D-002

防犯向け多カメラストリーム映像解析の実装と性能評価

An Implementation and Evaluation of Multi-Camera Video Analysis for Surveillance

江田 毅晴 †　村松 沙那恵 †　三上 啓太 †　史 旭 †
Takeharu Eda　Sanae Muramatsu　Keita Mikami　Shi Xu

1　はじめに

昨今，深層学習 (Deep learning) により，様々な機械学習応用分野においてブレークスルーが起きている．本稿では，深層学習を用いたリアルタイム防犯向けカメラ映像解析の実装例とその処理性能評価，および考察について述べる．

2　問題設定：防犯カメラ映像からの人物検索

ビルテナントやショッピングモールでは，通常数十〜数百台程度の防犯カメラが設置されているが，それらの映像を人間がリアルタイムに監視し続けることは不可能であり，犯罪やトラブルが起きた時の捜査用に録画保存されているだけのことが多い．もし，これらの映像を計算機のみで人間同等のレベルで監視し続けることが出来れば，大幅な犯罪抑制やコスト削減の効果が期待できる [2], [5].

本稿では，多数の防犯カメラ映像からリアルタイムに人物のトラッキングを行うユースケース（犯人捜しや迷子探し）を想定し，深層学習技術を組み込んだ映像解析システムを実装し，どの程度の処理性能を実現できるのか検証を行った．多数の映像ストリームを柔軟かつ高速に処理することを目指し，ストリーム処理エンジンである SensorBee を用いて処理フローを実装し，深層学習を利用した人検知および再照合をプラグインとして呼び出す．処理実行中の CPU や GPU といったリソースの状況を監視し，提案システムの性能について報告する．

3　実装例

3.1　要件

深層学習には，データ拡張や正規化といった前処理が必要である．アプリケーションとしては，解析結果（バウンディングボックスやクラスラベル）を画像に重ねたり，統計をとったりといった後処理も必要であるため，一般的な映像解析処理を容易に実装できる必要がある．我々のユースケースでは，リアルタイムに多数の映像解析処理を行うために，軽量な実装を行いたい．実際の環境では，カメラのスペックやクオリティが異なることが多く，処理を変えたいことがよくあるため，拡張性も重要である．また，将来的に AI 処理は エッジサーバに処理のオフローディングが進むことが予想されるため，実装アルゴリズムをポータブルに移行できると良い．上記を踏まえ，我々は下記の項目を実装物への要件と考えた．

1. 深層学習を含む映像解析処理を実装可能であること．
2. 容易に拡張可能であること
3. フットプリントが小さく軽量であること．
4. 実行環境のポータビリティを持つこと．

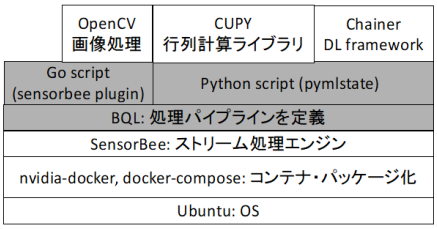

図 1　ソフトウェア構成.

3.2　ソフトウェア構成

選択した構成を図 1 に示す．映像ストリームエンジンとして SensorBee (http://sensorbee.io)．深層学習フレームワークとしては，Chainer (https://chainer.org) を採用している．SensorBee は，Preferred Networks によって開発されたストリーム処理エンジンであり，Chainer をはじめとする機械学習処理を容易に組み込め，宣言的言語 BQL による処理フローの記述を可能とする．ストリームエンジンとして代表的な Apache Storm [6] 等と異なり，スタンドアローンでの処理をフットプリント小さく（30MB 以下）実装する用途に向いている．

SensorBee では，BQL によって，入力 (SOURCE) から出力 (SYNC) までの処理を，宣言的に定義する．ネットワークカメラや動画ファイルの入力は URI やファイル名を指定し 1 行で定義することができ，フレーム by フレームでの処理を重ね，処理パイプラインを定義する．BQL で対応できない処理は，ユーザ定義関数をプラグインとして実装することで，BQL から呼び出すことが可能になる．また，機械学習モデルをユーザ定義 state として読み込み，容易に推論処理を処理パイプラインに組み込むことができる．

3.3　実装パイプライン

実際に実装した，リアルタイムな人物トラッキングの処理パイプラインを図2に示す．防犯カメラストリームに対してフレーム単位で人物検知処理を行い，人物を

† NTT ソフトウェアイノベーションセンタ
第二推進プロジェクト

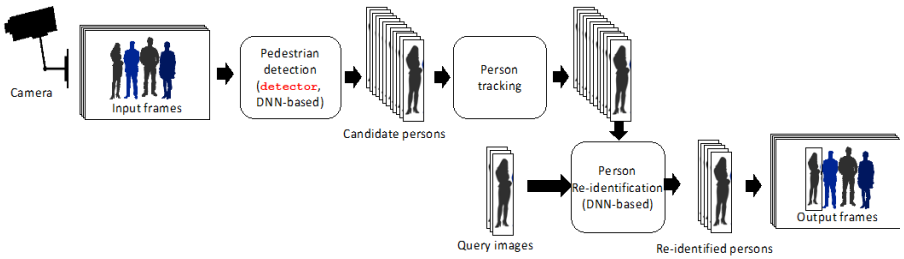

図 2　実装した処理パイプライン．

```
 1 CREATE PAUSED SOURCE camera TYPE opencv_capture_from_uri WITH uri="rtsp://*.*.*.*:554";
 2 ....
 3 LOAD STATE detector TYPE pymlstate SET gpu=0, thresh=0.7, image_size = [640, 480];
 4 CREATE STREAM regions AS SELECT RSTREAM
 5     frames:*,
 6     pymlstate_predict("detector", frames:*) AS regions
 7    FROM frames [RANGE 1 TUPLES];
 8 ....
 9 CREATE SINK mjpeg_server TYPE scouter_mjpeg_server WITH port=10080, quality=100;
10 INSERT INTO mjpeg_server FROM styled_tracked_frames;
```

図 3　パイプラインを定義する BQL 文（一部抜粋）．

含むバウンディングボックスを得る．続いて，検知された人物のフレーム間でのトラッキング処理を行った後，あらかじめ登録されたクエリ画像と同一人物かどうか照合（人物再照合）を行い，同一人物と判定された場合は，映像フレームにバウンディングボックスを重ね，強調表示した上で，ユーザインタフェースに表示する．人物検知と再照合において，深層学習を採用し chainer モデルによる推論を python で実装し，ユーザ定義 state(pymlstate) 経由で実行する．

図 3 に処理パイプラインを実装した BQL 文の一部を抜粋したものを示す．カメラソースを開き (1 行目)，python で実装された DNN モデルによる物体検知を行い (3〜7 行目)，結果を加工したものを，mjpeg にて配信する (9, 10 行目)．`detector` は，pymlstate 経由で呼ばれる python クラスであり，DNN の推論を実装してある．BQL では，この例のように，画像ソースに対する一連の処理を宣言的に定義し実行することができる．

SensorBee でのタプル（ストリームの 1 レコード）は，メインメモリ上で処理されるため，DNN の前処理・後処理は，python クラス内でできるだけ GPU 内で処理を行うように実装している．DNN に関連しないいくつかの処理（crop や処理結果映像作成）は，SensorBee プラグインとして，go で実装した．

3.4　実装詳細

3.4.1　コンテナ化によるポータビリティの向上

実装システムは多数の OSS および独自ライブラリを利用している．また，深層学習周りのソフトウェアは進化が速い上，NVIDIA ハードウェアとの互換性にも注意する必要があり，構築のコストが課題となった．そこで，すべてのソフトウェアを含む Docker イメージ（実際には nvidia-docker, docker-compose を利用）を構築し，デプロイおよびメンテナンスを効率化している．映像ストリーム毎に，Docker コンテナを起動し，SensorBee による処理パイプラインに流すことにより，多カメラ解析時でも監視や異常解析が容易になる．

コンテナ化のもう一つの恩恵として，ポータビリティの向上がある．我々の実装では，1 コンテナ 1 カメラ方式にて実装しており，カメラ IP アドレス，BQL の revision，利用 GPU，FPS 等を変数に，デプロイ時に柔軟に解析方法を指定することができる．Jetson や Raspberry Pi といった ARM 上での Docker コンテナの動作も確認されており，今後よりヘテロな環境でも，ポータブルに映像解析サービスを実装していくことが可能になる．

3.4.2　CNN による再照合

採用した再照合アルゴリズムは，人物の画像を入力とし，CNN を用いて画像を特徴ベクトル空間に射影する．その後，特徴空間内での距離の近さにより，同一人物かどうかを判定する．実際のユースケースを考えると，別の日に再訪したか知りたいことも多く，服装が違うと照合できない全身画像による照合は，不十分であったため，全身と顔の両方の特徴を組み合わせた照合を行っている．全身を先に検知してから顔を検知することで検知対象画像を狭め，処理の効率化を図っている．

3.4.3　推論のバッチ処理化

深層学習においては，学習時は精度と計算時間のバランスから，入力をバッチ単位で GPU に転送するミニバッチ学習法がとられる．実際には，画像の GPU コピーのオーバヘッドは小さくないため，推論時にもバッチによる恩恵が得られる [1]．我々は，再照合処理時に，フレーム内の候補人物画像をできるだけまとめてバッチに充填して処理することで，GPU コピーのオーバヘッ

ドを下げ処理の高速化を行っている．

上記の議論より，実装物は，3.1 にて述べた 4 つの要件を満たすことが分かる．

4 性能評価

まず，単一のサーバでの単体性能を見極める．そのうえで，100 カメラストリームを処理可能な環境を用意して，問題なく 100 カメラストリームを処理できるか検証を行った．

4.1 単体性能試験

4.1.1 条件

表1に，単体性能試験で用いた映像のスペックを示す．防犯カメラに映る映像から顔を切り出し，照合を実現す

映像スペック	解像度	1920x1080
	FPS	5
	平均登場人数	5 ～ 20
	カメラ台数	12, 32, 40
クエリ数		5

表 1 映像スペック．

るには，フル HD(1920*1080) の解像度が必要である．フレームレートについては，実用上の精度およびネットワークトラフィックを考慮し，5fps とした．映像中の登場人数や同時に処理するカメラ台数は可変とし (図4)，1 台のサーバでどの程度の処理が出来るか計測した．

図 4 評価映像の例．混雑状況，実映像，空いている状況．

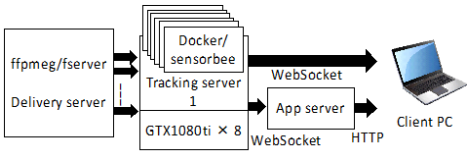

図 5 評価環境．

評価環境の構成を図5に示す．カメラ映像については，ffserver を用いて疑似的にカメラ数分の映像ストリームを配信した (delivery server)．映像解析サーバ (tracking server) は，Supermicro 4028GR-TRT2 に，NVIDIA GeForce GTX1080ti を 8 枚，CPU は Xeon E5-2698 v4 2.2 × 2，メモリ 256GB を搭載している．別のサーバ (App server) にアプリケーションサーバを実装し，映像をクライアントに配信する場合は，直接，映像解析サーバに WebSocket 接続する構成としている．

4.1.2 結果

表6に 60 分間処理し続けた際の各処理の平均 FPS のうち代表的な結果を示す．処理は上から順番に進み，途

	混雑時	実映像	
	12カメラ	32カメラ	40カメラ
カメラ映像取得	5.0	5.0	5.0
人検知	5.0	5.0	5.0
トラッキング	5.0	5.0	5.0
再照合	4.9 (5.0)	4.9 (5.0)	4.8 (5.0)
配信処理	4.9 (5.0)	4.9 (5.0)	4.8 (5.0)

図 6 各処理での FPS.

中で処理落ちが発生すると 5FPS を下回る．一旦コマ落ちが発生すると，それ以降の処理全てにコマ落ちが起きる．() で表示している数値は，後半 30 分での FPS を示している．今回，最初に照合用クエリをインデックス化する処理が並列で動作するため，一時的にコマ落ちが発生しているが，60 分後には，この例については全てコマ落ち無く，処理を実施できていることが分かる．実際には，これ以上にカメラ台数を増やしたケースも計測したが，コマ落ちが発生し続け，リアルタイム処理が不可能であることが分かった．

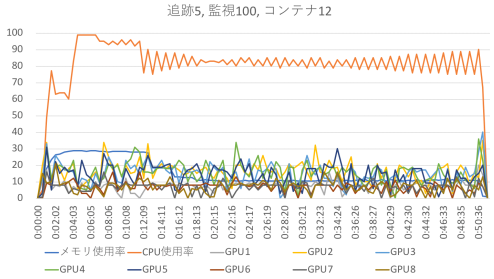

図 7 空いている動画でのリソース状況．

図7, 8に，リソースの利用状況を示す．混雑状況は 12

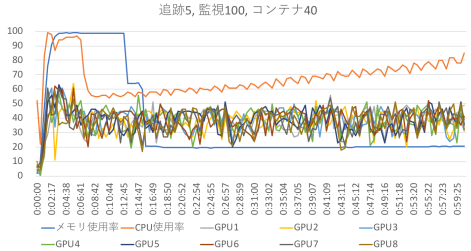

図 8 混雑状況でのリソース状況．

カメラ、空いている状況では 40 カメラまでが映像取得から遅延なく処理できる限界値であり，それ以上のカメ

ラ数を処理するには，CPU がボトルネックとなることが分かった．これらの結果から，今回の実装システムでは概ね検知人数に応じて，処理カメラ数の限界があると考えられる（約 240 人 =20 人 ×12 カメラ）．

4.2 100 カメラ性能試験

4.2.1 条件

100 カメラから配信があったときに，3 台の前述のスペックのサーバにリアルタイム処理が可能かどうか検証を行った．天井や壁に設置される通常の監視カメラの画角で顔を含む人間を照合するには，カメラに映る人の人数は，せいぜい 10 人程度が限界であり，実際我々が内部で撮影した映像では，平均すると 6 人程度だったため，平均して 6 人が映る映像を用いて評価を行った．

実際の防犯カメラは，必ずしも顔が映るポイント（チョークポイント [4]）に設置されるわけではない．むしろ，ほとんどの防犯カメラは，設置場所の制約や景観上の理由により，顔照合には不適切な箇所に設置されている．その傾向を踏まえ，全身照合は 100 映像すべてに対して実施するが，顔に対する処理は 15 映像のみに実施している．

4.2.2 結果

図 4 に 100 映像ストリームを 3 日間処理し続けた際の，各処理の平均 FPS を示す．単体試験と同様にリソース利用状況についても 監視したが，常時安定して

	Tracking server 1	Tracking server 2	Tracking server 3
カメラ映像取得	5.0	5.0	5.0
人検知	5.0	5.0	5.0
トラッキング	5.0	5.0	5.0
再照合	5.0	5.0	5.0
配信処理	5.0	5.0	5.0

図 9　各処理での FPS.

おり，特に問題なく，3 サーバにて 100 カメラ映像を処理可能であることが分かった．

4.3 考察

図10に，100 カメラ処理中の 1 台のサーバの 8 枚の GPU Utilization を示す．GPU の利用率は非常にばら

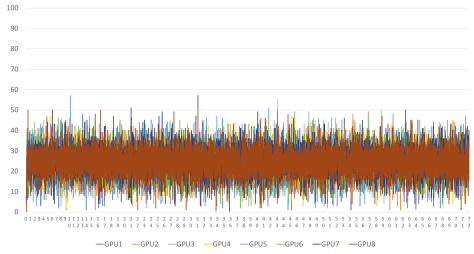

図 10　100 カメラ処理中の GPU Utilization.

ついているとともに，いずれも低い値に留まっている．逆に，CPU 利用率は高い値にあることから，収容数を増やすのにボトルネックになっているのは，GPU ではなく CPU であることが分かる．DNN を用いたアプリケーションとは言え，前・後処理の全てを GPU 内で実施することは難しく，ストリーム処理のほとんどは CPU で処理される．より高収容を目指すには，CPU 処理の GPU オフローディングや CPU 処理の高速化・最適化を検討する必要がある．

また，各パイプラインの中では，できるだけ バッチに固めて DNN の推論を行ってはいるものの，カメラ毎に独立のモデルを持つ実装方式では，限界があることを示している．今回，柔軟性を優先しストリーム処理エンジンを用いた実装を行ったが，今後は GPU 利用率を上げるように DNN 推論を集約する処理が効果的であると考えられる [1].

実験結果としては報告しないが，ネットワークトラフィックにも注意を払う必要があった．FullHD 画質の MJPEG 配信では，カメラ台数が 100 台規模になると，簡単に既設の 1Gbps 帯域を使いつくしてしまう．通常の防犯システムでは，H.264 を用いた圧縮によりトラフィックや保存データサイズの縮小を行うが，H.264 を用いる場合，フレームで切り出した際のノイズが大きくなるり，検知照合の精度に影響を与える．利用可能なネットワーク帯域の上での最適な画質・アルゴリズム・FPS の選択が重要になる．

5 まとめ

今回の実装・評価により，100 台の防犯カメラの映像を，3 台の GPU サーバを使うことで，リアルタイムに人物検索を実現できることを確認した．リソース利用状況の分析により，深層学習を活用したアプリケーションであっても，GPU よりも先に CPU やネットワークトラフィックがボトルネックとなることが分かった．

より高収容化・高速化を達成するために，例えば NoScope [3] のように処理映像を間引き，推論回数を減らす方法が考えられる．NoScope の特化モデルは精度に関する保障がないため，そのまま防犯分野に適用はできないが，精度を落とさない工夫が出来るならば検討の余地はある．また，実装のポータビリティを活かし，エッジサーバや AI カメラに処理の一部をオフローディングすることで，収容率向上やネットワークトラフィック低減することも検討していく．

参考文献

[1] TensorFlow Serving Batching Guide. https://github.com/tensorflow/serving/blob/master/tensorflow_serving/batching/README.md.

[2] Ganesh Ananthanarayanan et al. Real-Time Video Analytics: The Killer App for Edge Computing. *Computer*, pages 58–67, 2017.

[3] Daniel Kang et al. NoScope: Optimizing Neural Network Queries over Video at Scale. In *VLDB*, 2017.

[4] Y. Wong et al. Patch-based probabilistic image quality assessment for face selection and improved video-based face recognition. In *Biometric Workshop(CVPR Workshops)*, 2011.

[5] 株式会社富士経済. セキュリティ関連市場の将来展望, 2017.

[6] 黒崎裕子, 竹房あつ子, 中田秀基, and 小口正人. Apache storm を用いたリアルタイム動画像データ解析フレームワークの性能解析. In *DEIM Forum*, 2016.