Progetto Ingegneria Informatica
Datastream Change Detection

Fioravanti Massimo

Supervisori:
Boracchi Giacomo, Miele Antonio, Carrera Diego

3 settembre 2018

Indice

1 Introduzione

1.1 Obiettivi

Questo documento intende presentare un'implementazione OpenCL del permutation test, ovvero un test di verifica di ipotesi, con lo scopo di offrire velocità di computazione maggiore tramite l'uso schede grafiche, hardware specializzato per il calcolo parallelo. In particolare, i risultati raggiunti sono:

- **Test multivariati** l'analisi di campioni multidimensionali risulta particolarmente difficile e costosa, la nostra implementazione risolve tale problema per costruzione.
- **Memoria lineare** il nostro algoritmo richiede memoria pari a quella necessaria per allocare l'input in ingresso. Tale soluzione è ideale poiché la memoria disponibile sulle schede grafiche è limitata e garantire tale proprietà nel problema affrontato è difficile.
- **Accesso sequenziale** il nostro algoritmo accede alla memoria in maniera sequenziale ed esattamente una volta per ogni elemento, tale soluzione è ideale poiché l'accesso non ordinato rallenta significativamente l'efficienza delle schede grafiche.

1.2 Descrizione del problema

Varie branche scientifiche e economiche moderne, come ad esempio la diagnostica medica, si basano sulla capacità di discernere le differenze tra popolazioni di oggetti. Se è possibile descrivere le popolazioni tramite variabili numeriche allora ne consegue il problema rientra nel campo della matematica. Le procedure che permettono di operare tali distinzioni sono dette test di verifica di ipotesi e la loro formulazione rigorosa giace nel dominio della statistica. Talvolta il costo computazionale necessario per risolvere tali operazioni è assai elevato, in particolare perché la complessità degli algoritmi è spesso legata alla dimensione del campione. Inoltre se si tenta di analizzare un evento reale che avviene ad alta frequenza è importante riuscire a garantire la maggiore velocità di analisi possibile, poiché ciò permette di estendere l'uso dei test di ipotesi a nuovi ambiti. Per queste ragioni è interessante studiare implementazioni efficienti dei test statistici.

Permutation Test In alcune situazioni è possibile porre delle assunzioni riguardo quali sono i parametri delle popolazioni che si vuole studiare, in tali casi è spesso possibile creare dei test specifici particolarmente potenti. Tali test prendono il nome di test parametrici. Altre volte le caratteristiche delle popolazioni sono completamente ignote, poiché magari derivano da eventi naturali. I test ideati per lavorare con queste popolazioni prendono il nome di test non parametrici. I test di permutazione sono particolari all'interno di questa categoria perché il loro funzionamento li rende adatti alla parallelizzazione, e di conseguenza candidati ideali ad essere computati su schede grafiche.

2 Fondamenti teorici

2.1 Statistica

Definiamo come statistica una funzione che associa a una popolazione un numero reale che la rappresenta. Ad esempio la funzione che associa ad una popolazione di lunghezze la media di tali valori è una statistica.

2.2 Test di ipotesi

Per test di verifica di ipotesi si intende un test ideato per verificare su un ipotesi risulta essere vera o meno, e per ipotesi si intende un affermazione riguardante oggetti del mondo reale. I test di ipotesi di dividono in test di ipotesi deterministici o statistici, ed è su questi ultimi che ci concentreremo.

L'ipotesi da verificare prende il nome di H_0 o ipotesi nulla. Si definisce invece l'ipotesi contraria come H_1 o ipotesi alternativa. Come conseguenza di un test vi possono essere due tipi di errori: rifiutare l'ipotesi nulla quando essa era vera, come ad esempio non identificare una malattia in un paziente malato, oppure accettare l'ipotesi nulla quando essa è falsa, ad esempio identificare una malattia in un paziente sano. Il primo caso prende il nome di errore di primo caso, e il secondo errore di secondo tipo. Minimizzare gli errori del primo tipo senza curarsi degli errori del secondo è triviale, poiché è sufficiente accettare ogni volta l'ipotesi. Similarmente è vero l'inverso, rifiutare sempre l'ipotesi H_0 permette di minimizzare gli errori del secondo tipo ma garantisce di massimizzare quelli del primo. Di conseguenza la maniera corretta di formulare un test é garantire costanti gli errori del primo tipo, ad esempio il 5% dei casi, e si desidera minimizzare gli errori del secondo tipo. La soglia mantenuta costante è detta α. Il valore calcolato come 1 meno errori del secondo tipo è detto potenza del test.

Come conseguenza del fatto che gli errori del primo tipo devono essere costanti ne consegue che per certe statistiche è possibile calcolare dei valori di soglia che quando confrontati con la statistica valutata per una popolazione che rispetta l'ipotesi h_0 soddisfano la richiesta che gli errori del primo tipo siano esattamente $\alpha\%$. Questi valori, detti soglie, sono poi utilizzati come di valori di confronto nel caso di popolazioni che si vuole determinare se soddisfano l'ipotesi nulla o meno, e in questa maniera si può ricavare una procedura algoritmica che permetta di eseguire il test di ipotesi.

Ad esempio, supponiamo di voler notare cambiamenti nell'altezza media di due gruppi di persone di dimensione d. Si ha che l'ipotesi nulla vale:

$$H_0 : \mu 1 = \mu 2$$

mentre l'ipotesi alternativa:

$$H_1 : \mu 1 \neq \mu 2$$

dove μX è la media del campione X.

Poiché le funzioni che descrivono le popolazioni sono sconosciute, anche le varianze sono sconosciute, di conseguenza si utilizza la varianza campionaria

$$s^2 = \frac{\sum_{n=1}^{d}(X_n - \mu)^2}{n-1}$$

per calcolare la statistica a cui siamo interessati, definita come

$$S = \frac{\mu 1 - \mu 2}{\sqrt{(s_1^2 + s_2^2)/d}}$$

Intuitivamente tale valore si minimizza quando le medie campionarie sono molto simili oppure quando le varianze sono molto grandi, cioè quando è più probabile che i campioni soddisfino l'ipotesi nulla.

Per d che tende all'infinito la funzione S può essere approssimata da una T di student, la quale è una funzione di cui ne possiamo calcolare l'integrale. Ciò permette di calcolare la soglia a cui siamo interessati.

Accettiamo l'ipotesi nulla se il valore di S giace nell'intervallo [x, ∞] dove X è il valore tale che:

$$\int_X^\infty t\ di\ student = \alpha$$

Poiché questo garantisce che la percentuale di falsi allarmi sia alpha.

Permutation Test Nel caso precedentemente indicato era possibile trovare una formula chiusa che esprimeva la soglia in funzione di α, altre volte ciò risulta impossibile. In tali situazioni è necessario affidarsi ad altre soluzioni e una di queste è il permutation test che permette di stabilire se due gruppi provengono dalla stessa popolazione. La considerazione sulla quale si basa un permutation test è che se si permutano i due gruppi casualmente allora in media α % delle volte la statistica calcolata avrà un valore maggiore della soglia ignota, poiché permutando il campione allora sarà impossibile distinguere gli elementi appartenenti al primo gruppo da quelli appartenenti al secondo. Se si ordinano le statistiche valutate sulle popolazioni permutate e si seleziona quella nella posizione numero permutazioni * (1-α) tale valore risulta essere una stima della soglia, ed è quindi possibile eseguire test di ipotesi statistica pur essendo del tutto ignari della distribuzione di una popolazione. Quindi, dato un campione C, una statistica $S(x)$, un numero di permutazioni n e un livello di significatività α, T è definito come

$$T := \{S(P_1(C)), S(P_2(C)), \ldots, S(P_{\mathrm{n}}(C))\}$$

dove $\mathrm{P_x}$ è una permutazione randomica, inoltre T è ordinato in ordine crescente.

H_0 è rifiutata se:

$$S(C) > T[(1-\alpha) * N]$$

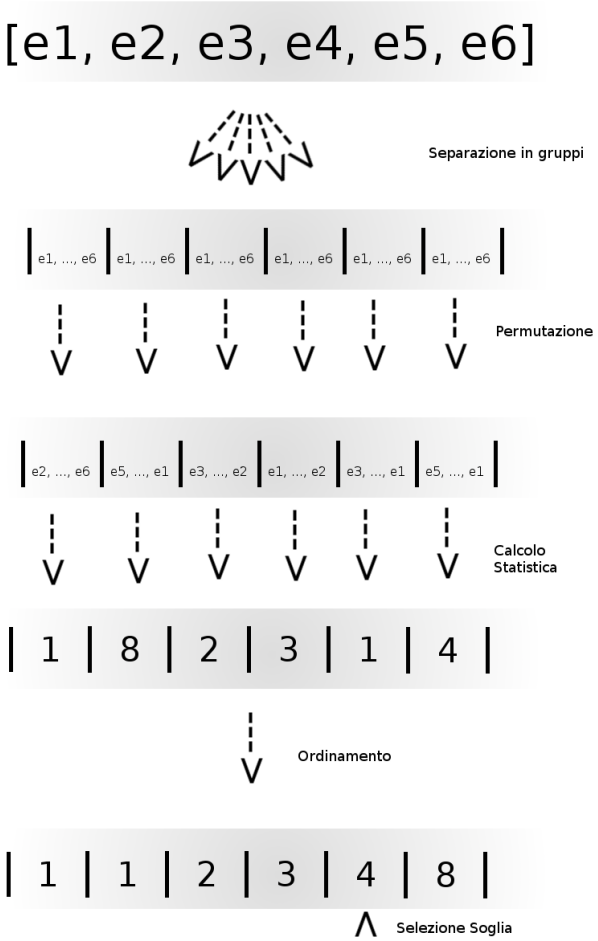

Figura 1: Schema riassuntivo Permutation Test

Procedura Riassumendo, dati un numero di permutationi N e una soglia α la sequenza di operazioni necessarie per operare un test di ipotesi basato sul permutatin test è la seguente.

1. Copiare i dati originali
2. Permutare casualmente i dati originali.
3. Calcolare la statistica e salvare il risultato.
4. Se il numeri di risultato raccolti è inferiore a N tornare al punto 1.
5. Ordinare i risultati.
6. Confrontare la statistica della popolazione originale contro l'elemento (1-α) * dimensione campione. Se la statistica è maggiore, allora sia accetta l'ipotesi nulla, altrimenti la si rifiuta.

3 Implementazione

3.1 Implementazione C

L'algoritmo è composto da due fasi, il calcolo delle statistiche per ogni permutazione e il confronto dei risultati ottenuti. Poiché il risultato deve essere un booleano che indica se le popolazioni dispongono delle stesse caratteristiche, e la statistica è indipendente dal test allora ne consegue la intestazione della funzione c può essere la seguente

```
int runPermutationTest(Data *data, float (*statistic)(Data* data));
```

Data è la struct contenente è definita come segue

```

typedef struct data
{
  float* sample;
  unsigned size;
  unsigned cutPoint;
  float alpha;
  float iterations;
} Data;
```

dove sample è il puntatore al primo elemento dell'array dei dati che include entrambe le popolazioni, cutPoint è la dimensione della prima popolazione (e di conseguenza l'indice del primo elemento della seconda) e size è la somma della dimensione del primo campione e del secondo. La dimensione del secondo campione può essere ricavata come size - cutPoint. Alpha è il livello di significatività del test, mentre iterations è il numero di iterazioni da effettuare.

Si noti anche il secondo parametro

```
float (*statistic)(Data* data)
```

tale scrittura indica che il secondo parametro è una funzione che accetta un puntatore a Data e ritorna un float, cioè una funzione che rappresenta una statistica.

La funzione risulta quindi essere:

```

int runPermutationTest(Data *data, float (*statistic)(Data* data))
{
  float statistics[data->iterationsCount];

  //calcolo della statistica delle popolazioni non permutate
  float sampleStatistic = statistic(data);

  //calcolo delle statistiche permutate
  for (unsigned a = 0; a < data->iterations; a++)
  {
    permutate(data);
    statistics[a] = statistic(data);
  }

  //confronto della statistica con quelle permutate
  sort(statistics, data->iterations, sizeof(float));
  int k = data->iterations-((int)(data->alpha * data->iterations));
  return statistics[k] > sampleStatistic;

}
```

La parte in cui si gestisce il caso statistics[k] == sampleStatistic del return è stato ignorato in favore della semplicità della descrizione. In tali situazioni è sufficiente generare un numero casuale tra 0 e 1 e accettare il risultato se tale valore è inferiore ad alpha. Ciò garantisce che anche nel caso si esegua una permutazione sola il numero di falsi allarmi sarà esattamente alpha.

OpenCL Gli ultimi 15 anni hanno visto l'affermarsi della tecnologia delle schede grafiche come strumento utile alla computazione ad alta velocità. Progettate inizialmente per la creazione di immagini 3D da presentare a schermo, tali unità di calcolo presentano caratteristiche che le rendono adatte a eseguire calcoli in parallelo. Una scheda grafica è composta svariate unità di calcolo ognuna disposta dei propri registri privati, queste unità sono poi riunite in gruppi i quali condividono una memoria locale e infine tutte loro accedono alla stessa memoria globale. I gruppi condividono anche parte delle componenti hardware, in particolare quelle legate al controllo del flusso del programma. Ciò implica che mentre il consumo energetico è minore rispetto alle cpu convenzionali, le unità di calcolo all'interno di un gruppo devono necessariamente eseguire le stesse operazioni. Tale tecnica prende il nome di SIMD, cioè Single Instruction Multiple Data, poiché disponendo di registri diversi per ogni nucleo di calcolo è possibile raggiungere risultati distinti su ognuno di essi, anche se la procedura è la stessa.

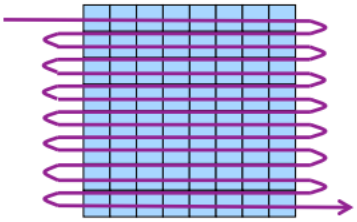

Figura 2: Processo Sequenziale

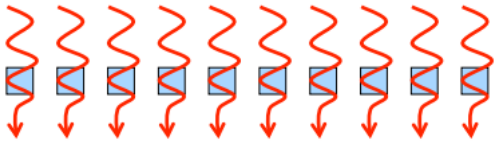

Figura 3: Processo SIMD

Tale architettura permette quindi di accelerare tutti gli algoritmi che possono essere eseguiti in parallelo. Questo è esattamente il caso riguardante i permutation test, se si seleziona una statistica adatta, poiché le permutazione non necessitano di conoscere quale fosse stato il risultato della permutazione precedente, mentre i calcolo della statistica segue le stesse operazioni in ogni unità di calcolo.

Si è mostrato come l'implementazione in un linguaggio di programmazione convenzionale risulta essere triviale, discuteremo ora di come implementare tale

algoritmo in OpenCL, cioè uno dei linguaggi utilizzati dalle schede grafiche. Un programma OpenCL prende il nome di kernel. Un kernel è fondamentalmente una funzione che viene invocata da un normale programma risiedente nella macchina ospite della scheda. L'invocatore deve specificare quali sono i parametri di invocazione, e quale è l'area memoria dove il kernel può scrivere il risultato computato. Inoltre è necessario specificare quante unità di calcolo vogliamo dedicare all'operazione, anche se tale valore non è accessibile all'interno di un programma OpenCl.

Ad esempio supponiamo di voler scrivere un kernel che raddoppia il valore di ogni elemento di un array

```
__kernel void double(__global const float *in, __global float *out)
{
  unsigned int i = get_global_id(0);

  out[i] = in[i] * 2;
}
```

Il programma invocante caricherà nel vettore in i dati da raddoppiare, creerà l'array out dal quale desidera ricavare i risultati e infine lancerà il kernel, indicando che desidera dedicare un numero di core pari alla dimensione dell'array.

Ogni core utilizzerà la funzione get_global_id per ricavare il proprio offset rispetto alla prima unità di calcolo e utilizzerà questo valore per scoprire quale numero deve raddoppiare e dove deve essere salvato.

La keyword __global serve per indicare alla scheda che è un riferimento alla memoria condivisa tra tutte le unità di calcolo.

Implementazione permutation test Per scrivere un algoritmo in OpenCL è necessario identificare quale parti dell'implementazione descritta nella sezione 3.1 sono adatti ad essere selezionati per essere svolti in parallelo.

I dati originali vengono ospitati sulla scheda, il ciclo contenuto del ciclo for viene spostato, nel kernel che si desidera scrivere, mentre l'ordinamento e il confronto vengono operati sulla cpu.

Risulta quindi che un kernel è composto dalla permutazione e dalla valutazione della statistica.

```
  __kernel void p_test
  (
  __global const float* in,
  __global float* out,
  unsigned int cutPoint,
  unsigned int sampleSize,
  )
  {
    unsigned int i = get_global_id(0);
    float[sampleSize] permutated;

    permutate(in, permutated, i);

    b[i] = evaluateStatistic(in, sampleSize, cutPoint, vectorSize);
  }
```

La scrittura efficiente della statistica verrà analizzata nella sezione 4, e l'intero problema si riduce alla capacità di permutare i dati in ingresso.

Ciò è possibile basandosi sulla seguente osservazione: Sia P un numero primo qualsiasi, allora

$$\forall X, Y \in [0, \frac{P}{2}] : X \neq Y \Rightarrow (X * X)\%P \neq (Y * Y)\%P.$$

Ciò significa che se due numeri minori di $\frac{P}{2}$ sono diversi allora è diverso il loro quadrato modulo P, e questo garantisce che se i risultati calcolati dalla espressione

$$F(x) = (x * x)\%P$$

non si ripetano mai. Questa proprietà permette di costruire un una funzione biettiva da $[0, P] \Rightarrow [0, P]$, cioè una permutazione pseudorandomica che opera in tempo e memoria costante.

Il codice risulta quindi essere:

```
unsigned int singlePermutate(unsigned int x, unsigned int prime)
{
  if (x <= prime / 2)
    return x * x % prime;
  else
    return prime - (x * x % prime);

}
```

Si noti che alcune ottimizzazioni sono necessarie per rompere la simmetria centrale dovuta al funzionamento della formula, ma verranno tralasciate per semplicità.

Da questa funzione segue che la variabile X rappresenta il seme del generatore di numeri casuali, e quindi è sufficiente a rappresentarne completamente lo stato.

Il kernel diviene quindi, a patto che sample size sia uguale un numero primo .

```
  __kernel void p_test
  (
  __global const float* in,
  __global float* out,
  unsigned int cutPoint,
  unsigned int sampleSize,
  )
  {
    unsigned int i = get_global_id(0);
    float[sampleSize] permutated;

    for (int a = 0; a < sampleSize; a++)
      permutated[i] =int[singlePermutate(a, sampleSize)];

    b[i] = evaluateStatistic(in, sampleSize, cutPoint, vectorSize);
  }
```

Risulta quindi che tutto ciò che rimane da fare è ordinare i risultati e confrontarli con la statistica dei dati non premutati. Tale computazione può essere svolta dal programma risiedente nella memoria principale della macchina, poiché è necessario attendere che tutte le statistiche vengano valutate e nel caso si utilizzasse più di una scheda non vi è alcuna garanzia che esse terminino in contemporanea.

3.2 Test Multivariati

Queste ottimizzazioni riguardano esclusivamente la maniera in cui si accede ai dati depositati nella memoria della scheda. Di conseguenza è possibile caricare dati sotto forma di struct, invece che semplici float. Risulta quindi raggiunto uno degli obiettivi di tale documento, disponiamo di un implementazione di un test di ipotesi capace di operare su campioni multivariati che non necessita di precalcolare le soglie.

4 Ottimizzazione

4.1 Permutazioni a spazio costante e visita ordinata

Poiché le computazioni avvengono in parallelo, non è possibile permutare sul posto i dati in ingresso, poiché ciò richiederebbe una quantità di memoria pari alla dimensione dei dati in ingresso moltiplicati per il numero di permutazioni, poiché ogni unità di calcolo deve allocare localmente i dati permutati. Risulta quindi imperativo utilizzare un algoritmo a spazio costante, poiché la memoria disponibile per ogni singolo core è limitata. Si mostrerà ora come è possibile computare statistiche che operano in maniera sequenziale senza precalcolare la permutazione da applicare ai dati.

Prendiamo come esempio la statistica che date due popolazioni ritorna la differenza delle loro medie.

```
float evaluateStatistic
(
  __global const float* in,
  const unsigned int sampleSize,
  const unsigned int cutPoint
)
{
  float meanSample1 = 0;
  float meanSample2 = 0;

  for (int a = 0; a < cutPoint; a++)
    meanSample1 += in[a];
  meanSampel1 /= cutPoint;

  for (int a = cutPoint < sampleSize; a++)
    meanSample2 += in[a];
  meanSample2 /= sampleSize - cutPoint;

  return abs(meanSample1 - meanSample2);
}
```

grazie alla funzione di permutazione possiamo mappare gli indici sequenziali al loro equivalente permutato, così facendo perdiamo però la possibilità di capire se l'elemento appartiene al primo gruppo o al secondo. La funzione deve quindi essere modificata in:

```
  float evaluateStatistic
  (
  __global const float* in,
  int prime,
  int index,
  const unsigned int sampleSize,
  const unsigned int cutPoint
  )
  {
    float meanSample1 = 0;
    float meanSample2 = 0;

    for (int a = 0; a < sampleSize; a++)
    {
      int realIndex = permutateSingle(index++, prime);

      if (realIndex < cutPoint)
        meanSample1 += in[realIndex];
      else
```

```
20          meanSample2 += in[realIndex];
21      }
22
23      meanSampel1 /= cutPoint;
24      meanSample2 /= sampleSize - cutPoint;
25
26      return abs(meanSample1 - meanSample2);
27  }
```

Si noti che questo algoritmo dispone di un altra caratteristica, l'accesso alla memoria è sequenziale e tutti i nuclei accedono contemporaneamente allo stesso elemento. Poiché le memorie globali delle schede grafiche sono molto più lente che le memorie private, questa soluzione migliora sensibilmente le performance dell'algoritmo.

4.2 Campioni di dimensione qualsiasi

Come precedentemente indicato, la natura del generatore di numeri casuali impone l'uso di campioni la cui dimensione è un numero primo. Tale restrizione è particolarmente insoddisfacente, quindi si discuterà ora di come rimuoverla: Se la dimensione del campione è troppo distante da un numero primo è possibile allocare come variabile locale un array la cui dimensione è pari alla differenza tra la dimensione e il minimo primo maggiore della dimensione stessa. Ogni volta che si estrae dalla permutazione un numero maggiore della dimensione del campione si salva il valore proveniente dall'array di input nell'array locale in maniera sequenziale. Dopo aver processato tutti gli elementi dell'array globale si processano nuovamente tutti quelli allocati localmente, con la garanzia che ora l'indice estratto sarà minore della dimensione del campione, poiché tutti gli indici maggiori sono già stati valutati.

```
1  float evaluateStatistic
2  (
3  __global const float* in,
4  int prime,
5  int index,
6  const unsigned int sampleSize,
7  const unsigned int cutPoint
8  )
9  {
10   float meanSample1 = 0;
11   float meanSample2 = 0;
12   float[prime - sampleSize] local;
13   int lastAllocatedIndex = 0;
14
15
16   for (int a = 0; a < sampleSize; a++)
17   {
18
19     int realIndex = permutateSingle(index++, prime);
20     if (realIndex > sampleSize)
21       local[lastAllocatedIndex++] = in[realIndex];
22     else if (realIndex < cutPoint)
23       meanSample1 += in[realIndex];
24     else
25       meanSample2 += in[realIndex];
26   }
27
28   for (int a = 0; a < prime - sampleSize; a++)
```

```
29    {
30      int realIndex = permutateSingle(index++, prime);
31
32      if (realIndex > sampleSize)
33        local[lastAllocatedIndex++] = local[realIndex];
34      else if (realIndex < cutPoint)
35        meanSample1 += local[realIndex];
36      else
37        meanSample2 += local[realIndex];
38    }
39
40    meanSampel1 /= cutPoint;
41    meanSample2 /= sampleSize - cutPoint;
42
43    return abs(meanSample1 - meanSample2);
44  }
```

Disponiamo quindi di un algoritmo che accede in maniera sequenziale a tutti gli elementi in memoria ed è capace di permutarli in parallelo utilizzando una quantità di memoria, oltre a quella richiesta dallo spazio dedicato all'input e all'output, costante.

5 Analisi e Conclusioni

5.1 Controllo dei falsi allarmi

La correttezza dell'implementazione di un test di ipotesi può essere controllata assicurandosi che in condizioni di ipotesi nulla il test produca esattamente $\alpha\%$ falsi allarmi. Il test deve poi presentare la maggiore potenza possibile in caso di ipotesi contraria. Per utilizzare una metrica di riferimento abbiamo utilizzato i test inclusi nella libreria standard di matlab. Mostriamo quindi il risultati ottenuti dall'uso di una semplice statistica, la differenza delle medie:

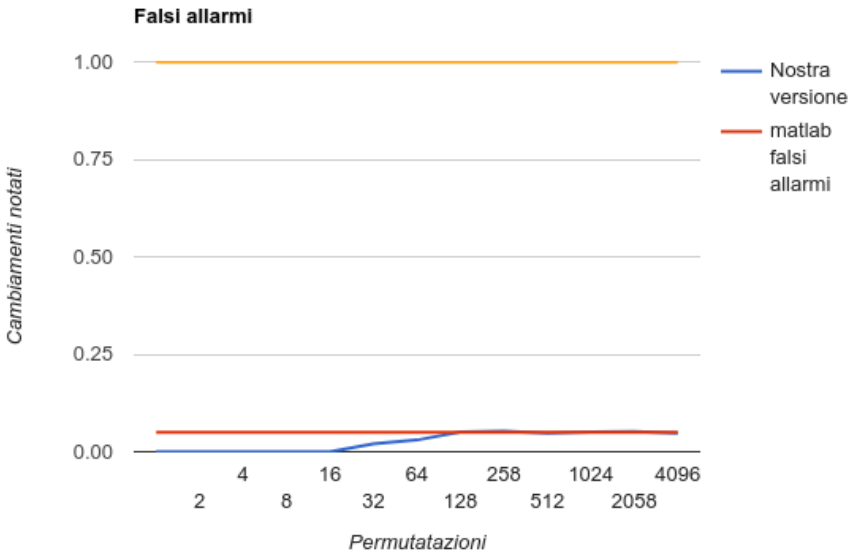

Figura 4: Falsi allarmi

La linea rossa è generata da matlab non dipende dal numero di permutazioni, quindi è costante a 0.05, come ci si attende. La linea blu rappresenta invece la nostra versione del permutation test, la quale per bassi valori di permutazione è del tutto incapace di notare cambiamenti nei valori in ingresso. Quando le permutazioni aumentano i falsi allarmi si stabilizzano velocemente ad α.

Ora che siamo certi che sotto ipotesi nulla il nostro algoritmo si comporta correttamente, possiamo quindi analizzare il caso in cui i gruppi provengono da popolazioni diverse.

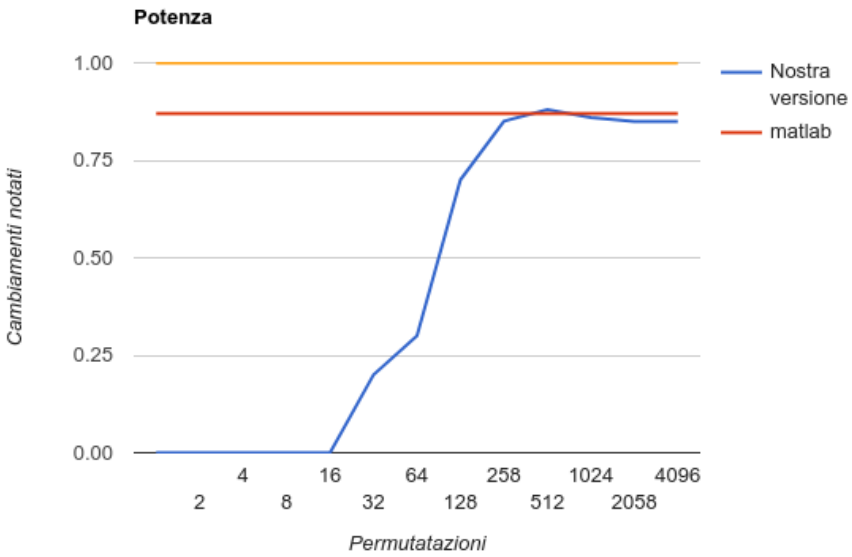

Figura 5: Potenza test

Come nel caso precedente si nota che pur fallendo per permutazioni in quantità minute risulta che il test produce risultati simili a quello standard. Si noti inoltre che come nel caso precedente non sembra esserci alcun vantaggio nell'utilizzare un grande numero di permutazioni, poiché quando raggiungono l'ordine delle centinaia la potenza si stabilizza. Questo significa che, date le ottimizzazioni sull'uso della memoria, se si fissa un numero di permutazioni, è possibile processare set di dati con utilizzi di tempo e memoria che dipendono linearmente solo dalla dimensione dei dati in ingresso. Poiché è evidente che per processare un set di dati è per lo meno necessario valutare almeno una volta tutti gli elementi, ciò risulta essere il migliore risultato possibile.

Ciò che rimane è mostrare la velocità di throughput che è possibile raggiungere è superiore ad un implementazione basata su cpu.

5.2 Benchmark

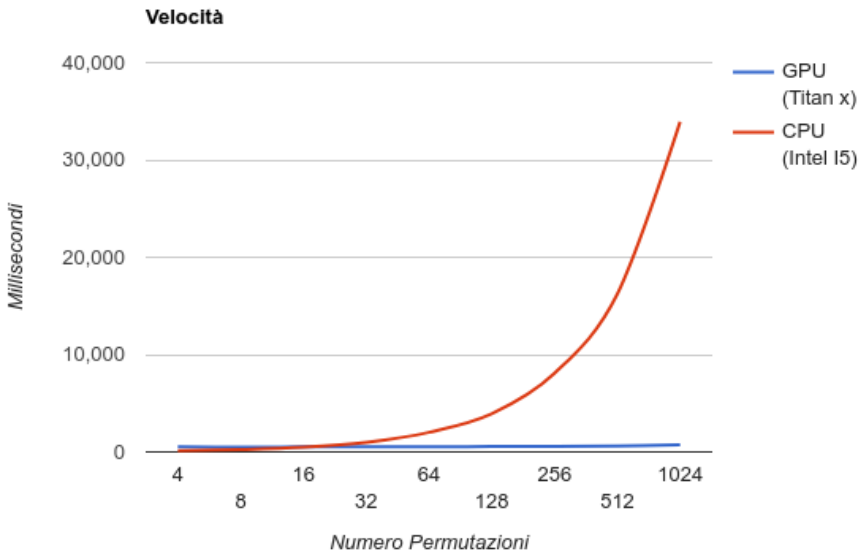

Figura 6: Confronto GPU vs CPU

Il tempo necessario al completamento del calcolo dell'implementazione su cpu dipende linearmente dalla quantità di permutazioni, e allo spostarsi verso destra sull'asse X diventa rapidamente troppo lento. La implementazione GPU invece sembra rimanere a tempo costante, ciò è dovuto al fatto che le permutazioni non superano in quantità il numero di core a disposizione sulla scheda e quindi tutti il calcoli possono avvenire allo stesso momento. Come ci si aspettava l'implementazione su GPU risulta essere meno performante per piccoli valori di permutazioni, dove i vantaggi dell'architettura parallela sono ridotti, ma non appena si supera una soglia critica la implementazione su gpu risulta più efficiente sotto tutti gli aspetti.

Studiano più accuratamente solo i risultati della gpu risulta inoltre:

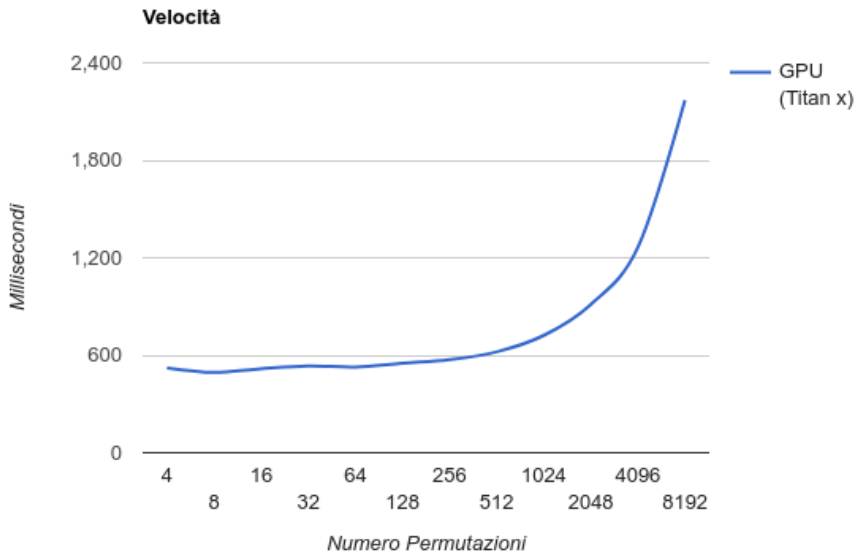

Figura 7: Velocità GPU nel dettaglio

Per piccoli valori di permutazioni il tempo richiesto è effettivamente costante, mentre esso ritorna a dipendere linearmente dal numero di permutazioni non appena la scheda viene saturata. Anche in questo caso il tempo per impiegare 8192 permutazioni è circa 2.5, inferiore a quello necessario per compierne 512 su una cpu. Si noti inoltre che non si erano trovati significativi incrementi di potenza quando il numero di permutazioni superava l'ordine delle centinaia, di conseguenza non vi è motivo di operare nello spazio in cui il tempo di esecuzione dipende linearmente dal numero di permutazioni. Ciò trasforma la nostra implementazione in un algoritmo che opera a tempo costante su set di dati di dimensione fissata dotato di potenza simile a quella dei test disponibili nelle librerie standard ma con un througthput maggiore e capace di operare su dati multivariati, il che era l'obiettivo che si intendeva raggiungere in questo documento.

Le ottimizzazioni inoltre producono i seguenti incrementi in performance:

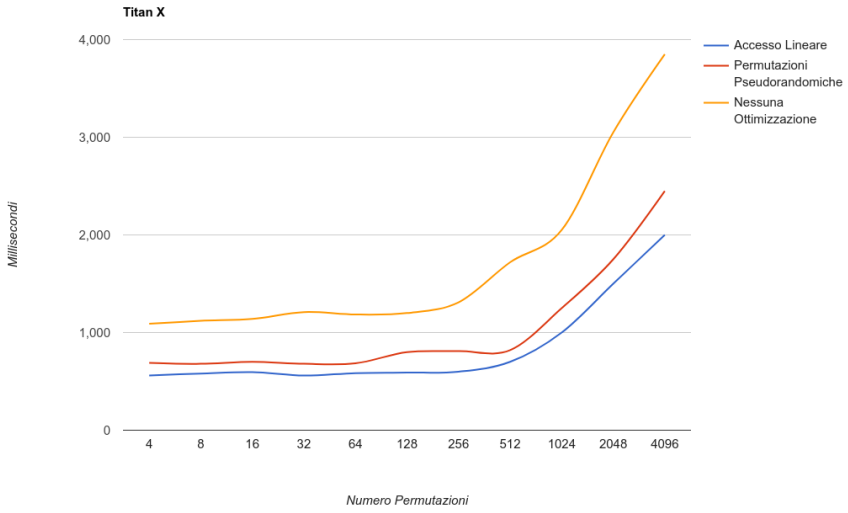

Figura 8: Confronto ottimizzazioni

La linea blu rappresenta l'implementazione OpenCL triviale, come ci si aspetta risulta essere due volte peggiore che la implementazione ottimizzata. La linea rossa rappresenta invece la versione dell'algoritmo dotata di permutazione pseudorandomica ma di cui non è garantito l'accesso ordinato alla memoria. In questo caso ne prestazioni sono paragonabili alle migliori rilevate, sebbene una diminuzione del tempo di calcolo di un sesto giustifica l'uso di codice più complesso e difficile da mantenere.

5.3 Lavori Futuri

A seconda di quale sia il campo di applicazione di un test di ipotesi ad alta velocità è possibile che ulteriori caratteristiche oltre a quelle indicare siano richieste. Si può immaginare che la necessità di analizzare eventi che avvengono ad alta frequenza sia spesso associata alla necessità di processare tali dati sotto forma di stream continui fino a che non viene notato un cambiamento nella natura dei dati osservati. Una delle soluzioni di tale problema prende il nome di Change Point Model (CPM) e consiste nell'analizzare i dati per ogni possibile posizione del change point, per poi selezionare la posizione che più si è discostata dalla soglia. Il tentativo di integrare il permutation test e il CPM si è rivelato più complesso del prevsito, le integrazioni più semplici sono inefficienti, poiché le ottimizzazioni necessarie ad uno risultano in un peggioramento delle prestazioni dell'altro. Questo limite pare significativo e ne consegue che la risoluzione possa essere soggetto di ulteriori studi.