
**UNIVERSITÀ POLITECNICA DELLE MARCHE**

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

CORSO DI MISURE E STRUMENTAZIONE PER L'AUTOMAZIONE

# Algoritmo di feature tracking basato su event-camera con un approccio multithreading

Davide Manco

Roberto Broccoletti

Paolo Di Massimo

# Introduzione

## Scopo del progetto

L'obiettivo del nostro lavoro è stato quello di ottimizzare un algoritmo esistente di feature tracking basato su event-camera, mediante il multithreading. Lo scopo è stato quindi quello di cercare di rendere l'algoritmo funzionante in real-time, in modo da poter, in futuro, implementare un'efficiente odometria visuale.

## Requisiti per la lettura

Per comprendere al meglio questa relazione è consigliato leggere:

- Event-based Vision: A Survey (Gallego et al. 2019) https://arxiv.org/pdf/1904.08405
- EKLT: Asynchronous Photometric Feature Tracking Using Events and Frames (Daniel Gehrig et al. 2019)
  http://rpg.ifi.uzh.ch/docs/IJCV19_Gehrig.pdf
- Implementazione originale https://github.com/uzh-rpg/rpg_eklt
- ROS http://wiki.ros.org/ROS/Introduction

## EKLT

L'algoritmo utilizzato come punto di partenza per il nostro lavoro è EKLT, descritto nel paper "EKLT: Asynchronous Photometric Feature Tracking Using Events and Frames" (Gehrig et al. 2019 ).

EKLT è, fondamentalmente, un feature tracker basato su eventi, che estrae le feature a partire dai frame, per poi tracciare tali feature utilizzando solo gli eventi, cercando quindi di combinare i vantaggi delle camere event-based e delle camere tradizionali.

Le camere a eventi consentono di rilevare movimenti rapidi sia della camera che del soggetto, con una latenza molto bassa, e risultando inoltre molto robusto a motion blur. In queste camere ogni pixel lavora in maniera indipendente e asincrona, andando a misurare l'intensità luminosa che lo colpisce, e scatenando un evento nel momento in cui l'intensità luminosa ha una variazione superiore a una certa threshold.

L’algoritmo in esame va quindi ad utilizzare frame presi da una camera tradizionale, estrae le feature, e, prima della valutazione del frame successivo, va a tracciare le feature estratte utilizzando gli eventi.

Più dettagliatamente, nel momento in cui viene arriva un frame (dalla camera tradizionale), viene utilizzato Harris per estrarre i corner dal frame. Ogni corner corrisponde al centro di una patch di dimensione fissa, e per ognuna di esse si memorizza una lista di eventi che ricadono all’interno della patch stessa. Nel momento in cui sono stati collezionati uno specifico numero di eventi (calcolato con una formula per ogni diversa patch) si va a minimizzare la funzione obiettivo: tale funzione obiettivo è la differenza tra la variazione di luminosità calcolata con gli eventi e la variazione di luminosità stimata mediante gradienti. Nell’andare a minimizzare tale funzione obiettivo si cercano i parametri, ovvero la matrice di Warp e il vettore optical flow, usati poi per aggiornare la nuova posizione di ogni patch.

Andando a testare questo algoritmo ci siamo però accorti di come sia estremamente inefficiente, e inutilizzabile in applicazioni reali. Difatti, per processare 10 secondi di frame ed eventi sono necessari in media 1000 secondi (su un Intel i7 CPU, with 64 bits, 2.60 GHz, single-threaded).

Il nostro lavoro parte dunque da qui, con lo scopo di velocizzare l’algoritmo descritto, con il multithreading.

# Approccio

Per prima cosa, abbiamo cercato di capire quale parte dell’algoritmo fosse più onerosa computazionalmente. Abbiamo quindi condotto vari test e ci siamo accorti che, essendo il flusso di eventi molto elevato, e dato che ogni evento può attivare un’ottimizzazione su una patch, tutta questa fase di aggiornamento delle patch risulta molto onerosa.

Più nel dettaglio, come possiamo vedere nel codice riportato, *processEvents* contiene il main loop in cui vengono processati tutti gli eventi. Per ogni evento che arriva viene fatto *updatePatch* su ogni patch; tale funzione è riassumibile nei seguenti passi:

1) controlla che l’evento sia all’interno della patch;
2) se l’evento è all’interno della patch lo inserisce nel batch degli eventi di quella patch;

3) se il numero di eventi è pari alla batch size, viene attivata l'ottimizzazione e aggiornate le matrici di warp, il vettore di optical flow e il centro della patch;
4) viene, infine, svuotato il batch degli eventi legati a quella determinata patch.

NB: l'ottimizzazione su una patch dipende dall'ottimizzazione precedente, per cui per la singola patch le ottimizzazioni sono incrementali.

```
1.  void Tracker::processEvents()
2.  {
3.      ...
4.
5.      while (ros::ok())
6.      {
7.          // blocks until first event is found
8.          waitForEvent(ev);
9.
10.         ...
11.
12.         // go through each patch and update the event frame with the new event
13.         for (Patch &patch: patches_)
14.         {
15.             updatePatch(patch, ev);
16.             // count tracked features
17.             if (patch.lost_)
18.                 num_features_tracked--;
19.         }
20.
21.         ...
22.     }
23. }
```

Data, dunque, la complessità computazionale di questa fase, abbiamo deciso di concentrare i nostri sforzi sul rendere l'esecuzione del main loop parallela.

Il problema di parallelizzare questo workflow risiede nell'avere in media 1 o 2 ottimizzazioni da eseguire per ogni singolo evento che arriva. Per cui non si possono parallelizzare le ottimizzazioni per singolo evento, perché la singola ottimizzazione dura troppo poco per riempire lo slot temporale che il sistema dedica a un singolo thread e si finirebbe a generare un overhead eccessivo. Il problema principale è che le ottimizzazioni vengono scatenate in maniera sequenziale, a causa delle sequenzialità del flusso di eventi; inoltre, come abbiamo anticipato prima, il calcolo delle ottimizzazioni per le singole patch è un processo incrementale e quindi non parallelizzabile.

Si nota quindi come, gestire meno patch, equivale a gestire meno eventi e di conseguenza fare meno ottimizzazioni e quindi velocizzare l'esecuzione. Da questa considerazione parte il nostro lavoro.

# Soluzione

Come anticipato nel precedente paragrafo, l'idea è stata quella di distribuire uniformemente le patch ad un numero configurabile di thread. Ciascun thread esegue il main loop proprio come l'algoritmo originale, andando però a gestire solo una parte delle patch. Uno dei problemi fondamentali da considerare è che nel singolo frame sono presenti delle parti con un'alta densità di feature, che si traducono poi in alta densità di eventi causati dal movimento; viceversa saranno presenti parti con poche feature, e quindi pochi eventi. Di conseguenza pensare di dividere il frame in K parti (con K thread), può risultare poco conveniente, perché potremmo avere dei thread con molte patch da gestire e dei thread con poche patch. Questo comporterebbe uno sbilanciamento del carico, con dei thread che lavorano maggiormente di altri, risultando quindi più lenti, e non portando dei vantaggi nel processo complessivo, data la necessità di dover sincronizzare i thread.

Abbiamo quindi deciso di distribuire le patch basandoci sulla distanza di ognuna di esse dal punto in alto a sinistra del frame; sono state quindi ordinate le patch in base a questa distanza, per poi assegnarle alternativamente ai thread (prima patch al primo thread, seconda patch al secondo thread e così via). In questo si ottiene un corretto bilanciamento del carico poiché le zone più dense di patch sono ripartite sui diversi thread.

Per fare in modo che il risultato ottenuto sia equivalente a quello dell'algoritmo originale, è stato necessario sincronizzare i diversi thread all'arrivo di ogni nuovo frame. Quindi il parallelismo avviene tra due frame: i diversi thread lavorando sulle diverse patch, vanno a processare gli eventi che gli competono tra un frame e il successivo, ottenendo quindi un risultato analogo identico al caso originale. Nel momento in cui tutti i thread arrivano all'evento subito antecedente al successivo frame si fa una valutazione complessiva come nel caso single-thread: ovvero si decide se dover ricalcolare nuove patch, nel caso in cui il numero di patch sia inferiore alla soglia minima prestabilita. Per ottenere un corretto parallelismo, al momento della sincronizzazione si va a fare una ridistribuzione delle patch sui diversi thread, in modo da mantenere il carico sempre bilanciato.

La differenza principale, quindi, con l'algoritmo originale, è che tra un frame e l'altro gli eventi non sono processati da un thread ma da più thread, distribuendo in questo modo il numero di ottimizzazioni e abbassando il tempo di esecuzione complessivo.

# Implementazione

Quello che verrà descritto sono solamente le modifiche al codice originale effettuate da noi, e si sono concentrate sui file *tracker.h, tracker.cpp* e *patch.h.* Per il resto, fare riferimento al codice originale.

## Tracker.h

Questo file è l'header file di *tracker.cpp* in cui viene definita la classe Tracker, e tutti i suoi attributi e metodi.

Partendo dalle strutture dati, nel codice sottostante possiamo vedere ciò che abbiamo aggiunto. Dato che ogni thread ha una sua lista di eventi, abbiamo definito un vettore di *EventBuffer* in cui ogni elemento del vettore è l'EventBuffer del rispettivo thread. Per eseguire le varie sincronizzazioni fra i thread abbiamo definito diversi mutex, ognuno con uno scopo ben specifico, come riportato nei commenti.

Vediamo, inoltre, la definizione del vettore contenente i thread e due vettori che indicano rispettivamente se un thread è entrato nella fase di sincronizzazione e se un thread è in esecuzione.

```
1.  std::vector <EventBuffer> events_;
2.
3.  // mutex
4.  std::vector <std::unique_ptr<std::mutex>> events_mutex_; // mutex used when events are add
    ed or used on the patches
5.  std::mutex images_mutex_; // mutex used when a new image arrives
6.  std::vector <std::unique_ptr<std::mutex>> distribute_mutex; // mutex used when patches are
     changed or distributed
7.  std::mutex file_mutex; // mutex used to synchronize writing on file
8.  std::mutex thread_sync_mutex; // mutex used to synchronize thread
9.
10. // threads
11. std::vector <std::thread> threads; // vector containing all the threads
12. std::vector<bool> entered_thread; // states if a certain thread entered the sync part of t
    he algorithm
13. std::vector<bool> thread_running; // states that a certain thread is running
14.
```

Per quanto riguarda l'inserimento degli eventi all'interno dei vettori definiti nel codice precedente, esso viene effettuato con il seguente metodo.

```
1.  inline void insertEventInSortedBuffer(const dvs_msgs::Event &e) {
2.      for (size_t i = 0; i < events_.size(); i++) {
3.          std::unique_lock <std::mutex> lock(*events_mutex_[i]);
4.          events_[i].push_back(e);
5.
6.          // insertion sort to keep the buffer sorted
7.          // in practice, the events come almost always sorted,
8.          // so the number of iterations of this loop is almost always 0
9.          int j = (events_[i].size() - 1) - 1; // second to last element
10.         while (j >= 0 && events_[i][j].ts > e.ts) {
11.             events_[i][j + 1] = events_[i][j];
12.             j--;
13.         }
14.         events_[i][j + 1] = e;
15.     }
16. }
```

In particolare, vediamo che, per ogni lista di eventi, blocchiamo il relativo mutex per evitare che il thread vada a leggere mentre è in corso la scrittura degli eventi sulla lista. Vediamo, infatti, nella successiva funzione, come viene effettuata la lettura degli eventi.

```
1.  inline void waitForEvent(dvs_msgs::Event &ev, int thread) {
2.      ros::Rate r(100);
3.
4.      while (true) {
5.          {
6.              std::unique_lock <std::mutex> lock(*events_mutex_[thread]);
7.              if (events_[thread].size() > 0) {
8.                  ev = events_[thread].front();
9.                  events_[thread].pop_front();
10.                 return;
11.             }
12.         }
13.         r.sleep();
14.         VLOG(1) << "Thread " << thread << " - Waiting for events.";
15.     }
16. }
```

Come accennato in precedenza, qui troviamo il mutex relativo alla lettura degli eventi, che appunto sarà bloccato se è in corso una scrittura; in caso contrario, bloccherà lui stesso il mutex per evitare che venga effettuata la scrittura nello stesso momento.

Vediamo adesso la funzione che viene richiamata ad ogni evento, e trova l'immagine successiva a quella attuale più vicina all'evento. La nostra aggiunta è stata quella di inserire il parametro op che indica l'operazione da effettuare; se esso è uguale a UPDATE_IMAGE allora significa che l'immagine attuale deve essere aggiornata con l'immagine successiva. Questo parametro è stato aggiunto, perchè solo quando tutti i thread hanno raggiunto l'evento antecedente alla successiva immagine, il primo thread che lo raggiunge aggiorna l'immagine corrente con la successiva.

```
1.  inline bool updateFirstImageBeforeTime(ros::Time t_start, ImageBuffer::iterator &t_image_i
    t, int thread, OperationType op) {
2.      std::unique_lock <std::mutex> images_lock(images_mutex_);
3.      bool next_image = false;
4.      auto next_image_it = current_image_it;
5.
6.      while (next_image_it->first < t_start) {
7.          ++next_image_it;
8.          if (next_image_it == images_.end())
9.              break;
10.
11.         if (next_image_it->first < t_start) {
12.             next_image = true;
13.             if (op == OperationType::UPDATE_IMAGE)
14.                 current_image_it = next_image_it;
15.             break;
16.         }
```

```
    }

    return next_image;
}

```

Con la funzione seguente, facciamo un semplice ordinamento delle patch, basato sulla distanza euclidea dal punto (0,0), coincidente con il punto in alto a sinistra dell'immagine.

```
void sortPatches(Patches &patches) {
    static const auto sort_function = [](const Patch &first, const Patch &second) -
> bool {
        auto firstDistance = first.center_.x * first.center_.x + first.center_.y * first.c
enter_.y;
        auto secondDistance = second.center_.x * second.center_.x + second.center_.y * sec
ond.center_.y;
        return firstDistance < secondDistance;
    };

    std::sort(patches.begin(), patches.end(), sort_function);
}
```

Con *distributePatches* distribuiamo le patch dell'immagine sui diversi thread scegliendo il thread a cui assegnarle, in base al modulo K (con K numero di thread) dell'indice. Il vettore *lost_indices* contiene gli indici delle patch perse, ed è un vettore ridondante utilizzato dall'algoritmo originale per effettuare delle piccole ottimizzazioni. Specificando come parametro *op* il valore INFER_LOST_INDICES costruiamo questo vettore basandoci sull'attributo *lost_* di ogni singola patch.

```
void distributePatches(OperationType op = OperationType::NONE) {
    patch_per_thread.clear();

    for (int i = 0; i < number_of_threads; i++) {
        patch_per_thread.push_back(std::vector<Patch>());
    }

    // If the patches are sorted this part distribute patches in order to have
    // that nearby patches are assigned to different threads.
    // Doing this we have that if an area of the frame is particularly dense of features t
hen
    // the relative computation will be distributed among all the threads.
    for (int i = 0; i < patches_.size(); i++) {
        int thread = i % number_of_threads;
        patch_per_thread[thread].push_back(patches_[i]);
    }

    // create lost indices using the lost_ attribute of the patches
    if (op == OperationType::INFER_LOST_INDICES) {
        for (int i = 0; i < lost_indices_.size(); i++) {
            lost_indices_[i] = std::vector<int>();
            for (int j = 0; j < patch_per_thread[i].size(); j++)
                if (patch_per_thread[i][j].lost_)
                    lost_indices_[i].push_back(j);
        }
    } else {
        for (int i = 0; i < lost_indices_.size(); i++)
            lost_indices_[i] = std::vector<int>();
    }
}
```

La funzione seguente ricostruisce il vettore di patch complessivo a partire dai singoli vettori di patch di ogni thread. Ricostruisce, inoltre, anche il vettore di indici persi globale che può essere anche qui definito a seconda dell'operazione scelta con il parametro *op*. Se op è ugale a SORT_BY_LOST_INDICES  allora le patch vengono ordinate in maniera tale da avere quelle perse alla fine del vettore. Da ciò, si ottengono gli indici delle patch perse; questo ci consente poi di distribuire le patch tra i vettori in modo tale da avere una quantità di patch perse simile in tutti i thread. Questo viene effettuato solo quando non si devono estrarre nuove feature e si deve solo ribilanciare il carico tra i thread. Alternativamente, se op è uguale a INFER_LOST_INDICES, avviene una cosa analoga a quanto descritto nella funzione precedente.

```
1.  void mergeThread(std::vector <std::vector<int>> &lost_indices,
2.              std::vector<int> &global_lost_indices, OperationType op = OperationType::NONE)
    {
3.      patches_.clear();
4.      int k = 0;
5.      int max_size = 0;
6.
7.      for (int i = 0; i < number_of_threads; i++) {
8.          for (int j = 0; j < patch_per_thread[i].size(); j++)
9.              patches_.push_back(patch_per_thread[i][j]);
10.         if (events_[i].size() > max_size) {
11.             k = i;
12.
13.             max_size = events_[i].size();
14.         }
15.     }
16.
17.   for (int i = 0; i < number_of_threads; i++) {
18.       if(i != k && events_[i].size() < events_[k].size()) {
19.           events_[i] = events_[k];
20.       }
21.   }
22.
23.    // sort the vectors in order to have the lost features at the end.
24.    // doing this, we'll distribute the lost patches among all threads
25.    if (op == OperationType::SORT_BY_LOST_INDICES) {
26.        static const auto fun = [](const Patch &a, const Patch &b) {
27.            if (!a.lost_ && b.lost_)
28.                return true;
29.            else return false;
30.        };
31.
32.        std::sort(patches_.begin(), patches_.end(), fun);
33.
34.        int i;
35.        for (i = 0; i < patches_.size(); i++) {
36.            if (patches_[i].lost_)
37.                break;
38.        }
39.
40.        for (; i < patches_.size(); i++)
41.            global_lost_indices.push_back(i);
42.    } else if (op == OperationType::INFER_LOST_INDICES){
43.        for (int i = 0; i < patches_.size(); i++) {
44.            if (patches_[i].lost_)
45.                global_lost_indices.push_back(i);
46.        }
47.    } else {
```

```
48.         for (int thr = 0; thr < lost_indices.size(); thr++)
49.             for (int num : lost_indices[thr])
50.                 global_lost_indices.push_back(num + thr * patch_per_thread[thr].size());
51.
```

## Patch.h

Le uniche modifiche effettuate su questo file sono state lo spostamento del vettore dei gradienti da tracker.h all'interno della stessa classe patch, così che ogni patch contenga al suo interno i suoi gradienti evitando problemi di coerenza con gli indici.

Inoltre, è stato inserito un costruttore di copia poiché quello originale impostava le patch come perse di default.

## Tracker.cpp

Durante la fase di inizializzazione in *init_patches* viene effettuato un *sortPatches*, in maniera tale da avere le patch ordinate per poi poterle distribuire.

Con *DECLARE_int32(thread_number);* abbiamo definito un altro parametro nel file di configurazione per specificare il numero di thread.

È stato modificato il costruttore del tracker perché sono state inizializzate per ogni thread la coda di eventi, il mutex sulla coda degli eventi e tutte le altre strutture di supporto come si vede nel codice.

```
1.  Tracker::Tracker(ros::NodeHandle &nh/*, viewer::Viewer &viewer*/)
2.          : nh_(nh), got_first_image_(false), sensor_size_(0, 0),/* viewer_ptr_(&viewer),*/
    it_(nh) {
3.      event_sub_ = nh_.subscribe("events", 10, &Tracker::eventsCallback, this);
4.      image_sub_ = it_.subscribe("images", 1, &Tracker::imageCallback, this);
5.
6.      std::thread eventProcessingThread(&Tracker::processEvents, this);
7.      eventProcessingThread.detach();
8.
9.      number_of_threads = FLAGS_thread_number;
10.
11.     // initialize all the structures that will be used for the threads
12.     for (int i = 0; i < number_of_threads; i++) {
13.         events_.emplace_back();
14.         events_mutex_.push_back(std::make_unique<std::mutex>());
15.         distribute_mutex.push_back(std::make_unique<std::mutex>());
16.         lost_indices_.push_back(std::vector<int>());
17.         entered_thread.emplace_back(false);
18.         thread_running.emplace_back(true);
19.     }
20.     }
```

Successivamente, abbiamo splittato la funzione originale *updatePatch* in *checkUpdatePatch* e *optimizePatch*, non alterando il funzionamento, ma solo per scopi organizzativi.

Le due successive funzioni riguardano le callback relative all'arrivo di nuovi eventi e nuove immagini. L'unica modifica effettuata è stata aggiungere un mutex per evitare che mentre viene scritta un immagine venga allo stesso tempo gestito un evento in arrivo.

```
1.  void Tracker::eventsCallback(const dvs_msgs::EventArray::ConstPtr &msg) {
2.      std::unique_lock <std::mutex> images_lock(images_mutex_);
3.
4.      ...
5.
6.  }
7.  void Tracker::imageCallback(const sensor_msgs::Image::ConstPtr &msg) {
8.      std::unique_lock <std::mutex> images_lock(images_mutex_);
9.
10.     ...
11.
12. }
```

*addFeature* è la funzione che si occupa di riunire tutte le patch in un unico vettore, trovare le patch perse, e sostituire con delle nuove patch calcolate con Harris sulla nuova immagine. Per fare ciò sono stati inseriti diversi mutex che impediscono la modifica delle strutture dati mentre si sta facendo questa operazione. Viene effettuato inoltre il *mergeThread* con l'operazione INFER_LOST_INDICES per riottenere gli indici delle patch perse. Una volta effettuato l'aggiunta delle nuove patch esse vengono riordinate e ridistribuite ai vari thread.

```
1.  void Tracker::addFeatures(std::vector <std::vector<int>> &lost_indices, const ImageBuffer:
    :iterator &image_it,
2.                            int thread) {
3.      // These mutex will prevent concurrent changes of the patches vectors
4.      std::vector <std::unique_lock<std::mutex>> mutexVector;
5.      for (int i = 0; i < distribute_mutex.size(); i++)
6.          mutexVector.push_back(std::unique_lock<std::mutex>(*distribute_mutex[i]));
7.      for (int i = 0; i < events_mutex_.size(); i++)
8.          mutexVector.push_back(std::unique_lock<std::mutex>(*events_mutex_[i]));
9.
10.     // in this part we need to have all the patches in a single vector in order to repleni
    sh lost features, so
11.     // we merge every thread patch vector
12.     std::vector<int> global_lost_indices;
13.     mergeThread(lost_indices, global_lost_indices, OperationType::INFER_LOST_INDICES);
14.
15.     // find new patches to replace them lost features
16.     Patches patches;
17.     extractPatches(patches, global_lost_indices.size(), image_it, thread);
18.
19.     if (patches.size() != 0) {
20.         // pass the new image to the optimizer to use for future optimizations
21.         optimizer_.precomputeLogImageArray(patches, image_it);
22.
23.         // reset all lost features with newly initialized ones
```

```
        resetPatches(patches, global_lost_indices, image_it);

        sortPatches(patches_);
        distributePatches();
    }

    mutexVector.clear();
}
```

Con la funzione seguente sono inizializzati i parametri (matrice di Warp, vettore optical flow) delle patch estratte nell'immagine precedente.

```
void Tracker::bootstrapAllPossiblePatches(const ImageBuffer::iterator &image_it) {
    std::vector <std::unique_lock<std::mutex>> mutexVector;
    for (int i = 0; i < distribute_mutex.size(); i++)
        mutexVector.push_back(std::unique_lock<std::mutex>(*distribute_mutex[i]));

    for (int j = 0; j < number_of_threads; j++) {
        for (int i = 0; i < patch_per_thread[j].size(); i++) {
            Patch &patch = patch_per_thread[j][i];

            // if a patch is already bootstrapped, lost or has just been extracted
            if (patch.initialized_ || patch.lost_ || patch.t_init_ == image_it->first)
                continue;

            // perform bootstrapping using KLT and the first 2 frames, and compute the ada
  ptive batch size
            // with the newly found parameters
            bootstrapFeatureKLT(patch, images_[patch.t_init_], image_it->second, j, i);
            setBatchSize(patch, patch.gradients.first, patch.gradients.second,
                         FLAGS_displacement_px);
        }
    }

    mutexVector.clear();
}
```

Descriviamo ora la funzione principale, quella che contiene il main loop di ogni thread.

L'inizializzazione si occupa di aspettare che arrivi una nuova immagine, di inizializzare l'ottimizzatore e di distribuire le patch fra i thread. Inoltre vengono inizializzati dei timer usati per calcolare le performance.

```
void Tracker::processEvents() {
    // blocks until first image arrives and sets current_image_it_ to first arrived image

    waitForFirstImage(current_image_it_);

    // initializes patches and viewer
    init(current_image_it_);

    int viewer_counter = 0;
    VLOG(1) << "Thread number:" << number_of_threads;

    // we'll need to distribute the patches among all the threads in order to have a right
   load for every thread
```

```
12.     distributePatches();
13.     std::chrono::time_point <std::chrono::system_clock> begin = std::chrono::system_clock:
    :now();
14.     ros::Time init = current_image_it_->first;
```

All'interno di questo for vengono creati tutti i thread che eseguiranno una funzione contenente il main loop.

```
1.  for (int i = 0; i < number_of_threads; i++) {
2.          most_current_time_.push_back(current_image_it_->first);
3.          VLOG(3) << "Timestamp of thread " << i << ":" << std::to_string(current_image_it_-
    >first.toSec());
4.          threads.push_back(std::thread([&, i]() { ... }));
5.      }
6.  }
```

Come si può vedere dal codice, per ogni nuovo evento vengono controllati i mutex, necessari per mantenere la coerenza,  poi si lavora come nel caso singolo thread.

```
1.  threads.push_back(std::thread([&, i]() {
2.      dvs_msgs::Event ev;
3.
4.      int prev_num_features_tracked = 0;
5.      std::chrono::time_point <std::chrono::system_clock> start = std::chrono::system_clock:
    :now();
6.
7.      // main loop for every thread
8.      while (thread_running[i]) {
9.          // blocks until first event is found
10.         waitForEvent(ev, i);
11.
12.         const cv::Point2f point(ev.x, ev.y);
13.
14.         // go through each patch and update the event frame with the new event
15.         int num_features_tracked = 0;
16.
17.         {
18.             // critical section of the thread, we'll need to avoid that
19.             // the patches change within this part
20.             std::unique_lock <std::mutex> lock(*distribute_mutex[i]);
21.
22.             num_features_tracked = patch_per_thread[i].size();
23.
24.             for (size_t j = 0; j < patch_per_thread[i].size(); j += 1) {
25.                 if (checkUpdatePatch(patch_per_thread[i][j], ev)) {
26.                     optimizePatch(patch_per_thread[i][j], i, j);
27.                 }
28.
29.                 // count tracked features
30.                 if (patch_per_thread[i][j].lost_)
31.                     num_features_tracked--;
32.             }
33.         }
34.
35.         // keep track of the most current time with latest time stamp from event
36.         if (ev.ts >= most_current_time_[i])
37.             most_current_time_[i] = ev.ts;
38.
```

La differenza sostanziale la si ritrova nella sincronizzazione dei thread. Quindi ogni thread deve controllare di aver raggiunto l'immagine subito successiva a quella attuale e se l'ha raggiunta si blocca in attesa che arrivino tutti gli altri thread allo stesso punto. Il primo thread che arriva all'evento subito antecedente al successivo frame, sarà poi responsabile di fare una serie di operazioni.

```
39.         // check if this thread has reached the timestamp of a new image
40.         if (updateFirstImageBeforeTime(most_current_time_[i], current_image_it_, i, Operat
    ionType::NONE)) {
41.             // this thread is ready to be synchronized
42.             entered_thread[i] = true;
43.
44.             // the first thread to pass over this mutex will lock other threads
45.             std::unique_lock <std::mutex> lock(thread_sync_mutex);
46.
47.             // if this is the first thread to reach the new image, then it will wait (acti
    vely) for the other threads
48.             while (updateFirstImageBeforeTime(most_current_time_[i], current_image_it_, i,
49.                                               OperationType::NONE)) {
50.                 // check if all the threads are ready to be synchronized
51.                 bool allIn = true;
52.                 for (int i = 0; i < number_of_threads; i++)
53.                     if (!entered_thread[i])
54.                         allIn = false;
55.
56.                 if (!allIn)
57.                     continue;
58.
59.                 ros::Time min = most_current_time_[0];
60.
61.                 for(auto && t : most_current_time_) {
62.                     if(t < min)
63.                         min = t;
64.                 }
65.
```

Le operazioni riguardano l'aggiornamento dell'immagine corrente con il successivo frame, che poi viene eventualmente utilizzato per calcolare le nuove patch, in caso ci fossero troppe poche feature complessivamente (cioè considerando la somma di tutte le feature perse da ogni thread). Effettua anche il bootstrap delle feature sulle patch che non sono state inizializzate.

```
66.                 if (updateFirstImageBeforeTime(min, current_image_it_, i,
67.                                                OperationType::UPDATE_IMAGE)) // enter if n
    ew image found
68.                 {
69.                     VLOG(2) << "Thread " << i << " updated new image.";
70.
71.
72.                     // bootstrap patches that need to be due to new image
73.                     if (FLAGS_bootstrap == "klt") {
74.                         bootstrapAllPossiblePatches(current_image_it_);
75.                     }
76.
77.                     // calculate the overall quantity of lost indices
78.                     int sum = 0;
79.                     for (auto vec : lost_indices_) {
80.                         sum += vec.size();
81.                     }
```

```
82.
```

Nel caso in non venga effettuato l'addFeatures, vengono comunque riunite le patch con l'opzione OperationType::SORT_BY_LOST_INDICES, e con l'opzione OperationType::INFER_LOST_INDICES viene lanciata la funzione per ridistribuire sui thread le patch. Poi rilascia tutti i lock ed i thread continuano il loro lavoro parallelamente, lavorando tra due frame, fino alla successiva sincronizzazione.

```
83.                     // replenish features if there are too few
84.                     if (sum > FLAGS_max_corners - FLAGS_min_corners) {
85.                         addFeatures(lost_indices_, current_image_it_, i);
86.                     } else {
87.                         // if there are enough features we'll only redistribute them to al
    l the threads
88.                         // in order to rebalance to overall load
89.                         std::vector <std::unique_lock<std::mutex>> mutexVector;
90.                         for (int i = 0; i < distribute_mutex.size(); i++)
91.                             mutexVector.push_back(std::unique_lock<std::mutex>(*distribute
    _mutex[i]));
92.                         for (int i = 0; i < events_mutex_.size(); i++)
93.                             mutexVector.push_back(std::unique_lock<std::mutex>(*events_mut
    ex_[i]));
94.
95.                         std::vector<int> global_lost_indices;
96.                         mergeThread(lost_indices_, global_lost_indices,
97.                                     OperationType::SORT_BY_LOST_INDICES);
98.                         distributePatches(OperationType::INFER_LOST_INDICES);
99.                     }
100.
101.                        // erase old image
102.                        auto image_it = current_image_it_;
103.                        image_it--;
104.                        images_.erase(image_it);
105.
106.                        for (int i = 0; i < number_of_threads; i++)
107.                            entered_thread[i] = false;
108.
109.                        break;
110.                    }
111.                }
112.            }
113.
114.            if (prev_num_features_tracked > num_features_tracked) {
115.                VLOG(2) << "Thread: " << i << " - Tracking " << num_features_tracked <<
     " features.";
116.            }
117.
118.            prev_num_features_tracked = num_features_tracked;
119.
120.        }
121.    }));
122.    }
```

Mantenere una struttura unica creata a partire dalle strutture dati separate, ci ha permesso tramite le fasi di sincronizzazione di tenere sempre il carico bilanciato, di mantenere sotto controllo l'andamento del lavoro dei thread e di ottenere sia a livello logico che pratico lo stesso identico comportamento dell'implementazione originale. Inoltre, facendo così, gran parte del codice già presente è stato riutilizzato,

perché sono state necessarie modifiche unicamente per separare ed unire le strutture dati solo ai fini del calcolo parallelo.

## Installazione ed utilizzo

Per installare ed utilizzare, si può fare riferimento alla pagina github del progetto originale, l'unica differenza è che nel file **eklt.conf** dentro la cartella **config** si può specificare il numero di thread che si vogliono utilizzare. Altra piccola differenza è nel nome per la build catkin che non è più **eklt** ma è **eklt_multithreading**, stessa cosa per quanto riguarda il file .launch che ora è **eklt_multithreading.launch** per lanciarlo con il comando roslaunch. Infine è stato disattivato il viewer, quindi non c'è più la possibilità di valutare in tempo reale l'andamento dell'algoritmo. Rimane possibile scrivere sul file tracks, le tracce, per poi plottarle successivamente. Nel caso si volesse fare odometria, i dati che vengono scritti sul file di track, devono essere anche mandati su un topic.

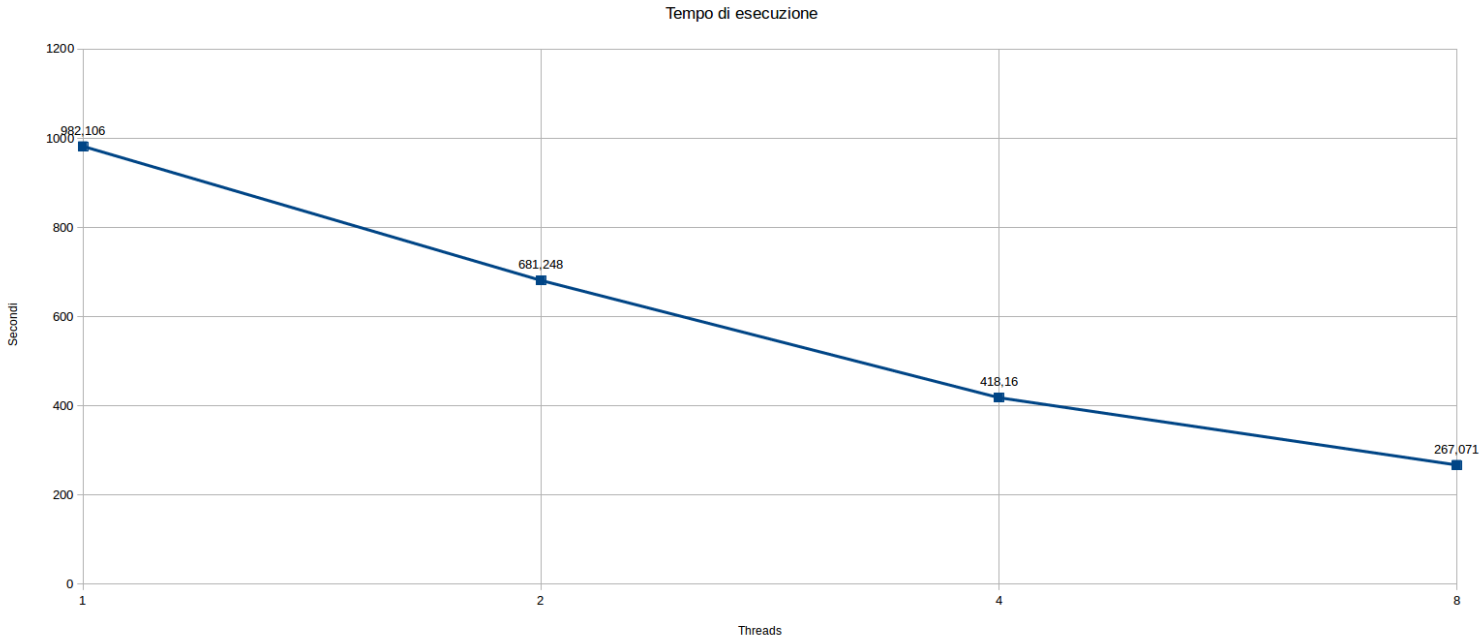
## Valutazione delle performance

Sono state effettuate svariate prove con diverse configurazioni di numero di thread. È stato utilizzato lo stesso rosbag (http://rpg.ifi.uzh.ch/datasets/davis/boxes_6dof.bag) per tutte le prove. Fra una esecuzione e l'altra con stessa configurazione il tempo varia leggermente quindi riportiamo i valori medi. Facciamo lavorare l'algoritmo solo su i primi 10 secondi di dati del rosbag (per questioni di semplicità). Per fare le prove sono state tolte tutte le parti di codice contenenti operazioni di input/output eccetto, ovviamente, quelle relative ai dati letti dal topic. Tutto è stato testato con Ubuntu 16.04 e ROS Kinetic.

La configurazione è:

- Minimo numero di patch: 60
- Massimo numero di patch: 100

Il resto della configurazione è derivata da quella base dell'implementazione originale.




| **Threads** | 1 (Originale) | 2 | 4 | 8 |
|---|---|---|---|---|
| **Secondi** | 982 | 681 | 418 | 267 |
| **Tempo medio per 1 secondo** | 98.2 | 68.1 | 41.8 | 26.7 |

È possibile notare che il tempo scende in funzione della quantità di feature associate ad ogni thread, come previsto inizialmente. Ad ogni modo il rapporto tra thread e tempo di esecuzione crea una curva che tende ad appiattirsi con il crescere del numero di thread, con un andamento simil-logaritmico. L'unico modo di renderlo ancora più veloce è cercare di trovare un giusto compromesso tra il numero di thread ed il numero di patch assegnate ad ogni thread. Considerando infine che eventualmente nel worst-case scenario, con un flusso di eventi molto denso le performance rischiano di peggiorare notevolmente, dato che rimane un algoritmo estremamente dipendente dal numero di eventi, non avendo nessuna politica particolare di scelta sugli eventi da processare.