Llama: Navigating Trade-offs in Diverse Video Pipelines

Mark Zhao*, Francisco Romero*, Neeraja J. Yadwadkar, Christos Kozyrakis
Video traffic is exploding in scale

82% of internet traffic by 2022\(^1\)

\(^1\)Cisco annual internet report (2018-2023)
Video pipelines are ubiquitous

- **Video pipelines**: Series of processing or analytics operations performed on a video
- Different performance/cost **targets** for each video-pipeline pair

“Add a vintage filter to the video”

“Identify cars and faces from the traffic feed”

“Blur Bob’s face in all frames”
Applications have different cost/latency targets

“Blur Bob’s face in all frames”

Priority

Live news

Hidden camera show
Using operation knobs to make the trade-offs

• Users configure operation *knobs* to best meet their targets
  • *<Hardware, resource allocation, batch size, resolution, ...>*

- 4 vCPU
  - Unit batch
  - 4 vCPU
  - Unit batch
  - 11x11 kernel
- 4 vCPU
  - High-resolution

Slower, but cheaper

- 2 GPU
  - Unit batch
  - 2 GPU
  - Batch-4
  - 11x11 kernel
- 1 GPU
  - High-resolution

Faster, but pricier
Configuring pipelines to meet targets is difficult

- Sequential, parallel, and branching paths
- Input-dependent behavior
- Large configuration space
- Diverse resource requirements
Key insights

- Exhaustive profiling *per-video* is expensive and time-consuming
- Static configurations means we cannot account for non-determinism

Decisions need to be made *per-video, per-frame*

Break down an application’s end-to-end pipeline target to a per-operation execution target

“Finish in 30s”
Serverless computing to the rescue...?

Per-function resource allocation with fine-grained billing

😊 Pipeline operations can be run as functions
😊 Resources can be allocated each time an operation executes

😢 No heterogeneous hardware
😢 Burst resource constraints
😢 Lack of target awareness

Cannot trade off performance and cost
Llama

• A heterogeneous serverless video processing and analytics framework
• Meets application targets by:
  • Breaking down an end-to-end target to a per-operation execution target
  • Dynamically configuring operation knobs to meet target
  • Balancing per-operation targets across heterogeneous backends
Llama design

• Specification phase: Profile operations and describe pipeline

• Online phase: Manages the *runtime*, configuring *invocations* to meet the target
Llama design

• **Specification phase:** Profile operations and describe pipeline

• Online phase: Manages the *runtime*, configuring *invocations* to meet the target
Specification phase

Generate operation profiles and expose programming interface

- **Pipeline SDK**: Used to specify pipeline operations and dependencies as a DAG
Pipeline SDK example - AMBER alert

decode = llama.decode("my_video.mp4")
preprocessObjDet = llama.preprocessObjectDetect(decode)

objectDetect = llama.objectdetect(preprocessObjDet, branch=True)
if objectDetect.class() == "person":
    preprocessFaceRecog = llama.preprocessFaceRecognition(objectDetect)
    faceRecognition = llama.faceRecognition(preprocessFaceRecog)
if objectDetect.class() == "car":
    preprocessCarRecog = llama.preprocessCarRecognition(objectDetect)
    carRecognition = llama.carRecognition(preprocessCarRecog)

target = "fastest"  # "cheapest", "120s"
llama.run([faceRecognition, carRecognition], target)
Specification phase

Generate operation profiles and expose programming interface

- **Pipeline SDK**: Used to specify pipeline operations and dependencies as a DAG
- **Operation Library**: Stores operation executables and configuration specifications for reuse across pipelines
Specification phase

Generate operation profiles and expose programming interface

- **Pipeline SDK**: Used to specify pipeline operations and dependencies as a DAG
- **Operation Library**: Stores operation executables and configuration specifications for reuse across pipelines
- **Operation-Profiler**: Collects performance and resource statistics for each operation to generate configuration specification
Llama design

- **Specification phase**: profile operations and describe pipeline

- **Online phase**: Manages the runtime, configuring invocations to meet the target
Lifecycle of a task: Manager

- Tracks dynamic execution state
- Updates in-memory configuration profiles using feedback
- Resolves higher-level logic (e.g., branching, reduction)
- Triggers new tasks when dependencies are resolved
Lifecycle of a task: Configurator

Configures individual tasks (per-frame execution of an operation) to efficiently meet the end-to-end performance target.
**Configurator: slack calculation**

- **Challenge:** End-to-end latency target -> per-task assignments?
- **Solution:** Decompose target into a per-invocation “slack”
  - Pessimistically allot slack, recover excess in future invocations

---

### Object Detection Configuration

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$0.02</td>
<td>1s</td>
</tr>
<tr>
<td>B</td>
<td>$0.01</td>
<td>2s</td>
</tr>
</tbody>
</table>
Configurator: evaluating configurations

**Challenge:** What is the “best” configuration?

**Solution:** Minimize objective function

- Optimize for cost-efficiency if slack is met
- Optimize for throughput if slack is violated

Delayed batching if slack will not be violated

\[
\text{obj}(\text{config}, \text{slack}) = \frac{\text{cost}(\text{config})}{\text{bSize}(\text{config})} + \frac{\alpha (\text{lat}(\text{config}) \cdot \text{resource}(\text{config}))}{\text{bSize}(\text{config}) \cdot \text{resource}_{\text{total}}(\text{engine}(\text{config}))}
\]

Cost per frame

- \(\text{cost}(\text{config})\)
- \(\frac{\alpha (\text{lat}(\text{config}) \cdot \text{resource}(\text{config}))}{\text{bSize}(\text{config}) \cdot \text{resource}_{\text{total}}(\text{engine}(\text{config}))}\)

Cost plus weighted throughput

Available configurations

\(~\mathcal{O}(10-100)~\)
Configurator: estimating queueing

Tasks will queue locally; we need accurate estimates of queueing delay

**Challenge:** How do we accurately estimate queueing delays when...
- Configuration decisions of queued tasks affect backend load
- Observed queueing time may drastically differ from expected latencies

**Solution:** *Early speculation and late commit*
- Speculate a configuration as early as possible to estimate queueing time
- Commit a configuration as late as possible to maintain maximum flexibility
Configurator: priority commit

**Challenge:** Operations can unnecessarily block resources

**Solution:** *Priority commit*

- Operations that benefit *most* from a backend have priority
- Others will wait for resources if slack will not be violated

<table>
<thead>
<tr>
<th>Config</th>
<th>Object Detection</th>
<th>Face Recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Latency</td>
</tr>
<tr>
<td>A (CPU)</td>
<td>$0.02</td>
<td>1s</td>
</tr>
<tr>
<td>B (GPU)</td>
<td>$0.01</td>
<td>2s</td>
</tr>
</tbody>
</table>
Lifecycle of a task: Scheduler

- Schedules configured invocations on appropriate backend
- Manages connections and error handling
- Provides Manager with results and performance metadata
Evaluation

We deployed Llama on Google Cloud
  • Serverless CPU & GPU backends
    • Prototype serverless GPU using Nvidia MPS

Pipelines (video length, video resolution)
  • AMBER alert (10min, 1080p)
  • Face blurring (10min, 720p)
  • Denoising (10min, 720p)
  • Toonify (10min, 720p)

Comparison systems
  • gg (Fouladi et al., ATC 2019)
    • gg-branch: variant that supports conditional paths
  • Nexus (Shen et al., SOSP 2019)
  • Scanner (Poms et al., SIGGRAPH 2018)
    • Scanner-CPU & Scanner-GPU
How well does Llama trade-off cost and performance?

**Llama-fast**: target set to minimize latency  
**Llama-cheap**: target set to minimize cost

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Llama-fast</th>
<th>Llama-cheap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Cost</td>
</tr>
<tr>
<td>Toonify</td>
<td>289s</td>
<td>$2.66</td>
</tr>
<tr>
<td>Face blurring</td>
<td>94s</td>
<td>$0.68</td>
</tr>
<tr>
<td>Denoising</td>
<td>242s</td>
<td>$1.93</td>
</tr>
<tr>
<td>AMBER alert</td>
<td>329s</td>
<td>$2.16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Specified Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Target</td>
</tr>
<tr>
<td>Toonify</td>
<td>450s</td>
</tr>
<tr>
<td>Face blurring</td>
<td>100s</td>
</tr>
<tr>
<td>Denoising</td>
<td>320s</td>
</tr>
<tr>
<td>AMBER alert</td>
<td>450s</td>
</tr>
</tbody>
</table>

Llama can trade-off cost and performance, and meet a specified target.
How well does Llama perform against baselines?

Latency normalized to llama-fast: lower is better

Cost normalized to llama-cheap: lower is better

*Nexus supports DNN-only pipelines (AMBER Alert); it cannot run Face Blurring, Denoising, or Toonify

Llama achieves a 10x speedup and 18.8x cost reduction on average
Toonify Demo

See demo at:
https://www.youtube.com/watch?v=5q4GsV3VhjU&feature=emb_logo
Future work

• Virtualizing accelerators for serverless frameworks
  • GPUs, FPGAs, TPUs, ...

• Optimizations within and across pipelines
  • Merging operations, co-locating shared operations between tenants

• Extensibility to other domains
  • High task-level parallelism, hardware-accelerated, non-deterministic pipelines
  • Suggestions?
Conclusion

• End-to-end video processing and analytics engine
• Leverages heterogeneous serverless backends to automatically explore performance/cost trade-offs for users
• Achieves a 10x latency speedup and 18.8x cost reduction on average compared to state-of-the-art frameworks

Questions?

{myzhao,faromero}@stanford.edu