Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available.

2016 © Microsoft Corporation. All rights reserved.
Tiny functions for codecs, compilation, and (maybe) soon everything

Keith Weinstein


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**
System 1: ExCamera (fine-grained parallel video processing)


https://ex.camera
What we currently have

Google Docs

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

Google Docs for Video?

- 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
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

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

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

serial ↓

\[\text{encode}(i[1:200]) \rightarrow \text{keyframe}_1 + \text{interframe}[2:200]\]

parallel ↓

[thread 01] \text{encode}(i[1:10]) \rightarrow \text{kf}_1 + \text{if}[2:10]
[thread 02] \text{encode}(i[11:20]) \rightarrow \text{kf}_2 + \text{if}[12:20]
[thread 03] \text{encode}(i[21:30]) \rightarrow \text{kf}_3 + \text{if}[22:30]
\vdots
[thread 20] \text{encode}(i[191:200]) \rightarrow \text{kf}_{191} + \text{if}[192:200]

finer-grained parallelism \Rightarrow more key frames \Rightarrow 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] vpxenc $\rightarrow$ keyframe, interframe[2:n]

Google's VP8 encoder

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

Our explicit-state style decoder

\[ \text{decode(state, frame)} \rightarrow (\text{state}', \text{image}) \]
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

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

Adapt a frame to a different source state
rebase(state, image, interframe) → interframe
6. [Parallel] Upload finished video

thread 1

thread 2

thread 3

thread 4
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?
How well does it compress?

![Graph showing quality (SSIM dB) vs. average bitrate (Mbit/s) for different compression methods: vpx (1 thread), vpx (multithreaded), and ExCamera[6, 1].]
How well does it compress?

![Graph showing quality vs. average bitrate for different compression methods.](image-url)
<table>
<thead>
<tr>
<th>Format</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.8-minute 4K Video @20dB</td>
<td></td>
</tr>
<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>ExCamera [6, 16]</td>
<td>2.6 mins</td>
</tr>
</tbody>
</table>
4. [Parallel] last thread’s state $\sim$ encode

Our explicit-state style encoder

$encode(state, image) \rightarrow interframe$
6. [Parallel] Upload finished video
4. [Parallel] last thread’s state $\sim$ encode

Our explicit-state style encoder

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

Adapt a frame to a different source state

rebase(state, image, interframe) $\rightarrow$ interframe'
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** or **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

- $\frac{3}{4}$ Media

- Roughly an Exabyte in storage

- Can we save backend space?
JPEG File

- **Header**
- **8x8 blocks of pixels**
  - DCT transformed into 64 coefs
    - Lossless
  - Each divided by large quantizer
    - Lossy
  - Serialized using Huffman code
    - 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

The diagram compares the decompression speed (in Mbyte/s) against compression savings (in percent) for different JPEG compression methods. The methods include JPEGrescan (progressive), MozJPEG (arithmetic), and packjpg (global sort + big model + arithmetic). The diagram indicates that packjpg provides a better trade-off between speed and savings compared to the other methods.
Challenge: distributed filesystem with arbitrary chunk boundaries

server #272
bytes 0..N-1

server #140
bytes N..2N-1

server #803
bytes 2N..end
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
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

jpeg
bytes 0..N-1

jpeg
bytes N..2N-1

jpeg
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 filesystem 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
Results

![Graph showing compression speed vs compression savings for JPEGrescan (progressive), MozJPEG (arithmetic), and packjpg (global sort + big model + arithmetic).]
Deployment

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

![Graph showing power usage over time](image-url)
Lepton concluding thoughts

- 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
How well does it compress?

![Graph showing quality (SSIM dB) vs. average bitrate (Mbit/s)]

- vpx (1 thread)
- ExCamera[6, 16]
- ExCamera[6, 1]

±3%

https://snr.stanford.edu/salsify
WebRTC (Chrome 65)
Current systems do not react fast enough to **network variations**, end up congesting the network, causing **stalls and glitches**.
Enter Salsify

- Salsify is a new architecture for real-time Internet video.
- Salsify tightly integrates a video-aware transport protocol, with a functional video codec, allowing it to respond quickly to changing network conditions.
- Salsify achieves 4.6x lower p95-delay and 2.1 dB SSIM higher visual quality on average when compared with FaceTime, Hangouts, Skype, and WebRTC.
Today's systems combine two *loosely-coupled* components.
Enter Salsify

- Salsify is a new architecture for real-time Internet video.
- Salsify tightly integrates a video-aware transport protocol, with a functional video codec, allowing it to respond quickly to changing network conditions.
- Salsify achieves 4.6x lower p95-delay and 2.1 dB SSIM higher visual quality on average when compared with FaceTime, Hangouts, Skype, and WebRTC.
Two distinct modules, two separate control loops

- **Video Codec**: 24 frames/s
- **Transport Protocol**: 300 packets/s

Target bit rate

Compressed frames
Salsify’s architecture:
Video-aware transport protocol

transport protocol & video codec
Video-aware transport protocol

“What should be the size of the next frame?”

* without causing excessive delay

- There’s no notion of bit rate, only the next frame size!

- Transport uses **packet inter-arrival time**, reported by the receiver.
The sender does not transmit continuously

- Pauses between frames give the receiver a "pessimistic" view of the network.
- Receiver treats each frame of the video as a separate packet train.
Salsify’s architecture:
Functional video codec
Transport tells us how big the next frame should be, but...

It’s challenging for any codec to choose the appropriate quality settings upfront to meet a target size—they tend to over-/undershoot the target.
How to get an accurate frame out of an inaccurate codec

- **Trial and error:** Encode with different quality settings, pick the one that fits.

- **Not possible with existing codecs.**
After encoding a frame, the encoder goes through a state transition that is impossible to undo.
Functional video codec to the rescue

\[ \text{encode}(\text{state}, \text{frame}) \rightarrow \text{state}', \text{frame} \]

Salsify’s functional video codec exposes the state that can be saved/restored.
Order two, pick the one that fits!

- Salsify’s functional video codec can **explore different execution paths** without committing to them.

- For each frame, codec presents the transport with *three* options:
  - A slightly-higher-quality version,
  - A slightly-lower-quality version,
  - Discarding the frame.
Codec → Transport

“Here’s two versions of the current frame.”

target frame size 30 KB
Transport → Codec

“I picked option 2. Base the next frame on its exiting state.”
Codec → Transport

“Here’s two versions of the latest frame.”
Transport → Codec

“I picked option 1. Base the next frame on its exiting state.”
Codec → Transport
“Here’s two versions of the latest frame.”
Transport → Codec

“I cannot send any frames right now. Sorry, but discard them.”
Codec → Transport

“Fine. Here’s two versions of the latest frame.”

![](image)

**target frame size** 50 KB
Goals for the measurement testbed

- A system with reproducible input video and reproducible network traces that runs unmodified version of the system-under-test.

- Target QoE metrics: per-frame quality and delay.
Evaluation results: **Verizon LTE Trace**

![Graph showing video quality and delay for Skype, WebRTC (VP9-SVC), FaceTime, and Hangouts. The graph indicates that WebRTC (VP9-SVC) and FaceTime perform better than Skype and Hangouts in terms of both video quality and delay.](image-url)
Evaluation results: **Verizon LTE Trace**

- **Video Quality (SSIM dB)**
- **Video Delay (95th percentile ms)**

- Skype
- WebRTC (VP9-SVC)
- WebRTC
- FaceTime
- Status Quo (conventional transport and codec)
- Hangouts
Evaluation results: Verizon LTE Trace

- Skype
- WebRTC (VP9-SVC)
- WebRTC
- FaceTime
- Status Quo (conventional transport and codec)
- Salsify (conventional codec)
- Hangouts

Video Delay (95th percentile ms) vs Video Quality (SSIM dB)
Evaluation results: Verizon LTE Trace

- Skype
- WebRTC (VP9-SVC)
- WebRTC
- FaceTime
- Status Quo (conventional transport and codec)
- Salsify (conventional codec)
- Hangouts

Video Quality (SSIM dB) vs Video Delay (95th percentile ms)
Evaluation results: AT&T LTE Trace

- WebRTC (VP9-SVC)
- FaceTime
- Hangouts
- Skype
- Salsify

Video Quality (SSIM dB) vs. Video Delay (95th percentile ms)
Evaluation results: **T-Mobile UMTS Trace**

- **WebRTC (VP9-SVC)**: High quality, lower delay.
- **Salsify**: High quality, higher delay.
- **Skype**
- **FaceTime**
- **Hangouts**

The graph shows the relationship between video quality (SSIM dB) and video delay (95th percentile ms). WebRTC (VP9-SVC) is indicated with a brown circle, indicating better performance compared to other services.
Evaluation results: **Emulated Wi-Fi (no variations, only loss)**

![Graph showing video quality vs. video delay for different applications.](Image)

- **WebRTC (VP9-SVC)**: High video quality and lower video delay.
- **Salsify**: Moderate video quality with a higher video delay.
- **FaceTime**: Lower video quality with a lower video delay.
- **Hangouts**: Lower video quality with a higher video delay.
- **Skype**: Lower video quality and higher video delay.

The graph illustrates that **WebRTC** offers better performance in terms of video quality and delay compared to the other applications.
Improvements to *video codecs* may have reached the point of diminishing returns, but changes to the architecture of *video systems* can still yield significant benefits.
Takeaways

- Salsify is a new architecture for real-time Internet video.
- Salsify tightly integrates a **video-aware transport protocol**, with a **functional video codec**, allowing it to **respond quickly to changing network conditions**.
- Salsify achieves **4.6x lower p95-delay** and **2.1 dB SSIM higher visual quality** on average when compared with FaceTime, Hangouts, Skype, and WebRTC.
- The code is open-source, and the paper and raw data are open-access: https://snr.stanford.edu/salsify
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
Renting 10,000 cores for 10 seconds is powerful, but how to program?

Bespoke systems like ExCamera and Pywren (Berkeley) commingle:

- the algorithm
- the schedule (what to parallelize)
- the execution (AWS Lambda, IBM OpenWhisk, etc.).

Proposal: a general IR for lambda-style granular computing.

Abstract algorithm from execution, somewhat like LLVM IR or Halide.
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**.
  - A codelet + the named data it will be applied to.
  - Data named by hash of contents.
  - “Microcontainer” can only access named data and nothing else.
What about dynamic task graphs?

- Example: detect faces (0..N), then recognize each face
- Idea: API call to spawn a new task
  - But language-dependent & hard to cache and outsource...
- gg’s approach: tail-recursion as only mechanism
- Thunk’s value is another thunk (forced recursively)
Thunks for compilation (GNU hello)

dirname.c -> dirname.i -> dirname.o -> libc
string.h

closeout.c -> closeout.i -> closeout.o -> libhello.a

stdio.h

hello.c -> hello.i -> hello.o

hello

hello (stripped)
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": "9f1d127592e2bee6e702b66c9114d813d059f65e8c6b79db2127e7d6d1b3384b",
      "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...
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.
Compiling Mosh (mobile shell) with 1,000-way parallelism

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

Jobs:
- Preprocess
- Compile
- Assemble
- Archive and link

Time (s)

Worker #
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 ➔

time (s)

0 5 10 15 20 25 30

worker #

0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000

5080 5095 5115

worker #