query], prompt="query: ",
normalize_embeddings=True)
    # Walk the graph on the GPU; neighbors holds row ids into
docs
    distances, neighbors = cagra.search(
        cagra.SearchParams(), index, cp.asarray(q), k
    )
    return [docs[i] for i in cp.asarray(neighbors)
[0].get().tolist()]
```

This function answers one query at a time, which is the least efficient way to use a GPU. A real server gathers the queries that arrive within a few milliseconds of each other and searches them as a batch. The function already accepts that shape, since the query array can have any number of rows, and cuVS provides a dynamic batching layer that does the gathering for you, which we come to in the tuning guide.

Measuring it

The pipeline so far runs, but there is no way to tell how well. The missing piece is a recall measurement, and it takes only a few more lines, because the exact answer is always available from a brute-force index over the same vectors.

```
import numpy as np
from cuvs.neighbors import brute_force
k = 10
sample = model.encode(sample_queries, prompt="query: ",
normalize_embeddings=True)
sample = cp.asarray(sample)
# Ground truth from an exact search over the same vectors
exact = brute_force.build(cp.asarray(embeddings),
metric="inner_product")
_, truth = brute_force.search(exact, sample, k)
_, found = cagra.search(cagra.SearchParams(), index, sample, k)
truth = cp.asarray(truth).get().tolist()
found = cp.asarray(found).get().tolist()
recall = np.mean([len(set(t) & set(f)) / k for t, f in zip(truth,
found)])
```

The one decision that matters here is where `sample_queries` comes from. It should be a sample of real queries, or the closest thing to them you can obtain, and not a held-out set of documents. In the asymmetric case the two come from different distributions, and an index that finds the neighbours of documents reliably can do noticeably worse on the queries it will actually serve. A few hundred to a few thousand queries is enough. With this number in hand, every parameter in the next section becomes something you can evaluate and not something you have to take on trust.

What the example leaves out

Several things a production system needs are absent, and most of them are outside cuVS altogether. Documents longer than the model's input limit, 512 tokens in this case, have to be split into chunks before embedding, and the chunks are what get indexed. Row numbers change whenever the index is rebuilt, so a real system keeps a stable document identifier alongside each row. Results from vector search are often fused with a lexical ranking and passed to a reranking model before being shown to anyone (hybrid search). And the corpus changes, which raises everything discussed in the previous section. Those concerns are what a vector database exists to handle, and they are the reason most users of cuVS meet it inside one.

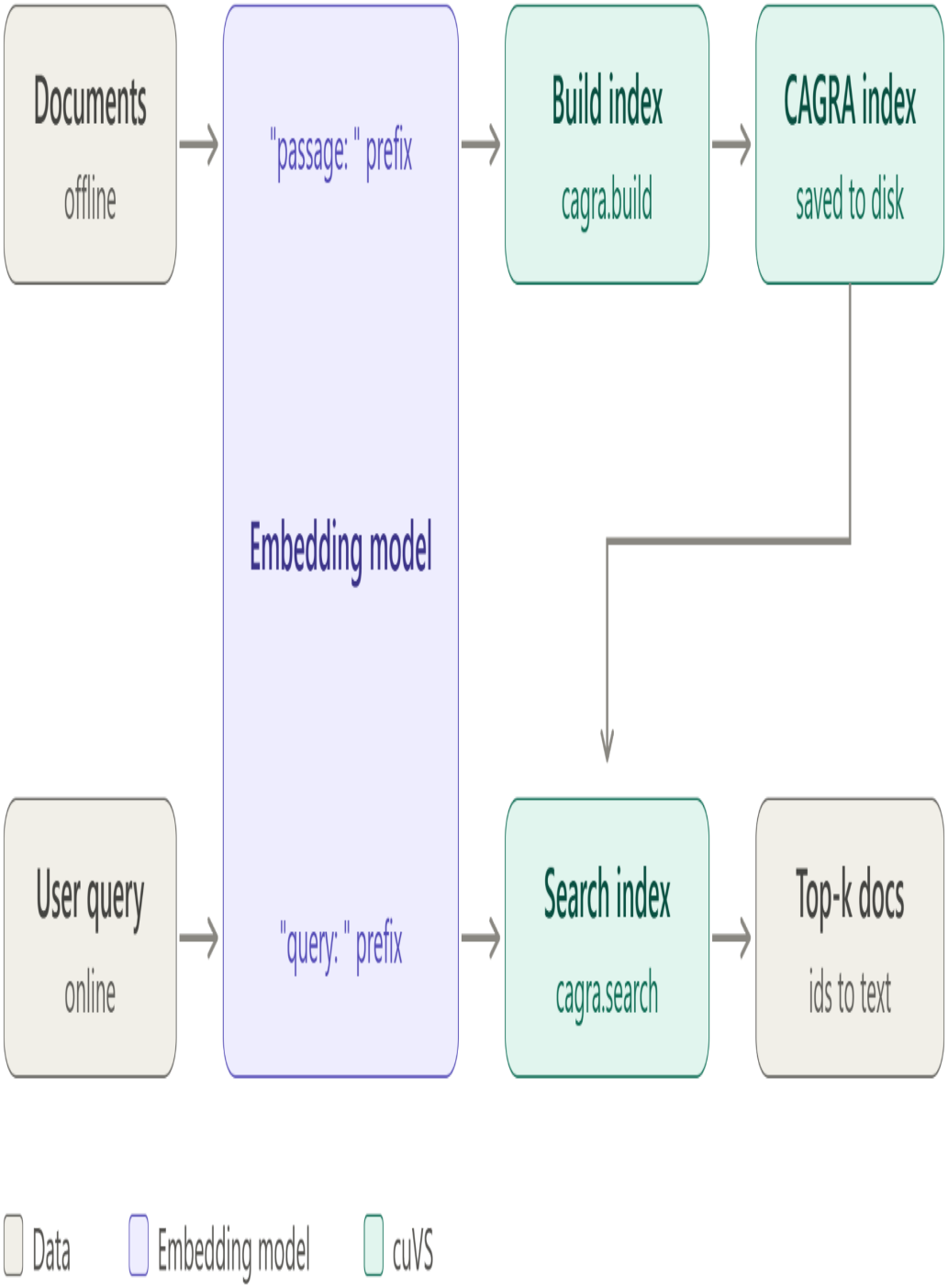

Documents
offline
"passage: " prefix
Build index
cagra.build
CAGRA index
saved to disk
Embedding model
User query
online
"query: " prefix
Search index
cagra.search
Top-k docs
ids to text
Data
Embedding model
cuVS

Figure 2-6. Example Semantic Search Pipeline Figure

Performance Tuning Guide

Every parameter in cuVS trades among four things: recall, search speed, build time, and memory. No setting is good universally, so tuning begins with the measurement built at the end of the previous section: ground truth from an exact search, and recall computed against it for a sample of real queries. Time each configuration as well and you have a curve of recall against throughput from which to pick an operating point.

It helps to treat this the way you would treat training a model. Sweep broad ranges before fine-tuning individual values, since a coarse sweep finds the useful region faster. Tune against one set of queries and confirm the best candidates on a second, held-out set, because it is quite possible to overfit an index to a few hundred queries. For large search spaces, ordinary hyperparameter optimisation tools work well with recall and throughput as joint objectives. cuVS ships a benchmarking tool, cuvs-bench, that automates these sweeps across algorithms and datasets and includes a tuning mode of its own.

One principle applies to every index: tune search parameters before build parameters. They are free to change, since nothing has to be rebuilt, and they often suffice.

IVF tuning

IVF-Flat and IVF-PQ share two parameters that matter more than all the others: n_lists at build time and n_probes at search time.

Start with n_lists, aiming for roughly 1,000 to 10,000 vectors per list. At a hundred million vectors this agrees with the square-root rule from the IVF-Flat section, and at a billion it calls for more and smaller lists than the square root would give, which is the better guide at that scale. Then tune n_probes, the main knob for recall and latency, which defaults to 20. What matters is the ratio between the two: probing 20 of 1,024 lists scans about

2% of the dataset, and the same 20 of 16,384 lists scans about 0.1%. For very large datasets, reducing kmeans_trainset_fraction shortens the build considerably, since the centroids do not need every vector to be well placed.

IVF-PQ adds the compression parameters. Its pq_dim is the number of subvectors, the m of the IVF-PQ section, and pq_bits is the size of each code. A good starting point is pq_dim at one quarter of the vector dimension with pq_bits at 8. There are constraints: pq_dim times pq_bits must be a multiple of 8, a pq_dim that is a multiple of 32 performs best, and ideally it divides the vector dimension evenly. Compress harder only while recall holds. The payoff is large. The documentation's example of a million 128-dimensional vectors comes to about 73 MiB as an IVF-PQ index, against 488 MiB of raw 32-bit data.

The most effective recall tool for IVF-PQ is not a PQ parameter at all but refinement, as described earlier: over-fetch by a factor of around four and rerank against the original vectors, which can stay in host memory. This pairs well with aggressive compression, because the throughput gained from smaller codes and tables can pay for the refinement step.

Leave the low-precision options (lut_dtype, internal_distance_dtype, coarse_search_dtype) until the basic balance of recall and latency is right, since each adds another source of approximation. The first of these shrinks the per-probe lookup table discussed in the IVF-PQ section. In NVIDIA's DEEP-100M experiments, 16-bit types improved throughput without reducing recall, while an 8-bit lookup table cost significant recall.

CAGRA tuning

The main search knob is itopk_size, the length of the candidate list in the beam search. It defaults to 64, must be at least k, and plays the same role as ef_search in HNSW. Raising it usually improves recall at the cost of throughput. The secondary knob is search_width, the number of candidates expanded per iteration, which defaults to 1. Increasing it helps when searches are getting trapped in one region of the graph. Leave

max_iterations and algo on their automatic settings unless you have a specific reason not to.

If no search setting reaches your recall target, move on to the graph. Increase graph_degree first, which gives every node more edges to follow but costs memory and work per hop. If graph quality is still the limit, increase intermediate_graph_degree, for the reasons given in the CAGRA section: it is the k of the initial neighbour graph, and data of high intrinsic dimensionality needs more of it. The defaults are 64 and 128. At four bytes per edge, a million vectors at degree 64 is about 244 MiB, usually small next to the vectors themselves, and halving the degree to 32 is a reasonable trade when memory is tight and recall targets are moderate. On clustered datasets the guarantee_connectivity option is worth trying, as it forces the initial graph to be connected.

The choice of initial graph builder is the memory decision described in the section on building neighbour graphs, and by default cuVS makes it for you. Two further options are worth knowing. If you plan to serve from CPUs, specify the build with HNSW's own M and ef_construction and let cuVS derive the CAGRA parameters, so that you do not tune twice. And if GPU memory and not recall is the constraint, CAGRA-Q replaces the full vectors with a compressed copy.

Queries that do not look like the data

One effect warrants separate mention, because it is easy to miss and it bears hardest on graph indexes. A search graph is built entirely from the corpus. Its edges connect documents to nearby documents, and its pruning decides which edges are redundant by looking at documents alone. In the asymmetric case the queries come from a different distribution, and a query may sit in a region of the space that the graph's structure does not serve well. The symptom is recall on real queries that is markedly lower than recall measured with held-out documents as queries, which is why the previous section insisted on the former. The remedy within cuVS is to give the search more slack, with a longer candidate list, a wider search, or a higher graph degree, and to accept that the operating point will sit further

along the curve than a benchmark on symmetric data would suggest. The partition-based indexes are less sensitive, since a query only has to land near the right centroids.

General tips

Tune with filters if your workload has them; a benchmark without them says little, as the filtered-search section explained.

Batch your queries. A GPU searching one query at a time is mostly idle. IVF methods benefit most, since their unit of work is queries times probes, but CAGRA throughput also rises steeply with batch size. For online serving, cuVS provides a dynamic batching API that gathers concurrent requests into batches, and for services receiving many small requests CAGRA's persistent search mode keeps a kernel resident on the GPU and feeds it incoming work, avoiding a launch per request.

Keep data on the device. Pass CuPy arrays or PyTorch CUDA tensors rather than NumPy arrays, so that nothing crosses the PCIe bus on each call, and copy results back once and not per query. If your embedding model runs on the same GPU, the embeddings never need touch host memory at all.

Use a pooled memory allocator. Raw CUDA allocations are slow, and the IVF indexes allocate each list separately. With RMM (Chapter 5) this is a few lines at startup:

```
import rmm
pool = rmm.mr.PoolMemoryResource(rmm.mr.CudaMemoryResource(),
initial_pool_size=2**30)
rmm.mr.set_current_device_resource(pool)
```

Shrink the vectors. Before reaching for a compressed index, make the vectors themselves smaller, which reduces the memory and the distance cost of every index at once. Storing 16-bit floats in place of 32-bit halves dataset memory with negligible recall loss for most embedding models, and 8-bit integers quarter it. Reducing dimension compounds with either: truncate the embeddings if the model was trained with MRL, or use the PCA preprocessor if it was not, and renormalise afterwards if you rely on

inner product and cosine ranking identically. Measure the effect against the right baseline: compute the ground truth once from the original full-size vectors and score every configuration against it, since recall against an exact search over the shrunken vectors says nothing about what the shrinking cost.

Estimate memory before you build. The documentation gives formulas for the build and search peaks of every index, and recommends reserving 20% headroom for graph builds that go through IVF-PQ and 30% for NN-Descent. An out-of-memory failure forty minutes into a build is an expensive way to learn the same thing.

cuVS Inside Other Systems

Persistence, stable identifiers, updates, hybrid ranking with lexical search: these are what a database provides, and they are why most people who use cuVS never call it themselves. Vector databases, traditional databases, and search engines have all converged on the same feature set, and the vector search core inside each is built from the handful of algorithms this chapter has described, which is why one library can sit inside all three. Most integrations use the GPU only to build: a CAGRA graph is constructed, converted to HNSW, and served by the system's existing CPU search path. Table 2-2 summarises the integrations at the time of writing; version requirements, licence tiers, and supported index types change quickly, so treat each product's documentation as the authority.

Table 2-2. Systems that embed cuVS

System	GPU role	Route to cuVS	Notes
FAISS	Build and search, or build only	C++ core	Optional backend sinc May 2025 for the exist GPU flat and IVF clas adds GpuIndexCagra, IndexHNSWCagra for CPU serving
Milvus	Build and search; build only from 2.6.4	C++ core, via Knowhere	GPU_CAGRA, GPU_IVF_FLAT, GPU_IVF_PQ, GPU_BRUTE_FORC index types
Apache Solr	Build only	cuvs-lucene, on the Java API	Vector field algorithm cagra_hnsw
Elasticsearch	Build only	Java API directly	Node setting vectors.indexing.use_g hnsw and int8_hnsw index types
OpenSearch	Build only, on remote workers	Through FAISS	Serving cluster needs GPU; managed on Amazon OpenSearch Service

System	GPU role	Route to cuVS	Notes
Weaviate	Build only	Go bindings	Weaviate contributed t Go bindings and pooled/pinned memory work upstream; announced 2025, not generally available at t time of writing
Oracle AI Database 26ai	Build only, on a remote GPU container	Not documented	Vector Index Service builds the graph in a Private AI Services Container and returns i as an In-Memory Neighbor Graph index GA March 2026
Kinetica	Build and search	Not documented	CAGRA index on vect columns; HNSW rema the automatically maintained option for mutable data
KDB.AI	Build and search	Not documented	Separate kdbai-db-cuv server image bundles t GPU dependencies; standard client APIs unchanged

System	GPU role	Route to cuVS	Notes
KIOXIA AiSAQ	Build only	C++ core	DiskANN fork; uses th Vamana builder and k-means, not CAGRA; available as a disk inde in Milvus 2.6.4+

FAISS matters beyond itself because it is embedded in other systems, and OpenSearch reaches cuVS through it. FAISS users keep their existing code and get cuVS's implementations underneath, while cuVS stays free to evolve its own API; the trade-off is that FAISS exposes only part of what cuVS can do, so applications that need the full range of tuning parameters or the newest features are better served by calling cuVS directly. Milvus has the deepest integration and is the clearest example of the division of labour noted under multi-GPU search: it handles sharding and replication across machines itself and uses cuVS to make each node fast. The three Lucene-based systems all build on the GPU and serve on CPUs, but each reaches cuVS by a different route, as Figure 2-6 shows: Solr through the cuvs-lucene vector format, which any application using Lucene codecs can adopt; Elasticsearch through the Java bindings directly; and OpenSearch by moving the build out of the cluster altogether, to remote GPU workers that build a CAGRA graph through FAISS and hand back an index for CPU search.

Not every consumer is a database. cuML, and tools built on it such as BERTopic, use cuVS to build neighbour graphs that feed UMAP and HDBSCAN and never become a search index.

One caution applies to all of them. Once graph construction moves to the GPU it stops being the bottleneck, and something else takes its place: JSON parsing, the network, storage. NVIDIA reports Lucene index builds 40 times faster with cuVS and end-to-end Solr indexing 6 times faster. Both

numbers are true, and the gap between them is the point. Before attributing a speedup to cuVS, check what it measures.

cuVS ships on a two-month cadence and NVIDIA publishes no roadmap for it, so its direction has to be read from its releases, and the recent ones are telling. Out-of-core builds, ABI versioning with automated checks for breaking changes, static linking, arm64 packages, segment merging, and index formats that other engines can read are not features an individual data scientist asks for. They are what the builders of databases and search engines ask for, and they confirm the strategy this chapter began with: cuVS is an engine, built to sit inside other people's products.

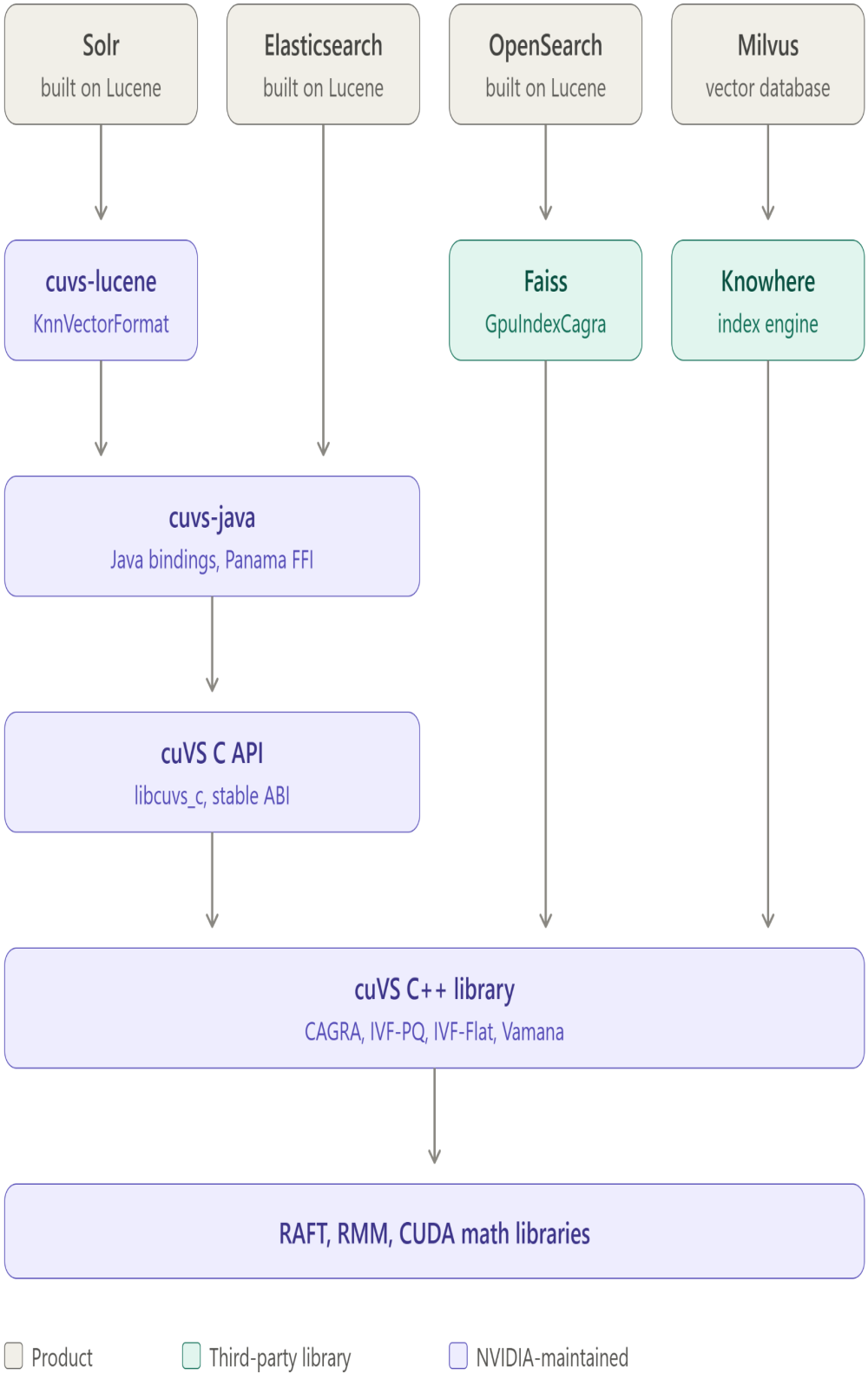

Solr
built on Lucene
Elasticsearch
built on Lucene
OpenSearch
built on Lucene
Milvus
vector database
cuvs-lucene
KnnVectorFormat
Faiss
GpuIndexCagra
Knowhere
index engine
cuvs-java
Java bindings, Panama FFI
cuVS C API
libcuvs_c, stable ABI
cuVS C++ library
CAGRA, IVF-PQ, IVF-Flat, Vamana
RAFT, RMM, CUDA math libraries
Product
Third-party library
NVIDIA-maintained

Figure 2-7. All roads lead to cuVS

About the Author

Max Buckley is the founder of Mechafauna, a robotics startup focused on optimizing AI inference for Nvidia edge hardware. Prior to this he had a brief stint as the Head of Knowledge Research at Exa, where he lead research on leveraging machine learning to improve retrieval and search infrastructure for the agentic AI era. Before Exa, he spent twelve and a half years at Google, finishing as a Senior Software Engineer, working on applied machine learning systems. He is an active open-source contributor to various projects including several of NVIDIA's. He hosts the Signal and Stories podcast, writes regularly on machine learning, GPU, and AI infrastructure topics for an audience of 32,500+ on LinkedIn. He is based in Zurich where he lives with his wife. He brings to this book a working practitioner's view of the NVIDIA ecosystem, direct relationships with the engineers building it, and the contributor's perspective of someone who has shipped code into it.