Tiny functions for codecs, compilation, and (maybe) soon everything

Keith Winstein


Stanford University
Saratoga High School
Dropbox
UC San Diego
MIT
Granular, functional interfaces to computing resources will enable new applications.

It’s worth refactoring megamodules (codecs, TCP, compilers, machine learning) using ideas from functional programming.

Saving and restoring program states is a powerful tool.
Tiny functions for four different purposes

**ExCamera:** Fast interactive video encoding

**Lepton:** JPEG recompression in a distributed filesystem

**Salsify:** Videoconferencing with co-designed codec and transport protocol

**gg:** general IR for lambda-style granular computation
Tiny functions for four different purposes

**ExCamera:** Fast interactive video encoding
  - “functional” video codec for fine-grained **parallelism**

**Lepton:** JPEG recompression in a distributed filesystem

**Salsify:** Videoconferencing with co-designed codec and transport protocol

**gg:** general IR for lambda-style granular computation
Tiny functions for four different purposes

**ExCamera**: Fast interactive video encoding
- “functional” video codec for fine-grained **parallelism**

**Lepton**: JPEG recompression in a distributed filesystem
- “functional” JPEG codec for boundary-oblivious **sharding**

**Salsify**: Videoconferencing with co-designed codec and transport protocol

**gg**: general IR for lambda-style granular computation
Tiny functions for four different purposes

ExCamera: Fast interactive video encoding
- “functional” video codec for fine-grained parallelism

Lepton: JPEG recompression in a distributed filesystem
- “functional” JPEG codec for boundary-oblivious sharding

Salsify: Videoconferencing with co-designed codec and transport protocol
- “functional” codec to explore an execution path without committing

gg: general IR for lambda-style granular computation
Tiny functions for four different purposes

**ExCamera:** Fast interactive video encoding
  - “functional” video codec for fine-grained **parallelism**

**Lepton:** JPEG recompression in a distributed filesystem
  - “functional” JPEG codec for boundary-oblivious **sharding**

**Salsify:** Videoconferencing with co-designed codec and transport protocol
  - “functional” codec to **explore an execution path** without committing

**gg:** general IR for lambda-style granular computation
  - “functional” thunk abstraction to efficiently **outsource to cloud functions**

https://ex.camera
What we currently have

- People can make changes to a word-processing document
- The changes are instantly visible for the others
What we would like to have

- People can interactively edit and transform a video
- The changes are instantly visible for the others
"Apply this awesome filter to my video."
"Look everywhere for this face in this movie."
"Remake Star Wars Episode I without Jar Jar."
The Problem
Currently, running such pipelines on videos takes hours and hours, even for a short video.

The Question
Can we achieve interactive collaborative video editing by using massive parallelism?
The challenges

- Low-latency video processing would need thousands of threads, running in parallel, with instant startup.

- However, the finer-grained the parallelism, the worse the compression efficiency.
Enter *ExCamera*

- We made two contributions:
  - Framework to run **5,000-way parallel jobs** with IPC on a commercial “cloud function” service.
  - Purely functional video codec for **massive fine-grained parallelism**.
- We call the whole system *ExCamera*. 
Cloud function services have (as yet) unrealized power

- AWS Lambda, Google Cloud Functions
- Intended for event handlers and Web microservices, *but*...
- Features:
  - Thousands of threads
  - Arbitrary Linux executables
  - Sub-second startup
  - Sub-second billing — 3,600 threads for one second → 9¢
**mu**, supercomputing as a service

- We built *mu*, a library for designing and deploying general-purpose parallel computations on a commercial “cloud function” service.
- The system starts up thousands of threads in seconds and manages inter-thread communication.
- *mu* is open-source software: [https://github.com/excamera/mu](https://github.com/excamera/mu)
Now we have the threads, but...

• With the existing encoders, the finer-grained the parallelism, the worse the compression efficiency.
Video Codec

- A piece of software or hardware that compresses and decompresses digital video.
How video compression works

• Exploit the temporal redundancy in adjacent images.

• Store the first image on its entirety: a **key frame**.

• For other images, only store a "diff" with the previous images: an **interframe**.

In a 4K video @15Mbps, a key frame is ~1 MB, but an interframe is ~25 KB.
Existing video codecs only expose a simple interface

\[
\text{encode}([\text{keyframe}, \text{interframe}[2:n]]) \rightarrow \text{keyframe} + \text{interframe}[2:n]
\]

\[
\text{decode}(\text{keyframe} + \text{interframe}[2:n]) \rightarrow [\text{keyframe}, \text{interframe}[2:n]]
\]
Traditional parallel video encoding is limited

```
encode(i[1:200]) → keyframe₁ + interframe[2:200]

-thread 01- | thread 02- | thread 03- | ... | thread 20-
encode(i[1:10]) → kf₁ + if[2:10]
encode(i[11:20]) → kf₁₁ + if[12:20]
encode(i[21:30]) → kf₂₁ + if[22:30]
```

Finer-grained parallelism ⇒ more key frames ⇒ worse compression efficiency
We need a way to start encoding mid-stream

- Start encoding mid-stream needs access to intermediate computations.
- Traditional video codecs *do not* expose this information.
- We formulated this internal information and we made it explicit: the “state”.
The decoder is an automaton
The state is consisted of reference images and probability models.
What we built: a video codec in explicit state-passing style

• VP8 decoder with no inner state:
  \[ \text{decode}(\text{state}, \text{frame}) \rightarrow (\text{state'}, \text{image}) \]

• VP8 encoder: resume from specified state
  \[ \text{encode}(\text{state}, \text{image}) \rightarrow \text{interframe} \]

• Adapt a frame to a different source state
  \[ \text{rebase}(\text{state}, \text{image}, \text{interframe}) \rightarrow \text{interframe'} \]
Putting it all together: ExCamera

- Divide the video into tiny chunks:
  - [Parallel] **encode** tiny independent chunks.
  - [Serial] **rebase** the chunks together and remove extra keyframes.
1. [Parallel] Download a tiny chunk of raw video
2. [Parallel] \texttt{vpxenc} $\rightarrow$ keyframe, interframe[2:n]

Google's VP8 encoder

\texttt{encode(img[1:n])} $\rightarrow$ keyframe + interframe[2:n]
3. [Parallel] \( \text{decode} \rightarrow \text{state} \rightarrow \text{next thread} \)

Our explicit-state style decoder

\[
\text{decode}(\text{state}, \text{frame}) \rightarrow (\text{state}', \text{image})
\]
4. [Parallel] last thread’s state $\rightarrow$ encode

Our explicit-state style encoder

$\text{encode}(\text{state, image}) \rightarrow \text{interframe}$
5. [Serial] *last thread’s state* → *rebase* → *state* → *next thread*

Adapt a frame to a different source state

\[ \text{rebase}(\text{state, image, interframe}) \rightarrow \text{interframe}' \]
5. [Serial] last thread’s state $\rightarrow$ rebase $\rightarrow$ state $\rightarrow$ next thread

Adapt a frame to a different source state

$\text{rebase}(\text{state, image, interframe}) \rightarrow \text{interframe}´$
6. [Parallel] Upload finished video
Wide range of different configurations

ExCamera \([n, x]\)

number of frames in each chunk
Wide range of different configurations

ExCamera $[n, x]$  

number of chunks "rebased" together
How well does it compress?

- **Quality (SSIM dB)**
- **Average bitrate (Mbit/s)**

Graph showing the relationship between average bitrate and quality for different VPX configurations.
How well does it compress?

The diagram shows the relationship between quality (measured in SSIM dB) and average bitrate (in Mbps) for different encoders:

- **ExCamera[6, 1]**
- **vp x (1 thread)**
- **vp x (multithreaded)**

The graph plots the quality on the y-axis and the average bitrate on the x-axis, illustrating how each encoder performs at different bitrates and qualities.
How well does it compress?

quality (SSIM dB)

average bitrate (Mbit/s)

±3%

ExCamera[6, 16]

ExCamera[6, 1]
### 14.8-minute 4K Video @20dB

<table>
<thead>
<tr>
<th>Encoder Type</th>
<th>Encoding Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpxenc Single-Threaded</td>
<td>453 mins</td>
</tr>
<tr>
<td>vpxenc Multi-Threaded</td>
<td>149 mins</td>
</tr>
<tr>
<td>YouTube (H.264)</td>
<td>37 mins</td>
</tr>
<tr>
<td><strong>ExCamera[6, 16]</strong></td>
<td><strong>2.6 mins</strong></td>
</tr>
</tbody>
</table>
ExCamera concluding thoughts

- Functional video codec lets ExCamera **parallelize** at fine granularity.

- Many interactive jobs call for similar approach:
  - Image and video filters
  - 3D artists
  - Compilation and software testing
  - Interactive machine learning
  - Database queries
  - Data visualization
  - Genomics
  - Search

- Distributed systems will need to treat application state as a first-class object.

- Every program soon: **do in 1 hour** vs. **do in 1 second for 9¢**
Overview

**ExCamera:** Fast VP8 encoding using “functional” VP8 codec

**Lepton:** JPEG recompression with VP8 and “functional” codec

**Salsify:** Videoconferencing using “functional” VP8 codec

**gg:** general IR for lambda-style granular computation
System 2: Lepton (distributed JPEG recompression)

Storage Overview at Dropbox

- ¾ Media
- Roughly an Exabyte in storage
- Can we save backend space?
JPEG File

• Header

• 8x8 blocks of pixels
  – DCT transformed into 64 coefefs
    o Lossless
  – Each divided by large quantizer
    o Lossy
  – Serialized using Huffman code
    o Lossless

Image credit: wikimedia
Idea: save storage with transparent recompression

- **Requirement:** byte-for-byte reconstruction of original file

- **Approach:** improve bottom “lossless” layer only
  - Replace DC-predicted Huffman code with an arithmetic code
  - Use a probability model to predict “1” vs. “0”
Prior work

Decompression speed (Mbits/s) vs. Compression savings (percent)

- **JPEGrescan** (progressive)
- **MozJPEG** (arithmetic)
- **packjpg** (global sort + big model + arithmetic)

Better geometrical representation of prior work in JPEG related algorithms.
Challenge: distributed filesystem with arbitrary chunk boundaries

server #272

server #140

server #803
Challenge: distributed filesystem with arbitrary chunk boundaries

Server #272 representing bytes 0..N-1
Server #140 representing bytes N..2N-1
Server #803 representing bytes 2N..end

Server #272
Server #140
Server #803

Lepton
Lepton
Lepton

bytes 0..N-1
bytes N..2N-1
bytes 2N..end
Requirements for distributed compression

- Store and decode file in independent chunks
  - Can start at any byte offset

- Achieve > 100 Mbps decoding speed per chunk

- Don’t lose data
  - Immune to adversarial/pathological input files
  - Every time program changed, qualify on a billion images
  - Three compilers (with and without sanitizers) must match on all billion images
When the client retrieves a chunk of a JPEG file, how does the fileserver re-encode that chunk from Lepton back to JPEG?
Making the state of the JPEG encoder explicit

- Formulate JPEG encoder in **explicit state-passing style**

- Implement DC-predicted Huffman encoder as a pure function
  - Takes “state” as a formal parameter
  - Can resume anywhere, even in the middle of a Huffman codeword

- State contains everything required to resume from midstream
  - 16 bytes: partial Huffman codeword, prior DC values for each component
Decompression speed (Mbits/s) vs. Compression savings (percent) for different JPEG variants:

- **JPEGrescan** (progressive)
- **MozJPEG** (arithmetic)
- **packjpg** (global sort + big model + arithmetic)

*Better* direction indicated by arrow.
Decompression speed (Mbits/s)
Compression savings (percent)
Better
Lepton
MozJPEG (arithmetic)
JPEGrescan (progressive)
packjpg (global sort + big model + arithmetic)
Deployment

• Lepton has encoded 150 billion files
  – 203 PiB of JPEG files
  – Saving 46 PiB
  – So far...
    o Backfilling at > 6000 images per second
Power Usage at 6,000 Encodes

![Chart showing power usage over time.](chart.png)
A little bit of functional programming can go a long way.

Functional JPEG codec lets Lepton *distribute* decoding with arbitrary chunk boundaries and *parallelize* within each chunk.
Overview

**ExCamera:** Fast VP8 encoding using “functional” VP8 codec

**Lepton:** JPEG recompression with VP8 and “functional” codec

**Salsify:** Videoconferencing using “functional” VP8 codec

**gg:** general IR for lambda-style granular computation

https://github.com/excamera
Two control loops at arm’s length: Skype/Facetime/Hangouts/WebRTC

- Video codec
- Transport protocol
- Compressed frames

Occasional updates (e.g., new bitrate)
The problem: codec and transport protocol are too decoupled

1. Transport tells encoder what “bitrate” to target

2. Encoder chooses a “frame rate” and delivers bits to transport

3. Transport stuck sending what encoder created
   - Even if overshot/undershot capacity estimate

4. Can pause encoder *input*, but once frame has been encoded, it *must be sent*
Salsify: one control loop, with **variable** frame rate and bitrate

transport protocol &
video codec
Enter Salsify

• Salsify introduces:
  • A video-aware congestion control.
  • A network-aware video codec.
  • Salsify’s traffic more closely match the network’s evolving capacity.
  • Salsify consistently achieved higher video quality and lower delay than Hangouts, Skype, FaceTime, and WebRTC.
Salsify’s goal: match video to varying network capacity

- Every iteration of the control loop, encode:
  - Higher-quality version of frame
  - Lower-quality version of the same frame

- Three options to send, based on sizes of encoded output:
  1. Higher-quality version
  2. Lower-quality version
  3. **Nothing: skip frame after encoding it**

- Base next frame on previous frame actually sent

- Never send a frame unless network is ready
End-to-end measurement of realtime video

- Reproducible input video
- Reproducible network traces
- Run unmodified version of the system-under-test

- Target QoE metrics:
  - Frame delay (from the camera at the sender, to the screen at the receiver)
  - Image quality
Network throughput (synthetic network outage)

- Throughput (Mbps)
- Time (s)

Graph showing throughput for Salsify, Skype, and WebRTC over time.
Video frame delay (synthetic network outage)

- Salsify
- Skype
- WebRTC

Frame delay (s) vs. time (s)
Network trace (Verizon LTE)

- Video Quality (SSIM dB)
- Video Delay (95th percentile ms)
- WebRTC (VP9-SVC)
- Skype
- FaceTime
- Hangouts
- Salsify

Comparison:
- WebRTC (VP9-SVC)
- Skype
- FaceTime
- Hangouts
- Salsify (no grace period)
- Salsify (conventional codec)

Better performance on the right side of the graph.
Network trace (T-Mobile UMTS)

- Video Quality (SSIM dB)
- Video Delay (95th percentile ms)
- WebRTC (VP9-SVC)
- Skype
- FaceTime
- Hangouts
- Salsify

WebRTC (VP9-SVC) has better video quality with lower video delay compared to other apps.
Salsify concluding thoughts

- Functional codec lets Salsify explore an execution path without committing to it.

- Exposing codec state allows codec and transport control loops to merge.

  - **Result**: Salsify responds to network variation more gracefully.

  - **Conclusion**: Codecs may have reached point of diminishing returns. Video **systems** still have low-hanging fruit.
Overview

**ExCamera:** Fast VP8 encoding using “functional” VP8 codec

**Lepton:** JPEG recompression with VP8 and “functional” codec

**Salsify:** Videoconferencing using “functional” VP8 codec

**gg:** general IR for lambda-style granular computation
Problem: lots of cores in the cloud, but how to use

- Compiling large programs can take hours

- Desire: **outsource** arbitrary computations to thousands of cores in cloud

- But: cloud far away
  - simply sending code (with NFS) has way too many roundtrips
  - no caching

- Solution: package computation as DAG of **thunks**
Approach: model and thunk everything

- Locally, “run” program under *model substitution*

- E.g. for software compilation, model:
  - preprocess
  - compile
  - assemble
  - link
  - ar, ranlib, strip

- Each model produces a **thunk**.
  - closure that allows deterministic delayed execution of any pipeline stage
  - same result anywhere
Thunks for compilation (GNU hello)
Thunks for ExCamera (video encoding)
Example thunk

{
  "function": {
    "exe": "gcc",
    "args": [
      "gcc", "-g", "-O2", "-c", "-o", "TEST_remake.o", "remake.i"
    ],
    "hash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
  },
  "infiles": [
    {
      "filename": "remake.i",
      "hash": "9f1d127592e2bee6e702b66c9114d813d059f65e8cdb79db2127e7d6d1b3384b",
      "order": 0
    }
  ],
  "outfile": "TEST_remake.o"
}
To execute, lazily force the thunk

- Thunks are self-contained and can be forced locally or in the cloud.
- Run 1,000+ thunks in parallel on Lambda/OpenWhisk
- Can trust others’ assertions
  - “File with this hash → contents” (easy to detect invalid claims)
  - “Thunk with this hash → result” (can prove a claim is invalid)
- Thunks could compile, encode video, map, reduce...
Compiling Mosh (mobile shell) with 1,000-way parallelism

- Fetching the dependencies
- Executing the thunk
- Uploading the results

工人编号

时间 (秒)

- preprocess
- compile
- assemble
- archive and link

工作已完成
Compiling FFmpeg with 1,000-way parallelism

- Fetching the dependencies
- Executing the thunk
- Uploading the results

↓ preprocess, compile and assemble

↑ archive, link and strip

job completed

archive, link and strip

time (s)

worker #
Tiny functions, executed everywhere, for lots of things . . .

- Granular, functional interfaces to computing resources will enable new applications.
- It’s worth refactoring interfaces to today’s megasystems (codec, compiler, TCP, machine learning . . .).
- **Saving and restoring** program states is a powerful tool.

- **ExCamera**: video encoding with thousands of tiny tasks
- **Lepton**: JPEG recompression
- **Salsify**: real-time video with “functional” codec and transport
- **gg**: general IR for lambda-style granular computation

Thank you: NSF, DARPA, Google, Dropbox, VMware, Facebook, Huawei, SITP, Platform Lab.
ExCamera timings

<table>
<thead>
<tr>
<th>threads</th>
<th>start</th>
<th>25 s</th>
<th>50 s</th>
<th>75 s</th>
<th>100 s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. [Parallel] Download 6-frame chunk of raw or mezzanine video
ExCamera timings (vpxenc)

2. [Parallel] vpxenc → keyframe, interframe[5]
3. [Parallel] **decode** → **state** ↦ *next thread*
4. [Parallel] last thread’s state $\leadsto$ encode-given-state
5. [Serial] last thread’s state $\sim$ rebase $\rightarrow$ state $\sim$ next thread
ExCamera timings (rebase)

5. [Serial] *last thread’s state* $\leadsto$ *rebase* $\rightarrow$ *state* $\leadsto$ *next thread*
5. [Serial] \textit{last thread's state} $\leadsto$ \textbf{rebase} $\rightarrow$ \textit{state} $\leadsto$ \textit{next thread}
5. [Serial] last thread’s state $\rightsquigarrow$ rebase $\rightarrow$ state $\rightsquigarrow$ next thread
5. [Serial] *last thread’s state* $\rightsquigarrow$ *rebase* $\rightarrow$ *state* $\rightsquigarrow$ *next thread*
5. **[Serial]** *last thread’s state* $\leadsto$ **rebase** $\rightarrow$ **state** $\leadsto$ **next thread**
5. [Serial] *last thread’s state* $\leadsto$ *rebase* $\rightarrow$ *state* $\leadsto$ *next thread*
5. [Serial] last thread’s state $\leadsto$ rebase $\rightarrow$ state $\leadsto$ next thread
5. **[Serial]** last thread’s state $\leadsto$ rebase $\rightarrow$ state $\leadsto$ next thread
5. [Serial] last thread’s state $\leadsto$ rebase $\rightarrow$ state $\leadsto$ next thread
5. [Serial] *last thread’s state* $\leadsto$ **rebase** $\rightarrow$ **state** $\leadsto$ *next thread*
5. [Serial] *last thread’s state* $\rightsquigarrow$ *rebase* $\rightarrow$ *state* $\rightsquigarrow$ *next thread*
ExCamera timings (rebase)

5. [Serial] last thread’s state $\rightsquigarrow$ rebase $\rightarrow$ state $\rightsquigarrow$ next thread
5. [Serial] last thread's state $\leadsto$ rebase $\rightarrow$ state $\leadsto$ next thread
ExCamera timings (rebase)

5. [Serial] last thread’s state $\leadsto$ rebase $\rightarrow$ state $\leadsto$ next thread
ExCamera timings (upload)

6. [Parallel] Upload finished video
Why lambda?

- Cheap source of massively parallel compute
  - 3,600 lambdas for 1 second each → 9 cents

- Lambdas launch quickly vs. VM
  - Lambda: less than 0.1 seconds
  - VM: more than 30 seconds

- Benefits are mostly to individual developer or sporadic jobs
  - Economics depend on multiplexing with diverse kinds of jobs on same infrastructure

- If user has enough money or demand to justify a 5,000-core build farm, lambdas less beneficial.