00:00:00:03 - 00:00:02:04
Before we get into the details of CETRA

00:00:02:04 - 00:00:03:14
it's worth spending a moment on

00:00:03:14 - 00:00:06:11
why GPUs are a good fit for this problem.

00:00:06:11 - 00:00:08:00
A modern CPU is designed

00:00:08:00 - 00:00:10:00
for speed on sequential tasks.

00:00:10:00 - 00:00:12:03
It has a small number of powerful cores,

00:00:12:03 - 00:00:13:21
typically somewhere between 8

00:00:13:21 - 00:00:16:13
and 64 on a high-end workstation,

00:00:16:13 - 00:00:17:10
each capable

00:00:17:10 - 00:00:19:11
of executing complex instructions

00:00:19:11 - 00:00:21:05
very quickly.

00:00:21:05 - 00:00:23:06
A GPU, by contrast,

00:00:23:06 - 00:00:25:10
has thousands of much simpler cores

00:00:25:10 - 00:00:27:12
designed to execute the same instruction

00:00:27:12 - 00:00:30:21

on many pieces of data simultaneously.

00:00:30:21 - 00:00:32:05
The raw floating point

00:00:32:05 - 00:00:34:05
throughput of a high-end GPU

00:00:34:05 - 00:00:35:10
can be an order of magnitude

00:00:35:10 - 00:00:37:08
or more greater than a CPU,

00:00:37:08 - 00:00:40:19
but only if you can keep all those cores busy.

00:00:40:19 - 00:00:42:09
Transit detection is a good fit

00:00:42:09 - 00:00:43:08
for this model.

00:00:43:08 - 00:00:44:15
The core operation -

00:00:44:15 - 00:00:45:21
comparing a transit model

00:00:45:21 - 00:00:47:03
to a light curve

00:00:47:03 - 00:00:49:14
at a given transit time and duration -

00:00:49:14 - 00:00:51:05
is the same operation

00:00:51:05 - 00:00:52:11
repeated millions of times

00:00:52:11 - 00:00:54:11
across a grid of parameters.

00:00:54:11 - 00:00:55:13
Each evaluation

00:00:55:13 - 00:00:57:24
is largely independent of the others.

00:00:57:24 - 00:00:58:17
That's exactly

00:00:58:17 - 00:01:01:05
the kind of problem GPUs are built for.

00:01:01:05 - 00:01:03:02
When you write code for a GPU,

00:01:03:02 - 00:01:04:02
the function that runs on it

00:01:04:02 - 00:01:05:16
is called a kernel.

00:01:05:16 - 00:01:07:22
You write a single kernel that describes

00:01:07:22 - 00:01:09:05
what one thread should do,

00:01:09:05 - 00:01:10:13
and then launch it across

00:01:10:13 - 00:01:13:11
a grid of thousands of threads simultaneously.

00:01:13:11 - 00:01:15:18
Each thread knows its own position

00:01:15:18 - 00:01:17:19
in that grid, its thread index,

00:01:17:19 - 00:01:19:09
and uses that to determine

00:01:19:09 - 00:01:21:21
which piece of data it should work on.

00:01:21:21 - 00:01:23:23
Threads are grouped into blocks,

00:01:23:23 - 00:01:27:03
and blocks are grouped into a grid.

00:01:27:03 - 00:01:28:06
Within the GPU,

00:01:28:06 - 00:01:29:04
the hardware unit

00:01:29:04 - 00:01:30:23
that manages a set of blocks

00:01:30:23 - 00:01:32:22
is called a streaming multiprocessor,

00:01:32:22 - 00:01:34:01
or SM,

00:01:34:01 - 00:01:35:20
a self-contained processing unit

00:01:35:20 - 00:01:37:18
that manages its own set of threads

00:01:37:18 - 00:01:39:15
and its own shared memory.

00:01:39:15 - 00:01:41:10
When designing a GPU algorithm,

00:01:41:10 - 00:01:43:04
deciding how to map your problem

00:01:43:04 - 00:01:44:13
onto this hierarchy -

00:01:44:13 - 00:01:47:01

which dimension corresponds to which parameter,

00:01:47:01 - 00:01:48:19
how big your block should be -

00:01:48:19 - 00:01:50:20
is one of the most important decisions

00:01:50:20 - 00:01:51:22
you make.

00:01:51:22 - 00:01:53:13
CETRA is implemented

00:01:53:13 - 00:01:55:23
using NVIDIA's CUDA platform.

00:01:55:23 - 00:01:56:19
In practice,

00:01:56:19 - 00:01:57:21
this means writing

00:01:57:21 - 00:01:59:21
kernel code in CUDA C,

00:01:59:21 - 00:02:00:09
which is then

00:02:00:09 - 00:02:02:17
compiled and launched from Python

00:02:02:17 - 00:02:04:17
using the PyCUDA library.

00:02:04:17 - 00:02:05:19
If you've not encountered

00:02:05:19 - 00:02:07:14
GPU programming before,

00:02:07:14 - 00:02:08:24
the learning curve is real,

00:02:08:24 - 00:02:10:06
but manageable.

00:02:10:06 - 00:02:11:19
And as we'll see, the performance

00:02:11:19 - 00:02:14:05
payoff is well worth the effort.

00:02:14:05 - 00:02:15:10
If there's one thing to take away

00:02:15:10 - 00:02:16:15
from this section,

00:02:16:15 - 00:02:18:09
it’s that on a GPU, memory

00:02:18:09 - 00:02:19:16
access patterns matter

00:02:19:16 - 00:02:22:22
at least as much as the computation itself.

00:02:22:22 - 00:02:25:07
GPUs have several tiers of memory,

00:02:25:07 - 00:02:26:24
each with very different speed

00:02:26:24 - 00:02:28:22
and capacity characteristics.

00:02:28:22 - 00:02:31:09
At the top of the hierarchy are registers,

00:02:31:09 - 00:02:32:17
a small amount of memory

00:02:32:17 - 00:02:34:04
private to each thread,

00:02:34:04 - 00:02:36:21
accessible in just a few clock cycles.

00:02:36:21 - 00:02:38:16
Below that is shared memory,

00:02:38:16 - 00:02:39:09
which is shared

00:02:39:09 - 00:02:41:06
between all threads in a block

00:02:41:06 - 00:02:43:06
and within the same SM.

00:02:43:06 - 00:02:45:15
It's still fast, a few clock cycles,

00:02:45:15 - 00:02:48:09
but limited in size, typically around 48

00:02:48:09 - 00:02:51:20
to 192 kB per SM, depending on the GPU.

00:02:51:20 - 00:02:53:12
Then there's global memory -

00:02:53:12 - 00:02:56:24
the main GPU DRAM, 40 GB on an A100 -

00:02:56:24 - 00:02:59:06
which is where most of your data lives.

00:02:59:06 - 00:03:01:06
It's large and relatively slow,

00:03:01:06 - 00:03:02:12
with access latencies

00:03:02:12 - 00:03:04:07
of hundreds of clock cycles,

00:03:04:07 - 00:03:05:16

though its bandwidth is high

00:03:05:16 - 00:03:07:17
if you access it in the right pattern.

00:03:07:17 - 00:03:09:05
Finally, there's host memory -

00:03:09:05 - 00:03:10:15
the RAM in your computer -

00:03:10:15 - 00:03:12:08
which the GPU can only access

00:03:12:08 - 00:03:15:04
by transferring data across the PCIe bus,

00:03:15:04 - 00:03:16:15
which is slower still.

00:03:16:15 - 00:03:18:15
The practical implication is this:

00:03:18:15 - 00:03:19:22
if your kernel repeatedly

00:03:19:22 - 00:03:21:04
reads from global memory,

00:03:21:04 - 00:03:22:14
you will spend most of your time

00:03:22:14 - 00:03:23:13
waiting for data

00:03:23:13 - 00:03:25:23
rather than doing computation.

00:03:25:23 - 00:03:28:10
The goal is to minimise the time spent

00:03:28:10 - 00:03:30:03
reading and writing data

00:03:30:03 - 00:03:31:17
from and to memory.

00:03:31:17 - 00:03:33:11
This principle directly shaped

00:03:33:11 - 00:03:35:08
several of CETRA's design decisions,

00:03:35:08 - 00:03:37:04
which we'll come to in the next section.

00:03:37:04 - 00:03:38:16
One other important concept

00:03:38:16 - 00:03:40:15
is coalesced memory access.

00:03:40:15 - 00:03:42:06
When threads in the same warp -

00:03:42:06 - 00:03:43:16
a group of 32 threads

00:03:43:16 - 00:03:44:22
that execute together -

00:03:44:22 - 00:03:46:15
access global memory,

00:03:46:15 - 00:03:48:20
the hardware can combine their requests

00:03:48:20 - 00:03:50:09
into a single transaction

00:03:50:09 - 00:03:53:04
if the addresses are contiguous.

00:03:53:04 - 00:03:54:23
If the addresses are scattered,

00:03:54:23 - 00:03:56:14
you pay the full latency cost

00:03:56:14 - 00:03:58:17
for each access individually.

00:03:58:17 - 00:04:00:03
Designing your data layout

00:04:00:03 - 00:04:00:24
and access patterns

00:04:00:24 - 00:04:02:20
so that neighbouring threads read

00:04:02:20 - 00:04:03:24
neighbouring addresses

00:04:03:24 - 00:04:06:22
is therefore an important optimisation.

00:04:06:22 - 00:04:07:24
These constraints -

00:04:07:24 - 00:04:09:02
the thread hierarchy,

00:04:09:02 - 00:04:10:06
the memory hierarchy

00:04:10:06 - 00:04:12:17
and the importance of access patterns -

00:04:12:17 - 00:04:14:01
are all important,

00:04:14:01 - 00:04:15:01
but they're secondary

00:04:15:01 - 00:04:17:12
to a more fundamental question:

00:04:17:12 - 00:04:18:16

is your algorithm

00:04:18:16 - 00:04:21:01
even the right shape for the hardware?

00:04:21:01 - 00:04:23:00
All the low-level tuning in the world

00:04:23:00 - 00:04:25:11
won't save a poorly structured algorithm.

00:04:25:11 - 00:04:26:11
And conversely,

00:04:26:11 - 00:04:27:21
getting the algorithm right

00:04:27:21 - 00:04:29:22
makes the lower-level optimisations

00:04:29:22 - 00:04:31:14
much easier to achieve

00:04:31:14 - 00:04:32:14
and in some cases

00:04:32:14 - 00:04:34:01
takes care of them naturally.

00:04:34:01 - 00:04:35:19
Algorithm design is arguably

00:04:35:19 - 00:04:36:21
the most important

00:04:36:21 - 00:04:39:13
part of GPU optimisation.

00:04:39:13 - 00:04:41:04
This is something I learned early

00:04:41:04 - 00:04:43:05
in the development of CETRA.

00:04:43:05 - 00:04:44:04
My first attempt

00:04:44:04 - 00:04:45:01
was essentially

00:04:45:01 - 00:04:46:00
a direct port

00:04:46:00 - 00:04:48:05
of the TLS approach to CUDA -

00:04:48:05 - 00:04:49:17
take the existing algorithm,

00:04:49:17 - 00:04:51:15
translate it into GPU kernels

00:04:51:15 - 00:04:52:19
and see what happens.

00:04:52:19 - 00:04:54:02
It didn't work well.

00:04:54:02 - 00:04:55:14
TLS relies on sorting

00:04:55:14 - 00:04:56:22
the light curve in phase space

00:04:56:22 - 00:04:58:10
for each trial period,

00:04:58:10 - 00:05:00:00
and sorting is an operation

00:05:00:00 - 00:05:01:11
that doesn't parallelise

00:05:01:11 - 00:05:03:05
efficiently on a GPU.

00:05:03:05 - 00:05:04:10
The alternative -

00:05:04:10 - 00:05:05:20
sorting on the CPU

00:05:05:20 - 00:05:08:12
and transferring the result to the GPU -

00:05:08:12 - 00:05:10:05
meant constant back and forth

00:05:10:05 - 00:05:12:08
between host and device memory,

00:05:12:08 - 00:05:13:20
and was in fact an order of magnitude

00:05:13:20 - 00:05:15:09
slower than TLS.

00:05:15:09 - 00:05:16:02
It became clear

00:05:16:02 - 00:05:18:02
that a new approach was needed,

00:05:18:02 - 00:05:18:20
one designed

00:05:18:20 - 00:05:19:15
from the ground up

00:05:19:15 - 00:05:20:24
with the GPU’s strengths

00:05:20:24 - 00:05:22:18
and constraints in mind.

00:05:22:18 - 00:05:24:07
That's what CETRA is.

00:05:24:07 - 00:05:26:21

I started developing CETRA in 2022,

00:05:26:21 - 00:05:28:06
having already experimented

00:05:28:06 - 00:05:30:01
on an earlier project with Numba,

00:05:30:01 - 00:05:30:24
a Python library

00:05:30:24 - 00:05:32:21
that can compile GPU kernels

00:05:32:21 - 00:05:34:13
directly from Python code.

00:05:34:13 - 00:05:36:09
Numba is a natural entry point

00:05:36:09 - 00:05:37:14
for GPU programming

00:05:37:14 - 00:05:38:06
if you're coming from

00:05:38:06 - 00:05:40:16
a scientific Python background,

00:05:40:16 - 00:05:41:16
and it does support

00:05:41:16 - 00:05:42:23
shared memory and

00:05:42:23 - 00:05:44:10
fine-grained kernel control.

00:05:44:10 - 00:05:45:18
But I hit a limitation

00:05:45:18 - 00:05:47:01
that I needed to work around

00:05:47:01 - 00:05:48:03
that was only accessible

00:05:48:03 - 00:05:50:06
via CUDA C directly,

00:05:50:06 - 00:05:52:07
so I made the switch at that point.

00:05:52:07 - 00:05:53:20
Having made it, I stuck with it

00:05:53:20 - 00:05:55:15
and I think it was the right call.

00:05:55:15 - 00:05:56:09
CUDA C

00:05:56:09 - 00:05:57:19
gives you full explicit

00:05:57:19 - 00:05:58:13
control over

00:05:58:13 - 00:06:00:21
every aspect of kernel execution,

00:06:00:21 - 00:06:03:03
and the documentation is excellent.

00:06:03:03 - 00:06:04:22
For a novice GPU developer,

00:06:04:22 - 00:06:06:21
that combination is hard to beat.

00:06:06:21 - 00:06:08:11
The obvious question is

00:06:08:11 - 00:06:10:11
why not use a more portable framework?

00:06:10:11 - 00:06:12:10
Something that would run on AMD

00:06:12:10 - 00:06:14:06
hardware as well as NVIDIA.

00:06:14:06 - 00:06:16:15
JAX is the most commonly suggested

00:06:16:15 - 00:06:17:17
alternative today,

00:06:17:17 - 00:06:20:08
and it does now support AMD GPUs.

00:06:20:08 - 00:06:22:00
But back in 2022,

00:06:22:00 - 00:06:24:18
AMD support for JAX was in its infancy,

00:06:24:18 - 00:06:25:12
and in any case,

00:06:25:12 - 00:06:27:09
JAX is designed around high-level

00:06:27:09 - 00:06:28:09
array operations

00:06:28:09 - 00:06:30:15
and automatic differentiation -

00:06:30:15 - 00:06:32:07
it would abstract away exactly

00:06:32:07 - 00:06:34:02
the low-level memory control

00:06:34:02 - 00:06:36:16
that CETRA's key optimisations depend on.

00:06:36:16 - 00:06:38:22

OpenCL would have been the right

00:06:38:22 - 00:06:41:04
portability choice, and with hindsight,

00:06:41:04 - 00:06:42:08
I might have gone that route

00:06:42:08 - 00:06:43:18
from the start.

00:06:43:18 - 00:06:44:17
The programming model

00:06:44:17 - 00:06:46:01
is similar enough to CUDA

00:06:46:01 - 00:06:47:21
that the transition isn't large.

00:06:47:21 - 00:06:50:01
Some effort has been made to port CETRA

00:06:50:01 - 00:06:50:24
to OpenCL,

00:06:50:24 - 00:06:52:19
and early testing on AMD hardware

00:06:52:19 - 00:06:54:09
looks promising.

00:06:54:09 - 00:06:55:01
If you're starting

00:06:55:01 - 00:06:56:07
a similar project today,

00:06:56:07 - 00:06:58:00
my suggestion would be:

00:06:58:00 - 00:06:59:07
learn the fundamentals

00:06:59:07 - 00:07:00:21
with Numba or CUDA,

00:07:00:21 - 00:07:02:20
then consider whether portability matters

00:07:02:20 - 00:07:06:00
enough to your use case to justify OpenCL

00:07:06:00 - 00:07:07:03
from the outset.