MANNING REPORT

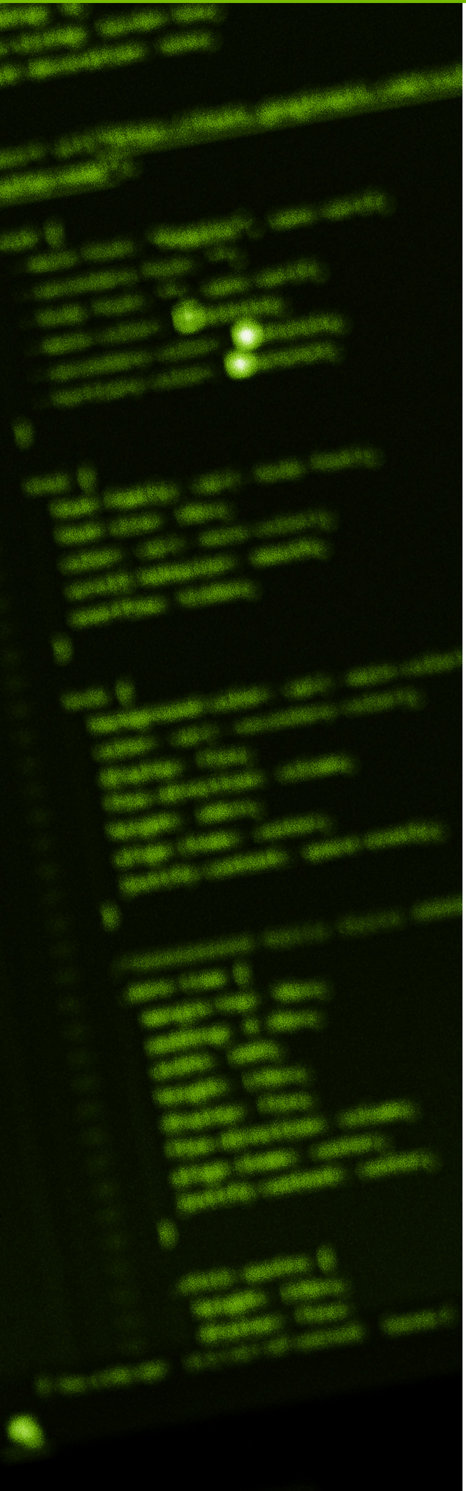
Hardware > Software > Process

Data Science in a Post-Moore's Law World

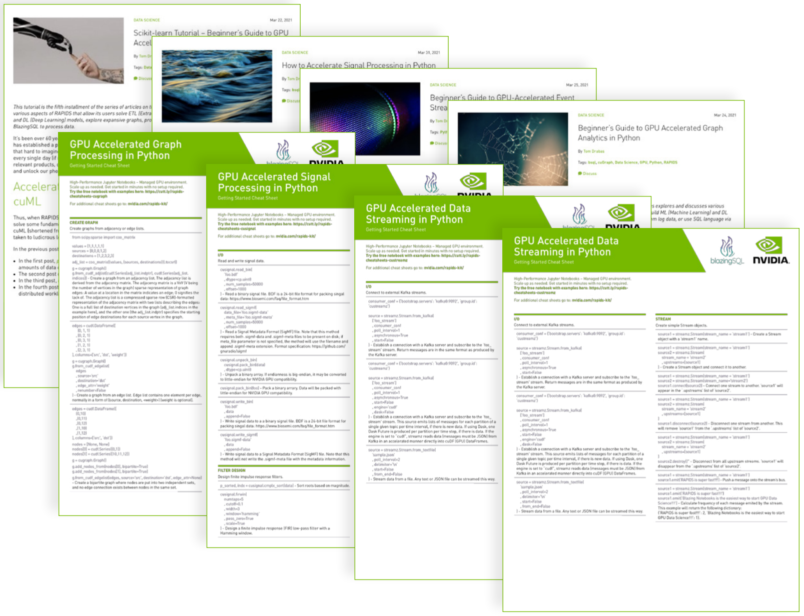
by Paco Nathan and Dean Wampler
with contributions from Jim Scott

MANNING + NVIDIA

Free Getting Started Kit for Accelerated Data Science



This kit includes **nine cheat sheets** and **nine tutorials** so you can learn how to effortlessly accelerate your Python workflows.



Scikit-learn Tutorial – Beginner's Guide to GPU Accel

How to Accelerate Signal Processing in Python

Beginner's Guide to GPU-Accelerated Event Strea

Beginner's Guide to GPU Accelerated Graph Analytics in Python

GPU Accelerated Graph Processing in Python

GPU Accelerated Signal Processing in Python

GPU Accelerated Data Streaming in Python

GPU Accelerated Data Streaming in Python

Getting Started Cheat Sheet

Access this kit to learn how to GPU-accelerate the following:

- DataFrames in Python
- DataFrames for Pandas Users
- Querying data using SQL
- Distributed Computing in Python
- Machine Learning in Python
- Graph Processing in Python
- Data Streaming in Python
- Processing Cyber Security Logs in Python
- Signal Processing in Python

Download Here: nvidia.com/rapids-kit/

Hardware > Software > Process

Data Science in a Post-Moore's Law World

PACO NATHAN
DEAN WAMPLER
with contributions by JIM SCOTT

MANNING
SHELTER ISLAND

contents

foreword

Since the turn of the century, I have found myself creating, consuming, and processing copious amounts of data. The idea of a terabyte of data was ludicrous, and this was well before the term *big data* came into existence. When big data came onto the scene, driven by Hadoop, not many solutions were available. Still, the community came together quickly to figure out how to apply this technology stack to business problems. The shift in technology stacks was enabled mainly by the Java programming language. Enterprise software was being developed with Java to target various hardware platforms and abstract the hardware.

At that time, I was embracing this technology stack and working in the digital mapping industry. I then began applying these technologies in the digital advertising technology industry, pushing boundaries only seen in that industry: upward of 60 billion advertising requests per day. I then turned to apply my experience to the consumer packaged goods retail analytics space. These technologies were very complex, and many found them unwieldy.

It was circa 2010 when I met Paco. I was a co-founder of the Chicago Hadoop Users Group (CHUG) and invited Paco to speak about an open source project, Apache Mesos. He explained the value of the software and how it would help drive better hardware utilization in the data center. Shortly thereafter, I came to know Dean, a physicist, who was applying his depth of knowledge at a hedge fund company. In our intersecting roles, we became collaborators and friends, and we all stayed tied to use cases in and around the big data application space. I place Dean and Paco in a group of people I associate with who, like me, are all lifelong learners—eager to push the boundaries and not afraid to fail, as long as we can learn a better way to apply technology.

Within just a few short years of big data taking over the technology sector, the term *data science* came into being: using data to drive a business, most commonly associated with in-depth data analytics, streaming data, and applying machine learning to impact a business.

Fast-forward to the COVID-19 pandemic. Paco delivered a keynote presentation at the Big Things Conference that made me realize there was an opportunity in front of us to drive a new era of education. I quickly brought up the topic with Paco to discuss the paradigm change that was underway. I escalated the conversation by bringing Dean into this story because he has experienced these same things and, quite frankly, three brains are better than two.

For the last 20 years, we have been abstracting away the hardware to make it easy to move our software between servers. However, the data science space has recently been driven by the Python programming language, not Java. Open source software like RAPIDS, and popular frameworks like PyTorch and TensorFlow, enable data scientists and software engineers to take advantage of new hardware architectures that deliver magnitudes of improved performance at a fraction of the cost.

We have grown accustomed to ignoring hardware and sticking with the status quo, changing the software as we go, and altering our processes every step of the way to scale out our solutions. In reality, this paradigm shift is offering us a new perspective. Instead of ignoring the hardware, we can embrace the advancements provided by the hardware architecture. This results in minimal or, potentially, no changes to the software and no change to the processes to support these systems. This approach simplifies the business's operations and unlocks magnitudes of additional productivity and value by flipping the traditional approach upside down.

We are at an inflection point in enterprise operations and all data science–related solutions. This report provides historical reference points and describes how we have arrived where we are today thanks to approaches like the agile methodology and co-development of hardware and software solution stacks.

I hope that every reader will take away a series of learnings from this report. At the top of that list is not to fear making changes at the hardware layer. The benefits are abundant; the new approach will provide further opportunities and reduce the amount of time to value for your business. Thirty years ago, people waited for code to compile. Now people are waiting around for everything data-related. This is the chance to take advantage of saving time: it is our most precious resource and most significant opportunity to create an advantage in a competitive industry.

This report will serve as a blueprint for taking advantage of all the new technologies to drive your business ahead of the competition. I you derive significant value from it. Paco, Dean, and I are here to help you through this journey.

—Jim Scott, NVIDIA

Hardware > Software > Process: Data Science in a Post-Moore's Law World

Software engineers have been taught the following:

- *Process* (such as Agile) is generic and equally applicable for all projects.
- *Software* emphasizes general-purpose approaches, largely agnostic about the hardware.
- *Hardware* is low-level and out of view, hidden behind abstractions, a resource that keeps getting faster, better, and cheaper, thanks to Moore's Law.

Organizations have historically planned for their needs, driven processes to meet those needs, implemented their business solutions in software, and let IT worry about the hardware. It's time to invert this perspective. Since around 2009, GPUs have been essential for deep learning, driving innovative breakthroughs like AlexNet in 2013, which was built on the CUDA API for NVIDIA GPUs. Simultaneously, the death of Moore's Law has pushed parallel distributed processing in other areas of data science and general services. Even general-purpose *system-on-chip* (SoC) designs from Qualcomm and Apple now included dedicated neural engines for *deep learning* (DL).

This report explores the hardware innovations that enable applications that would have been impossible without them. These innovations, in turn, demand rethinking how we write data science applications and the processes we follow to build them. In this report, you will learn:

- How hardware choices affect software performance
- What software idioms take full advantage of hardware acceleration, and which idioms undermine this acceleration
- How to refine development processes and leverage emerging trends to maximize business outcomes
- Which are the emerging opportunities where changing processes can take better advantage of these historic hardware innovations.

1 Backstory: Pythonistas, One and All

This report explores hardware innovations that enable previously impossible applications. They demand a rethinking of how we write data science applications and the processes we follow to build them.

In many ways, the Python language has become a *lingua franca* for work in data analytics (or data science—whatever you want to call it). The language is quite simple to learn and offers many popular libraries—pandas, scikit-learn, NumPy, PyTorch, spaCy, NetworkX, and others—that have become *de facto* standards for data science work.

Early development of popular Python libraries like SciPy focused on medical imaging use cases and related work in science. Since 2007, the Cython optimizing static compiler has provided a native interface for C and C++ code, which brought in sophisticated numerical libraries and allowed work in Python to get close to the hardware for optimal performance. These factors converged in some of the most popular use cases in data science—such as deep learning for computer vision—which relied on both the numerical libraries in Python and the ability to access the hardware.

One striking aspect of the Python language is how compact the source code tends to be. That generally makes Python code simpler to read and reuse, which is especially important for open source projects.

The Python language is also relatively forgiving for people who aren't expert at software engineering. It was made simple by design. While the Python 3.x releases have begun to incorporate more sophisticated programming language features such as *type annotations*, *futures*, and so on, the use of these is not *required*. The barrier to entry is low.

Given the accessibility of learning and using the language, many people have adopted Python for a broad range of popular use cases. An important point to understand is that the Python interpreter alone has always been relatively slow, especially for use cases that require lots of number crunching. The use of Cython and native libraries became important for data science needs early on since it allows for more effective use of the underlying hardware. Consequently, a dichotomy emerged where, on the one hand, Python provides easy-to-use, concise APIs; but on the other hand, some popular use cases need high performance. As we'll see in this report, *idiomatic* use of Python provides a bridge between the two and is crucial for making effective use of the hardware.

However, the idioms of Python and almost all other programming languages, including the languages used to create the native libraries, have assumed a hardware architecture typical for CPUs, especially x86 and its successors. Hardware is growing more heterogeneous, with GPUs firmly established for graphics processing first and then becoming necessary for the performance demands of data science, new CPU architectures like ARM, and even new kinds of silicon specifically for deep learning.

To that end, NVIDIA developed RAPIDS, a suite of data science libraries for GPUs. RAPIDS re-implements several open source libraries, popular in data science work, to make it much simpler to leverage GPU hardware for accelerating end-to-end data workflows. The objective is to provide these capabilities with minimal code changes and almost no new tools to learn. The APIs used in RAPIDS are designed to have a familiar look and feel for data scientists who typically work in Python. Under the hood, RAPIDS relies on several open source projects, notably

- Numba—An open source JIT compiler that translates NumPy calls and other Python code into fast machine code using LLVM
- Apache Arrow—A language-independent columnar memory format for efficient data operations on contemporary hardware, including GPUs

Combined, these provide support for *Multi-Node Multi-GPU* (MNMG) deployments. Key benefits include capabilities to allow processing and model training to scale up for the use of much larger datasets while enabling end-to-end pipeline accelerations with substantially lower serialization costs.

Let's step back for a moment to unpack this last statement. One essential challenge in data analytics is that when a problem grows bigger than can be computed on a single machine, what do you do? Since the mid-2000s, distributed frameworks such as Hadoop, Spark, and so on have provided open source solutions that use CPU clusters to handle these kinds of problems. Meanwhile, since about 2009, the use of GPUs has been growing in specific areas of machine learning such as deep learning for computer vision—where the use of CPUs would otherwise be slow. However, what do you do when an important application runs too slowly on CPUs but won't fit on a single GPU? In other words, how can you leverage a cluster of CPUs and GPUs? The popular data frameworks have never quite addressed this question. So RAPIDS provides a vital missing piece through MNMG architectures that uses both multiple GPUs and multiple CPUs. In our current era of AI enthusiasm, the co-evolution of hardware and software has become crucial to AI's success.

In our current era of AI enthusiasm, the co-evolution of hardware and software has become crucial.

Even so, the perception persists in industry that using GPUs is (1) expensive and (2) somewhat esoteric. While that was partially true in the past, recent advancements in both hardware and software have largely erased these barriers. Even though you may need to pay more per hour for a cloud instance that includes GPUs, in most cases, the overall costs will be lower by leveraging their hardware acceleration—there's generally a tipping point in terms of dataset size beyond which it becomes more expensive *not* to use GPUs. The economics of *total cost of ownership* (TCO) resolve the perception that GPUs are expensive. Figure 1.1 compares the speedup and costs of using a CPU versus a GPU for an example machine learning use case: *hyper-parameter optimization* (HPO) for training a large *random forest* model. Effectively, this case shows a 700% reduction in overall costs through hardware acceleration on GPUs, so the solution becomes *both* faster *and* cheaper. We see similar ROIs across a range of AI projects at scale. GPUs have become a ubiquitous HPC workhorse and value driver.

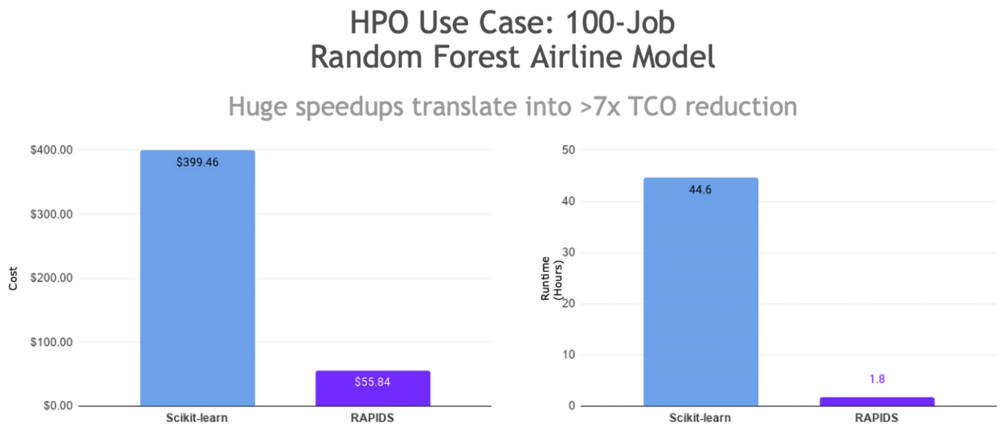


Figure 1.1 Example of reducing overall costs (700%) by using GPUs[1]

The perception that GPUs are esoteric devices to use is resolved by using RAPIDS and the evolving MNMG kinds of architectures. Meanwhile, using idiomatic programming in Python is necessary so that the underlying hardware can optimize these kinds of data workloads.

Using idiomatic programming in Python is necessary so that the underlying hardware can optimize these kinds of data workloads.

[1] Source: https://docs.rapids.ai/overview/latest.pdf. Based on sample random forest training code from the cloud-ml-examples repository, running on Azure ML: 10 concurrent workers with 100 total runs, 100 M rows, and fivefold cross-validation per run. GPU nodes: 10× Standard_NC6s_v3, 1 V100 16G, vCPU 6 memory 112G, Xeon E5-2690 v4 (Broadwell)—$3.366/hour. CPU nodes: 10× Standard_DS5_v2, vCPU 16 memory 56G, Xeon E5-2673 v3 (Haswell) or v4 (Broadwell)—$1.017/hour.

2 Early ML in Production: The Two Cultures

Let's roll back the clock to the release of Python 2.0 on 16 October 2000. Coincidentally, just a few months afterward, several leading software engineers met in Snowbird, Utah, to discuss lightweight software development methods, after which they published the *Manifesto for Agile Software Development*. What followed was a vast rethinking of software engineering practices.

Until that time, a naive *waterfall model*[2] dominated software engineering: projects started with carefully defined requirements, then an architecture and design were developed for the implementation, and then the code was written for the implementation. There was no concept of iterative feedback to improve the upstream artifacts. In contrast, the *Agile Manifesto* emphasized delivering results in small increments with frequent iterations, continuous learning, obtaining frequent feedback from stakeholders, and using minimally sufficient process steps—some pretty significant changes that impacted software itself.[3] For example, the prior emphasis on grand architecture and design shifted to more organic growth. Building software for all eventualities was flipped to "don't write the code until you actually need it."

These essential principles apply to a wide range of software work, but if you read the *Manifesto*, you'll notice that the word *data* is not used even once. That reflected the mindset in software engineering at the time. Subsequent generations of mainstream software developers have been taught that

Coding is pre-eminent; data is secondary.

But what about processes for data analysis projects? Later in the same year that the *Agile Manifesto* was published, UC Berkeley professor Leo Breiman wrote a controversial paper called "Statistical Modeling: The Two Cultures." This paper chronicled a sea change from an emphasis on building statistical models of data to a process of training an algorithm to simulate the data transformation process in question. The latter approach has come to dominate data science even though the paper came out long before that term became popular.

We'll need to back up a few decades to understand why Breiman's observations in 2001 created such a stir. Throughout the 1960s, 1970s, and 1980s, John Tukey and colleagues had promoted the idea of *exploratory data analysis*, where computing could be used to study a dataset—which represented a radical shift in thinking. For example, software could fit distributions or generate plots or other graphics much better than a person with pencil and paper. Even so, that kind of approach reinforced a *data model* view of the world: data was assumed to be generated by mechanisms that could be described by statistical theory. Statisticians spent their time proving that point,

[2] Waterfall became an industry standard after it was adopted as a required standard for software development by the US Department of Defense. Tragically, the standard ignored a crucial detail in the seminal paper by Winston Royce: the essential need for feedback loops. Hence, we used the term *naive*.

[3] Manufacturing industries made this transition long ago. In fact, many ideas in Agile are inspired by manufacturing, such as Toyota's Lean Manufacturing.

focused on the theory. Meanwhile, a completely different community of practitioners had emerged: they focused on *algorithmic modeling* instead, which relied more on empirical work than theoretical modeling. In other words, given a mountain of data at Amazon about customers who prefer to buy particular products, they use algorithms to predict "People who bought Xyzzy also bought Wubble" and then make that recommendation. Skip the part about proving any statistical theory regarding Xyzzy or Wubble since the predictions are required within milliseconds. These new practitioners leveraged *machine learning* algorithms, which worked fast and ignored much of the theory required for *statistical learning*.

Large-scale commercial successes with machine learning at Amazon, Google, Yahoo!, eBay, LinkedIn, etc., sealed the deal. Within the next several years, big data, data science, data engineering, machine learning, data privacy, and other aspects of data began to loom large. In contrast to the perspectives that followed from the *Agile Manifesto*, a quite different worldview emerged related to data practices:

Learning is pre-eminent; data is a competitive differentiator.

However, making coding pre-eminent is at odds with making learning pre-eminent. While there are areas of overlap, such as the emphasis on incremental delivery and continuous feedback, the code-centric agile processes have become canon among software developers, while the data-centric[4] model is how the tech "unicorn" companies have diverged so dramatically with first-mover advantage in AI.

Two decades later, we're mostly stuck between agile processes and learning-driven processes while the landscape has become more complex. Coding wants *agility* as well as *deterministic, repeatable* processes, such as automated, repeatable *CI/CD* (continuous integration/continuous delivery) pipelines. In contrast, data-centric approaches embrace *uncertainty* within the data and even take advantage of it through probabilistic methods such as machine learning. They also place a premium on what we could call *legibility,* by which we mean clarity that our interpretation and application of our data accurately reflects its inherent information and value. In other words, when your code runs and passes its automated tests, producing expected results, it's good to go. However, when you're working with data, you'll most likely transform that data in many ways through multiple stages of processing; nuances about meaning and interpretation tend to get obscured at each stage, bias may be introduced along the way, and even at the end of the workflow you still may not know what a "correct" result should be. Saying "Well, the program ran okay, didn't it?" is not enough for a data-centric approach. Instead, you must make sure the data stays *legible* (capable of being discovered or understood) at every stage of processing. The code and data views of the world especially don't agree on the latter part, and much of what we'll explore throughout this report is about improving how you work with data.

[4] Andrew Ng has a good presentation about the priority of "data-centric" approaches in MLOps, showing a contrast between efforts to improve code (much of the history of IT) versus the upside from efforts to improve data (better leverage).

Fortunately, the tech stack has evolved considerably over this period. While the authors of the *Agile Manifesto* were primarily concerned with source code and team process, and Breiman was mostly concerned with use cases of machine learning, neither had any clue that *cloud computing* would emerge within a few years. Nor did they foresee how rapidly the layers underneath all that coding and modeling would evolve. The emergence of Apache Spark, Dask, Ray, PyTorch, etc., has had an enormous impact on what can be performed with a few lines of Python code. Moreover, the hardware capabilities have completely changed: 2001 was a time when single-core CPUs ran on spinny disks (that were prone to failure) with small amounts of memory and relatively slow network connections, and gigabytes were considered large amounts of data. Parallel processing required highly specialized software tools and skills. Now, especially when it comes to machine learning work, a small fragment of code can make entire GPU clusters do backflips, moving through terabytes with ease.

Even so, "with great power comes great responsibility," and parallel processing doesn't come for free. A notion of *design patterns*[5] was introduced into software engineering in the 1990s and became an important source of reusable programming abstractions above the level of particular APIs. Similar kinds of design patterns emerged for working with data at scale and building data science workflows; however, each ecosystem's patterns are poorly understood by the other.

Fortunately, data scientists working in Python have developed their own effective design patterns for data that are often expressed as *idiomatic* programming practices. By using the proper design patterns, the underlying automation for data workflows can work optimally, such as leveraging the available hardware and distributed computing.

At the same time, software engineers have developed mature DevOps processes for creating automated, repeatable deployments of services with essential monitoring and management of deployed artifacts. An emerging consensus in the data world is the need to adopt and adapt these processes, called *MLOps* or *ModelOps*.

3 Evolution of Hardware and Software: When Moore Becomes Less

Another interesting thing happened during the same two decades: Java became a dominant programming language for enterprise applications. Along with that, widespread use of the JVM (Java virtual machine) kept Java source code an arm's length away from the hardware. The appeal was twofold: Java applications were portable across different hardware architectures and even operating systems, and the JVM "walled garden" provided an extra measure of security. Translated: we didn't need to worry about hardware since it lives way down at a lower level, and it will just keep getting better/faster/cheaper.

[5] The book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by the "Gang of Four" (Gamma, Helm, Johnson, Vlissides; Addison-Wesley Professional, 1994).

For a long time, Moore's Law—which says that the number of transistors in chips doubles every two years—has served as an approximate proxy for this performance growth, as shown in figure 3.1. But notice the transition that started around 2005, where single hardware thread (core) performance and clock speed began to level off, while the number of cores began growing to compensate. To continue improving performance, software applications written since that time have exploited concurrency and distribution over multiple cores and even multiple CPUs.

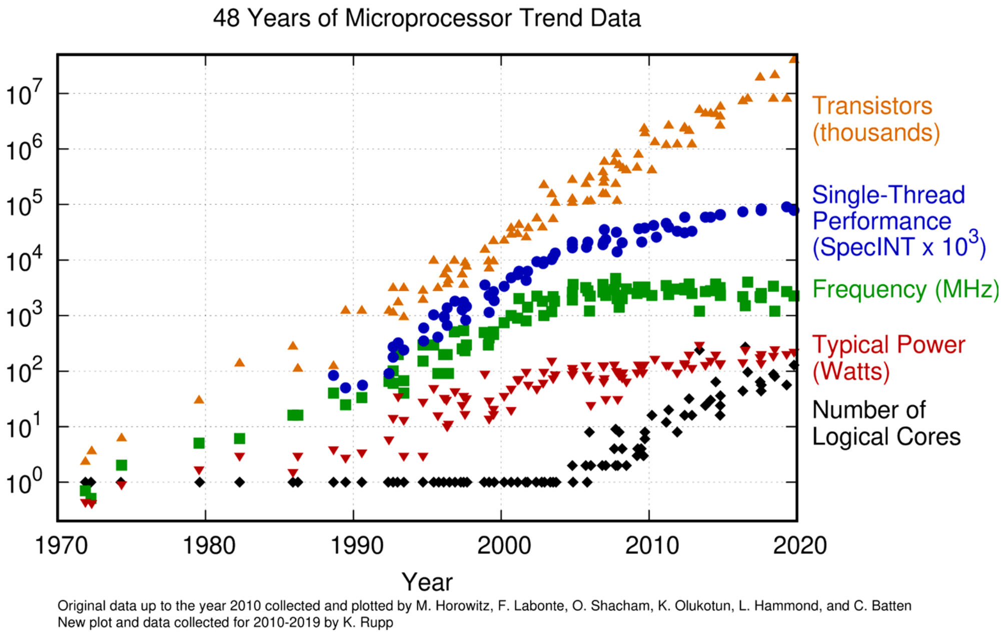


Figure 3.1 Moore's Law for microprocessors (CPUs): transistor count versus time. Source: https://github.com/karlrupp/microprocessor-trend-data.

However, the ever-increasing demand for optimal performance is outstripping Moore's Law. The computing requirements to train deep learning models continue growing at least 10 times as fast as Moore's Law. The breakthroughs in AI since the early 2010s have pushed the envelope for computing. Deep learning use cases in computer vision, speech, natural language, and other applications placed a premium on efficient access to hardware accelerators. Data rates required to train a machine learning model grew by orders of magnitude, as did the size of models in terms of their parameters. Hardware resources required to train large models grew at an even faster pace.

GPUs have become the essential hardware accelerators for meeting these requirements. The transistor counts are similar to those shown for CPUs, but the architectures are very different, with orders of magnitude more cores that are designed to run in groups called *warps* rather than each core running independently. For floating-point calculations, like those required for deep learning (as well as the original target application, computer graphics), GPUs can provide significantly more FLOPS (floating-point operations per second) than more general-purpose CPUs.

However, our software has to know how to exploit this advantage. Because of Python's ease of use, its active user community made machine learning more accessible, while at the same time, its ability to wrap calls to hardware-optimized libraries enabled performance to be maximized. Hence, Python became the dominant language for building these kinds of AI applications. This happened despite the JVM's dominance for other enterprise applications because it has been harder to integrate native-optimized libraries with the JVM's walled garden. The TIOBE Programming Community Index shows that the popularity of Python has grown markedly in the last three years or so. Python is now ranked as the third most popular language in the index, after C and Java. Ironically, the popular systems that grew dominant during this period have tended to be C/C++ under the hood: most of the PyData stack is Cython-based, which, among other benefits, makes it easy to integrate high-performance C/C++ libraries with Python code. Similarly, spaCy, TensorFlow, Ray, and RAPIDS also leverage C/C++ kernels.

Beyond being simply a function of model size, the requirements for nearly every stage of a typical data science workflow have increased dramatically:

- *Data preparation and curation*—Data preparation and curation rates have grown due to the need for larger training sets.
- *Feature engineering*—Feature engineering costs have grown as well, amplified by increased attention to fairness, bias, and privacy concerns that tend to be introduced at this stage, in addition to traditional data governance requirements (to track and control access to data).
- *Training and optimization*—Not only have model training and evaluation increased, but innovations in *AutoML* methods have amplified the computing needs at this stage, such as hyperparameter optimization.
- *Deployment*—Deployment constraints, such as *model distillation* for low-power use cases, use of federated learning and differential privacy for efficiency and privacy preservation, and other last-mile considerations also add to the compute budget for ML.

Hardware acceleration offers dramatic performance increases, which also reduce overall costs.

Meanwhile, hardware acceleration offers dramatic performance increases, which also reduce overall costs. Going forward, because of the growing requirements of production workflows, effective machine learning work in both research and industry will rely on making more effective use of hardware. That, in turn, will require code to be written in ways that hardware can optimize.

4 Building Data Workflows: Thinking Sparse and Dense

To effectively use machine learning, it's important to understand the kinds of data that is typically used with machine learning models and how data characteristics affect performance. There may be *structured* data, as in what gets managed in a database. There may also be images and video, and possibly audio—which are much less structured. There's probably some *machine data*: somewhere in the late 1990s, the world reached a point where most of the data online had been created by machines, such as log files. Also, there's lots and lots of text, because humans tend to communicate with each other (and sometimes with machines) through text.

In contrast, at the level of hardware acceleration, we typically work with *numeric data* that's organized in vectors, matrices, and tensors. As we build data workflows to produce and evaluate machine learning models, the different kinds of input data require different kinds of processing. Much of this work is about transforming data down to the numeric matrices the machines want and back again to representations that people can comprehend. Figure 4.1 illustrates how it all begins, with typical sources of data and their basic characteristics.

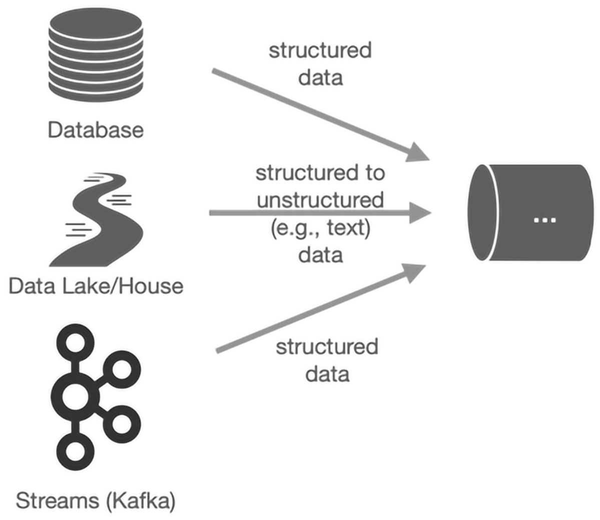


Figure 4.1 Different ingestion options, feeding data into the left-hand side of a data pipeline

In a data-centric *process*, the aim is to write your *software* to leverage the *hardware* most effectively. When working with data, there are a couple of important ways to think about this problem at scale: one way is *sparse*, the other is *dense*, and you need to know when to use each to construct effective workflows. Early stages of a workflow tend to be more sparse and use more symbolic representations—which are closer to the business use case, closer to what the stakeholders recognize, and largely divorced from representations that hardware uses internally. This kind of processing tends to happen in CPU-land and is usually less compute-intensive, although your mileage may vary. The downstream stages, such as training and evaluating ML models, tend to be more dense and also require numeric representations—which are closer to the hardware, reflecting more how the data is processed at scale as vectors, matrices, and tensors, with a less obvious connection to the business use case. This kind of processing typically can be optimized by the use of GPUs, although not always.

In other words, we take raw data from real-world sources, which may be a collection of text documents, but by the time we start to train a machine learning model, this data must be transformed into densely packed matrices of numbers. Once a ML model gets deployed in production, its payload of customer input and inferred results must be transformed back to the real world of symbols that customers can understand: instead of returning a list of product ID numbers, your ecommerce website probably spells out recommendations as a narrative, such as "People who bought book X, also bought book Y and book Z."

While programming languages are quite good at converting between symbols and numbers, they're not especially good at distinguishing the sparse and dense needs within a workflow. Meanwhile, the hardware is mostly focused on the denser parts. So to leverage the hardware, as part of the team process, we must apply the sparse and dense approaches in a workflow at the appropriate places.

Let's step back for a moment to consider an idealized data workflow. The typical steps are as follows:

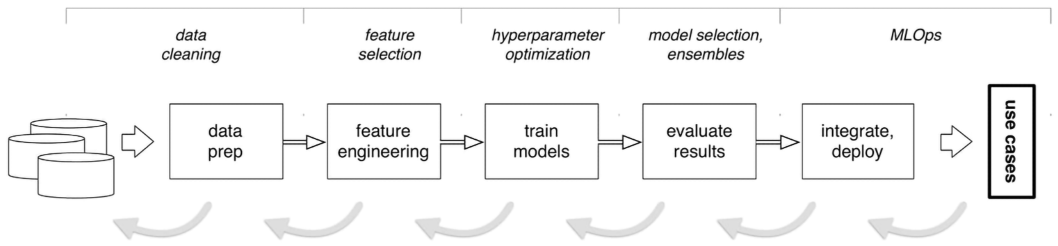
1. Data preparation
2. Feature engineering
3. Training models
4. Evaluating results
5. Testing and validation
6. Integrating/deploying models into use cases
7. Monitoring, measuring, making inquiries

In addition, workflows must leverage feedback at each stage—for example, to improve the quantity and quality of labels, clarify which features or embeddings to use, resolve issues of introducing bias, and so on. Figure 4.2 illustrates this process.

Recognize that machine learning takes advantage of the structure it finds within training data—in other words, the patterns. Making these patterns more evident in the data is what the early stages of a workflow are all about. Overall, the general process in data science involves the transition from relatively unstructured data sources into more refined data representations, which are more structured.

Data Science Process

unstructured data → structure
managing and leveraging dimensionality



Of course, these feedback loops are where we apply so much of the science and engineering. These feedback loops are also where **graph-based data science** provides powerful tooling.

Figure 4.2 Typical stages in an idealized data workflow

Typically, within a workflow, we focus on *data preparation* during the early stages, working with one or more datasets. You'd rarely ever use raw data to build a machine learning model. Instead, we transform data to build *features*[6] for our training sets, for a few good reasons:

- Different business logic often operates on different data sources.
- Improving the predictive power of the resulting model (accuracy, recall, etc.).
- Avoiding biased inferences (fairness/bias).
- Increasing the stability[7] of training different models over time as data for use case changes (MLOps issues).
- Optimizing to reduce the costs of training and inference.

Often the data preparation requires transformations: join, filter, aggregate, etc., which may commonly be handled through SQL queries on a database. There may also be a need for *dimensionality reduction*, shifting the problem space from sparse data in a large number of dimensions (i.e., the real world) to dense data in fewer dimensions. For example, an insurance workflow may need to make decisions about 100 million

[6] Chris Albon provides an excellent tutorial about feature engineering and related work at https://chrisalbon.com/#machine_learning.

[7] For a good introduction to an important part of ML that's not widely implemented yet, see the frequently cited paper "On the Stability of Feature Selection Algorithms" by Sarah Nogueira, Konstantinos Sechidis, and Gavin Brown, *Journal of Machine Learning Research* vol. 18, pp. 1–54, 2018, www.cs.man.ac.uk/~gbrown/stability.

documents, from 20 million customers, involving 300 different domain-specific terms and conditions; however, within the workflow, the problem gets reduced to 12 different customer segments with a clustering algorithm. The workflow's dimensionality has been reduced dramatically, often using what's called an *embedding*—in other words, a relatively low-dimensional space into which you can translate high-dimensional vectors.[8] That makes it simpler to train ML models with large inputs, such as *sparse vectors* representing sets of words. This is one example of data preparation or *preprocessing* work to refine the data so that it's more readily usable by algorithms and hardware.

Definition: Embedding

A common practice in data science where data points in a sparse high-dimensional space are translated into a denser low-dimensional space through a process called *dimensionality reduction*. The effects tend to create better predictive power for machine learning models, for example, by clarifying the features used to train models.

Exploiting the property of *sparseness* in data is what people tend to think about for big data approaches—finding the proverbial needle in a haystack. A range of common approaches use a general category of *non-negative matrix factorization*:[9] clustering algorithms, principal component analysis, singular value decomposition, autoencoders, and so on. To understand why this principle of sparsity is important, consider that probably no Netflix customer has rated every film on Netflix—but Netflix still needs to be able to recommend new films based on what it knows about what everyone else liked. If real-world data were more evenly distributed and symmetric, then building a really good recommender system could be a simple matter of statistics. Instead, content-recommender systems in ecommerce, social networks, genomic analysis, antifraud anomaly detectors, and many other big data applications are informed by the asymmetry of their underlying data and relations. We hear terms such as *power law, broad-tailed*, and *Zipfian* to describe the highly skewed distributions that are often encountered in the wild and are responsible for much of the sparseness found in real-world data.

Figure 4.3 illustrates what we mean. The sparse matrix for the actual data on the left has many zeroes. The two computed matrices on the right are dense, with few if any zero cells. When they're multiplied together, the resulting 4 × 4 matrix is approximately equal to the original matrix on the left.

In fact, Texas A&M hosts a "museum" of known data patterns in sparse matrices, called the SparseSuite Matrix Collection (https://sparse.tamu.edu/interfaces). These patterns have been used over the years to help determine how to optimize GPU hardware.

---

[8] For example, tSNE and UMAP are common methods for visualizing this kind of translation. For a good overview, see https://umap-learn.readthedocs.io/en/latest/how_umap_works.html.

[9] See https://en.wikipedia.org/wiki/Non-negative_matrix_factorization.

$$\begin{bmatrix} 0 & m_{12} & m_{13} & 0 \\ m_{21} & 0 & m_{23} & 0 \\ 0 & m_{32} & 0 & m_{34} \\ m_{41} & 0 & 0 & m_{44} \end{bmatrix} \approx \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \end{bmatrix}$$

Figure 4.3 Sparse versus dense representation of matrix data[10]

The key takeaway here is to *recognize the kind of processing needed in the early parts of a workflow and perform it there—don't conflate it with other work later in a pipeline.* Refine your data so that it's ready for number crunching by hardware, but also have well-understood ways to transform it back into something you'll be able to interpret downstream. This latter point becomes especially important for human-in-the-loop, explainable AI, rules based on domain expertise, compliance audits, and so on.

As development and deployment processes have matured, a boundary has emerged between the processing steps leading to the creation of dense, usable data for model training and scoring versus the downstream processes. A new category of framework called a *feature store*[11] is emerging as a boundary where the dense data begins to be represented. Data engineering teams often take responsibility for developing the repeatable *MLOps* processes that process the sparse data into features, ending at the feature store. The data science teams define the set of features and continue developing models with this data, but production model training and serving also become part of the MLOps pipeline developed and managed by data engineers.

After progressing through the sparse processing stages—once the dimensions have been reduced and the features have been prepared—the challenges of distributed processing shift focus to the *density* of the data, generally working with *numeric* representations. Approaches used for graph algorithms, deep learning, visualization, etc., generally all require numeric representation. This is where "thinking dense" becomes important.

A good example is the PageRank algorithm used to rank web pages based on which other pages link to them. Figure 4.4 is a sketch of the entire process. Web crawlers download copies of web pages and preprocess them to extract their links in the form of an *adjacency matrix*, along with the most significant words for each page.

[10] In a real dataset, the number of total cells on the right-hand side would be dramatically smaller than the number on the left-hand side.

[11] A feature store is a centralized storage where the features used within an organization to train ML models are curated. This popular approach was introduced by the Uber Michelangelo project in 2017. For a good overview, see "Feature Store: The Missing Data Layer in ML Pipelines?" by Kim Hammar and Jim Dowling, and also "Meet Michelangelo: Uber's Machine Learning Platform" by Jeremy Hermann and Mike Del Balso.

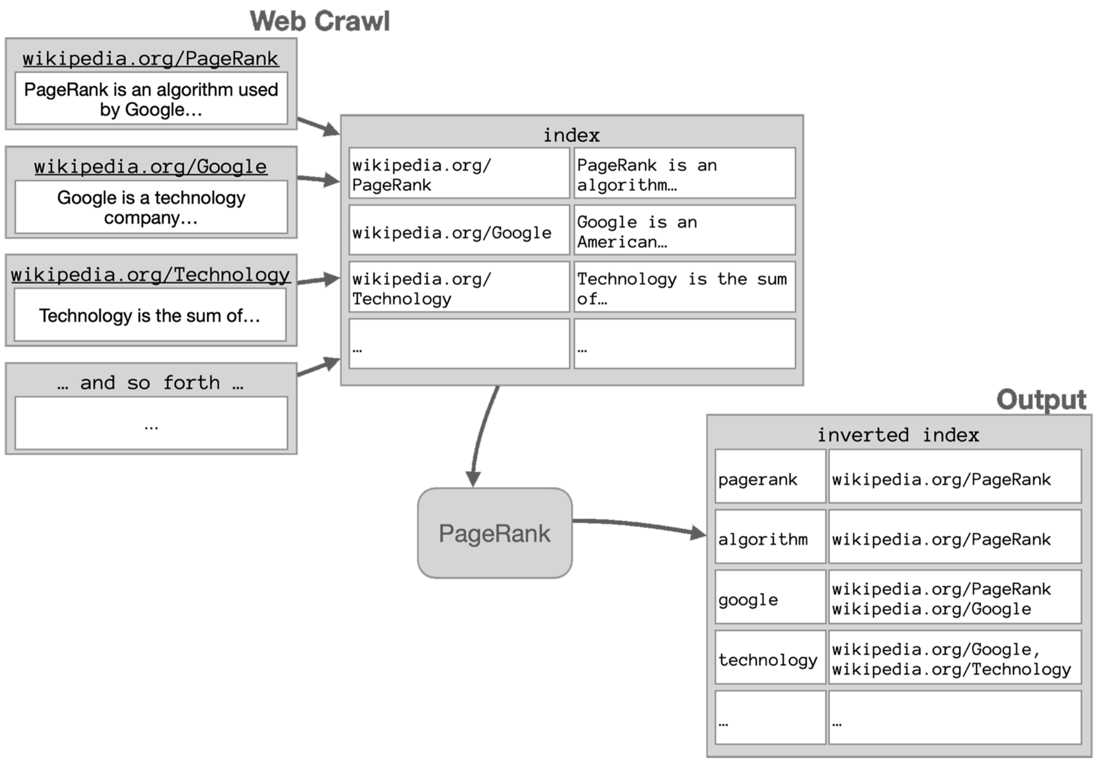


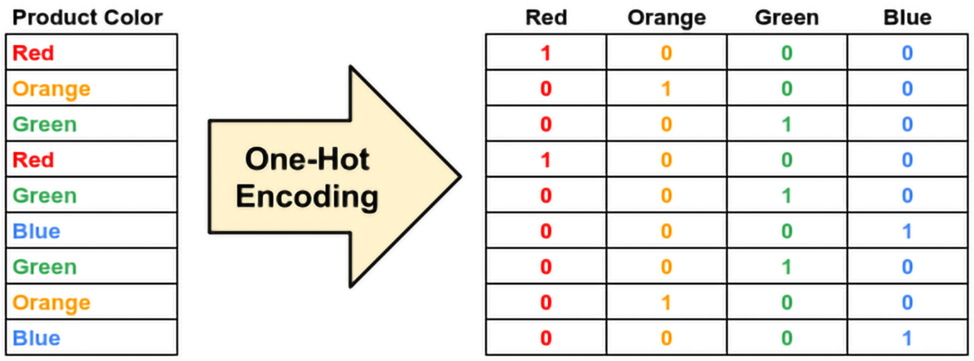
Figure 4.4 Example of input data and output data in a PageRank workflow

Significant words are typically sparse (occurring on just a few pages), while uninteresting words like *the*, *a*, *he*, *she*, etc. (also known as *stop words*) are dense, appearing on most pages. The PageRank algorithm uses the structure of the adjacency matrix to compute a score for each page based on how well-connected it is. Finally, the output of the PageRank algorithm and the most relevant words for each page are combined to generate an inverted index that maps each word to all the pages that contain it, sorted by their respective PageRank values.

The general pattern employed here can be summarized as follows:

1. Join lots of data (sparse, symbolic).
2. Prepare features (reduce dimensions, transform to numeric).
3. Number-crunch on a dense, numeric chunk of data—which can run fast with hardware acceleration.
4. Apply an inverse-transform to map the calculated results back to the sparse/symbolic space for use cases.

Examples of these patterns for data workflows include processing text input for scikit-learn pipelines and using `LabelEncoder`, where each string is encoded into a unique number, after which the processing runs on the dense, numeric representation. *Embedding* approaches[12] used with PyTorch for deep learning input are similar. Figure 4.5 shows a common form of encoding for deep learning, where a single feature with a fixed number of labels is transformed into columns of 1s or 0s; this is called *one-hot encoding*.



Product Color
Red
Orange
Green
Red
Green
Blue
Green
Orange
Blue

Red	Orange	Green	Blue
1	0	0	0
0	1	0	0
0	0	1	0
1	0	0	0
0	0	1	0
0	0	0	1
0	0	1	0
0	1	0	0
0	0	0	1

Figure 4.5 Example of one-hot encoding,[13] used for input data into neural networks in PyTorch

The next several sections about *process* depend on keeping a clear distinction between sparse and dense in your data workflows.

5 The Disconnect Between Software Abstractions and Implementation

A core part of the *hardware > software > process* problem boils down to this: while our programming languages have data types for representing *strings*, *floats*, *lists*, etc., they don't provide good ways to distinguish between "Run this part as sparse, symbolic data on distributed software" and "Now let's shift to dense, numeric arrays that must run on GPUs." Moreover, we've evolved software engineering practices that de-emphasize the data and discourage working directly with the hardware. Applying this practice across the board, circa 2021, can be an expensive mistake.

A disconnect exists between the abstractions that we tend to use in data workflows and how these are implemented in data frameworks. When people use the software abstractions improperly, either the workflow runs slowly or it doesn't run at all. In other words, the hardware—and the many layers of software in between the hardware and your software abstractions, such as the C++ in TensorFlow, spaCy, Ray, RAPIDS, Legion, etc.—doesn't have any opportunity to optimize the required work.

[12] For a good overview, see "Screening and Stabilizing Learned Image Manifolds" from Deep Learning Analytics.
[13] For example code, see "One-Hot Encode Nominal Categorical Features" in *Machine Learning Flashcards* by Chris Albon.

For example, is the representation of our data conducive to optimal performance as the hardware processes it at each stage of the workflow? Are we making assumptions about our algorithms that undermine hardware acceleration? People think much differently about *how* to process their data than distributed processors do. Consider figure 5.1, which illustrates schematically how data is moved from memory to the cache in a CPU.

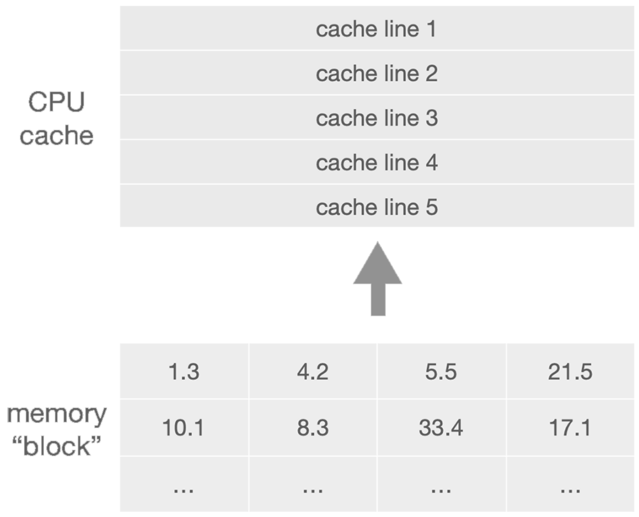


Figure 5.1 How a CPU looks at data

For example, due to fast CPU clock speeds, the time it takes to fetch data from memory takes tens of clock cycles, during which the processor core may sit idle. This means data should be structured in memory such that each record can fit on a single CPU *cache line* (typically 64 bytes); and larger chunks of data should be contiguous in memory so the cache can be kept hot with data just before it is needed, keeping the CPU cores busy. The best chunk size probably won't be apparent at the level of code in Python—or in C++, Java, etc. Moreover, we shouldn't assume that a compiler will make good decisions about this. Instead, we need to rely on other tools outside of the programming languages to determine how to optimize data workflows.

A related problem affected the performance of Spark on the JVM. The Java memory model is ideal for heterogeneous graphs of data, shown schematically as Before in figure 5.2. However, each traversal of an arrow requires a memory read to bring the item into the cache. When you have *billions* of structurally identical records, all that data movement per cache is wasted. It is far more efficient to use a compressed, contiguous representation, shown schematically as After, where one memory fetch brings the whole Person record or a sequence of records into the CPU cache. Spark

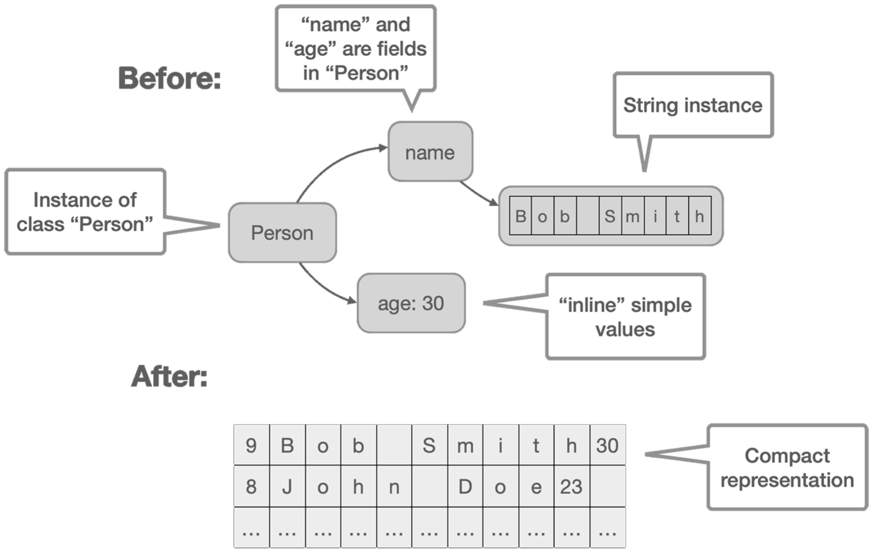


Figure 5.2 Before and After memory model in Spark

implemented its own memory manager so it can use a compressed representation like this, bypassing Java's memory model and greatly improving performance.

In several key ways now—most certainly within the space of AI applications—the hardware has been evolving more rapidly than the software, and the software has been evolving more rapidly than the processes we use to build it. How do we flip that around? How do we teach software engineers to leverage design idioms and processes that make the most of the hardware, because of the importance of how we need to use the data? And in turn, how do we teach the software to make the most of the hardware underneath it?

Fortunately, this is exactly where the *idiomatic* parts of Python come back into the picture, smack dead center in the middle of the camera lens's view. Idiomatic Python is not pedantic; instead, it plays an important role for two big reasons:

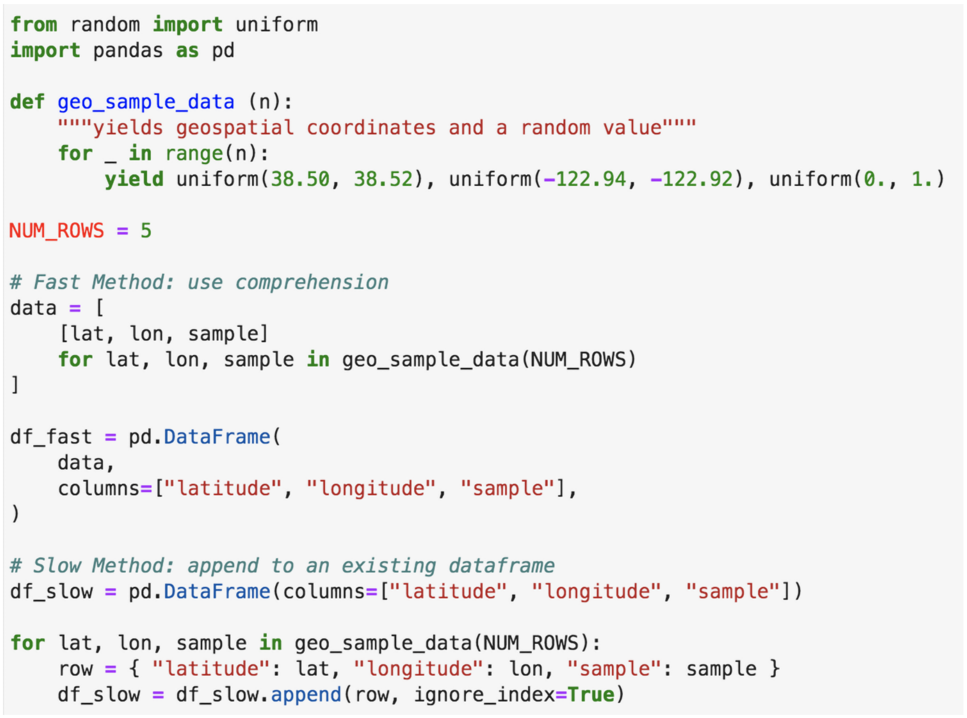
- *Software quality*—Enabling the tools to prevent or at least catch subtle errors
- *Performance*—Allowing the underlying Cython, C++, OS, Arrow, Dask, Ray, sklearn pipelines, etc., to do their magic to optimize pipelines for particular hardware.

Definition: Dataframe

One of the most commonly used objects in data science workflows for data organized as rows and columns. In other words, this is a two-dimensional, labeled data structure[14] where the columns may represent different types of data—similar to a spreadsheet or a SQL table.

[14] See https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html#dataframe.

Let's consider a question that shows up frequently on StackOverflow, where someone working in Python wants to use a pandas dataframe for a data science project. They've been taught to iterate through data using loops, so a natural conclusion is to build a dataframe within a loop—except that, as it turns out, Python runs as slow as molasses when you do that. Spoiler alert: the trick is to use a *list comprehension*, building a list of dictionaries first—as a dense representation of the rows you want in the dataframe. Then use the constructed list as input so that pandas can build the dataframe efficiently instead of appending one row at a time. It's a subtle nuance and a matter of idiomatic programming in Python, as shown in figure 5.3.



```python
from random import uniform
import pandas as pd

def geo_sample_data (n):
    """yields geospatial coordinates and a random value"""
    for _ in range(n):
        yield uniform(38.50, 38.52), uniform(-122.94, -122.92), uniform(0., 1.)

NUM_ROWS = 5

# Fast Method: use comprehension
data = [
    [lat, lon, sample]
    for lat, lon, sample in geo_sample_data(NUM_ROWS)
]

df_fast = pd.DataFrame(
    data,
    columns=["latitude", "longitude", "sample"],
)

# Slow Method: append to an existing dataframe
df_slow = pd.DataFrame(columns=["latitude", "longitude", "sample"])

for lat, lon, sample in geo_sample_data(NUM_ROWS):
    row = { "latitude": lat, "longitude": lon, "sample": sample }
    df_slow = df_slow.append(row, ignore_index=True)
```

Figure 5.3 Example code for comprehension (fast) versus comparing loop (slow) approaches for building a dataframe in Python[15]

Once you reach 10,000 or more rows, this idiomatic approach runs two orders of magnitude faster—just in software. The idiomatic approach also allows you to run the cuDF library in place of pandas and then run *much* faster on GPUs.

[15] Running this example on a CPU shows a 17× difference in execution times for a list comprehension versus using a loop. For extended analysis and discussions, see https://stackoverflow.com/questions/10715965/create-pandas-dataframe-by-appending-one-row-at-a-time.

Overall, this is an example of where features for distributed processing and the hardware to support them have evolved more rapidly than programming languages. Programmers write loops all the time. It's *what you were taught to do*, but that habit often undermines performance in a world of data at scale, where parallelism and distribution are the only competitive path forward.

A good point to keep in mind is that in data science, we generally *are not* writing programs anymore; we're integrating stages in pipelines.[16] This is great news for productivity and reliability. Writing low-level, bit-twiddling code is error-prone and time-consuming. Being able to integrate mature, powerful pipeline segments to solve our problems is an important advance. Hence, to resolve the disconnects between software abstractions and their implementations, we must write the code well enough that the underlying software frameworks and hardware accelerators can recognize design patterns for distributed processing and take advantage of them. Fortunately, there are tools and best practices to help with precisely that.

An example of this change is the evolution of the Spark APIs. In the initial *RDD (resilient distributed dataset) API*, the user chains together transformations inspired by *Scala's collection API*. However, when the *Catalyst optimizer* was introduced, users were encouraged to migrate to the newer *Spark SQL and related APIs*, where fewer assumptions are made about how results are computed, allowing Catalyst the freedom to use more aggressive optimizations. When you have an optimizer available, use it.

6 Key Abstractions

Let's briefly discuss a few data abstractions that aren't represented natively in programming languages but are essential for using hardware efficiently. If our programming languages supported these constructs natively, they could handle the required best practices for us to optimize performance. Instead, we have to apply a little discipline to use the right libraries in idiomatic ways, allowing those libraries to handle the optimal behaviors for us.

There are open source implementations in Python for each of these popular abstractions. RAPIDS provides a suite of open source libraries that build upon these popular abstractions to optimize for leveraging GPUs—for the dense portions of the workflows, as shown in the following table:

Abstraction	Python libraries	RAPIDS implementation
dataframe	pandas	cuDF
graph	NetworkX, iGraph	cuGraph
tensor	PyTorch, TensorFlow	NVTabular
ML pipeline	scikit-learn	cuML

[16] Chris Ré at Stanford has a really good talk about this point, "How Machine Learning Is Changing Software": https://youtu.be/45lqfnDM9Kw.

Following these abstractions helps reduce the *cognitive load* for an individual and helps a team collaborate on best practices.

6.1 Dataframes (When Everything Is a Table ...)

Perhaps the most common way to think of data is as a table, with rows and columns. Visually, tables look like rectangles, matrices, grids, cuboids, and so on. Spreadsheets, which we use daily, are the most common example. So are reporting tools and relational database tables with SQL. This form is ubiquitous, and it's surprising that programming languages don't support it directly.

However, it's not surprising that pandas, which launched in 2008, became wildly popular. The term *data science* was just starting to gain attention in industry. The dataframe objects that form the basis for pandas became a core concept for Python data science workflows. Otherwise, we were all writing custom code, over and over, to accomplish that—which pandas abstracted into a standard practice.

As a foundation for matrix operations, pandas is built atop NumPy—which was built with vector operations and hardware optimization in mind. In turn, lessons learned from using pandas led to a subsequent project: Apache Arrow,[17] mentioned earlier. Arrow uses an optimized C++ engine for efficient, in-memory *columnar storage* and manipulation of data objects independent of the programming languages and frameworks involved. The ongoing integration of Arrow and pandas enables much more efficient memory use than a pure Python implementation would allow. Arrow also provides APIs for many other languages to allow for cross-language tech stacks on the same shared memory. Arrow provides a reusable, memory-efficient storage system, akin to aspects of the shared memory described in the original Spark paper.

Figure 6.1 shows how Apache Arrow reduces the amount of system resources that are wasted simply through copying objects in memory back and forth across different frameworks. This helps resolve bandwidth bottlenecks. The practice of Unified Memory extends this approach, reducing the need to copy objects between CPU memory and GPU memory.

[17] This 2017 blog post by Wes McKinney explains many of the details, with additional project history: https://www.dremio.com/origin-history-of-apache-arrow.

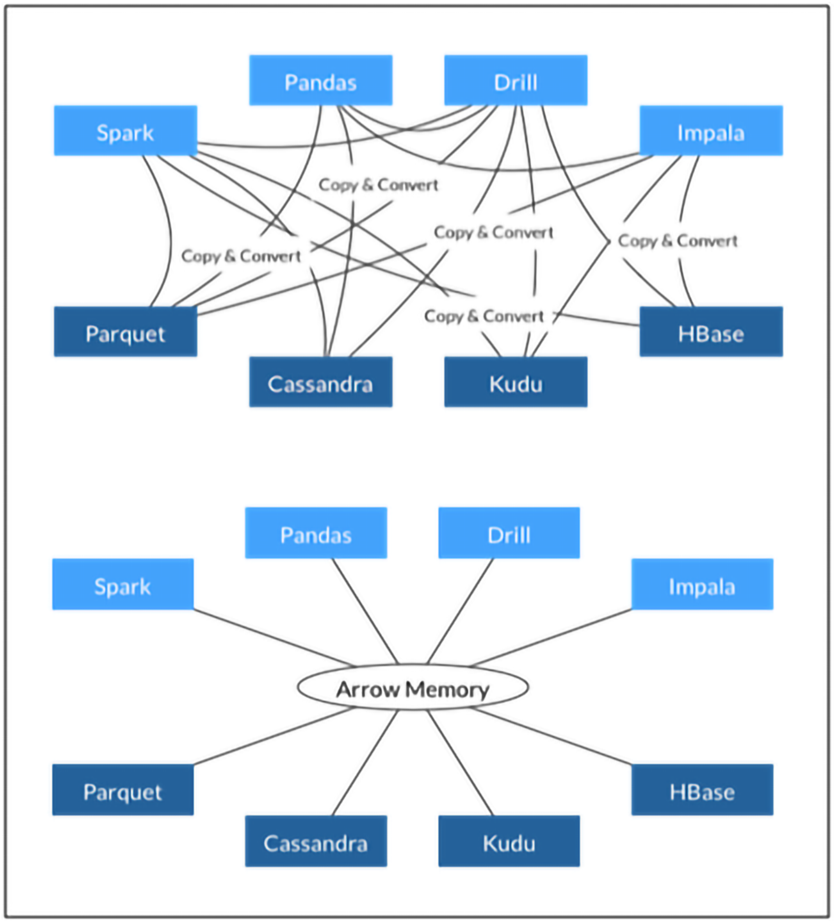


Figure 6.1 **Apache Arrow reduces copying memory back and forth.**[18]

As a complement to Arrow's in-memory representation, Apache Parquet provides a standard serialization format for columnar persistent storage along with support for features such as *predicate pushdown*,[19] which are widely used for optimizing queries in big data frameworks. In other words, only the columns required for a given query need to be decompressed and deserialized—which leads to significant performance gains. Parquet is leveraged by cuDF for fast data ingestion. The Parquet serialization format provides highly efficient columnar storage on disk, which maps closely to the needs of hardware optimizations and distributed software.

[18] Copied from the Apache Arrow overview page: https://arrow.apache.org/overview.

[19] For an overview, see "Understand predicate pushdown on row group level in Parquet with pyarrow and python" by Peter Hoffmann.

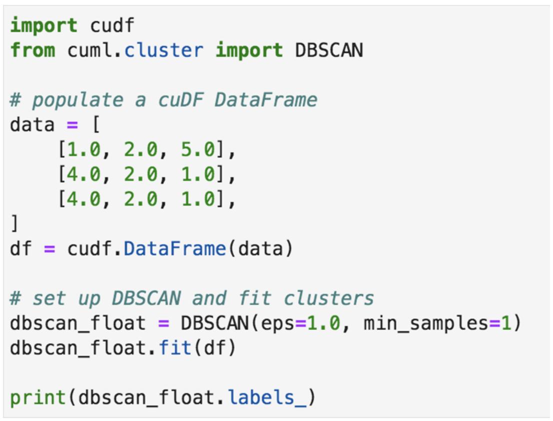
Another popular tool called Dask scales workflows[20] beyond single machines to run on clusters, even for work with extremely large datasets. This integrates closely with pandas, NumPy, scikit-learn, etc. Modin provides another abstraction that leverages Dask or Ray to scale pandas workflows to run on clusters. These kinds of workflow tools are highly recommended. They provide design patterns for the case where the processing would not fit on a single processor and must be distributed. In doing so, they make effective use of hardware while hiding the complexity from the user.

To make the most of your hardware for dataframes, use the cuDF library, which is foundational for RAPIDS and provides a GPU-optimized version of pandas. This fits well into the typical uses for pandas dataframes such as scikit-learn, Dask, etc. For the code repository, user documentation, and blog articles, see the following:

- https://github.com/rapidsai/cudf
- https://docs.rapids.ai/api/cudf/stable/
- https://medium.com/rapids-ai/tagged/dataframes

Dataframes in cuDF are built atop Apache Arrow's columnar memory. This is the core abstraction used throughout RAPIDS. By definition, this is the dense matrix of data in its numeric representation.

A key point to keep in mind is that while it's possible in Python to iterate through a pandas dataframe, it's horribly inefficient. In cuDF, the intent is to steer developers away from these kinds of *antipatterns*. So when you try to get away with a bad practice, cuDF will either prevent it at the API level or use the existing exceptions in pandas to communicate advice back to developers at runtime. Figure 6.2 shows an example of idiomatic cuDF, which is concise and allows cuDF to optimize performance internally.



```python
import cudf
from cuml.cluster import DBSCAN

# populate a cuDF DataFrame
data = [
    [1.0, 2.0, 5.0],
    [4.0, 2.0, 1.0],
    [4.0, 2.0, 1.0],
]
df = cudf.DataFrame(data)

# set up DBSCAN and fit clusters
dbscan_float = DBSCAN(eps=1.0, min_samples=1)
dbscan_float.fit(df)

print(dbscan_float.labels_)
```

Figure 6.2 Example code loading scikit-learn with a dataframe

[20] Dask-on-Ray is being used in petabyte-scale workloads at Amazon, etc.: https://docs.ray.io/en/master/dask-on-ray.html.

If you find that some coding logic *really* seems to need a loop, instead use a user-defined function (UDF), which can be applied to cuDF dataframes. This approach borrows from *functional programming* and extends the range where hardware optimizations can apply. Moreover, cuDF uses Numba for JIT compilation of UDFs.

Figure 6.3 shows typical performance gains when you allow RAPIDS to do the optimizations for pandas-based applications. Even on CPUs, looping through data yourself limits processing to a single CPU core, one datum at a time.

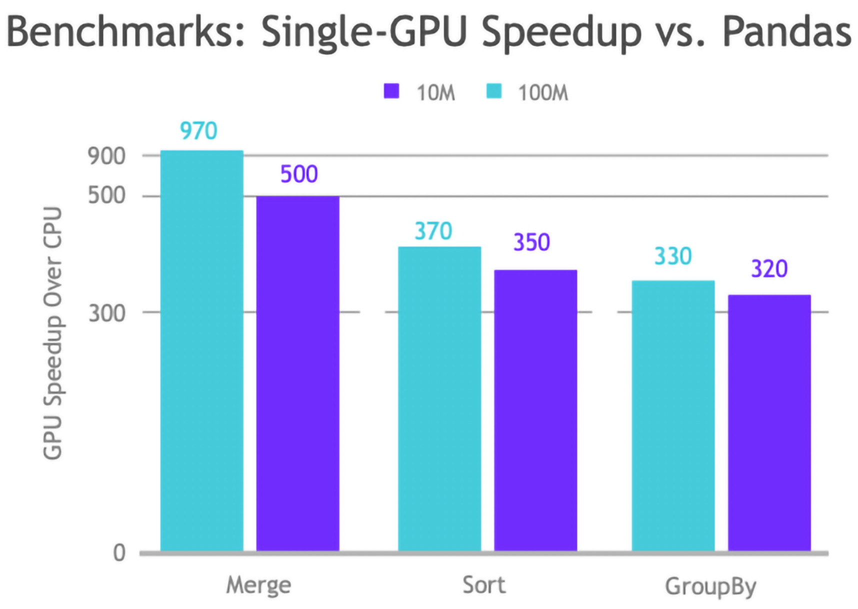


Figure 6.3 Speedup benchmarks using cuDF and GPUs[21]

Another key point is that Dask understands at a workflow level how to optimize across different workflow stages; this wouldn't be possible at the cuDF level of the stack. Dask-cuDF extends Dask where necessary so that its dataframe processing uses cuDF to leverage GPUs. Meanwhile, the RAPIDS Memory Manager handles much lower-level optimizations such as memory allocations—although you probably won't need to reach down into this level of detail. The bottom line is, if your workflow can run on a single GPU, use cuDF; however, when you need to distribute processing of tabular data across multiple GPUs—or perhaps input data from several files in parallel (as in sharding)—then use Dask-cuDF.

[21] cuDF v0.13, pandas 0.25.3. Running on NVIDIA DGX-1. GPU: NVIDIA Tesla V100 32GB. CPU: Intel® Xeon® CPU E5-2698 vs4 @2.20GHz. Other benchmark details at https://docs.rapids.ai/overview/latest .pdf.

6.2 Graphs (... Except When Something Is Not a Table)

Everything is a table, except when it's not. For example, spreadsheets and SQL queries both appear to have much to do with tables, except that under the hood, they are really *graphs*. For spreadsheets, there are dependency graphs. For SQL queries, there are *query plans* (DAGs, directed acyclic graphs)—plus *schema* represented in ERD (entity relationship diagram) format, which can become complex graphs, too. One truism about computing: when you attempt to understand the nuances of what's happening at runtime in a spreadsheet or a SQL query, you run headlong into graphs.

Everything is a table, except when it's not.

On the one hand, avoiding the key abstraction—for example, by making graph processing appear as if it were a table—tends to obscure the metadata and business rules in these systems. These abstraction mismatches become more difficult to troubleshoot, test, reuse, audit, and so on, which creates a form of technical debt.[22] On the other hand, there's a wide range of powerful *graph algorithms*, and these will not run effectively in spreadsheets, SQL queries, and the like. Graphs have been a central concept since the earliest years of computer science, so it's surprising that programming languages don't support graphs directly.

But for Python, a popular open source library called NetworkX provides for network analysis, graph algorithms, and integration into graph serialization formats and interactive visualization. Additional use cases for graphs in AI applications include *semantic inference*, *shape constraints* (e.g., audits), *probabilistic graphs* (e.g., causality), graph ML, etc. Open source libraries in Python, such as kglab, integrate NetworkX into these other kinds of graph[23] use cases.

In RAPIDS, the cuGraph library builds atop cuDF to provide a GPU-optimized version of NetworkX. It also includes some popular graph algorithms that are noticeably missing from NetworkX, such as Leiden. For the code repository, user documentation, and blog articles, see the following:

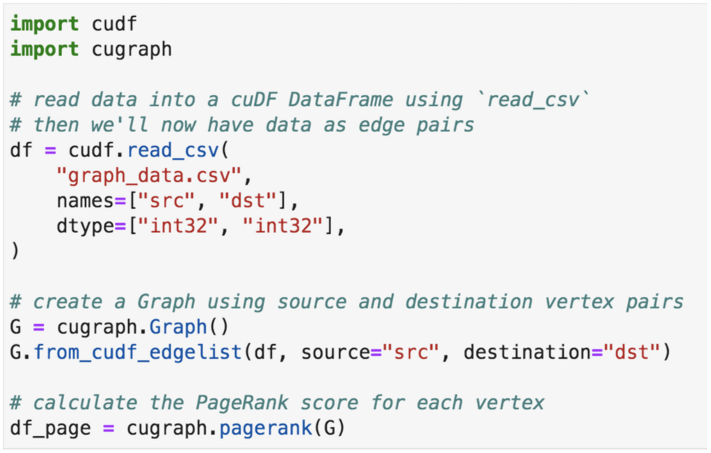
- https://github.com/rapidsai/cugraph
- https://docs.rapids.ai/api/cugraph/stable/
- https://medium.com/rapids-ai/tagged/graph-analytics

Similar to the approaches used by cuDF, the cuGraph API attempts to steer developers away from antipatterns, checking arguments and throwing exceptions if needed to

[22] Felienne Hermans is a leading researcher about spreadsheets and has identified many of the associated practices and antipatterns: https://www.felienne.com/archives/tag/spreadsheets.

[23] The iGraph library (https://igraph.org) is similar to NetworkX, and cuDF provides some of its features. In general, there is a wide range of graph libraries, although these two represent those that fit especially well with the PyData stack that's so popular in data science work; for libraries comparisons, see https://derwen.ai/docs/kgl/ack/#similar-projects. Of course, many popular libraries implement graph algorithms, such as https://github.com/alibaba/GraphScope—although it scales out on CPU clusters, and its API is quite different than NetworkX's.

provide better graph optimization with hardware. For example, at a low level, if a call runs out of memory and crashes, that can be caught. Also, calls to graph algorithms in cuGraph produce cuDF dataframes for their results, which allows for the use of method chaining on graph operations—with dataframes as the unit of work. This fits back into optimization strategy at the workflow level, e.g., with Dask. Figure 6.4 shows a concise use of the cuGraph API in action.



```python
import cudf
import cugraph

# read data into a cuDF DataFrame using `read_csv`
# then we'll now have data as edge pairs
df = cudf.read_csv(
    "graph_data.csv",
    names=["src", "dst"],
    dtype=["int32", "int32"],
)

# create a Graph using source and destination vertex pairs
G = cugraph.Graph()
G.from_cudf_edgelist(df, source="src", destination="dst")

# calculate the PageRank score for each vertex
df_page = cugraph.pagerank(G)
```

Figure 6.4 Example code for loading a graph from a dataframe

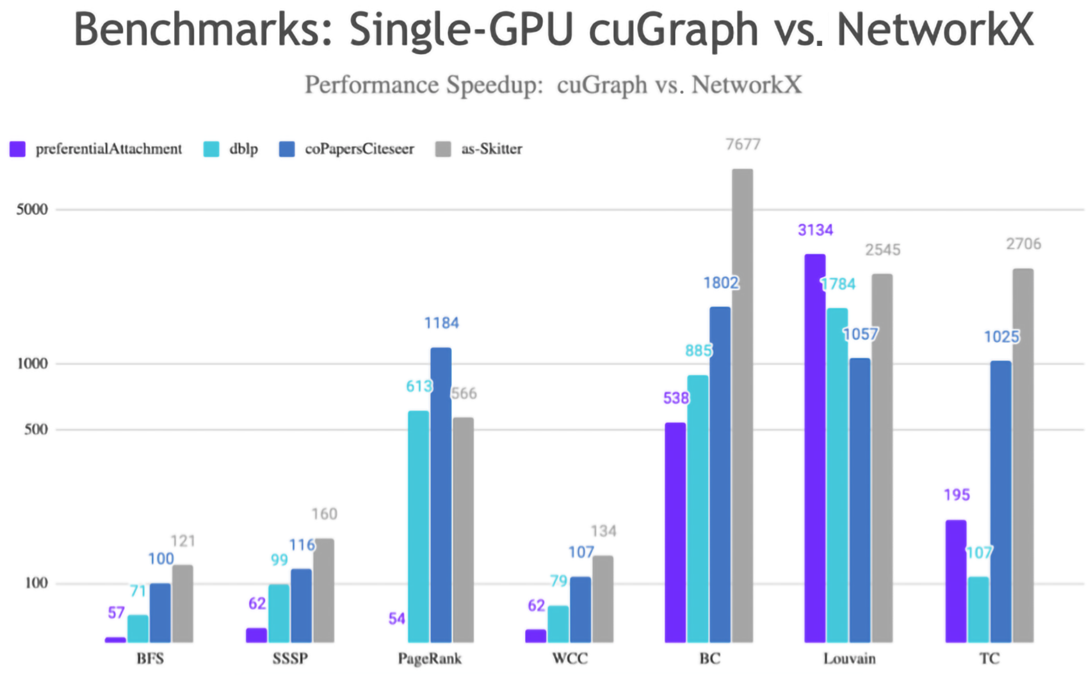
The vision for cuGraph is "to make graph analysis ubiquitous to the point that users just think in terms of analysis and not technologies or frameworks"—creating an accelerated unified graph analytic library. From a big-picture perspective, the timing for this could hardly be better. In February 2021, Gartner Research reversed its prior guidance from mid-2020 in a new report, stating, "By 2025, graph technologies will be used in 80% of data and analytics innovations, up from 10% in 2021, facilitating rapid decision making across the enterprise."

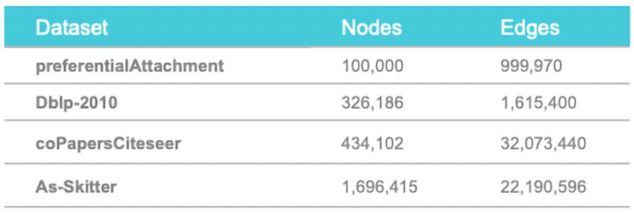
Clearly, the opportunities afforded by cuGraph force a rethink about parallelism for graph algorithms. On the one hand, many of the graph algorithms in NetworkX can be implemented as operations on dense matrices with numeric values. Therefore it makes sense that cuGraph uses cuDF dataframes internally to represent graph data. To load graph data into cuGraph:

- The simplest way is to pass a `networkx.Graph` object.
- The fastest way is to load from a cuDF dataframe.
- SciPy also has sparse matrices that cuGraph can operate on directly.

Note that *property graphs* are widely used in industry (e.g., in the commercial graph databases), and use cases for these can be accelerated by cuGraph plus GPU hardware, plus other parts of RAPIDS that are outside the scope of the NetworkX API.

In any case, the point is to open much broader use cases for graph analytics by making execution *fast*, especially with large graphs. For example, see "Tackling Large Graphs with RAPIDS cuGraph and CUDA Unified Memory on GPUs" by Alex Fender and Brad Rees for benchmark comparisons. The NVIDIA implementation of the popular PageRank (one form of graph algorithms used for measuring *node centrality*) with a Twitter dataset was speeded up approximately 1000×[24] using cuGraph and GPUs. As you might expect, using cuGraph in place of NetworkX greatly accelerates applications, as shown in figure 6.5.





Dataset	Nodes	Edges
preferentialAttachment	100,000	999,970
Dblp-2010	326,186	1,615,400
coPapersCiteseer	434,102	32,073,440
As-Skitter	1,696,415	22,190,596

Figure 6.5 Speedup of graph algorithm benchmarks (including PageRank) using cuGraph and GPUs[25]

[24] From the cuGraph documentation: "The compute power of the latest NVIDIA GPUs (RAPIDS supports Pascal and later GPU architectures) make graph analytics 1000x faster on average over NetworkX": https://docs.rapids.ai/api/cugraph/stable/cugraph_intro.html.

[25] From https://docs.rapids.ai/overview/latest.pdf.

From the perspective of a better process for leveraging GPUs, the principle is to think about how you store your data. While NetworkX uses dictionaries, which are inefficient, can you instead organize the data into numeric vectors? Also, how do you ask questions about the graph? Sometimes asking one question is less efficient than asking more than one question at a time, such as in the case of breadth-first search. Again, both of these points go into the dense kind of thinking about data workflows.

6.3 Pipelines (Workflows)

The notion of *pipelines* is generally not a first-class construct in programming languages. Remember, in data science and data engineering work, *you're probably defining pipelines*—typically much more so than you are writing custom code. There are related notions such as *method chaining* in Scala, Java, Python, etc., and even more generally, there's the notion of *function composition,* which can be used along with object-oriented programming in some languages (e.g., Scala, Java, Python, etc.) to create pipelines. In other cases, popular Python libraries such as spaCy allow for factory objects to create components that can be used to define pipelines.

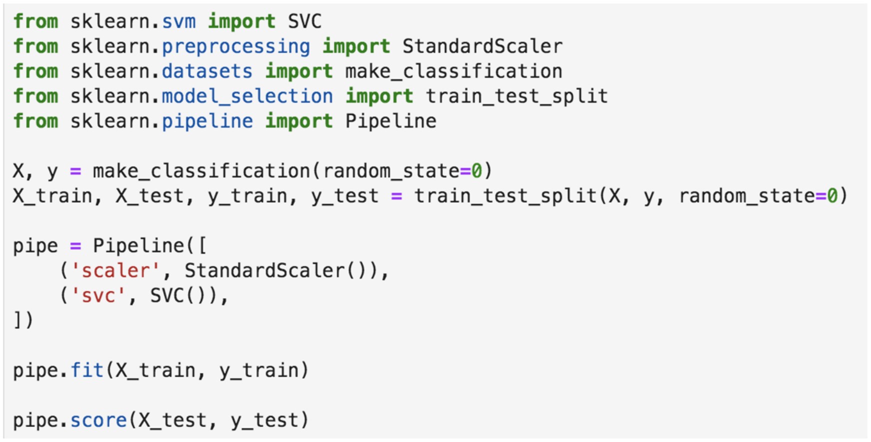
The popular scikit-learn open source library in Python provides simple and efficient tools for predictive data analysis. It's the core library in the PyData stack for running machine learning algorithms and defining data analytics pipelines in general. Other machine learning libraries in Python use scikit-learn as their foundation, such as lale for AutoML workflows.

In scikit-learn, a `Pipeline` object is defined explicitly. Within these pipelines, scikit-learn uses objects called *estimators*, i.e., any object that learns from data: a classifier, a regression or clustering algorithm, a transformer that extracts and filters useful features from raw data, and so on. By leveraging pipelines in scikit-learn, tools such as Dask can then wield their magic to perform effective optimization: thinking sparse and dense on behalf of data workflows. Even so, one of the confounding experiences for the developers of scikit-learn is to see just how much effort people put into writing custom code instead of simply using the built-in support for pipelines and composite estimators. When developers pipeline their data workflows idiomatically, they can unlock Dask's acceleration benefits. Figure 6.6 shows a typical, concise pipeline definition using scikit-learn.

In RAPIDS, the cuML library builds atop cuDF to provide a GPU-optimized version of scikit-learn. For the code repository, user documentation, and blog articles, see the following:

- https://github.com/rapidsai/cuml
- https://docs.rapids.ai/api/cuml/stable/
- https://medium.com/rapids-ai/tagged/machine-learning

NVIDIA works closely with the Dask team, and the cuML code base is designed with Dask in mind: e.g., where algorithms are split into tiers. Even so, when you look at the



```python
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

X, y = make_classification(random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('svc', SVC()),
])

pipe.fit(X_train, y_train)

pipe.score(X_test, y_test)
```

Figure 6.6 Example code for a pipeline in scikit-learn

cuML code, only about 4% is specific to GPUs. Most of the code is about smarter implementations of algorithms for parallelization in general.

The range of what's implemented in scikit-learn is vast, and not all of its estimators are in common use. So, to be clear, cuML is not identical to scikit-learn, but it runs the same models and implements the same API. The cuML team has been scraping Kaggle notebooks to run statistical analysis of where coverage is needed. The coarse-grained strategic questions are simpler to answer: which estimators in scikit-learn are found in the wild? But for a given estimator, determining which parameters to support is a trickier tactical question that depends on the most common (and most recommended) usage. Where parameters define algorithmic features, these are definitely prioritized for support. Nonetheless, there are finer-grained details where cuML usage may differ from scikit-learn results. For example, in poorly conditioned optimization models, slightly different rounding errors can occur when algorithms are run in parallel versus serial. But there are also optional slow paths in some estimators to fix some minor discrepancies at the end of a workflow, to guarantee the end results down to true machine epsilon, if needed. Figures 6.7 and 6.8 show typical speedups using cuML versus scikit-learn.

The key takeaway here is that to make the most of the hardware with machine learning pipelines, be sure to

1. Use the standard APIs in cuML.
2. Use numeric formats.
3. Don't use iteration loops.
4. Use the idiomatic scikit-learn approaches, such as pipelines.

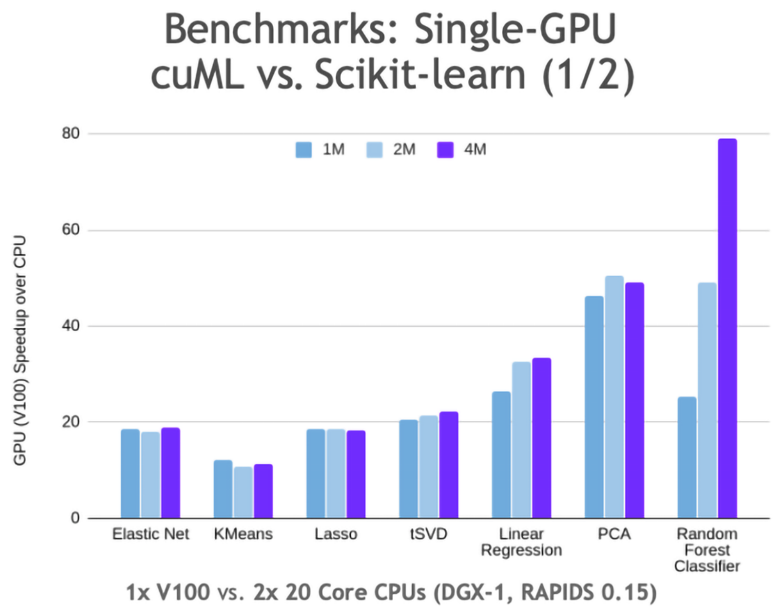


Figure 6.7 Speedup benchmarks using cuML and GPUs (1/2)[26]

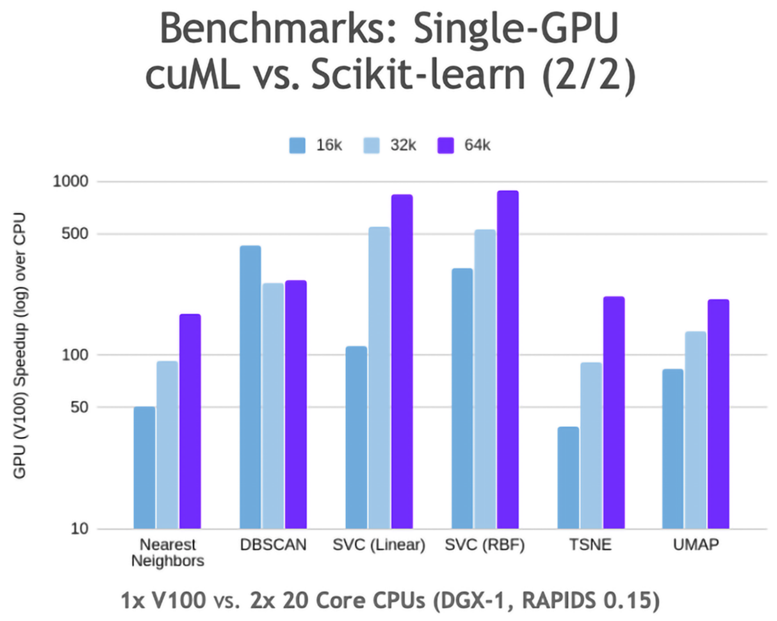


Figure 6.8 Speedup benchmarks using cuML and GPUs (2/2)

[26] Source: https://docs.rapids.ai/overview/latest.pdf.

There's a cognitive leap from writing vanilla Python code to using pandas. To use cuDF efficiently in place of pandas, you must learn to think about vectorizing critical sections of code. Using pipelines in scikit-learn requires even more of a cognitive leap since these pipelines represent so much work in so little code. Then, moving to cuML in place of scikit-learn, you must leap a bit further since there are even larger penalties for not using the idiomatic design patterns.

6.4 MLOps: Pipelines in Production

Pipelines are important in the early stages of data science processes where data gets explored and models are trained. Pipelines are also essential for production scenarios. They enable formalizing how production models are trained, deployed, and managed. Some pipeline implementations also support *streaming* scenarios, where data is scored against models as it arrives rather than captured and scored as batches later.

Formalizing the deployment process is an integral part of the emerging interest in *MLOps,* the extension of mature *DevOps* processes to the unique requirements of data science artifacts in production.

How formal, repeatable, and controlled to make pipeline deployments will depend on the answers to several questions. Two questions that we think everyone should consider are these:

- Is the data used for training and production scoring subject to careful data governance? Data with sensitive information requires access controls, provenance tracking, and other careful handling. Some data may be so sensitive that it can only be accessed in a carefully controlled production environment by a limited set of privileged team members, where even the data science team may not have free access for model development!
- What is the cost of failure? If a model makes a bad decision, what is the negative impact on the organization, customers, etc.? Concerns about embedded bias in models lead to requirements for repeatable training processes, where the metadata about any one model in production can be interrogated to debug issues, such as what datasets were used to train the model, when and how was the model trained, etc. This in turn means automation and repeatability of all aspects of the pipeline may be required, and visibility into model performance at runtime is essential.

DevOps emerged as a general approach to automate deployments and management of services, independent of what the services do. A DevOps *pipeline* for deployment and management might include any or all of the following stages, which typically start from the moment a software revision is pushed to a target repository branch or is tagged in some appropriate way:

1. Check out the repository branch or tag.
2. Perform a full build of the deployment artifacts.
3. Run all automated unit and integration tests.

4. Deploy to a staging environment, and run further acceptance tests.
5. Deploy to production.
6. Optionally, route a percentage of traffic to the new and old deployments for A/B or canary testing, or do traffic shadowing, as well, to allow additional burn-in testing for safety.
7. Remove older deployments, and route all traffic to the new deployment.
8. Monitor the service for health.
9. Scale instances of the service up and down on demand.
10. Route traffic away from the service in preparation for retirement.
11. Remove the service from production.

To be clear, this DevOps pipeline is different from the data science pipeline described earlier. These are nested pipelines. The latter can be managed by the former. When data science pipelines are managed, additional considerations lead to the MLOps specialization of DevOps:

- Data governance, such as access controls and provenance tracking, is required at all stages where data is used.
- Models are themselves data since they are intellectual property, and in some cases, sensitive information from the training data can be extracted from them.
- Model serving results (scores) are inherently nondeterministic, reflecting their probabilistic and statistical roots, which is an unfamiliar property for developers and administrators accustomed to other kinds of services.

The *production* models are trained in an MLOps pipeline build step, while the data science team designs and trains models to discover the optimal model architecture. To achieve the required repeatability, automation, and end-to-end governance, the production models are built by the pipeline using the *hyperparameters* determined during the discovery process (although some additional hyperparameters might be determined during the build).

All services require monitoring in production for health and performance. Model serving adds particular metrics to gauge model performance. In particular, *concept drift* is a decay process of sorts, where a model trained on data at a particular point in time becomes less and less effective as the data characteristics evolve over time.

Fortunately, MLOps principles apply equally well to CPU-based data pipelines and GPU-accelerated pipelines, although some details will vary. For more on MLOps, see this GradientFlow blog post[27] and this Martin Fowler blog post.[28]

[27] https://gradientflow.com/what-is-dataops/
[28] https://martinfowler.com/articles/cd4ml.html

6.5 Tensors (Deeper Data Representation)

You can think of a *tensor* as an N-dimensional matrix (there are a few subtle differences). It used to be the case that if you wanted to run numerical libraries on networks or graph data, first you needed to fit the data into a matrix. Graphs with *multiple* kinds of edges between two nodes, or attributes on nodes and edges, had to be simplified. These conversions, at best, resulted in overly complicated and expensive encoding. At worst, important and valuable details were lost.

Alternatively, a tensor is a better way to represent a network or a graph instead of trying to wedge the data into matrices. Prior to 2015, there weren't many open source libraries for working with tensor representations of data, especially not in Python. However, these days we hear the word *tensor* frequently in relation to work with neural networks, in TensorFlow, PyTorch, etc. So far, not many programming languages have tensors defined as first-class constructs. Instead, we need to rely on the idioms of popular deep learning frameworks—and follow their idiomatic approaches. These frameworks have made good use of GPU hardware optimizations from their inception.

Figure 6.9 shows how popular data science packages that use data as tensors can interoperate by sharing objects in device memory. This keeps data on the GPU to avoid costly memory copies back and forth between GPU and CPU memory.

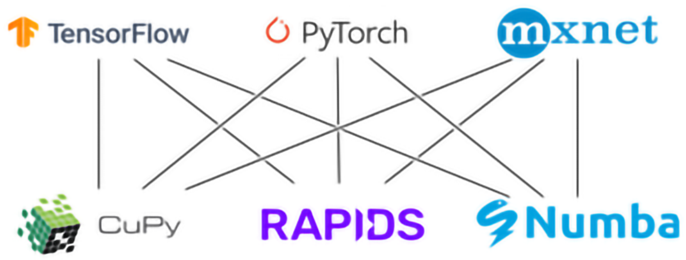


Figure 6.9 DL frameworks that use tensors, sharing memory objects

Going back to our earlier discussion about the kinds of data used for machine learning, it's important to note that the early breakthroughs[29] for deep learning almost all involved computer vision (CV) problems. Then, in late 2017, the notion of *embedded language models*[30] led to breakthroughs with deep learning in natural language processing

[29] See the "AlexNet" paper from 2012, "ImageNet Classification with Deep Convolutional Neural Networks," by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton: https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[30] The first was ELMo by Allen AI in 2017 (https://allennlp.org/elmo), followed by a general class of transformer models for NLP.

(NLP). The way that deep learning frameworks used GPUs was informed by the needs of CV, which were reused for NLP work. Notably, in CV

- There are large datasets.
- Each of the inputs is a relatively large object.
- The resulting model is massive (many parameters).

Also, understand the typical performance concerns for training deep learning models:

- Math-heavy parts such as convolutions, fully connected layers, or recurrent layers tend to be compute-bound.
- Loss calculation or bias normalization tends to be limited by memory.

Hence, the first round of popular AI applications involved CV and natural language. However, the adoption of deep learning has spread throughout industry, where it is being used for other kinds of problems.

One broad class of machine learning applications are *recommender systems* (recsys), which were the initial commercial successes that drove billion- and trillion-dollar companies: product recommendations on Amazon, search recommendations on Google, social recommendations on LinkedIn, media entertainment recommendations on Netflix, and so on—plus virtually all of online advertising. The recsys use cases based on deep learning will likely outnumber the CV use cases. However, the data needs for recsys applications are shaped quite differently:

- The resulting models aren't massive.
- Embeddings represent the bulk of what is stored on GPUs.
- Memory is the limiting factor (based on the embeddings), not compute.

There are also considerations about the costs and potential bottlenecks at training time versus inference time, for example, during model usage. Access to memory bandwidth therefore becomes the gating factor for recsys; moreover, these bandwidth bottlenecks occur in both training and inference, which can escalate costs. Consequently, early recsys use of GPUs was only 4–5× faster than CPU. In most cases, if you tried out GPUs two years ago for recsys, you probably concluded, "Zero benefit."

The kinds of layers that a neural network uses has an impact: they rely on different kinds of system resources and therefore may encounter different kinds of performance bottlenecks. Figure 6.10 shows typical kinds of layers in a deep neural network model and indicates whether they tend to be compute (CPU/GPU) bound or memory bound.

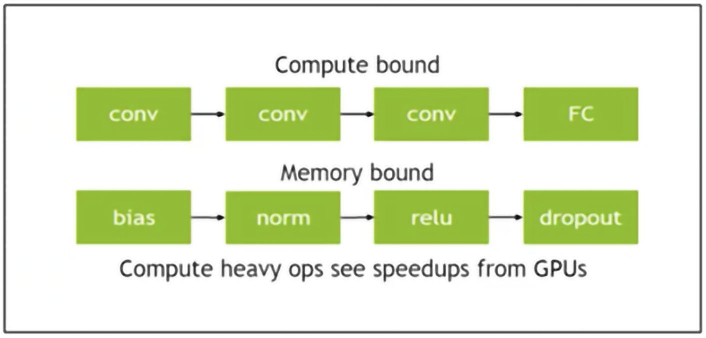


Figure 6.10 Typical layers[31] in deep learning, with their performance characteristics

After much performance analysis of recsys applications, NVIDIA has been building software atop the backbone of RAPIDS, which wasn't needed or even possible during the rise of CV (2009–current) or NLP (2017–current). The NVTabular package is "a feature engineering and preprocessing library for tabular data designed to quickly and easily manipulate terabyte-scale datasets used to train deep learning-based recommender systems." Translated: recsys is typically trained on data from log files—machine data. To be efficient, this kind of data requires substantially different preprocessing than images or text, as well as different hardware and software architecture. For the code repository, user documentation, and blog articles, see the following:

- https://github.com/NVIDIA/NVTabular
- https://nvidia.github.io/NVTabular/main/index.html
- https://developer.nvidia.com/nvidia-merlin
- https://medium.com/nvidia-merlin

Only within the last generation of GPUs has the hardware become well-suited for recsys use cases by improving memory bandwidth. Fortunately, embeddings tend to follow a Power Law distribution, so they can leverage caching better than what's been done previously with GPUs. The performance of *interconnects* is also getting better, such NVLink and RDMA.[32]

One interesting finding from the performance analysis of recsys on GPUs: overall performance was typically gated by the performance of the data loaders[33]—for example, the `DataLoader` objects in PyTorch. Recall that CV involves *really big* data objects for input, and by default, these are loaded one by one in TensorFlow or PyTorch. In

[31] Specifically, convolution (conv) and fully connected (FC) layers are compute intensive, while bias, normalization (norm), rectified linear units (relu), and dropout require lots of memory for recsys.

[32] See the 2013 "Taming Latency" talk by Jeff Dean for a good general discussion of optimizing software and hardware for training large neural networks, including latency for memory interconnects: https://youtu.be/S9twUcX1Zp0.

[33] See "Announcing the NVIDIA NVTabular Open Beta with Multi-GPU Support and New Data Loaders" (5 October 2020).

contrast, recsys uses *many* data input items that are much smaller, so the memory bandwidth bottlenecks will cause the default data loaders to run orders of magnitude slower. That was the first priority in software for NVTabular to fix!

Other typical recsys bottlenecks on GPUs have been mitigated by

- Pre-shuffling the training data—then the data movement becomes much more efficient for a GPU
- Changing the data access patterns to get everything ready for recsys upfront, prior to data loading
- Also speeding up PyTorch training of recsys by 10×

With these mitigations taken together, a typical recsys pipeline now runs 50× faster on GPU than when using CPU.

To be effective for recsys applications in the general case, *we must think about the problem across the board*: how long does it take a data scientist to go from problem formulation to production? This question is turning out to spotlight time-to-market (TTM) as one of the key performance indicators[34] for AI projects in production. And to that point, the NVTabular team is in a unique position. In industry, recsys teams tend to be tied very closely to near-term revenue; in other words, company executives watch the performance of recsys applications *closely*, especially in advertising. People working on these kinds of teams[35] in industry rarely get to step back, analyze the performance of their hardware in fine-grained detail, and then recommend a co-evolution strategy for hardware and software architecture together. Instead, their executives demand immediate results.

To be effective for recsys applications in the general case, we must think about the problem across the board: how long does it take a data scientist to go from problem formulation to production?

In response to these needs, the NVTabular team has leveraged their unique position to develop the hardware and software together to achieve optimal performance. This represents a rethink about recsys on GPU, considering how the hardware has changed so much since the early days of CV and how the software has also evolved, such as in terms of caching.

Other enhancements for recsys include the following:

- Note how trade-offs for memory bandwidth costs are similar to the costs of moving data in and out of the cloud. In the larger scope, allowing recsys pipelines to

[34] For two studies about TTM as a key performance indicator in AI products, see *Operationalizing AI* by John Thomas, Will Robert, and Paco Nathan (2021), https://www.oreilly.com/library/view/operationalizing-ai/9781098101329 and "2021 AI in Healthcare Survey Report" by Ben Lorica and Paco Nathan, https://gradientflow.com/2021aihealthsurvey.

[35] Per Paco's personal experience, having led recsys teams in advertising.

access cloud storage directly is another way that NVTabular helps scale out solutions even at the early workflow stages of data preparation.

- Exporting the pipeline used in training so that it can be reused at inference time in production.
- Work on ETL to capture the statistics of input data since recsys data tends to drift more than CV or NLP data and require more frequent updates of ML models.

Consequently, what NVTabular provides to customers includes a couple of design patterns plus their orchestration, for a continuous feed of data (required by the recsys definition) as well as continuous updates for models.

A key takeaway for leveraging hardware with tensors, especially for recsys applications: "Think about where data needs to be stored and used. Efficient and fast data transfer is super important."

7 Leveraging Design Patterns: Neither a Tinkerer nor a Trifler Be

If you want to be effective as a software engineer and make the code take advantage of the hardware to make the most of your data, then do just that: be an *engineer*, not a *tinkerer*. Based on using the key abstractions described previously, let's spell out the general form of a process for data workflows that use GPU hardware and the related software layers:

- Consider your use case from end-to-end, from product concept through into production use and iteration, and be especially mindful about TTM.
- Recognize the kind of processing needed in each stage of a workflow, and accomplish it there: don't conflate sparse thinking with dense thinking.
- Use the appropriate data abstractions, and follow idiomatic design patterns for their related libraries—dataframe/pandas/cuDF, graph/NetworkX/cuGraph, pipeline/scikit-learn/cuML, tensor/PyTorch/NVTabular—and also in workflow tools such as Dask.
- Apply software engineering tools that detect the use of these idioms and design patterns in Python—or antipatterns—as part of your continuous integration process. These techniques complement conventional unit testing and integration testing, which are used for detecting bugs and meeting requirements.
- Use profiling tools for performance analysis of your applications, applying them layer by layer as needed until you can identify the main gating factors.

Another part of the process is to engage with the developer community: SSO, Go.AI Slack, attending GTC conferences, GitHub issues for the open source projects, meetups, etc. It's surprising how much you can learn—best practices, new tooling, and so on—from other practitioners through these interactions.

We've discussed the first three points; next let's unpack the fourth: how to leverage software engineering tools to follow idioms and design patterns more closely. The next section explores the fifth point.

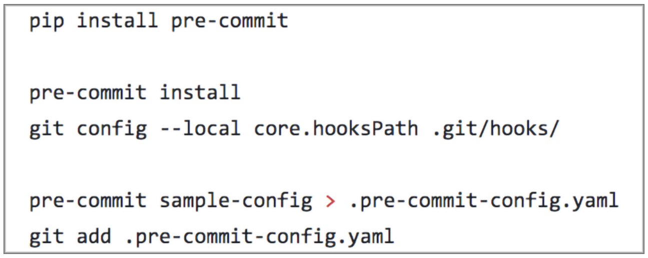
In Python software development, whether for data science or any other application area, the single best thing you can learn to do is to make good use of the standard tooling. That doesn't necessarily imply using IDEs, although these can help. It's more a matter of tools that can run in addition to your unit tests. Roughly speaking, these are *continuous integration* checks to analyze and verify changes in your source code before your commits.

- Use *type annotations* plus related annotation checkers such as Mypy to catch areas of code that need better specification. These help prevent bugs and can also help the underlying software layers optimize code execution.
- Use *linters* such as Pylint, Flake8, and Black—not only to check for *formatting* but also to identify sections of code that could be expressed more idiomatically. Think of these tools as a form of unit testing that pushes you to follow better design patterns: for example, using comprehensions. Recall that Python is quite forgiving and allows you to write code poorly; these tools push you toward more efficient idioms.
- Be sure to use *pre-commit* hooks in your coding projects to help employ all of the above as well as check for potential security vulnerabilities and other hazards.

For some good general guides to using pre-commit hooks in Python and these kinds of tools in general, see the following:

- https://pre-commit.com
- https://calmcode.io/pre-commit/the-problem.html
- https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
- https://docs.github.com/en/developers/webhooks-and-events/about-web-hooks

Figure 7.1 shows the commands you can use to install and begin using pre-commit hooks for a Git repository.



```
pip install pre-commit

pre-commit install
git config --local core.hooksPath .git/hooks/

pre-commit sample-config > .pre-commit-config.yaml
git add .pre-commit-config.yaml
```

Figure 7.1

8 *The Art of Profiling: Veni, Vidi, Metiri*[36]

The previous section discussed *process* approaches that are mostly automated, such as pre-commit hooks used prior to committing code into a Git repo. When antipatterns in the code are identified, you revise the source and then commit. Done and done. In contrast, this section explores tools that generally aren't quite as automated. You may do a little benchmarking automatically—ensuring that any code changes don't degrade your pipeline performance—but you probably don't budget hours of profiling for each pull request or dig into hardware profiling on every commit. Instead, these tools become engineering practices: used when needed. Profiling is always an iterative process of discovery and action, as illustrated in figure 8.1.

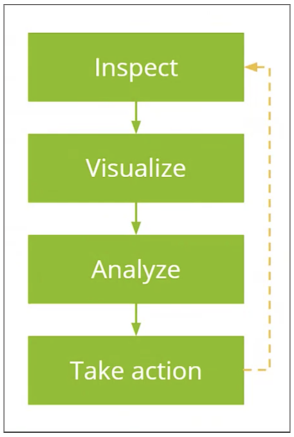


Figure 8.1 An iterative process as a best practice for using profilers

How you use the profiling tools is important. There are layers in the tech stack, and your profiling should progress in stages through those layers: Python, Cython, C++, CUDA. Have a strategy for profiling and how to use the feedback from it:

1. Start with coarse-grained analysis in Python.
2. Drill down to more fine-grained profiling tools to troubleshoot problems as they are identified.
3. Throughout, use an "inspect, visualize, analyze, take action" loop to guide your performance analysis.

Figure 8.2 illustrates the layers encountered with cuDF. The software layers are on the left: Python is at the highest level, farthest from the hardware, while CUDA is the lowest-level library for direct manipulation of GPUs. On the right are the patterns for

[36] "I came, I saw, I measured," with apologies to Julius Caesar.

how these are used, such as Dask-cuDF. For effective profiling, you need to progress through these layers using the interactive process.

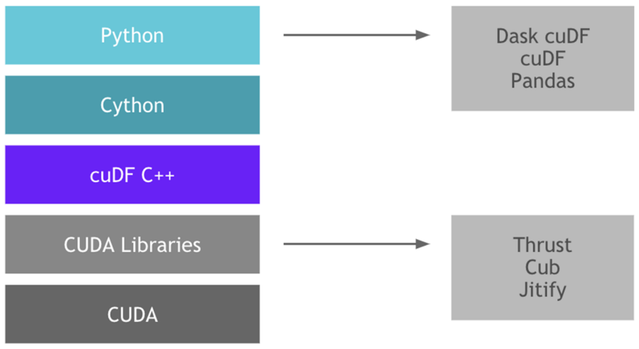


Figure 8.2 Software stack for cuDF, where the layers on the left correspond to patterns for usage on the right[37]

The following table describes several popular profilers for Python that analyze[38] for both compute-bound and memory-bound bottlenecks.

Library	Purpose	Usage
watermark	Jupyter magic extension for printing date and time stamps, library version numbers, and hardware information	Keeping track of the details for each configuration you're profiling
Fil	Tracing peak memory usage	Determining which section of code caused the high-water mark
objgraph	Tracing and visualizing the object graph	Finding out which objects are referencing which other objects
tracemalloc	Tracing memory blocks allocated by Python	Computing the differences between snapshots to detect memory leaks
SnakeViz	Browser-based graphical viewer for cProfile output	Using icicle charts and sunburst charts to visualize compute-bound functions
cProfile	Built-in profiler (in C, for less overhead) for deterministic profiling of the call stack	Capturing the full statistics of the run time for a Python application
pyinstrument	Statistical profiler of the call stack	Estimating compute times for particular sections of code (less distorted by overhead)

[37] Source: https://docs.rapids.ai/overview/latest.pdf.

[38] For example use of these profiling tools in a Python application, see the pi.ipynb notebook in the https://github.com/DerwenAI/ray_tutorial tutorial.

For other good overviews about using the Python profilers, see the following:

- https://scoutapm.com/blog/identifying-bottlenecks-and-optimizing-performance-in-a-python-codebase
- https://towardsdatascience.com/speed-up-jupyter-notebooks-20716cbe2025
- https://jakevdp.github.io/PythonDataScienceHandbook/01.07-timing-and-profiling.html

Most users will rarely ever need to punch down below the Python tools. However, if you need more fine-grained analysis, Nsight Systems provides profiling at the hardware level. Based on this, you can use NVTX decorators to annotate your Python, Cython, and C++ code to generate *timelines* for analyzing and visualizing how your code runs on GPUs. Nsight Systems also measures the use of other resources: memory, disk, etc.

Another tool called DLProf provides additional details specifically for deep learning workloads. This tool is agnostic about frameworks, supporting the GPU-optimized versions of TensorFlow, PyTorch, and TensorR—running atop Nsight Systems and using NVTX markers. DLProf analysis correlates to the layers and iterations of deep learning models as these are being trained. You interact with its UI as a TensorBoard plugin, much like other tools for analyzing DL workloads.

9 Looking Ahead

The intention of this report was to discuss the considerations needed in your software and process to take advantage of newer hardware. Of course, there isn't room on the page to dig into more details about the full range of options for RAPIDS and other libraries across a wide range of available hardware. This is pointed at people developing in Python for data science and data engineering purposes.

Why does this matter? Speed, cycle times, and time to market (TTM) are the key motivators. That's an enormous game-changer. To quote from a recent paper[39] from the RAPIDS team, "One of the core ideas and motivations behind the multifaceted and fascinating field of computer programming is the automation and augmentation of tedious tasks."

The TTM metric has become one of the most important key performance indicators for AI projects. People working in data science teams typically spend upward of 90% of their time *waiting* for ETL jobs to complete, *waiting* for models to finish training, and *waiting* to visualize results. Figure 9.1 provides a cheeky view of a day in the life of a data scientist when using CPU-based ETL versus GPU-accelerated ETL.

[39] "Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence," by Sebastian Raschka, Joshua Patterson, and Corey Nolet (31 March 2020): https://arxiv.org/abs/2002.04803.

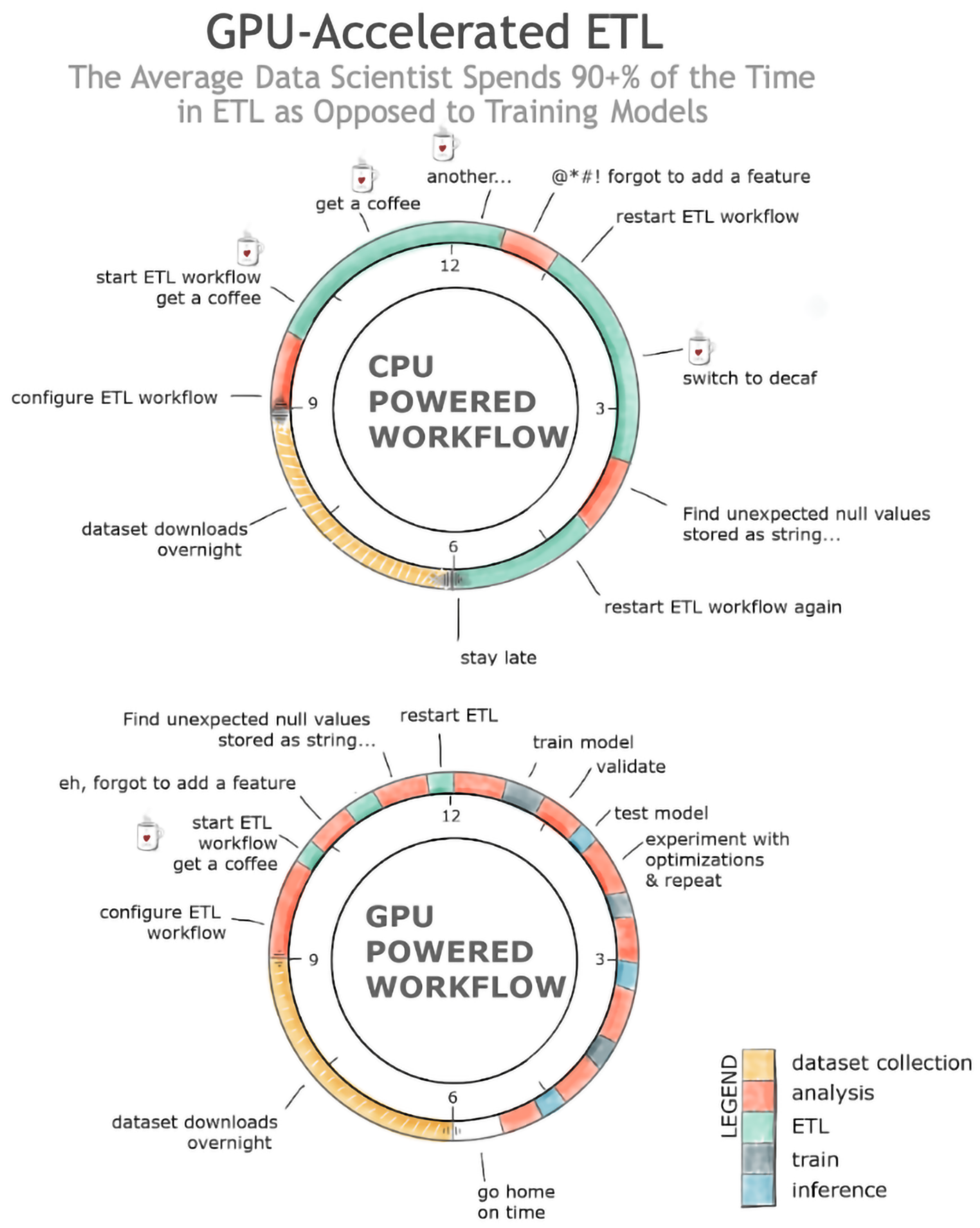

GPU-Accelerated ETL
The Average Data Scientist Spends 90+% of the Time in ETL as Opposed to Training Models
get a coffee
another...
@*#! forgot to add a feature
restart ETL workflow
start ETL workflow get a coffee
switch to decaf
configure ETL workflow
CPU POWERED WORKFLOW
Find unexpected null values stored as string...
dataset downloads overnight
restart ETL workflow again
stay late
Find unexpected null values stored as string...
restart ETL
train model
validate
eh, forgot to add a feature
test model
start ETL workflow get a coffee
experiment with optimizations & repeat
configure ETL workflow
GPU POWERED WORKFLOW
dataset downloads overnight
go home on time
LEGEND
dataset collection
analysis
ETL
train
inference

Figure 9.1 A hardware-accelerated process remaps data practices due to TTM.[40]

[40] Source: https://docs.rapids.ai/overview/latest.pdf.

In practice, all of the waiting implies some almost inevitable failure modes. First, your highly expert (and expensive) staff are probably going to go get coffee instead of focusing on hard problems. That creates a cognitive load for the individuals and slows down team collaboration overall. Second, your team will tend to have less time for *iterating* toward better solutions. The best practices that we've outlined almost all depend on iteration—lots of iterations. Take that away, and your team's software engineering process will have a regression back to *waterfall* practices. Again, how long does it take a data scientist to go from problem formulation to production? This is more than an argument about efficiency—it's an argument about bringing complex engineering problems and their appropriate tooling back to human scale.

Looking ahead, the demand for larger and more complex machine learning models and other data analytics will of course continue to drive the evolution of hardware: specifically, toward more complex workflows involving *Multi-Node Multi-GPU* (MNMG) architectures, which require a fine balance of resource use for compute, memory, storage, network, etc. This implies a co-evolution of the hardware and software together, which is definitely in progress. For one good example, take a look at Unified Memory, which simplifies the programming model of MNMG, providing a pool of memory shared by CPUs and GPUs. On the one hand, this reduces the bottlenecks of moving memory objects (deep copies) between devices, while on the other hand, it reduces the need for programmers to stop and think about which devices their code must run on, where, when, and why. While this doesn't resolve all the issues of distributed processing, it does confront some of the challenges of thinking sparse and dense and makes the lower layers of software much simpler to leverage.

The demand for larger and more complex machine learning models and other data analytics will continue to drive the evolution of hardware.

Moving up the software stack, some changes will be required in machine learning frameworks: for example, the notion of *model parallelism*, splitting the layers of deep learning models across multiple devices, for example, using MNMG architectures. In an even more general sense, a newer generation of distributed tooling is emerging, such as the Legion programming runtime environment and the Legate framework for scaling. Research from Stanford University and Los Alamos National Labs has produced a distributed processing framework for Python workloads that outperforms even the latest generations of Dask, Spark, Ray, JAX, and so on. In particular, Legate understands how to scale a data workload efficiently across a combination of CPU clusters *and* GPU clusters, both with native support. Meanwhile, it's also much more aware of the key abstractions mentioned earlier. The idea is to port a large set of Python libraries atop Legion. These will be naturally composable since they share a common data model and runtime, with a scheduler that is built for analyzing and resolving some of the bottlenecks we've described.

10 Conclusion

This report has explored hardware innovations that enable previously impossible applications, plus the rethinking of how we build data science workflows and the processes we follow. We hope you have found the report useful and thought-provoking. Here are the key suggestions we believe you should keep in mind when developing and running production data workflows:

- Demand for larger and more complex machine learning models and other data analytics drives a co-evolution of hardware and software, where it won't be possible to train models on a single processor and strategies for using MNMG clusters become essential.
- Hardware acceleration with GPUs offers dramatic performance increases, which also reduces overall costs. Even more important, this approach reduces the key TTM performance indicator.
- Write idiomatic code so the hardware can be used to its fullest extent. Use tools like linting, static analysis, and pre-commit hooks to help ensure this goal and iterate on profiling in layers.
- Try to develop an intuition about how the hardware works and how this helps when optimizing your applications. You don't need to be an expert, but understanding the fundamentals can enable you to realize solutions you may have once thought impossible.

Organizations can meet the increasing demands of data workflows by using powerful, distributed GPUs and CPUs in MNMG architectures. These require using highly optimized tools, such as RAPIDS, along with a data-centric approach. Using these tools, results can be obtained orders of magnitude faster, and that increase in speed itself enables entirely new application areas to thrive in the enterprise. Meanwhile, the data scientists and data engineers on your team can continue to use many of the same Python packages, design patterns, and frameworks they already know.

Acknowledgments

Many thanks to Jim Scott and Chase Hooley for all of their help and support on this report, and to NVIDIA software engineering leads Keith Kraus, John Zedlewski, Brad Rees, and Even Oldridge for their invaluable expert guidance. Thanks also to Michael Mikowski at Kubuntu Focus for help with the GPU cycles. And thanks to our reviewers for their expertise in the subject matter and providing super-helpful feedback and suggestions: Ben Lorica, Michael Mikowski, Will Roberts, Bruno Gonçalves, and John Kaufhold. Also, we're grateful to get to work with our editor Jennifer Stout and the kind staff at Manning Publications.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

—Martin Fowler

Better programmers write idiomatic code that hardware can optimize.

—Paco Nathan, Dean Wampler