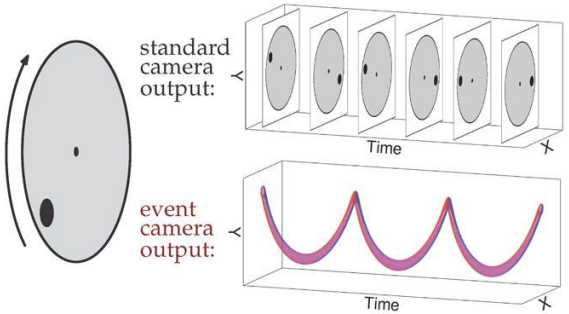
Ottimizzazione CUDA per rappresentazioni stacked di eventi generati da una videocamera neuromorfica

Simone Guidi
Matricola: 981961
Anno accademico 2025/2026

Videocamera ad eventi



Una videocamera ad eventi (o videocamera neuromorfica) genera milioni di eventi al secondo, corrispondenti ai pixel del sensore che riscontrano una variazione di luminosità superiore ad una determinata soglia.

Gli eventi sono restituiti dal sensore sotto forma di 3 informazioni:

- **Posizione** del pixel: coordinate X e Y
- **Polarità** dell'evento: 0 in caso di polarità negativa, 1 in caso di polarità positiva
- **Timestamp** dell'evento: valore intero che rappresenta i microsecondi trascorsi dall'inizio della ripresa

Per natura questi eventi sono **asincroni**, **indipendenti** e **sparsi**:
le prime due caratteristiche rendono questo problema particolarmente adatti ad processamento parallelo, mentre la sparsità degli eventi rende la sfida dell'ottimizzazione più complessa, aspetto che verrà affrontata nel corso del progetto.

Ambienti di testing

Per testare il codice ho utilizzato due ambienti:

- Server domestico:

  virtual machine con passthrough di una scheda **GTX 1050 Ti** (4GB RAM - Pascal), utilizzando una versione di cuda di fine 2019 per poter usare Nsight Compute

- Colab:

  TESLA T4 (16 GB RAM - Turing)

I test di performance li ho eseguiti sul server domestico, raccogliendo i risultati su un database **MongoDB** in modo da confrontare i risultati delle varie esecuzioni.

Per validare la correttezza dei risultati ho utilizzato il **checksum** della matrice risultato.

Il dataset

I dati degli eventi elaborati li ho ottenuti dal dataset **m3ed**, **urban_night_penno_big_loop.h5** (5.8GB), disponibile a questo link: https://m3ed.io/sequences/#urban-night

Ho optato per una scena notturna in modo da evidenziare i vantaggi dati dall'ampia **gamma dinamica** che una videocamera ad eventi è in grado di gestire.

Per i benchmark utilizzerò i primi **60 milioni di eventi**, raggruppati in stack da 200000 eventi, andando a generare un totale di 300 frame.

Rappresentazioni stacked di eventi

Histogram

- Due canali, uno per polarità
- Gli eventi vengono accumulati per pixel nei due rispettivi canali, ottenendo due immagini in scala di grigio

MDES (Multiple Density Event Stacking)

- Ogni canale rappresenta una finestra di eventi dimezzata rispetto alla finestra usata dal canale precedente
- Il numero di canali si può scegliere a seconda del contesto

Tencode

- Gli eventi sono codificati a seconda della polarità e tenendo conto del timestamp
- Il canale rosso è utilizzato per rappresentare gli eventi positivi, il blu quelli negativi, mentre quello del verde è utilizzato per codificare il timestamp

Prima versione CPU Naïve (python)

I dati del dataset vengono caricati dal **file H5**.

Gli eventi vengono gestiti da un ciclo for che rimappa le coordinate X,Y dell'evento in un **indice lineare**, sommando le varie occorrenze nella cella di memoria corrispondente.

Ho gestito l'indirizzo del **canale** direttamente nel calcolo dell'indice in modo da evitare un ciclo if per differenziare il canale.

Tempo di esecuzione: 42.45s

Checksum del risultato:
5074c6c9f7f8be5d7476ba2135821ced

```python
# Carico il dataset
with h5py.File(datasetPath, 'r') as hf:
    dataset = hf['/prophesee/left']
    totalEvents = endEvent - startEvent
    totalStacks = totalEvents // eventsPerStack

    # Definisco l'array del risultato, inizializzando tutte le celle a 0
    data = [0] * ((resolution[0]*resolution[1]*2) * totalStacks)

    X = dataset['x'][startEvent:endEvent]
    Y = dataset['y'][startEvent:endEvent]
    P = dataset['p'][startEvent:endEvent]

    startTime = time.time()
    for eventID in range(0, endEvent-startEvent, 1):
        # Linearizzo la posizione del pixel dell'evento, utilizzo il valore della polarità
        # indirizzare automaticamente al canale corrispondente
        idx = (eventID // eventsPerStack) * (resolution[0]*resolution[1]*2) + \
            (int(Y[eventID])*resolution[1] + int(X[eventID]))*2 + \
            (1-int(P[eventID]))
        if data[idx] < 255: data[idx] += 1

    endTime = time.time()
```

Prima versione CPU Naïve (C++)

Poiché la versione python risultava eccessivamente lenta, ho optato per creare una seconda versione naïve che utilizzi C++.

Per favorire il riutilizzo di codice e la gestione semplificata di strutture dati complesse, oltre all'importazione dei dati del file H5, ho utilizzato **ctypes** per chiamare la nuova versione c++, che permette di ottenere risultati nettamente superiori, e che utilizzerò come riferimento per le implementazioni successive.

Tempo di esecuzione: 0.4231s

Checksum del risultato:
5074c6c9f7f8be5d7476ba2135821ced
(valido)

```
uint8_t* histogramStack(uint16_t* x, uint16_t* y, uint8_t* p,
    int numEvents, int width, int height) {

    const int SIZE = width * height * 2;
    uint8_t* accBuffer = (uint8_t*)malloc(SIZE);

    std::fill(accBuffer, accBuffer + SIZE, 0);

    for (int i  = 0; i < numEvents; ++i) {

        uint16_t curr_x = x[i];
        uint16_t curr_y = y[i];
        uint8_t curr_p = p[i];
        int pos = (curr_y * width + curr_x)*2 +(1-curr_p);

        if (accBuffer[pos] < 255) {
            ++accBuffer[pos];
        }

    }

    return accBuffer;
}
```

Versione CUDA Naïve (PyCuda)

Per la versione cuda ho rimpiazzato l'utilizzo del ciclo for, associando ogni evento ad un thread.

Per mantenere la coerenza della somma ho adottato delle **atomicAdd**.

Tempo di esecuzione: 0.6386s

Checksum del risultato:
5074c6c9f7f8be5d7476ba2135821ced
(valido)

Il risultato tuttavia mostra che il costo di trasferimento dei dati in memoria GPU e l'utilizzo delle atomicAdd, pesano eccessivamente sulle performance di elaborazione, rendendo questa implementazione svantaggioso rispetto alla versione C++

```
kernel_code = """
__global__ void compute_histogram_stacking(unsigned int* d_input_x,
                                           unsigned int* d_input_y, char* d_input_p,
                                           unsigned int* d_output,
                                           unsigned int height, unsigned int width,
                                           unsigned int lastEventIdx,
                                           unsigned int eventsPerStack) {

    int eventIdx = blockIdx.x * blockDim.x + threadIdx.x;

    if (eventIdx < lastEventIdx) {
        int stackId = eventIdx / eventsPerStack;

        unsigned int resX = d_input_x[eventIdx];
        unsigned int resY = d_input_y[eventIdx];
        signed char resP = d_input_p[eventIdx];

        unsigned int idx = (stackId * height * width * 2) +
                           (resY * width * 2) +
                           (resX * 2) + (1 - resP);

        atomicAdd(&d_output[idx], 1);
    }
}
"""
```

Versione CUDA Ottimizzata

Per aumentare le performance del codice cuda ho applicato due strategie:

- Carico in memoria GPU dati a 2 byte (uint16) in modo da diminuire i tempi di trasferimento
- Converto il risultato in uint8 direttamente sulla GPU, in questo modo risparmio ulteriore banda in fase di trasferimento dei risultati

Tempo di esecuzione: **0.3499s**
Speed-up: 1.21

Checksum del risultato:
5074c6c9f7f8be5d7476ba2135821ced
(valido)

```
conv_kernel_code_short = """
__global__ void normalize_and_convert(unsigned short* d_input,
    unsigned char* d_output, int size) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        if (d_input[idx] > 255) d_input[idx] = 255;
        d_output[idx] = (unsigned char)d_input[idx];
    }
}
"""
```

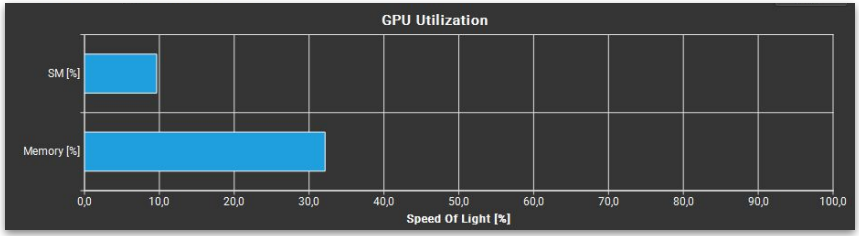
```
func(x_gpu, y_gpu, p_gpu, res_gpu,
        np.int32(h), np.int32(w),
        np.int32(total_events), np.int32(events_per_stack),
        block=(threads_per_block, 1, 1), grid=grid_size)

# Aspetto che la GPU finisca
driver.Context.synchronize()

# Normalizzo il risultato a 1 byte, in modo da velocizzare il trasferimento in memoria host
res_uint8_gpu = gpuarray.empty((total_stacks, h, w, 2), dtype=np.uint8)
norm_func = compiler.SourceModule(conv_kernel_code_short).get_function("normalize_and_convert")
size = total_stacks * h * w * 2
norm_func(res_gpu, res_uint8_gpu, np.int32(size),
        block=(512, 1, 1), grid=((size // 512) + 1, 1))

# Copio il risultato normalizzato sulla RAM dell'host
res_cpu= np.empty((total_stacks, h, w, 2), dtype=np.uint8)
driver.memcpy_dtoh(res_cpu, res_uint8_gpu.ptr)
```

Analisi dei dati di Nsight compute

Come è evidente dai dati di Nsight compute qui mostrati, la sparsità degli eventi impedisce una gestione ottimale della memoria, questo non permette alla GPU di lavorare a pieno ritmo, poiché siamo in un contesto fortemente **memory bound**.

Purtroppo non è possibile sfruttare la shared memory, perché non è sufficiente a contenere le informazioni di un singolo frame (921KB), aspetto che avrebbe permesso di ridurre l'accesso alla memoria globale.

Per questo motivo ho cercato di adottare un approccio diverso per cercare di ottenere risultati migliori, per farlo ho testato la possibilità di ordinare gli eventi prima di passarli alla GPU. Naturalmente questa operazione avrà un costo che analizzerò in rapporto ai possibili scenari di utilizzo.



smsp__pcsamp_warp_stall_wait	44.409 {119}
memory_l2_transactions_global	69.155.022 {6}
Memory Throughput	35.965.673.718,79

Versione con ordinamento

Ho implementato un ultima versione che va ad utilizzare un ordinamento bitonico, per ordinare gli indici degli eventi per ciascuno stack. Poiché l'algoritmo richiede lunghezze di array come potenze di 2, ho utilizzato il valore infinito (0xFFFFFFFF) per indicare gli elementi di padding, che saranno poi ignorati in fase di composizione del frame.

```
__global__ void compute_sorted_histogram(const unsigned int* index_array,
                                         unsigned short* d_output,
                                         int height, int width,
                                         int total_events) {

    int eventIdx = blockIdx.x * blockDim.x + threadIdx.x;

    if (eventIdx < total_events) {
        unsigned int idx = index_array[eventIdx];
        if (idx != 0xFFFFFFFF) atomicAddShort(&d_output[idx], 1);
        // Ignoro gli elementi di padding
    }
}
"""
```

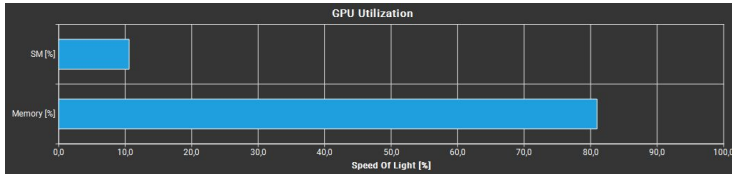
```
kernel_sorted_indexes = """
__global__ void compute_indices_padded(unsigned int *x, unsigned int *y, unsigned char *p,
                                       int tot_events, int events_per_stack, int padded_eps,
                                       unsigned int frame_size, unsigned int w,
                                       unsigned int *out) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int total_padded_size = (tot_events / events_per_stack) * padded_eps;

    if (i < total_padded_size) {
        int s = i / padded_eps;
        int idx_in_stack = i % padded_eps;
        int original_idx = s * events_per_stack + idx_in_stack;

        if (idx_in_stack < events_per_stack && original_idx < tot_events) {
            out[i] = s * frame_size + (x[original_idx] + y[original_idx] * w) * 2 + (1 - p[original_idx]);
        } else {
            out[i] = 0xFFFFFFFF;
        }
    }
}
```

Versione con ordinamento: risultati

La versione con ordinamento mostra un utilizzo della memoria nettamente superiore, e anche il valore di throughput è praticamente raddoppiato.

Tempo di esecuzione: **2.554s**

Checksum del risultato:
5074c6c9f7f8be5d7476ba2135821ced (valido)



Il tempo di esecuzione tuttavia riflette che il costo dell'ordinamento è troppo elevato per renderlo conveniente in questo scenario. Come implementazioni future si potrebbe testare con altri algoritmi di ordinamento, per vedere se è possibile ridurre l'impatto sui tempi.

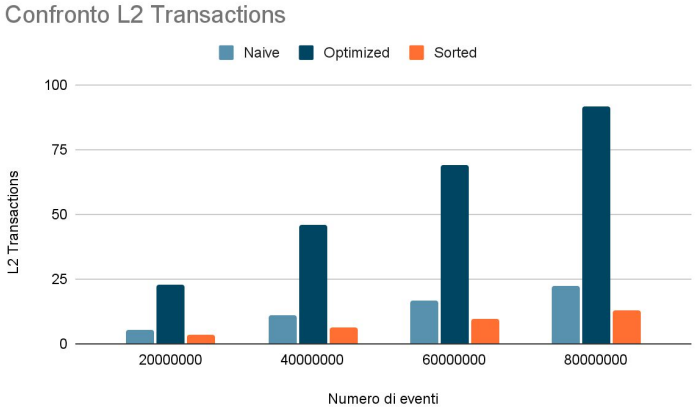
Tuttavia, i tempi di esecuzione escludendo la parte di ordinamento sono quasi **3 volte inferiori** ai tempi della versione CUDA ottimizzata, per questo motivo in scenari dove è possibile ordinare prima il dataset i tempi di processamento potrebbero subire uno speedup molto interessante.

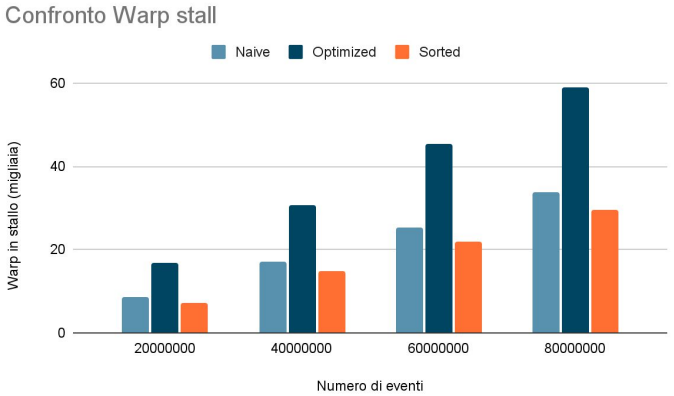
Confronti di stalli e accessi in memoria L2

Ho eseguito Nsight Compute su tutti e tre i codici cuda con un numero di eventi diverso al fine di confrontare i limiti delle 3 implementazioni.

Il codice con ordinamento mostra come previsto un migliore utilizzo della memoria dovuto agli **accessi coalescenti**. In particolare questo è evidente dal numero inferiore di L2 transactions e di stalli dei warp.

La versione ottimizzata invece è quella che presenta più fasi di stallo e più L2 transactions, questo è probabilmente dovuto al fatto che le GPU sono ottimizzate per operare a 32 bit. In particolare l'implementazione di atomicAdd a 16 bit causa probabilmente un maggior numero di letture e scritture che che fanno incrementare moltissimo i valori di L2 transactions e di stalli dei warp.





Confronti delle performance e conclusione

Ho infine eseguito centinaia di volte i 4 codici con un numero diverso di eventi da gestire, i risultati sono riportati nel grafico a fianco.

Dal grafico è evidente come le versioni CUDA (sorted) sia la meno performante, ed inoltre, così come la versione CUDA (naïve), sono particolarmente limitate dalla memoria disponibile sulla mia GPU (4GB), che non permette loro di processare più di 90 milioni di eventi.

La versione CUDA (optimized) con la mia scheda video riesce invece a gestire fino a 140 milioni di eventi mantenendo le performance più solide tra le versioni testate.