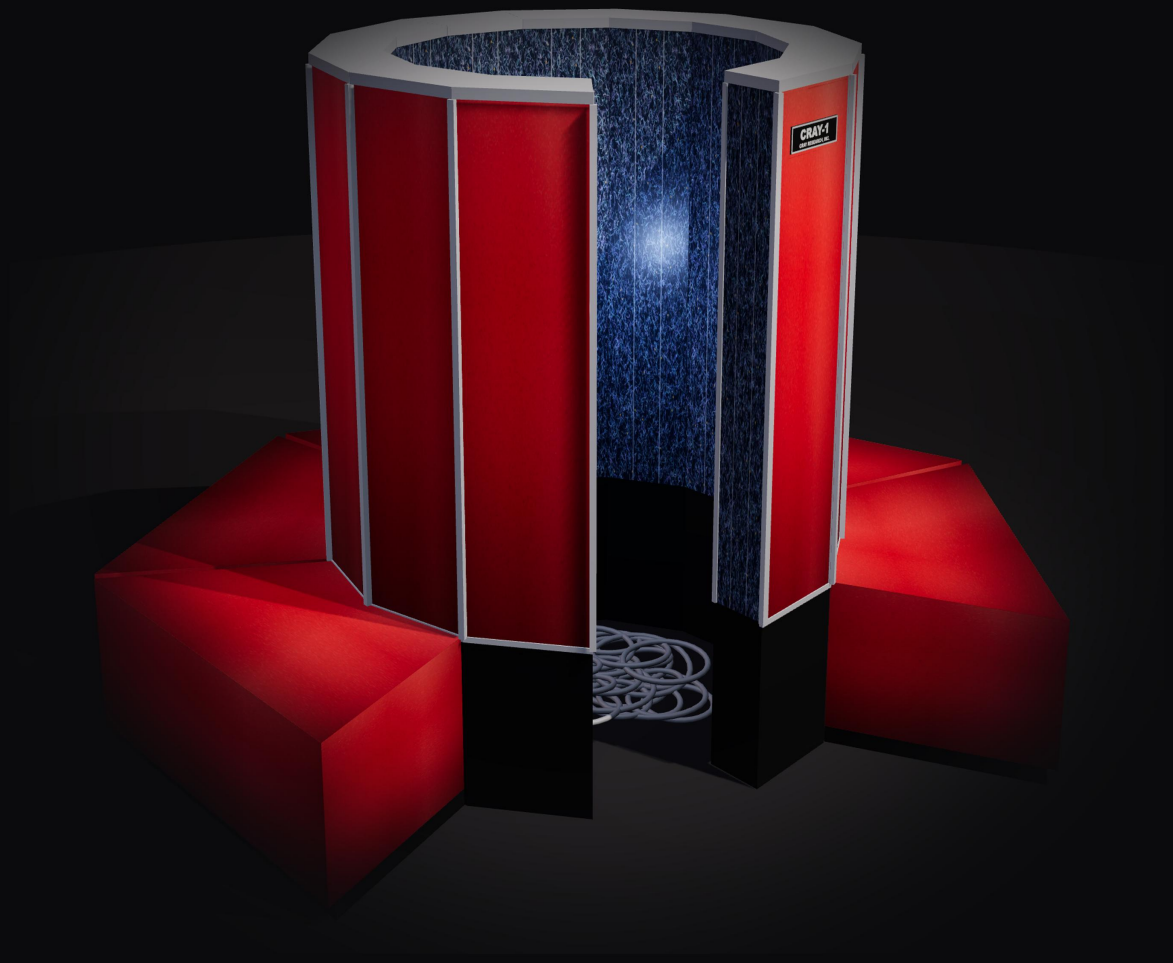
PERSONAL SUPERCOMPUTING WITH VECTORFALL



A GPU-hosted algorithms lab for you and your AI agents

Amir Husain

Best-selling author of *The Sentient Machine*

PERSONAL SUPERCOMPUTING WITH VECTORFALL

A GPU-hosted algorithms lab for you and your AI Agents

AMIR HUSAIN

First edition, for VectorFall 0.1.0

Personal Supercomputing with VectorFall: A GPU-hosted algorithms lab for you and your AI Agents

© 2026 Amir Husain. All rights reserved.

Published by AM Press, Austin, Texas, USA.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to publications@ampress.co.

Requests to the Publisher for permission should be addressed to the Permissions Department, AM Press, Austin, Texas, USA at publications@ampress.co.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

AM Press also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about AM Press products, visit www.ampress.co.

ISBN 978-1-965686-05-8 (paperback)

ISBN 978-1-965686-04-1 (Kindle)

First edition.

CONTENTS

PREFACE	Preface	6
INTRODUCTION	Introduction: The Work a Computer Can Do	10
PART I · THE LAB		
CHAPTER 1	A Supercomputer on Your Desk	20
CHAPTER 2	Installing and Running VectorFall	27
CHAPTER 3	A Tour of the Lab	36
CHAPTER 4	The Console: Python on the Card	53
PART II · YOUR PROBLEMS		
CHAPTER 5	Mapping Your Problems onto the Algorithms	68
CHAPTER 6	Tables: Something Odd in My Numbers?	85
CHAPTER 7	Team Building Inside a Company	94
CHAPTER 8	Economic Simulations That Look for an Outcome	109
CHAPTER 9	Audio Analysis: Sounds, Music and Speech	126
CHAPTER 10	Text: Trace It, Model It, Write It	143
CHAPTER 11	Pictures, Patterns and Worlds	166
PART III · AGENTS AND AUTOMATION		
CHAPTER 12	The vf Client, REST and JSON-RPC	187
CHAPTER 13	AI Agents over MCP	196
CHAPTER 14	Lineage: Your Card and the Supercomputers	217
CHAPTER 15	An Illustrated History of the Machines	224
CHAPTER 16	What Have Supercomputers Done for Us?	254
APPENDIX A	Model Reference	260

APPENDIX B	Supported Cards	279
APPENDIX C	vfctl and the Settings	281
APPENDIX D	Licence and Credits	283
BACK MATTER	Other books by the same author	285

PREFACE

PREFACE

In June 1997 the fastest computer on earth was ASCI Red, built by Intel for Sandia National Laboratories. It was the first machine to pass a trillion calculations a second on the standard benchmark. The United States had stopped testing nuclear weapons in 1992, and ASCI Red was one of the machines that took the place of the tests: it simulated the physics of the weapons and how the stockpile ages. It led the world's list of supercomputers for three years and grew to more than nine thousand processors. Only the mightiest nations could build a machine like it, and they used it for their most serious questions.

Today a used graphics card that sells for less than fifty dollars on eBay does a trillion calculations a second. The card used for most of this book, a Quadro K1200, measured 1,030 billion a second in Chapter 1, and it draws 45 watts, about as much as a desk lamp. Its trillion is counted in the single-precision arithmetic that graphics cards are built for, and ASCI Red's record was set in double precision; Chapter 14 makes the strict comparison. Even so, a machine that once needed a national laboratory now fits in your hand.

We rarely think of our computers this way. We use them to send messages, play games, read the web and, lately, talk to an AI. Few of us look at the machine on the desk and see a supercomputer, with the kind of computational power that the world's greatest nations had a generation ago. Once you do see it, a different question follows. What can I do with all this capability? What could it do for my business, my job, or my wish to learn and understand something new? Which problems can a home or a small company solve today that a superpower could not solve twenty-five years ago?

The answers in this book begin with ordinary questions: whether three months of shop receipts contain a warning, how to divide people into fair teams, where a familiar sound occurs in a long recording. Each one comes with a little work you can do by hand. Each also grows into a much larger experiment, one that the card can repeat a million times while you watch.

AI changes how you reach that power. Combining serious algorithms used to need a specialist who knew each one by heart. A language model puts a natural-language front end on them: you describe the problem, and an agent can find the right method and call it. VectorFall is my attempt to democratize supercomputing. It is a laboratory of algorithms that runs on an NVIDIA graphics card, with the arithmetic, a place to work, and several ways to check the results. I call this *personal supercomputing*, or HPC for citizens. You can use the browser, write a program, or give the same operations to an AI agent through the Model Context Protocol (MCP). The most powerful high-performance computing algorithms are at your fingertips, and at your agent's too.

The phrase *personal supercomputing* is meant literally. When I was at the University of Texas at Austin, our classes in supercomputing and high-performance computing ran on Crays. I learned massively parallel programming with MPI on a Cray T3E, and the lesson that stayed with me had less to do with any particular machine than with a way of working. Many hard problems give way to a great many simple calculations. You sample the uncertain costs a million times, let thousands of searchers wander through possible schedules, or move a crowd of simulated bodies and watch what forms. Statistical and approximate methods like these have always been hungry for arithmetic. For most of computing history only large institutions could feed them.

That has changed. The Introduction tells the story of the Crays I learned on, and Chapter 14 measures how a small card compares with them. This book is about what you can do with that arithmetic once it is yours: on your own desk, on your own data, at any hour, with nobody else waiting for the machine.

WHO THIS BOOK IS FOR

This book is for curious people with data and a question. You do not need to know CUDA. You do not need to be a statistician. If you can read a short Python program and type a command in a terminal, you can use everything in this book. If you are a programmer, the Console and the API give you all the power of the card, and Chapter 4 shows you how to write your own GPU code in a few lines.

You will also need some patience with your own questions. The lab is quick to give an answer, and an answer that arrives in a fraction of a second is easy to believe. Much of this book is about the moment after the answer arrives: comparing it with chance, with a simple method you could have used instead, and with a small case you can check by hand. I call that comparison the golden rule of the lab.

HOW THIS BOOK IS ORGANIZED

Each chapter starts with a real problem and a short session that solves it. Then the chapter explains what happened and shows the variations. You can read the chapters in any order after Part I.

The sessions are real. Each listing in the book ran on a VectorFall server, and the output printed beside it is what that server returned. You can read the book in an armchair and follow the reasoning from the printed output, or read it at the keyboard and run each listing as you go. Both ways work. The second is more fun, and your numbers will differ from ours in ways that teach something.

Part I, *The Lab*
: Chapter 1 runs your first session. Chapter 2 installs VectorFall and shows how to look after the server. Chapter 3 is a tour of every screen. Chapter 4 teaches the Console, the Python environment that runs your code on the card.

Part II, *Your Problems*
: Chapter 5 teaches you to recognize which algorithm fits your problem. Chapters 6 to 11 solve problems from tables, teams, money, sound, text and pictures.

Part III, *Agents and Automation*
: Chapter 12 covers the `vf` command-line client, REST and JSON-RPC. Chapter 13 connects AI agents over MCP. Chapter 14 compares your card with scores from earlier machines. Chapter 15 is an illustrated tour from the Cray-1 to the exascale era. Chapter 16 looks at some of the work done on those machines and offers related experiments for your own card.

Appendixes
: Appendix A lists every model. Appendix B lists the supported cards. Appendix C is the reference for `vfctl` and the settings. Appendix D gives the license and the credits.

CONVENTIONS USED IN THIS BOOK

Italic
: New terms, and the names of screens and panels.

`Constant width`
: Code, commands, file names, model names such as `sim.budget`, and parameters such as `--horizon`.

`Constant width bold`

In a shell session, the commands that you type.

A Console listing is Python for the VectorFall Console. A Shell listing contains commands to run in a terminal. Where the book includes captured output, the label names the card when that information is available.

The recorded examples in the main chapters were captured on a VectorFall 0.1 server with a Quadro K1200 (Maxwell, 512 cores, 4 GB, 45 W). Chapter 13 includes fresh catalogue captures on a GTX 1080 and describes the separate setup used for its agent experiments. The output is sometimes shortened for the page. Expect your timings and random samples to vary.

USING CODE EXAMPLES

The data files of the examples are in the `manual/examples/` folder of the VectorFall source. The release source includes the guide build tools and the example runner, `manual/run_snippets.py`. You can try the examples on your own server and compare the output with the captures in the book.

You can use the code examples in your own programs and documents. You do not need permission for this, unless you copy a large part of the code. An attribution is welcome: *Personal Supercomputing with VectorFall*, by Amir Husain (2026).

THE LICENSE IN ONE PARAGRAPH

VectorFall uses the PolyForm Noncommercial License 1.0.0. The repository's `LICENSE` defines permitted uses. For commercial use, see `COMMERCIAL-LICENSE.md`. Appendix D lists the license and credits.

ACKNOWLEDGMENTS

The music samples are by Kevin MacLeod. The spoken digits come from the Free Spoken Digit Dataset by Zohar Jackson and the contributors. The photographers of Wikimedia Commons supplied the sample photos. VectorFall stands on NumPy, CuPy, SciPy, FastAPI and the CUDA toolkit. Thank you to all of them.

INTRODUCTION

INTRODUCTION: THE WORK A COMPUTER CAN DO

THE CRAY AT AUSTIN

In 1999 I was a graduate student at the University of Texas at Austin, and I worked on a Cray T3E. Our high-performance computing classes ran on Crays. To program one of those machines was to learn a particular way of thinking. The work was to take a simple step and arrange for it to happen an enormous number of times, on many processors at once. You laid out the problem so that the processors would rarely wait for one another.

I returned to that experience in the first VectorFall video. It gives me a way to see a change that is otherwise easy to miss. The kind of calculation I once associated with access to a large machine is now within reach of a small one. A graphics card that fits in an ordinary desktop can now do arithmetic that once needed a machine room. What I want to bring home is more than the arithmetic. It is the habit that the arithmetic made possible. You pose a question in a form that a computer can explore. The computer explores it, and then you look carefully at what it found.

That Cray belonged to a period when a supercomputer could mean hundreds of processors, each with its own memory, joined by a fast network. The University of Texas traces the Texas Advanced Computing Center to a reorganization in 2001. The center inherited the original Lonestar, an 88-processor, liquid-cooled Cray T3E, from its predecessor. My graduate work there began in the period before that reorganization. (See UT Austin's history of the Texas Advanced Computing Center.) At the time, the machine at the top of the world's TOP500 list was ASCI Red at Sandia National Laboratories, the first computer to pass one trillion floating-point operations a second on the LINPACK benchmark. Chapter 15 tells its story. A trillion operations a second was the frontier of national laboratories. In Chapter 1 a 45-watt card comes

close to that rate on a single-precision matrix product. It finishes in less time than it takes to read this sentence.

Those two figures do not measure the same thing, and Chapter 14 explains the difference with care. Even so, the fact that a card on a desk stands anywhere near that frontier is the reason for this book.

WHY APPROXIMATE ANSWERS ARE POWERFUL

I have long been interested in using large computers to work on complex problems by statistical and approximate means. Many of the most interesting questions do not have a formula you can solve with a pencil. How likely is this project to go over budget? Which of a million possible schedules treats everyone fairly? Does this recording contain the same sound twice? For questions like these, an exact answer is either unknown or so expensive to compute that nobody will ever see it.

Computing gives you a particular kind of leverage here. You can settle on a small rule, apply it many times, and see an answer emerge from the collection of results. Sometimes that means repeatedly sampling uncertain costs. Sometimes it means moving thousands of bodies and finding out which structures persist. Sometimes it means letting a search explore thousands of possible schedules, then asking how much its best schedule improves on the one you could draw up yourself. These are all ways to turn a hard question into a lot of small operations a machine can do.

Think of a rule simple enough to fit on a single sheet of paper, such as choosing how long a job takes and moving on to the next one. With pencil and paper you can work out a particular schedule. With a machine you can work out ten thousand schedules built from the same rule. Suddenly you can ask about the spread of finish times or the conditions under which the schedule slips. The repeated arithmetic produces evidence for a different question. One schedule tells you what might happen. Ten thousand tell you how often it does.

There is a quiet honesty in this method. An approximate method comes with a measure of its own uncertainty. When you estimate a probability from random trials, you can calculate the error to expect from chance alone. That error shrinks in a predictable way: four times as many trials cut it in half. The first chapter of this book estimates π this way and prints both the error it made and the error that chance would predict. If the two agree, the method is working. If they disagree, something is wrong, and you know to look. That simple comparison, between what you got and what

chance or a plain method would give, runs through every chapter. I think of it as the golden rule of the lab.

This is also why the graphics card suits the work so well. The designers of the GPU built it to color millions of pixels, each by the same small program. A million simulated futures, a million random points or a million candidate routes are, to the card, much the same as a million pixels. The number of trials you could afford has always set the limit on statistical methods. When the trials become cheap, the methods become practical for everyday questions.

WHAT CAN I BRING TO THIS MACHINE?

I began the VectorFall video series with the Text Engine. I loaded a collection of my own writing and followed its phrases through several different views. One view could find a passage and show where it came from. Another tracked which words tend to follow which, and a third could generate a continuation. Those are distinct capabilities. In the video I let a generated paragraph run long enough for its limitations to show. A sentence can start well and drift before the end. By tracing a transition or opening the source passage, I could get past the impression made by the prose and ask what the model had actually used. The Text chapter takes up that question with a corpus you can examine yourself.

Audio made the same idea audible. In the second video I played a song, applied a low-pass filter to the entire recording, and played the result. A filter has a particular job: in this case it attenuates components above a cutoff. You can hear that effect, see it in a spectrum, and apply the same operation to more recordings without changing your instructions. Think of someone looking for a repeated warning chime in a field recording, or an archivist preparing several hours of speech. A chain of small operations becomes a repeatable procedure. As soon as you know which parts need to vary, you have a program.

In the third episode the data was a staff roster and an Austin route. These were opportunities to ask what an ordinary list leaves out. The nursing example needed skills, coverage rules, and limits on the number of shifts. The route needed the cost of each leg. In either case, putting the names into a table was just the beginning. Once you write down the rules, you can try a small example by hand and discover whether you meant the same thing as the person who gave you the problem. The machine can take over the tedious search after that.

The Austin route makes the cost of searching tangible. With a fixed starting point there are 120 ways to visit five other stops and return. With ten other stops there are more than three million. Each added stop multiplies the count again, which is why I described the problem in the video as one that scales in a horrendous way. Good route algorithms use structure to avoid or organize much of that work. The check that a route's reported total matches the sum of its legs is simple enough to carry out independently. Separating the search from the check is a habit that pays off across this book. It also tells you what to ask an agent to show when it brings an answer back.

TRY THE FUTURE MANY TIMES

One demonstration in the fourth episode started with a project budget. Under the assumptions in that example, a budget of $180,000 was exceeded in about 23% of the simulated trials. At $190,000 the proportion was about 14%. The point of the experiment was the difference between those two risks, and what someone deciding on a budget might do with it. In a planning meeting you might ask whether the extra amount would be better spent on contingency, on buying down a particular risk, or on sequencing the work differently. A model gives you somewhere specific to put each proposal.

Think of a simulated future as one line in a long notebook. A random draw selects a cost for each uncertain task. Some trials include a supplier delay. Others have a cost overrun without a delay. At the end you add the costs and make a mark beside that line if the total is over budget. A GPU is well suited to this kind of work because many of the lines can be filled in independently. You can wait for the counts to settle, then spend the rest of your time looking at the assumptions.

Two kinds of uncertainty live in that notebook. With a finite number of trials, rerunning the same model gives slightly different counts. Running more trials can narrow that variation. The model itself also carries assumptions about how often things fail, how events are related, and how much each failure costs. Those assumptions can move an answer even when the simulated counts have settled. A beginner often confuses the two and runs ten times as many trials to fix a doubt that lives in the assumptions. In the economics chapter you'll see how to explore both kinds of uncertainty, first by making repeated runs and then by changing the conditions you gave the machine.

Another demonstration in that episode used a simulated fluid to recognize spoken digits. Imagine tapping a bowl of water at a succession of points. Each tap changes the surface, and the shape that remains when the tapping stops carries some information about the sequence. A computer can simulate a response, read selected features, and learn which features tend to accompany each label. We can compare it with simpler methods and keep some recordings out of training to see whether the learned rule carries over. The metaphor gives you a picture of the mechanism. The held-back examples provide evidence about whether it works.

It is an odd idea, a bucket of simulated liquid that listens, and that is part of its charm. It is also exactly the kind of idea that a personal supercomputer invites. Nobody would book time on a national machine to see whether a fluid can tell a spoken "three" from a spoken "seven". On a card at your desk you can try it, measure it against chance, and keep or discard the idea on the evidence.

A SMALL CARD AND A DIFFERENT ARCHITECTURE

The language prototype in VectorFall grew from the same conviction. In a follow-up video, I showed a model that I trained from scratch to run on a Quadro K420. That is a one-gigabyte card about fifteen years old, with 192 CUDA cores as I recalled on camera. There is a natural cynicism about how much knowledge so small a model can hold, and the cynicism is fair. My aim was different. I wanted to give the model basic fluency in English and keep the knowledge, and many specialized capabilities, outside it. The knowledge lives in a store of passages it can search. Deterministic tools do other jobs, and a small router recognizes a simple arithmetic question and answers it at once.

This is an old interest of mine. Many years ago I did some work in what was called Neural MRI and filed patents in that area. Over the years I kept coming back to one question: how to separate basic language skills from knowledge. Knowledge changes anyway, so it should not be frozen into the static weights of a network. In the video I borrowed Marvin Minsky's phrase, a *society of mind*: deterministic and non-deterministic parts working together, each doing what it does efficiently. The network supplies the one thing it seems uniquely good at, fluent rewriting. Careful design supplies everything around it.

On camera the prototype wrote its answers at between about 23 and 32 tokens a second. Several clients shared the same K420 at the time, and the card had only 352

megabytes in use. When it had no source to draw on, its English stayed fluent and its facts did not, and I left those mistakes on screen. The book's own Language chapter treats the prototype in the same spirit, with its limits labeled. As I said in that video, there is nothing special about the K420. I picked it because I had it. The interesting work is in the design around a small, tight substrate.

HOW TO READ A FAST NUMBER

The last episode brought the experiments down to a Python Console. On camera the software calculated the mean and median of ten million numbers, and the display showed 102 milliseconds. That was one run of one demonstration. The exact speed on another machine depends on what is timed, where the data starts, and which library and hardware carry out the operation. You can try a similar question on your own card and decide whether you care about how fast the arithmetic runs or how long the entire job takes.

In the same episode the screen expressed a quick benchmark as a count of Cray-1s, reporting 1,016 in that demonstration. That comparison was meant to convey how much arithmetic can hide in an unremarkable card. It also raises the question of which arithmetic we are counting. A single-precision multiplication test and a historical double-precision linear solve report different things. The Lineage chapter sets up a narrower experiment with a measured result, a precision, a benchmark family and a residual.

The Cray-1 is a good yardstick for another reason. The National Center for Atmospheric Research took delivery of one in 1977. It paid about $8.86 million for the system and its disks, and the machine drew about 115 kilowatts. NCAR credits it with big advances in modeling climate and severe storms, and it ran there for almost twelve years. Those were serious problems, solved with patience on a machine that few people would ever touch. Chapter 15 tells that story with its sources. When a card in a desktop counts its speed in Cray-1s, the comparison carries all of that history with it.

One attraction of the old Quadro K420 in those videos was its modest cost. I corrected the model name in the second episode. The price was what I encountered shopping for a used card at the time, and card condition, availability and driver support vary. A little card can have a short memory and surprisingly quick arithmetic. A different GPU may have a stronger double-precision unit or enough memory to keep an entire

dataset on the device. By running a real question and watching what happens, you can decide which kind of strength matters to you.

PUT THE QUESTION WHERE YOU CAN SEE IT

The browser, the Console and an agent are three ways to approach the same lab. I use the browser when I want to turn one control and watch its effect. I use the Console when I want a procedure I can rerun and a record of how I got the result. An agent is handy when a question requires looking up tables, checking their shape and making a series of calls. VectorFall speaks the Model Context Protocol, so an agent such as Claude or Codex can hand its heavy arithmetic to your card instead of the cloud. As I put it in the video, that can save a lot of tokens. In each case the measurements still matter. Chapter 13 puts this to a test with a local model and a routing problem. You will see how often the tool-assisted runs found the correct optimum, and how much time and context they used to do it.

ALGORITHMS IN THE AGE OF AI

A language model can write a solver for almost any problem you describe: a route search, a scheduler or a Monte Carlo simulation. The program it writes may well be correct. I think there is a better division of labour. Let the model understand the question and choose the method, and let a known algorithm do the calculation. VectorFall is built for that arrangement, for four reasons.

The first reason is explainability. When an agent calls a VectorFall model, the answer is not constructed on the fly. A known algorithm does the calculation, the same one every time, with its parameters in the request and its method in the reply. Each model is a registered function with a description, an input schema and example inputs, and `vf test` runs every one of them against those examples. Even when an LLM starts the work, you can say which algorithm produced the number, with which settings, and what it was compared with.

The second reason is tokens. An agent that solves a problem from scratch writes the algorithm, writes a program around it, downloads libraries and scaffolding, fixes the errors, and only then runs it. Each step costs tokens and time, and the next question starts again from nothing. In VectorFall the algorithms, the data formats and the GPU code already exist, so they are quick to use. The agent reads a short description and

sends the inputs. Chapter 13 shows how far those descriptions can be compressed: loading tools by keyword cut the agent's input tokens by about 68 percent.

The third reason is the GPU. VectorFall's algorithms are written for the card first. A model that writes an ad hoc solution usually writes ordinary code for the processor. That code can be correct and still leave the most capable part of the computer idle. On the card, the work gains speed and room to scale: thousands of walkers or a million simulated futures instead of one. The processor also stays free for you. A large calculation on the CPU can slow down the whole computer; on the card, it runs in parallel while you carry on with other work.

The fourth reason is verification. When a well-understood problem has a known solution, available as a module or an MCP call, you can test that solution and rely on what you learned. You know what it computes, whether it is exact or approximate, and how it reports its own check against a baseline and against chance. A program generated afresh for each question has no such record, and you have to check it every time. Chapter 13 measures one case. A local 27B model working alone did not find the best route in any of 10 runs; the same model calling VectorFall found the proven optimum in all 10.

So VectorFall has two front ends. The web interface shows each capability visually: a person turns a control and watches the result. The MCP and API server lets an agent of your choice become the front end. The agent discovers the tools, reads their descriptions and calls them for you. Behind both, the same algorithms do the work on the same card.

WHY OWN A SUPERCOMPUTER

There is a difference between borrowing a large computer and owning one. Borrowed time is precious, so it tends to go to questions you are already sure about. A machine on your own desk changes that. It is always on, it answers in seconds, and the only jobs in its queue are yours. You can try a half-formed idea at the end of the day, look at the result, and throw it away without anyone keeping count. Your data stays on your computer: by default the lab answers only the machine it runs on.

Owning the machine also changes what you learn. When the card is yours, you watch its memory fill. You see a kernel take two seconds where you expected two milliseconds, and you find out why. The lessons that once needed a seat in a supercomputing class come to you through ordinary problems. A budget, a roster, a

song or a column of odd numbers will teach them. Those are the problems most people actually have.

A good place to start is a question you recognize. There may be one in a spreadsheet you have opened all year, in a recording you keep going back to, or in a schedule that becomes awkward whenever a person is away. Write down what you know, what you need to find, and how you would check one small case. That gives you a first experiment. The rest of the book gives you ways to run it, study it and ask a better question after you have seen the answer.

SOURCES FOR THE INTRODUCTION

- Amir Husain, *VectorFall*, Episodes 1 through 5, *Deep Tech with Amir Husain*. Episode 1, 00:38–01:32 and 10:11–10:59 (used GPUs, UT Austin and a language-model demonstration). Episode 2, 00:09–00:23 and 02:38–04:12 (card correction, audio filtering and MCP). Episode 3, 04:40–05:11 and 06:35–07:30 (rostering and routing). Episode 4, 01:59–02:24 and 13:20–14:15 (project-budget simulation and fluid reservoir). Episode 5, 00:47–01:07 and 02:42–02:59 (quick benchmark and Console demonstration).
- Amir Husain, *Vector LM: Under the Hood*, *Deep Tech with Amir Husain*. 00:18–01:45 (a model trained from scratch for the K420, and the separation of fluency from knowledge), 04:08 and 09:06 (token rates), 09:45–09:57 (memory in use), 15:35–16:13 and 17:46–18:50 (the neurosymbolic design and the society of mind).
- The University of Texas at Austin, *A Decade of Discovery*, June 21, 2011. The transition to TACC and the original Lonestar.
- ASCI Red, the Cray-1 at NCAR: TOP500 lists and the sources collected for Chapter 15 (`sources/lineage_data.json`).

PART I

THE LAB

CHAPTER 1

A SUPERCOMPUTER ON YOUR DESK

Suppose someone hands you a graphics card and tells you it can out-calculate the Cray supercomputers of the early 1990s. The claim sounds like a sales pitch. What would a fair test look like? It would ask the card to do real work, compare it with something you already trust, and show you how much of the result to believe.

Give the card three jobs before you form an opinion about it. The first is a race between two implementations of the same matrix multiplication. The second is a statistical estimate whose answer you already know. The third is a question about a roomful of people. These are small experiments chosen to make different parts of a computation visible: the time to do the arithmetic, the error you should expect, and the answer a program can deliver to another program.

If your main interest is a dataset of your own, keep it nearby as you try these examples.

THE FIRST SESSION

Open `http://localhost:8092` in a browser. (If VectorFall is not installed yet, Chapter 2 installs it in three commands.) Click *Console* in the rail on the left. Type this program and press *Run* (or `Ctrl`+`Enter`):

CONSOLE

```
def matmul(xp, n):
    a = xp.ones((n, n), dtype=xp.float32)  ❶
    return a @ a

vf.race(matmul, 2048)  ❷
```

```
OUTPUT · QUADRO K1200

6.9× faster
CPU                                                    130.8 ms
```

```
Card                                                          18.84 ms
done in 0.93 s
```

1. xp is np (CPU) or cp (the card)
2. the same function on the CPU and on the card

`vf.race` runs your function two times: one time with NumPy on the CPU, and one time with CuPy on the card. A 2048 by 2048 matrix product is about 17 billion floating-point operations. The card did it in less than 20 milliseconds, which is close to one trillion operations each second. The CPU in this test computer has 12 cores and a good BLAS library, so it is not slow. On a smaller computer the difference is much larger.

Where does the figure of 17 billion come from? Each entry of the product is a sum of 2,048 multiplications and about as many additions. There are 2,048 times 2,048 entries, so the total is two times 2,048 cubed, which is a little over 17 billion. In our capture the CPU took about 131 milliseconds and the card about 19, a factor of about seven. The habit to form is to count the operations first and then divide by the time.

Now ask a question that has a known answer. What fraction of random points in a square falls inside the circle that touches its sides? The answer is π/4. Throw two hundred million points on the card and see how close the estimate comes:

CONSOLE

```
n, inside = 0, 0
for _ in range(20):  (1)
    x = cp.random.random(10_000_000, dtype=cp.float32)
    y = cp.random.random(10_000_000, dtype=cp.float32)
    inside += int(((x * x + y * y) ⩽ 1.0).sum())
    n += x.size
est = 4 * inside / n
print("estimate of pi:", est)
print("error:         ", abs(est - math.pi))
print("expected error from chance alone: about", round(4 * math.sqrt((math.pi
/ 4) * (1 - math.pi / 4) / n), 6))
```

OUTPUT · QUADRO K1200

```
estimate of pi: 3.14148692
error:          0.00010573358979293346
expected error from chance alone: about 0.000116
done in 0.62 s
```

1. 20 batches of 10 million points fit on a 1 GB card

A Monte Carlo estimate always has an error from chance. Two hundred million points give an expected error of about one part in ten thousand. If your error is of that size, the method works. If it is much bigger, repeat the experiment and inspect the calculation. Many tools in VectorFall report a comparison or a check as part of the result.

Our run printed an estimate of 3.14148692, an error of about 0.000106 and an expected error of about 0.000116. The two errors agree. The formula on the last line comes from treating each point as a coin toss that lands inside the circle with probability $\pi/4$. The spread of the fraction of heads shrinks with the square root of the number of tosses. So a hundred times as many points buys only ten times the accuracy.

For the third experiment, leave the browser. Open a terminal and ask the server how many people must be in a room before two of them probably share a birthday:

SHELL

```
$ vf run forecast.birthday
```

OUTPUT · QUADRO K1200

```
23 people
With only 23 people in a room, it is more probable than not that two have the
same birthday.
  tone: good
  paths: 23700000
  ms: 214.1
  (also: chart, facts — use --json or --out to see them)
[forecast.birthday: 214.4 ms wall · 214.2 ms on the GPU in 1 runs]
```

The `vf` client sent one request. The server simulated rooms on the card and answered with a headline, a sentence and the numbers behind them. The browser, the API, the command line and an AI agent all reach the same models.

The answer, 23 people, surprises most people the first time. The trick is to count pairs. In a room of 23 there are 253 different pairs, and each pair is a small chance of a match. With that many small chances, a match becomes more likely than not. In our capture the server simulated 23.7 million paths in about 214 milliseconds. You could work out this answer with an exact formula.

WHAT HAPPENED

Three things happened in this session, and they are the core of VectorFall.

The card did the arithmetic. CuPy is NumPy for the GPU. Each operation on a `cp` array starts a *kernel* that runs on every element in parallel, on hundreds or thousands of cores. Python itself still runs on the CPU. It only tells the card what to do.

The answer came with a check. The π estimate came with the error that chance alone gives. The birthday answer is a simulation that you can compare with the exact formula. In Part II, each tool compares its result with a simple baseline: a greedy plan, a snake draft, last year's value, the most common class.

Every operation is a model. The birthday simulation is the model `forecast.birthday`. VectorFall 0.1 has more than two hundred models. Each one has a name, a title, typed parameters with limits, and a note that says if it runs on the GPU, on the CPU or on both. You can call each model in five ways:

WAY IN	HOW	WHO USES IT
The web interface	`http://localhost:8092`	you, with a mouse
The `vf` client	`vf run sim.budget --budget 200000`	you, in a terminal or a script
REST and OpenAPI 3.1	`POST /v1/models/sim.budget`	programs in any language
JSON-RPC 2.0	`POST /v1/rpc` with `"method": "sim.budget"`	programs that send batches
MCP	`http://localhost:8092/mcp`, tool `sim_budget`	AI agents

One function checks the parameters for all five. A bad value gets the same message everywhere.

HOW THE SERVER USES THE CARD

VectorFall is one server process in one container. It owns one card. A single *GPU worker* thread runs all the work for the card, one job at a time, from a queue. This design keeps the card healthy: two jobs never fight for memory, and a stuck job cannot block the server. The worker writes each job in the *Log*, and the top bar of the web interface shows the card's activity, memory, temperature and the speed of the last job.

Long work runs as a *job*. A job reports its progress, and you can cancel it. The interface stays responsive while the card works.

The Console is different. Your code runs in its own Python process, with its own access to the card. If your code loops forever, the *Stop* button ends that process and the rest of VectorFall continues.

> **NOTE** A card that also drives a display has a watchdog. The driver stops any single kernel that runs longer than about two seconds. VectorFall splits its own long work into short kernels. Chapter 4 shows how to do the same in your code. A card without a display has no such limit.

HOW MUCH COMPUTER IS THIS?

The *Machine* screen measures the card. You can also measure it from a terminal:

SHELL

```
$ vf run machine.benchmark
```

OUTPUT · QUADRO K1200

```
  sgemm_gflops: 1030.2
  sgemm_n: 2048
  fft_ms: 0.94
  scan_gbs: 27.0
[machine.benchmark: 655.3 ms wall · 295.3 ms on the GPU in 1 runs]
```

The first line is the speed of a 2048 by 2048 single-precision matrix product (SGEMM) from the cuBLAS library. On the Quadro K1200, a 45-watt card from 2015, that is close to one trillion floating-point operations each second. The Cray-1 had a peak of 160 million. Chapter 14 makes this comparison with care, because single precision and double precision are not the same thing.

Our capture reported 1030.2 GFLOP/s for the matrix product, 0.94 milliseconds for an FFT, and a memory scan at 27.0 GB/s. Each number measures a different part of the card. The matrix product shows how fast the cores can multiply and add when the data is ready for them. The scan shows how fast the card can read its own memory. When a program runs slower than the headline figure suggests, memory is usually the first place to look.

The power is real. But a GPU is not faster at everything. It is fast when the same operation applies to many numbers at the same time: matrices, images, audio spectra, millions of simulated futures, thousands of random walkers. It is slow at long chains

of decisions that depend on each other. Chapter 5 teaches you to see the difference in your own problems.

A kitchen makes the distinction concrete. A hundred cooks can chop a hundred onions in the time one cook chops one. They cannot bake one cake a hundred times faster, because each step waits for the last. Your problems will have both kinds of work in them. The art is to find the onions.

TRY THE RACE AT TWO SIZES

The result from one size is a snapshot. Run the same Console program at a smaller size, perhaps 128, and then again at 2,048. The number of operations grows with the cube of the dimension, so the larger job has much more arithmetic to do. The two calls both have a fixed cost involved in preparing and starting work. At the smaller size the fixed cost can be a large part of the experiment.

Run each size at least twice, make a note of the first run, and compare the warm runs separately. If the first run is slower, it may be initializing a context or loading a library. In an interactive session you may care about that first wait. If your program will do the job all day, the later runs tell you something more useful.

Time what matters to your question. A table stored on disk may need to be read, parsed and moved to the card before the arithmetic starts. If those steps take a second and the calculation takes ten milliseconds, making the calculation twice as fast will save five milliseconds on the whole job. A later chapter calls several operations on data kept on the card. Keeping data near its next use can matter more than improving the fastest operation in isolation.

WHAT YOU LEARNED

In one short session you raced the card against a capable CPU, estimated π and checked the error against chance, and asked the server a question from the terminal. Along the way you met the ideas that the rest of the book builds on:

- The Console runs Python with NumPy (`np`) on the CPU and CuPy (`cp`) on the card. `vf.race` compares the two.
- Every result in VectorFall comes with a check against chance or a baseline.

- Every operation is a model. You can call it from the browser, the `vf` client, REST, JSON-RPC and MCP.
- One GPU worker runs the jobs for the card. Long jobs show progress and you can cancel them.

TRY THIS NEXT

Change the π program to use two million points instead of two hundred million, by reducing the number of batches or the batch size. Before you run it, predict the expected error: a hundred times fewer points should make it about ten times larger. Then run it a few times and see whether the actual errors scatter around your prediction. If you want a second experiment, run `vf run machine.benchmark` twice and note how much the figures move between runs. That spread is the noise floor for every speed comparison you will make on this card.

End of the sample

The full book continues with 15 more chapters: installing VectorFall, a tour of the lab, Python on the card, tables, teams, budgets, sound, text and pictures, AI agents over MCP, and an illustrated history of the supercomputers.

Paperback (premium colour, 248 pages)

amazon.com/dp/1965686052

Kindle

amazon.com/dp/B0GYCTT9TN

VectorFall, the free guide and the install commands

amirhusain.com/software/vectorfall

As an Amazon Associate I may earn from qualifying purchases.
Copyright 2026 Amir Husain. Published by AM Press. Sample for personal reading.