



Università degli Studi di Catania
Dipartimento di Matematica e Informatica
C.d.L. Triennale in Informatica

“Taubin Smoothing Parallelization”

a.a. 2017/18

*Calanna Daniele
Torrisi Riccardo*

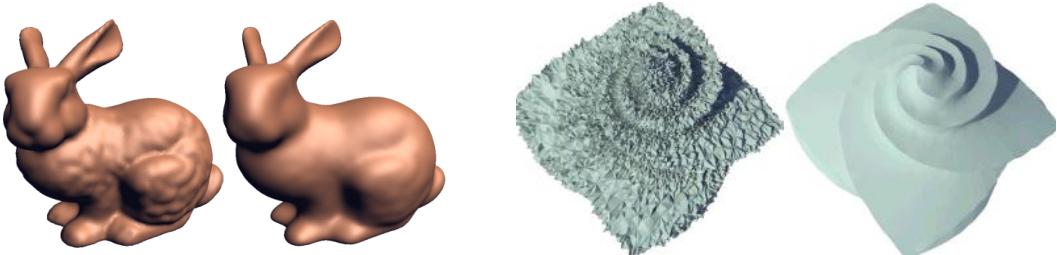


Indice generale

1. Introduzione.....	3
1.1 Gaussian Smoothing (Laplacian Smoothing).....	3
1.2 Taubin Smoothing.....	4
1.3 <i>Sistema cartesiano e meshes di riferimento.</i>	5
1.4 Tipi di dati.....	8
1.5 Il formato obj.....	8
2. Implementazione.....	9
2.1 Kernel smooth.....	13
2.2 Kernel smooth con local memory.....	14
2.3 Kernel smooth con accessi in coalescenza.....	14
2.4 Kernel smooth con accessi in coalescenza e local memory.....	16
3. Interoperabilità OpenCL & OpenGL.....	17
4. Confronto performance e risultati smoothing.....	18
4.1 Profiling meshes.....	18
4.2 Kernels Profiling.....	21
5. Conclusioni.....	26

1. Introduzione

In diversi software 3D, le meshes poligonali forniscono un modo semplice di rappresentare e manipolare complessi oggetti geometrici tridimensionali. Tra le possibili manipolazioni, una tra le più comuni è l'operazione di smoothing (smussamento) o denoising della geometria, il cui scopo è di rendere la superficie della mesh 3D più levigata, senza sfaccettature e spigolosità filtrando via il rumore agendo come filtro passa-basso.



La maggior parte dei metodi di smoothing esistenti soffrono però di diversi problemi. Uno tra i più importanti è il problema dello stiramento della mesh: quando lo smoothing è applicato un gran numero di volte la mesh tende a collassare in un unico punto. Un secondo problema è dovuto al lungo tempo computazionale richiesto per eseguire ciascun passo di smoothing.

Lo scopo di questa documentazione, è di descrivere la nostra implementazione dell'algoritmo di smoothing proposto da *Gabriel Taubin* in “*Curve and surface smoothing without shrinkage*” che riduce al minimo lo stiramento della mesh anche dopo aver effettuato un gran numero di iterazioni, sfruttando però quanto più possibile l’hardware parallelo della GPU, riducendo così il tempo totale richiesto per effettuare un gran numero di iterazioni di smoothing.

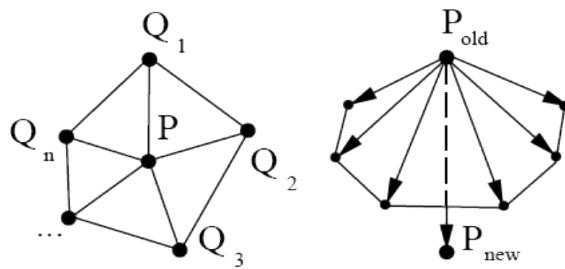
1.1 Gaussian Smoothing (Laplacian Smoothing)

L’idea che sta alla base di tutti gli algoritmi di smoothing consiste nel determinare per ciascun vertice della geometria 3D un “vettore spostamento” ottenuto tramite informazioni sui vertici adiacenti. Ciascun vertice viene successivamente spostato di una frazione del vettore spostamento ottenuto.

$$P_{\text{new}} \leftarrow P_{\text{old}} + \lambda \mathcal{U}(P_{\text{old}})$$

$$\mathcal{U}(P) = \frac{1}{\sum_i w_i} \sum_i w_i Q_i - P$$

$$\mathcal{U}_0(P) = \frac{1}{n} \sum_i Q_i - P,$$



La nuova posizione di ciascun vertice è spesso ottenuta quindi sommando a ciascuna vecchia coordinata il risultato dell’operatore U (detto *Umbrella operator*) moltiplicato per un certo fattore λ . L’operatore ad ombrello è solitamente calcolato effettuando una semplice media (o media pesata) delle distanze tra ciascun vertice e i suoi adiacenti. Questo algoritmo prende il nome di “*Smoothing Gaussiano*” o “*Smoothing Laplaciano*”.

1.2 Taubin Smoothing

La variazione proposta da Gabriel Taubin al fine di ridurre al minimo lo stiramento della mesh dopo un gran numero di iterazioni prevede che, per ogni iterazione di smoothing, vengano effettuati due passi consecutivi di smoothing Gaussiano.

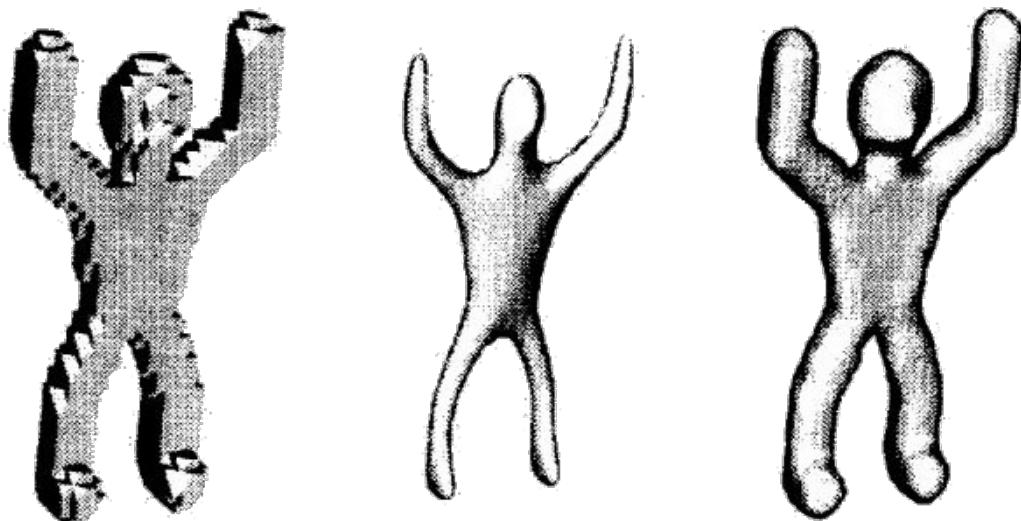
Dopo un primo passo di smoothing Gaussiano, applicato a tutti i vertici della mesh con un fattore positivo $\lambda > 0$, un secondo passo di smoothing Gaussiano è applicato nuovamente a tutti i vertici ma con un fattore negativo $\mu < 0$, maggiore in valore assoluto del fattore λ ($0 < \lambda < |\mu|$).

$$P_{\text{new}} \leftarrow P_{\text{old}} + \lambda \mathcal{U}(P_{\text{old}})$$

$$P_{\text{new}} \leftarrow P_{\text{old}} + \mu \mathcal{U}(P_{\text{old}})$$

Per ottenere un buon effetto smussato, è necessario iterare questi due passi un certo numero di volte. In questo modo eviteremo che la mesh collassi su se stessa, mantenendo buona parte dei dettagli.

Nell'illustrazione che segue è possibile apprezzare il miglioramento qualitativo che l'applicazione dell'algoritmo proposto da Taubin offre rispetto all'applicazione dello smoothing Gaussiano. La geometria a sinistra dell'immagine è la mesh con sfaccettature data in input ai due algoritmi di smoothing, la mesh centrale illustra il risultato dopo avere applicato lo smoothing Gaussiano (o smoothing Laplaciano), mentre a destra è illustrato il risultato ottenuto applicando l'algoritmo di smoothing proposto da Taubin.



1.3 Sistema cartesiano e meshes di riferimento

Il sistema cartesiano di riferimento scelto (*figura 1*) prevede l'asse x con coordinate crescenti verso il lettore per rappresentare la profondità, l'asse y con coordinate crescenti verso destra per rappresentare la larghezza e l'asse z con coordinate crescenti verso l'alto per rappresentare l'altezza della geometria 3D.

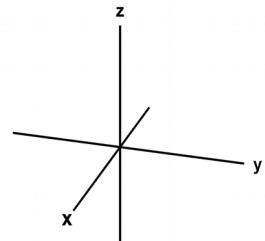


Figura 1) Sistema di riferimento cartesiano

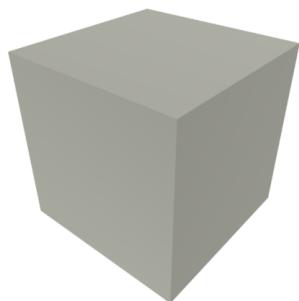


Figura 2) Rappresentazione 3D di un cubo

Al fine di fornire una visione completa e dettagliata delle strutture dati utilizzate e delle diverse implementazioni dell'algoritmo, si è scelto, senza ledere di generalità, di utilizzare come mesh di riferimento un semplice cubo 3D (*figura 2*), con facce triangolate (*figura 3*), per via del ristretto numero di vertici e di facce.

A seguire, vengono riportate immagini e tabelle che permettono una migliore comprensione delle informazioni contenute nelle strutture dati e del modo in cui tali informazioni sono relazionate fra loro, consentendo in generale una migliore comprensione dell'intero documento.

La seguente tabella (*tabella 1*) illustra la numerazione [1 ; 8] dei vertici del cubo di riferimento (*figura 3*) e la terna ordinata (x, y, z) di coordinate spaziali di ciascun vertice.

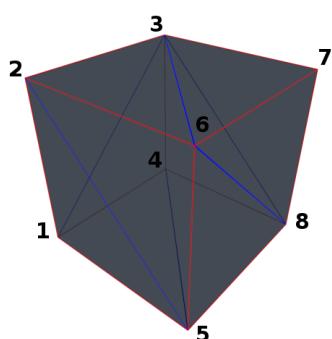


Figura 3) Numerazione dei vertici del cubo

Indice	Vertice	X	Y	Z
0	1	1,0	-1,0	-1,0
1	2	1,0	-1,0	1,0
2	3	-1,0	-1,0	1,0
3	4	-1,0	-1,0	-1,0
4	5	1,0	1,0	-1,0
5	6	1,0	1,0	1,0
6	7	-1,0	1,0	1,0
7	8	-1,0	1,0	-1,0

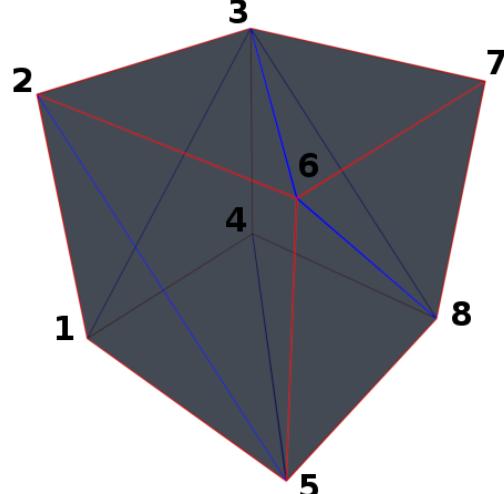
Tabella 1) Numerazione e terna (x, y, z) dei vertici

N.B. La numerazione dei vertici dovrà essere considerata “zero-based” [0 ; 7] solo se espressamente specificato.

Nelle seguenti tabelle sono illustrati gli indici dei vertici che compongono ciascuna faccia del cubo (*tabella 2*) , e il numero di vertici adiacenti di ciascun vertice (*tabella 3*).

Faccia	1° Vertice	2° Vertice	3° Vertice
1	5	1	4
2	5	4	8
3	3	7	8
4	3	8	4
5	2	6	3
6	6	7	3
7	1	5	2
8	5	6	2
9	5	8	6
10	8	7	6
11	1	2	3
12	1	3	4

Tabella 2) Facce del cubo



Vertice	# Vertici adiacenti
1	4
2	4
3	6
4	4
5	5
6	5
7	3
8	5

Tabella 3) Numero di adiacenti di ciascun vertice

Al fine di effettuare test e confronti esaustivi, sono state scelte 3 meshes di riferimento con diverse caratteristiche di cui vengono riportati i dettagli qui sotto:

“Suzanne Monkey”
(Blender Foundation)



- 2.016.578 : numero vertici
- 4.030.464 : numero facce/triangoli
- 12.094.080 : somma adiacenti di ogni vertice
- 3 : numero minimo adiacenti per vertice
- 8 : numero massimo adiacenti per vertice
- 5,999 : numero medio adiacenti per vertice

Figura 4: Mesh 3D "Suzanne"

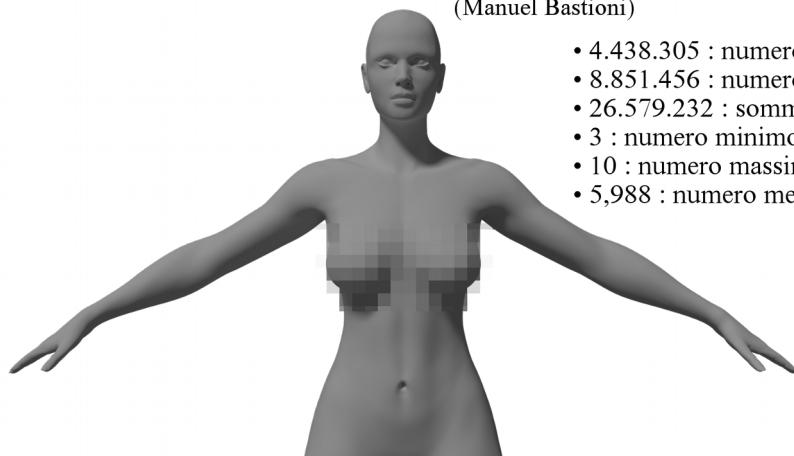
“Two wrestlers in combat”
(Geoffrey Marchal)



- 196.239 : numero vertici
- 391.068 : numero facce/triangoli
- 1.174.644 : somma adiacenti di ogni vertice
- 2 : numero minimo adiacenti per vertice
- 13 : numero massimo adiacenti per vertice
- 5,985 : numero medio adiacenti per vertice

Figura 5: Mesh 3D "Wrestlers"

“Human”
(Manuel Bastioni)



- 4.438.305 : numero vertici
- 8.851.456 : numero facce/triangoli
- 26.579.232 : somma adiacenti di ogni vertice
- 3 : numero minimo adiacenti per vertice
- 10 : numero massimo adiacenti per vertice
- 5,988 : numero medio adiacenti per vertice

Figura 6: Mesh 3D "Human"

1.4 Tipi di dati

La seguente tabella illustra alcuni dettagli riguardanti i tipi di dati maggiormente utilizzati all'interno del nostro progetto. E' quindi fornito il range di valori e la dimensione in memoria di ciascuno di essi.

Tipo	Dimensione	Range valori
char	1 byte	[0 ; 255]
int	4 byte	[- 2.147.483.648 ; 2.147.483.647]
unsigned int	4 byte	[0 ; 4.294.967.295]
uint4	16 byte	[0 ; 4.294.967.295]
uint8	32 byte	[0 ; 4.294.967.295]
float	4 byte	[1,2E-38 ; 3,4E+38]
float4	16 byte	[1,2E-38 ; 3,4E+38]

1.5 Il formato obj

Un file object (*.obj*) rappresenta un formato di file open-source, sviluppato da Wavefront Technologies, per definire geometrie 3D. Il formato *.obj* rappresenta solamente l'essenziale di una geometria 3D, ovvero definisce la posizione di ogni vertice, la posizione di ogni coordinata UV delle texture, le normali e le facce che compongono la geometria. Divenuto quasi uno standard, il formato obj viene spesso utilizzato come formato di passaggio tra diversi applicativi 3D.

La struttura di un file object è molto semplice, non necessita di alcun file aggiuntivo, o header all'inizio del file per essere letto e di solito inizia con linee di commenti precedute dal carattere '#'.

Ogni riga del file inizia con una keyword seguita dai dati ad essa riferiti. Le keyword utili alla nostra implementazione sono di seguito riportate, si omettono quelle non utilizzate ai fini del raggiungimento del nostro obiettivo finale.

Le coordinate dei vertici, le coordinate texture e le direzioni normali elencate nel file *.obj* sono numerate rispettivamente per ciascuna keyword in ordine di apparizione partendo da 1. Per le keyword utilizzate, i dati sono tutti espressi in float ad eccezione dei valori delle facce che sono invece espressi come combinazione di interi che fanno riferimento alle posizioni delle keyword *v*, *vn* e *vt*.

Keyword	Sintassi	Descrizione
v	v x y z w	Coordinate vertice
vt	vt u v w	Coordinate texture
vn	vn i j k	Direzione delle normali
f	f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3	Faccia

2. Implementazione

Durante lo sviluppo del progetto sono state implementate diverse versioni del kernel di calcolo responsabile di effettuare lo smoothing sulla geometria 3D. Per lo sviluppo è stato scelto il linguaggio di programmazione C++ e i test delle performance sono stati eseguiti utilizzando una scheda video “Nvidia GTX 970” e un processore “i5-3570k”, entrambi impostati con i valori delle frequenze di fabbrica.

Riassumiamo in breve i kernel implementati e le loro principali caratteristiche:

- ***smooth*** (kernel base)
- ***smooth_lmem*** (kernel con uso di local memory)
- ***smooth_coalescence*** (kernel con accessi in coalescenza)
- ***smooth_coalescence_lmem*** (kernel con accessi in coalescenza e local memory)

Ciascun kernel richiede una specifica pre-elaborazione dei dati extrapolati dal file object. In particolare risulta fondamentale per il tempo di esecuzione l'organizzazione che questi hanno prima di essere passati in input al kernel di calcolo.

Per questo motivo è stata inserita la possibilità di specificare da riga di comando delle opzioni aggiuntive che possono essere combinate tra loro, alcune delle quali modificano la disposizione dei dati che il kernel prende in input cambiando così le performance dello smoothing.

Comando	Varia disposizione dati	Descrizione
-p <platform_id>	No	Specifica piattaforma
-d <device_id>	No	Specifica device
-m <mesh_name>	No	Specifica mesh
-i <iterations>	No	Specifica numero di iterazioni
-f <smoothing_factor/s>	No	Specifica fattori λ e μ
-w <write_result_bool>	No	Scrivi risultato su file obj
-o sortVertex	Si	Ordina array vertici
-o sortAdjs	Si	Ordina array vertici adiacenti
-o localMemory	Si (internamente al device)	Usa kernel con localMemory
-o coalescence	Si	Usa kernel con coalescenza
-g <local_work_size>	Si (internamente al device)	Specifica dim. work-group

```
\GPGPU-Mesh-Smoothing>meshsmooth.exe -p 0 -d 0 -m suzanne -i 200 -f 5.4 -o lmem sortAdjs -g 32 -w 1
```

Si descrivono di seguito in breve le operazioni di inizializzazione dei dati e di estrapolazione delle informazioni necessarie all'algoritmo, a prescindere dalla scelta del kernel e delle opzioni specificate.

Inizialmente il nostro parser identifica i vertici della mesh e le relative coordinate spaziali e li inserisce nel vector “*vertex_vector*” (*figura 7*) con lo stesso ordine di numerazione del file obj.

1.0	-1.0	-1.0	1.0	-1.0	1.0	-1.0	-1.0	1.0	-1.0	-1.0	1.0	1.0	-1.0	1.0	1.0	-1.0	1.0	1.0	-1.0	1.0	1.0	-1.0	1.0	-1.0	-1.0
X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z		
v1	v2	v3	v4	v5	v6	v7	v8																		

Figura 7: vertex_vector: ciascun colore identifica un vertice della mesh di riferimento (cubo in figura 2/3)

Successivamente vengono identificate le facce della geometria, nel nostro caso composte sempre da 3 vertici connessi tra loro (mesh triangolate), ed inserite nel vector “*facesVertexIndex_vector*” (*figura 8*). Ciascuna faccia mantiene il riferimento ai suoi tre vertici tramite un numero intero che indica la posizione del vertice nel vector “*vertex_vector*”.

5	1	4	5	4	8	3	7	8	3	8	4	2	6	3	6	7	3	1	5	2	5	6	2	5	8	6	8	7	6	1	2	3	1	3	4		
f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12																										

Figura 8: facesVertexIndex_vector

Le operazioni sopra descritte vengono effettuate dalla classe “*OBJ*”, responsabile oltre che della lettura del file object, anche della scrittura del risultato finale in un nuovo file object.

Finite le operazioni di lettura e parsing, la classe “*Smoothing*” ha il compito di creare e inizializzare i buffer device, settare i valori da passare come parametri dei kernel ed accodare gli stessi in base alle opzioni specificate da riga di comando.

Tra i buffer da inizializzare è di particolare importanza il buffer “*cl_adjs_array*”, il quale conterrà le informazioni su quali siano i vertici adiacenti di ciascun vertice. Essendo lo smoothing un’operazione che non altera la relazione “è adiacente di” tra i vari vertici della mesh, riteniamo importante evitare che sia il kernel, ad ogni lancio, a dover interpretare per ogni vertice quale siano i suoi adiacenti. Il buffer è quindi creato e inizializzato, una sola volta, prima del primo lancio.

La rappresentazione di come il buffer “*cl_adjs_array*” viene popolato per la mesh di riferimento (*figura 2/3*) è illustrato in *figura 9*.

La numerazione è da considerarsi da ora in poi “zero-based” in quanto, per convenzione, nel linguaggio di programmazione utilizzato, il primo elemento di ogni array deve essere acceduto con indice 0; lo “0” farà quindi riferimento al vertice v1, “1” farà riferimento al vertice v2 e così via.

4	3	1	2	5	2	0	4	6	7	3	1	5	0	4	0	7	2	0	3	7	1	5	1	2	6	4	7	2	7	5	4	3	2	6	5					
v1	v2	v3	v4	v5	v6	v7	v8																																	

Figura 9: cl_adjs_array: ciascun colore identifica il vertice con cui si ha la relazione di adiacenza

Tra le opzioni che è possibile specificare prima dell'esecuzione dell'algoritmo, vi è l'opzione “*-o sortAdjs*” che consiste in un inserimento ordinato degli indici durante la scoperta degli adiacenti di ciascun vertice ad un costo in termini di tempo aggiuntivo che può essere considerato nullo.

Il risultato che si ottiene applicando l'opzione “*-o sortAdjs*” è illustrato in *figura 10*.

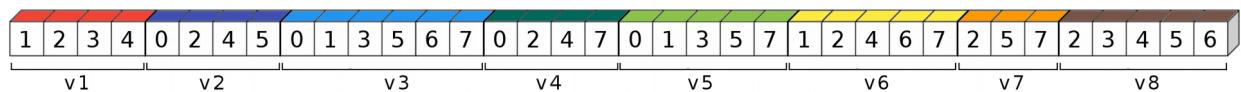
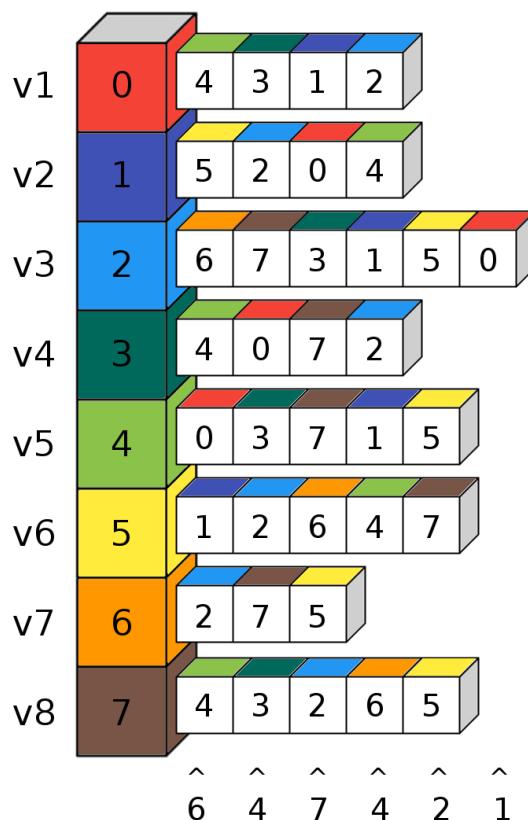


Figura 10: *cl_adjs_array*: indici adiacenti ordinati per ciascun vertice

Il possibile vantaggio dovuto alla scelta di effettuare l'inserimento ordinato dei vertici adiacenti (riassunto in *figura 11/12*), consiste nel ridurre il numero totale di accessi ad elementi diversi in global memory effettuati da ciascun work-group nella lettura di tali, determinando inoltre l'aumento della probabilità di fare cache-hit.

Questa riflessione si basa sull'ipotesi che molteplici richieste di lettura di un singolo dato (o di un numero ristretto di dati) potrebbero essere soddisfatte in un tempo minore rispetto ad un numero di richieste di lettura di dati diversi pari (o di poco inferiore) al numero di richieste.



*Figura 11: Accessi con adiacenti non ordinati.
Totale accessi ad elementi diversi: 24*

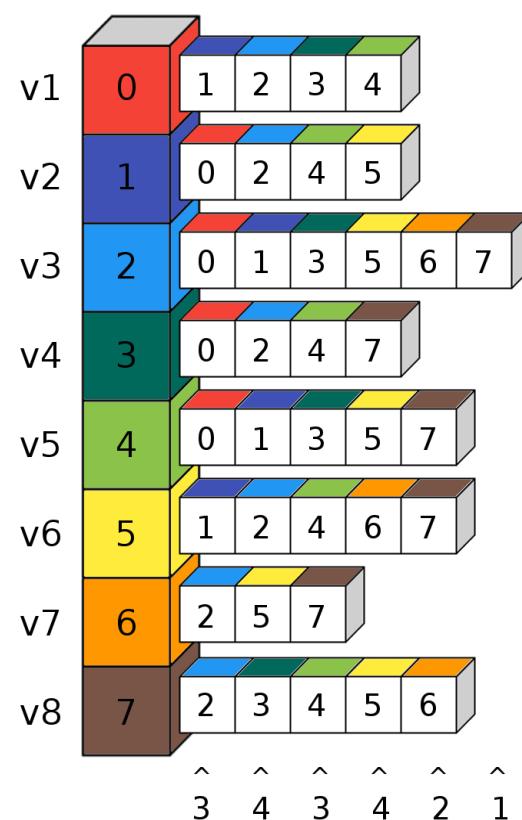


Figura 12: Accessi con adiacenti ordinati. Totale accessi ad elementi diversi: 17

Una seconda opzione disponibile è “*-o sortVertex*”. Come può essere intuito dal nome stesso, quest’ultima assegna alla classe “*Smoothing*” il compito di inizializzare il buffer “*cl_vertex4_array*” riordinando i vertici della mesh 3D contenuti nel vector “*vertex_vector*” in base al numero di vertici adiacenti in ordine decrescente.

A differenza dell’ordinamento degli adiacenti, l’opzione “*-o sortVertex*” ha un grosso impatto sul tempo di pre-elaborazione. Abbiamo osservato un aumento di tempo dal 60% al 80% in più rispetto all’inizializzazione del buffer senza ordinamento. Complice dell’aumento di tempo, ma non unico responsabile, è la necessità di dover riorganizzare i dati anche per il buffer “*cl_adjs_array*” secondo la nuova disposizione dei vertici (figura 13).

La disposizione dei vertici ottenuta applicando l'opzione “-o sortAdjs” è illustrato in *figura 13*.

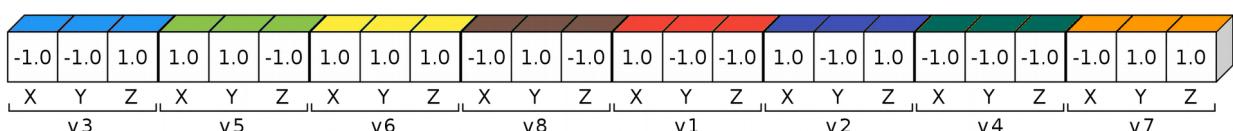


Figura 13: `cl_vertex4_array`: ciascun colore identifica un vertice della mesh di riferimento (figura 2/3)

La nuova organizzazione dei dati nel buffer “cl_adjs_array” è invece illustrata in *figura 14*.

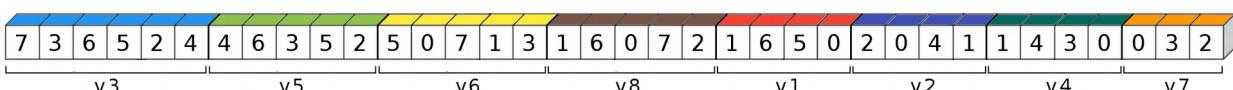


Figura 14: *cl adjs array*: ciascun colore identifica il vertice con cui si ha la relazione di adiacenza

Combinando invece le opzioni fin ora descritte (“`-o sortVertex sortAdjs`”), “`cl_adjs_array`” sarà come illustrato in *figura 15*.

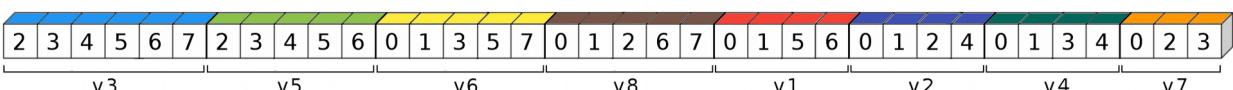
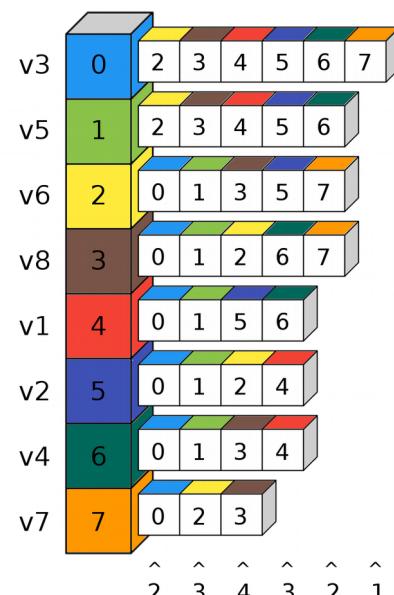
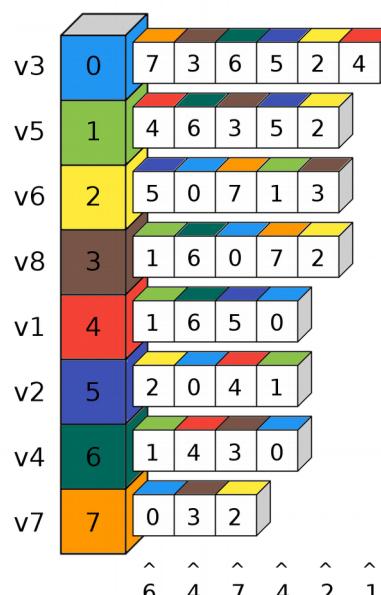


Figura 15: `cl_adjs_array`: indici adiacenti ordinati per ciascun vertice



Un occhio attento potrebbe aver notato che nel nome del buffer che ospita le coordinate dei vertici “*cl_vertex4_array*” è presente il numero 4. Difatti, nonostante le coordinate per ciascun vertice siano solamente 3 (x, y, z) vengono riportate nel buffer con una componente extra (w). Tale componente verrà sfruttata sia per permettere letture allineate in una sola transazione sia per contenere informazioni extra utili (diverse in base al kernel usato) risparmiando così memoria aggiuntiva da dover passare tramite ulteriori parametri. Il modo in cui la quarta componente è sfruttata verrà discussa nei paragrafi a seguire.

2.1 Kernel smooth

Iniziamo col descrivere come primo kernel “*smooth*”. I parametri richiesti dal kernel sono:

- **global const float4 * restrict vertex4_array**
- **global const unsigned int * restrict adjs_array**
- **global float4 * restrict result_vertex4_array**
- **unsigned int nels**
- **float factor**

Come anticipato precedentemente “*cl_vertex4_array*” o “*vertex4_array*” prevede 4 componenti per ciascun vertice della geometria. In questa versione del kernel la componente w di ciascun vertice è stata sfruttata per contenere due diversi dati utilizzando lo spazio di un float (4 bytes / 32 bit). Di questi 4 bytes, 26 bit sono stati assegnati per contenere l’indice di inizio degli indici dei vertici adiacenti al vertice stesso. Tale indice di inizio fa ovviamente riferimento alla struttura “*adjs_array*” o “*cl_adjs_array*” contenente tutti gli indici dei vertici adiacenti. I restanti 6 bit sono invece stati assegnati per memorizzare il numero di vertici adiacenti del vertice in questione. Si tratta dunque di un indirizzamento tramite indice e spiazzamento. Un’illustrazione di quanto appena detto può essere consultata in *figura 16*.

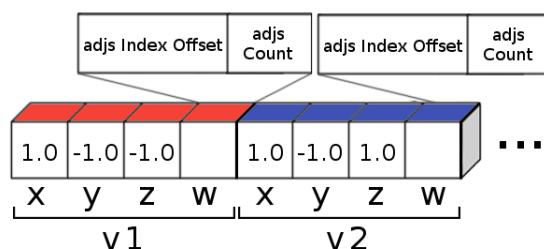


Figura 16: Suddivisione 4a componente

La divisione scelta permette di indirizzare fino a $2^{26} + 2^6 - 1$ elementi. Ciascun vertice potrà avere al più 64 adiacenti e la somma totale del numero di adiacenti di ciascun vertice non potrà eccedere 67.108.927 .

In *figura 17* è mostrato il codice che permette la scrittura all’interno della quarta componente tramite l’uso degli operatori “<<” e “>>” che agiscono sui bit eseguendo degli shift.

```
uint currentAdjsCount = obj_adjacents_arrayVector[i].size();
uint* adjIndexPtr = (uint*)(&vertex4_array[4*i+3]);
*adjIndexPtr = ((uint)currentAdjStartIndex)<<6;
*adjIndexPtr += (currentAdjsCount<<26)>>26;
```

Figura 17: Scrittura 4a componente

La *figura 18* mostra invece il codice eseguito dal kernel per estrarre le due informazioni.

```
const float lastComponent = vertex.w;
const unsigned int adjsInfo = *(unsigned int *)&lastComponent;
const unsigned int adjsIndexOffset = adjsInfo>>6;
const unsigned int adjsCount = ((adjsInfo)<<26)>>26;
```

Figura 18: Lettura 4a componente

Il resto del kernel consiste, per ciascun vertice, nel prelevare e sommare le coordinate dei vertici adiacenti, dividere il risultato per il numero di adiacenti ed infine scriverlo nel buffer “*result_vertex4_array*”.

2.2 Kernel smooth con local memory

Il kernel “*smooth_lmem*” prevede gli stessi parametri e lo stesso meccanismo riguardo la quarta componente del kernel “*smooth*” ad eccezione del parametro “*local float4 * restrict local_vertex4_array*” utilizzato per consentire l’uso della local memory. La differenza con il kernel precedente consiste nell’inserire in memoria locale le coordinate del vertice stesso nella posizione avente come indice la posizione del work-item all’interno del work-group ottenuta tramite la chiamata “*get_local_id(0)*”.

Durante le iterazioni per accedere alle coordinate dei vertici adiacenti un condizionale verifica se l’indice dell’adiacente a cui si sta facendo riferimento ricade all’interno del range degli adiacenti contenuti in memoria locale. In caso di successo, il dato viene prelevato dalla local memory invece che dalla memoria globale.

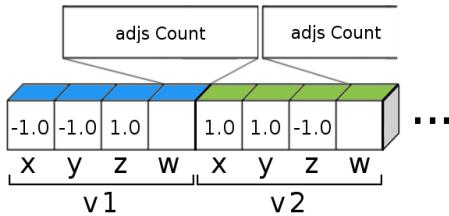
In questa versione del kernel, aumentare la dimensione del work-group è utile in quanto aumenta la possibilità di trovare i dati nella local memory ma, sia la quantità di local memory sia la restrizione necessaria di inserire una barriera, potrebbe non far sfruttare al 100% l’hardware. In base ai test effettuati, sul nostro hardware, la dimensione del work-group consigliata è 32.

E’ possibile specificare che venga utilizzata questa implementazione del kernel specificando tra le opzioni passate da riga di comando l’opzione “-o lmem” o “-o localMemory”.

2.3 Kernel smooth con accessi in coalescenza

Scopo principale del kernel “*smooth_coalescence*” è quello di permettere letture in coalescenza dal buffer “*cl_adjs_array*”. Per poter effettuare tutti gli accessi in coalescenza in memoria globale è indispensabile, per il nostro algoritmo, effettuare l’ordinamento dei vertici per numero di adiacenti tramite l’opzione “-o sortVertex”.

La necessità di effettuare l’ordinamento è dettata dal fatto che nelle comuni geometrie 3D, è molto improbabile che tutti i vertici della mesh abbiano lo stesso numero di adiacenti e ci si ritroverebbe molto spesso ad avere work-group con un range abbastanza ampio nel “numero di adiacenti” che porterebbe a complicazioni dal punto di vista implementativo e a molti mascheramenti durante le iterazioni per accedere agli adiacenti rendendo inutile l’intento di partenza.



In questa implementazione la quarta componente di ciascun vertice è interamente dedicata ad ospitare il numero di adiacenti di ciascun vertice espandendo così il limite precedente di 2^6 adiacenti a 2^{32} .

I parametri richiesti dal kernel “*smooth_coalescence*” sono:

- **global const float4 * restrict vertex4_array**
- **global const unsigned int * restrict adjs_array**
- **global const unsigned int * restrict adjsCounter_array**
- **global float4 * restrict result_vertex4_array**
- **unsigned int nels**
- **float factor**

Diversamente dai kernel finora descritti il kernel “*smooth_coalescence*” richiede un buffer aggiuntivo: “*cl_adjsCounter*” o “*adjsCounter_array*”. Questo buffer viene usato per tener traccia di quanti vertici hanno lo stesso numero di vertici adiacenti partendo dal numero di vertici aventi un solo adiacente fino al numero di vertici aventi il numero massimo in assoluto di adiacenti all’interno della mesh.

La dimensione di “*adjsCounter_array*” è pari al massimo numero in assoluto di adiacenti per vertice della mesh. Prendendo come riferimento la mesh del cubo (*figura 2/3*), la dimensione sarebbe quindi pari a 6 e l’array sarebbe popolato come in *figura 19*.



Figura 19: adjsCounter_array

Il buffer “*cl_adjs_array*” deve essere opportunamente risistemato fornendo i dati raggruppati secondo una nuova disposizione che non rappresenta più un raggruppamento degli adiacenti per vertici ma bensì un raggruppamento per “indici dei primi adiacenti di ciascun vertice”, “indici dei secondi adiacenti di ciascun vertice”, e così via per permettere le letture in coalescenza.

Il contenuto finale del buffer passato come parametro per la mesh di riferimento in *figura 2/3* sarebbe popolato come in *figura 20* o, nel caso in cui venisse specificata l’opzione “*-o sortAdjs*”, come in *figura 21*.

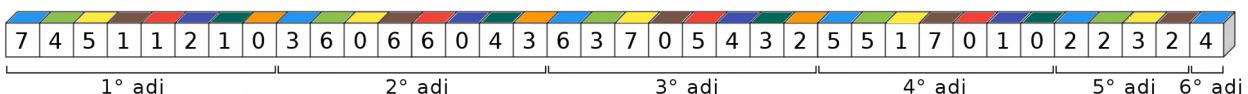


Figura 20: cl_adjs_array: kernel con accessi in coalescenza

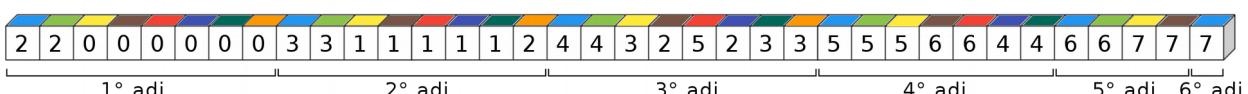


Figura 21: cl_adjs_array: kernel con accessi in coalescenza e indici adiacenti ordinati

La nuova sistemazione degli indici nel buffer “*cl_adjs_array*” ha però un grosso impatto sui tempi di pre-elaborazione. Oltre all’aumento di tempo richiesto per l’ordinamento dei vertici del buffer “*cl_vertex4_array*”, la nuova disposizione degli indici nel buffer “*cl_adjs_array*” richiede in media il doppio del tempo previsto dalla normale sistemazione discussa nelle versioni precedenti.

L’indice da utilizzare per accedere alle informazioni sugli adiacenti contenute nel buffer “*cl_vertex4_array*”, è adesso ricavato dal kernel sommando alla posizione globale, ottenuta tramite la chiamata “*get_global_id(0)*”, un offset. Questo offset serve per spostarsi tra gli indici degli adiacenti dello stesso vertice all’interno del buffer “*cl_adjs_array*”. Il codice che effettua questi calcoli è illustrato in *figura 22*.

```
int offset = 0;
for(int adjIndex=0; adjIndex<adjsCount; adjIndex++)
{
    current_adj = vertex4_array[adjs_array[i+offset]];
    umbrellaOperator += (current_adj - vertex);
    offset += adjsCounter_array[adjIndex];
}
```

Figura 22: Accesso informazioni adiacenti nel kernel "smooth_coalescence"

La restante logica per effettuare ciascun passo di smoothing, rimane invariata rispetto alle precedenti versioni.

Per specificare che venga utilizzata questa implementazione del kernel è necessario inserire da riga di comando l’opzione “-o coal” o “-o coalescence”.

2.4 Kernel smooth con accessi in coalescenza e local memory

Piccola variazione del kernel “*smooth_coalescence*” è il kernel “*smooth_coalescence_lmem*”. Quest’ultimo kernel prevede la stessa logica e gli stessi parametri del kernel descritto in precedenza con l’aggiunta del parametro “*const unsigned int adjsCounter_array_dim*” e del parametro “*unsigned int * restrict local_adjsCounter_array*” utilizzato per permettere l’uso della local memory.

In questa implementazione i primi elementi di ciascun work-group si occupano di copiare i dati contenuti nel buffer “*cl_adjsCounter_array*” in memoria locale evitando così accessi ripetuti in memoria globale inutilmente.

Definita con “*adjsCounter_array_dim*” la dimensione del buffer “*cl_adjsCounter_array*”, solo i work-item del workgroup con indice locale minore stretto di “*adjsCounter_array_dim*” scriveranno in memoria locale. I restanti work-item attenderanno tramite apposita barriera. (*vedi figura 23*)

```
const unsigned int li = get_local_id(0);
const unsigned int i = get_global_id(0);

if(li<adjsCounter_array_dim) local_adjsCounter_array[li] = adjsCounter_array[li];
barrier(CLK_LOCAL_MEM_FENCE);
if(i >= nels) return;

const float4 vertex = vertex4_array[i];
const float lastComponent = vertex.w;
const unsigned int adjsCount = *(unsigned int *)&lastComponent;

float4 current_adj;
float4 umbrellaOperator = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
int offset = 0;
for(int adjIndex=0; adjIndex<adjsCount; adjIndex++)
{
    current_adj = vertex4_array[adjs_array[i+offset]];
    umbrellaOperator += (current_adj - vertex);
    offset += local_adjsCounter_array[adjIndex];
}
```

Figura 23: Accesso informazioni adiacenti nel kernel "smooth_coalescence_lmem"

3. Interoperabilità OpenCL & OpenGL

OpenCL e OpenGL sono 2 APIs che supportano l'interoperabilità: una delle principali funzionalità è la capacità di condividere dati utilizzando gli stessi buffer senza dover quindi effettuare alcuna copia (“zero-copy”).

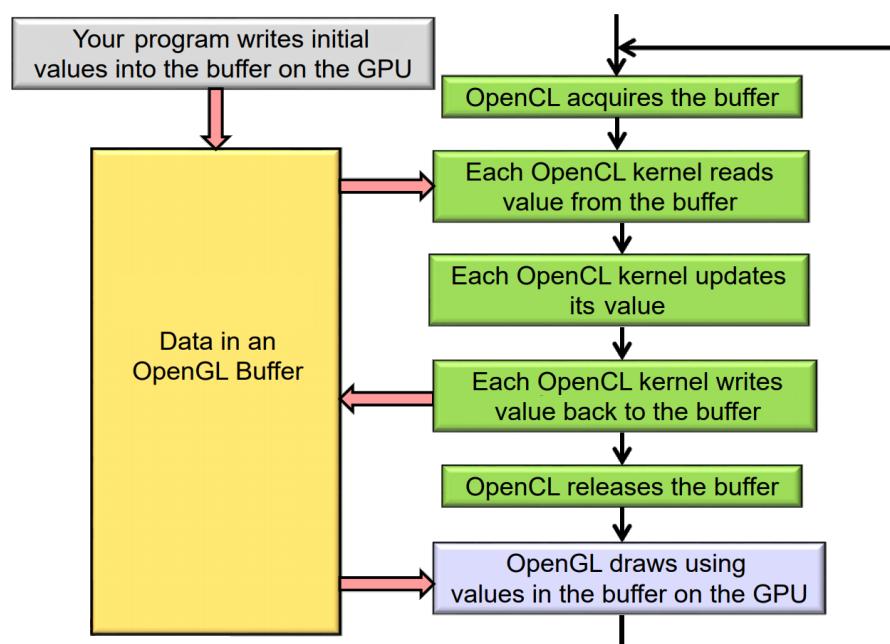
Nel nostro progetto, una volta che l'algoritmo di smoothing in OpenCL (eseguito dalla GPU) ha completato la sua esecuzione scrivendo i risultati in un specifico buffer, è necessario comunicare ad OpenGL di eseguire il rendering del risultato senza dover copiare i dati in un altro buffer.

La creazione del buffer condiviso richiede un contesto condiviso tra OpenCL e OpenGL che opera sullo stesso device. Tale contesto viene creato da GLFW (API per la creazione di finestre, contesti e gestione degli input) ed utilizzato da OpenGL per effettuare il rendering.

Il contesto condiviso può essere ottenuto chiamando da OpenCL “`clCreateContext`” passando come proprietà “`wglGetCurrentContext`”. Successivamente è possibile creare una nuova *command queue*, un nuovo *program* dal source che implementa lo smoothing e un *buffer condiviso* tra le due APIs.

Un OpenCL memory object viene creato da un OpenGL object e non vice versa. Nel nostro caso un OpenGL vertex buffer verrà trasformato in un OpenCL buffer tramite “`clCreateFromGLBuffer`”.

Dato che molte chiamate OpenCL e OpenGL non sono eseguite immediatamente ma vengono inserite nelle command queues, è richiesta una coordinazione per l'accesso alle risorse tra le due APIs. Prima di poter essere usati, i buffer condivisi necessitano quindi di essere acquisiti da un comando OpenCL in una command-queue. La chiamata “`clEnqueueAcquireGLObjets`” effettua questa operazione di acquisizione, ritornando un `cl_event` che dovrà essere inserito nella waiting-list del kernel di calcolo.



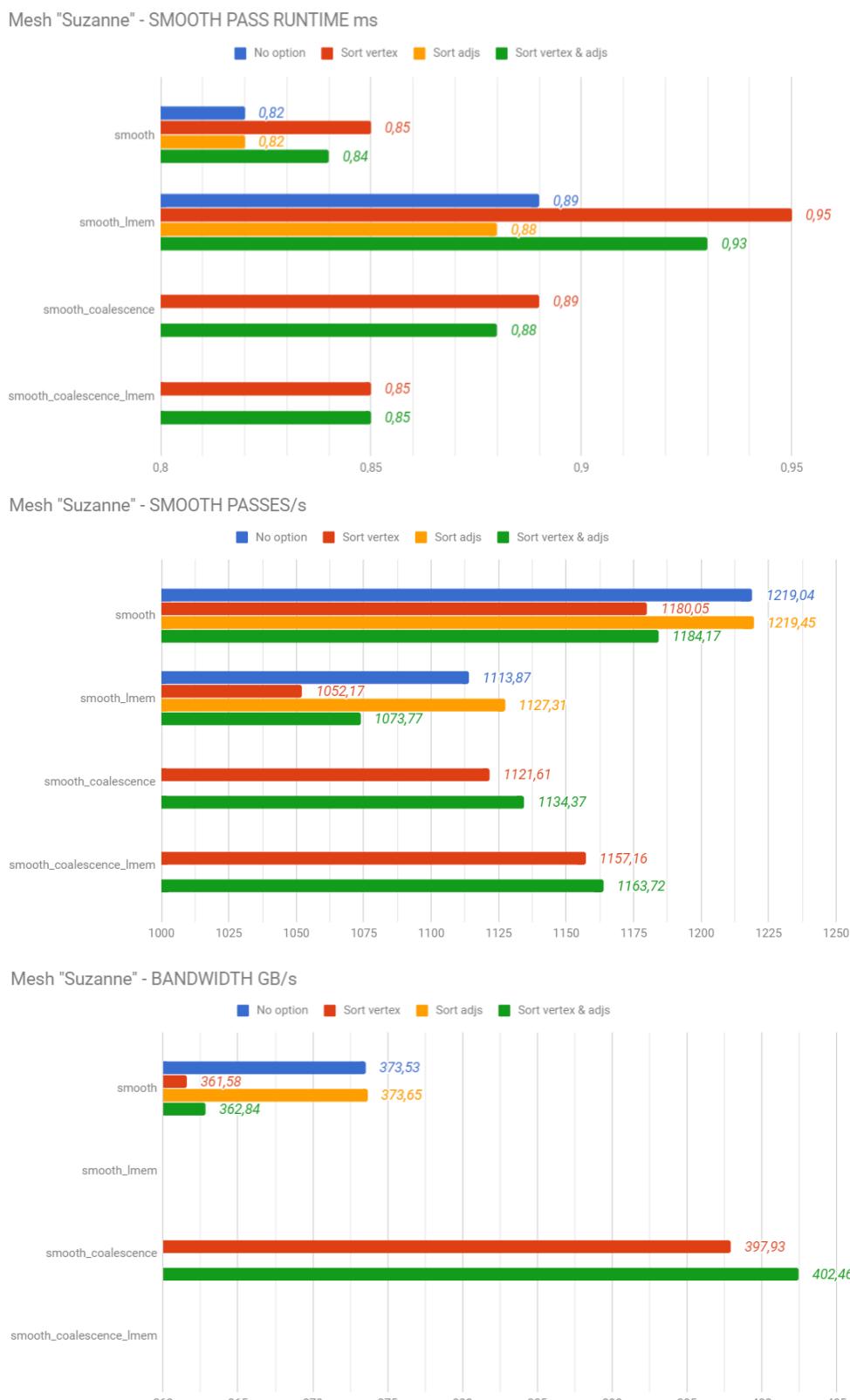
Un volta che il kernel di calcolo a concluso la sua computazione, è necessario rilasciare il buffer condiviso tramite la funzione “`clEnqueueReleaseGLObjets`”, in modo tale da renderlo nuovamente disponibile ad OpenGL.

4. Confronto performance e risultati smoothing

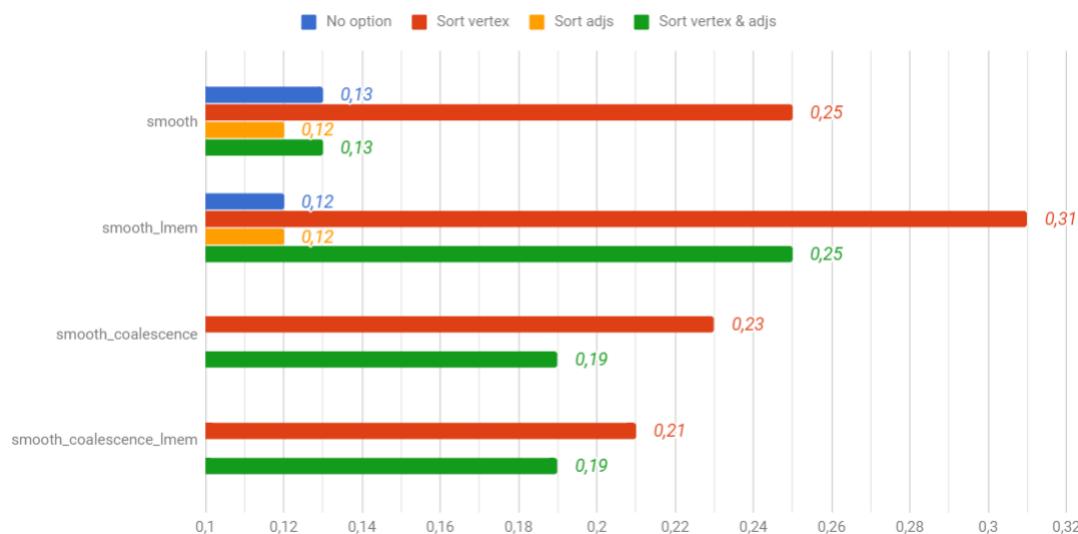
Di seguito vengono riportati i grafici dei test eseguiti per tutte le combinazioni possibili tra kernel e opzioni disponibili. Per una migliore analisi e comprensione, i dati sono stati riportati raggruppando i grafici prima per mesh in input e successivamente per versione di kernel.

I test sono stati eseguiti usando una scheda Nvidia GTX 970 e riportando la media di 200 iterazioni.

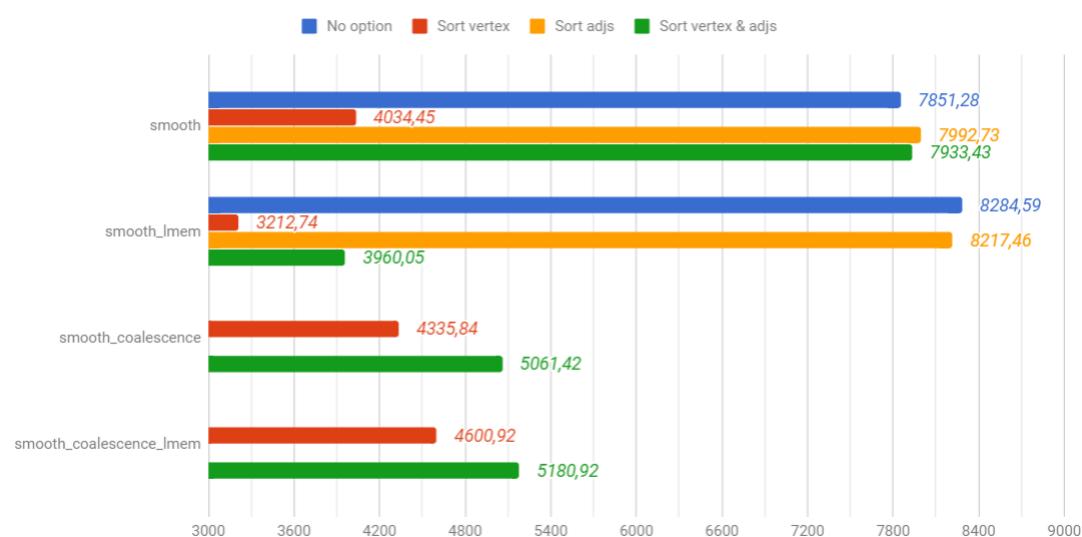
4.1 Profiling meshes



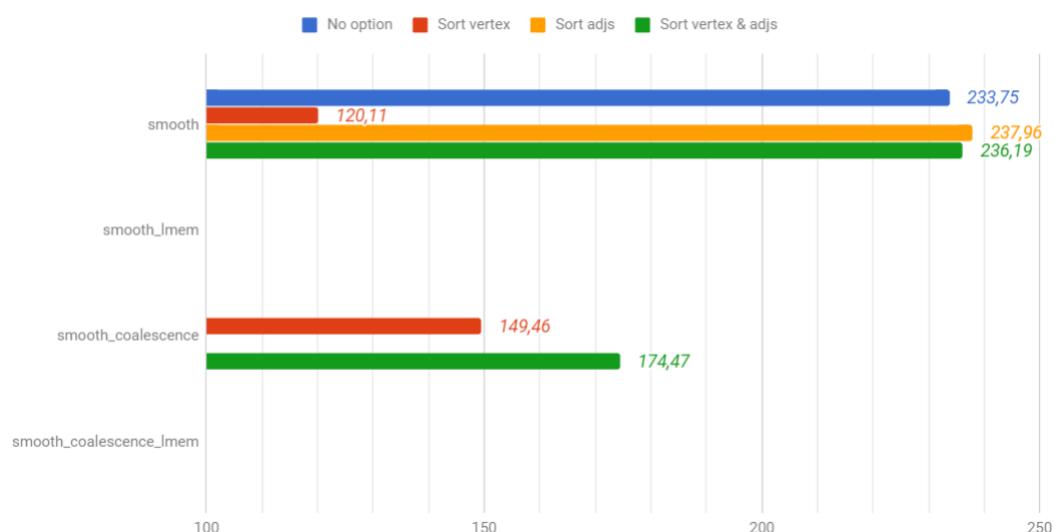
Mesh "Wrestlers" - SMOOTH PASS RUNTIME ms



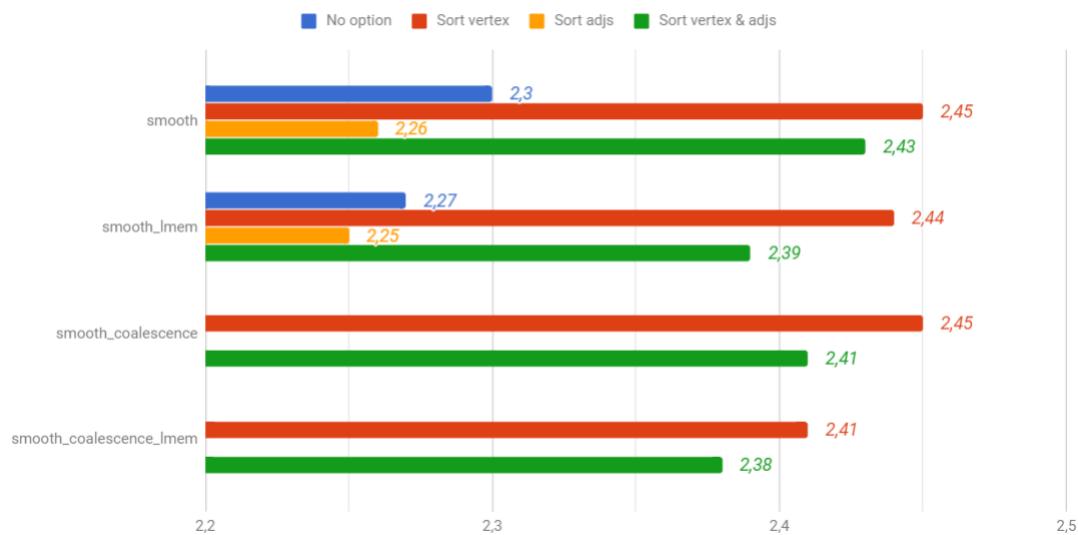
Mesh "Wrestlers" - SMOOTH PASSES/s



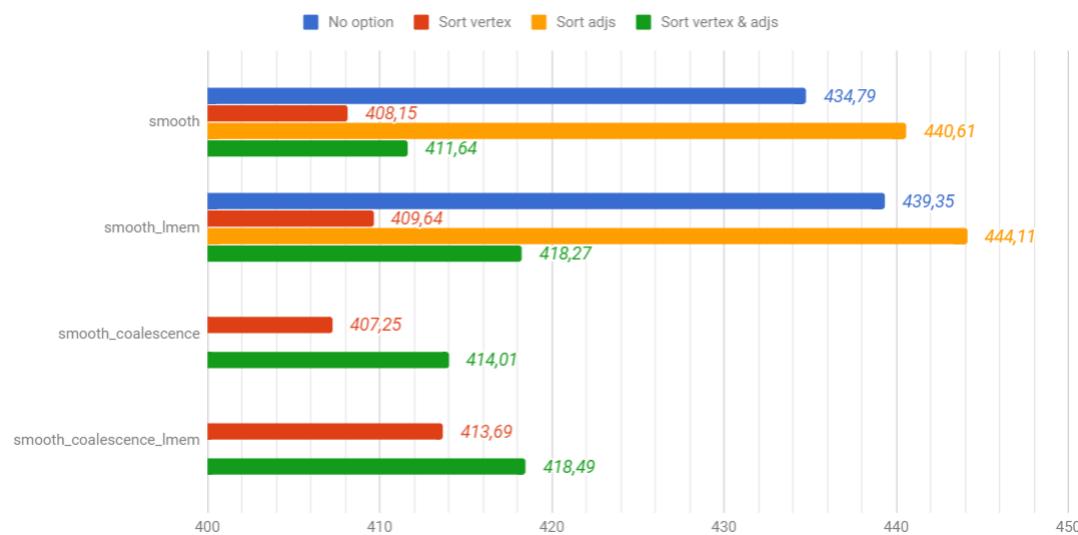
Mesh "Wrestlers" - BANDWIDTH GB/s



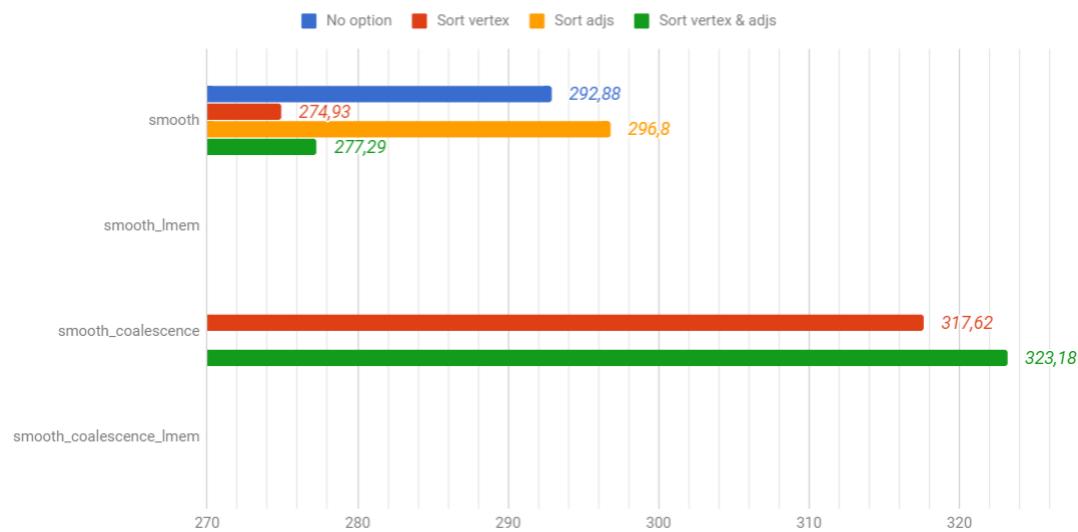
Mesh "Human" - SMOOTH PASS RUNTIME ms



Mesh "Human" - SMOOTH PASSES/s

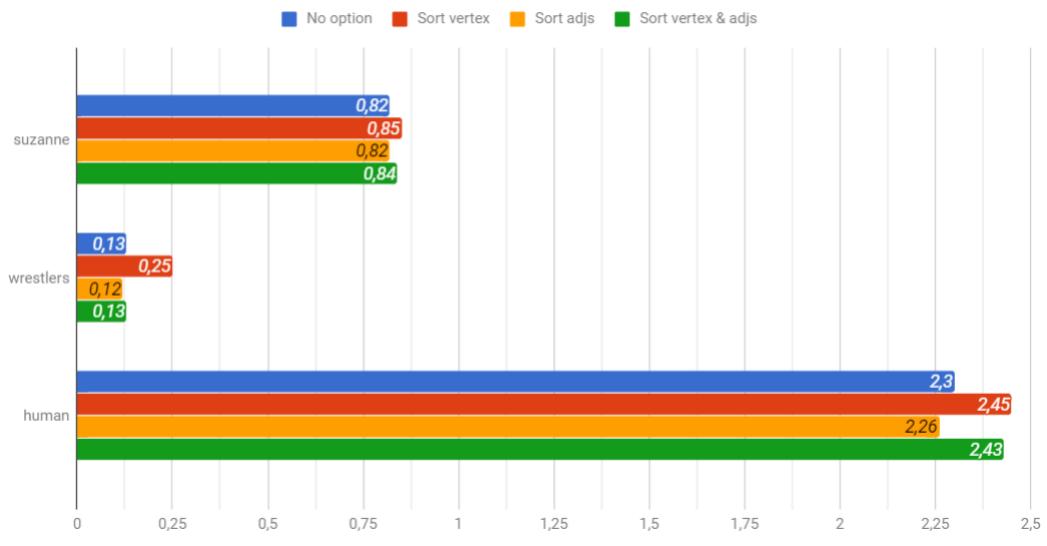


Mesh "Human" - BANDWIDTH GB/s

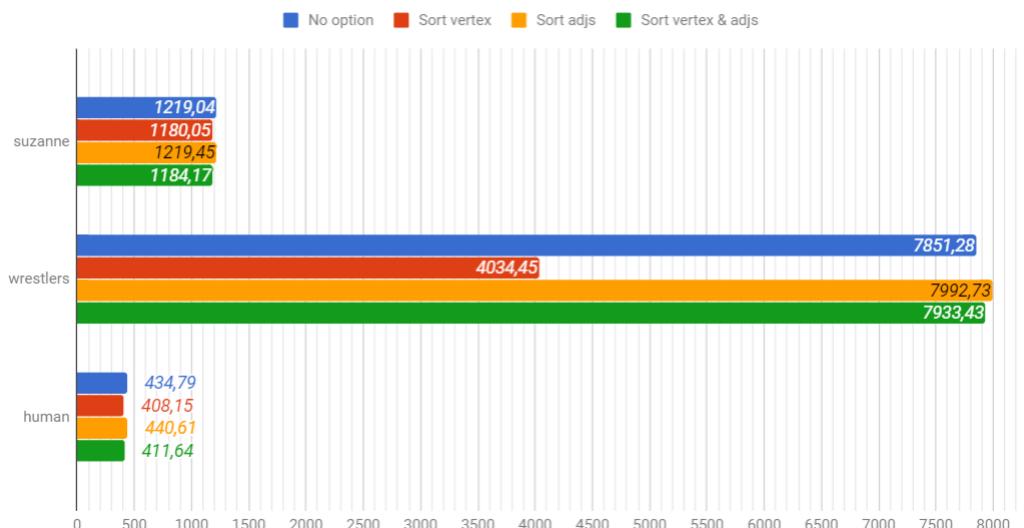


4.2 Kernels Profiling

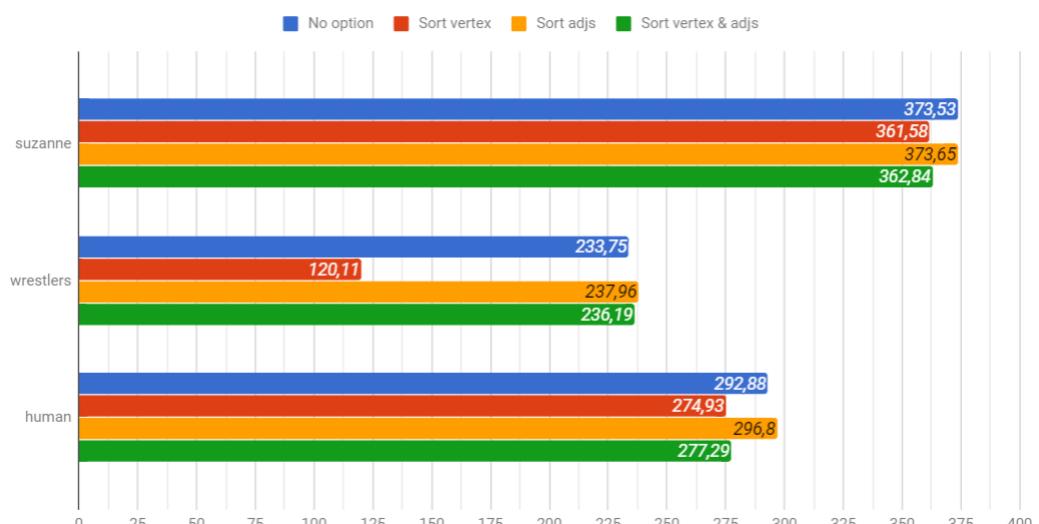
Kernel "smooth" - SMOOTH PASS RUNTIME ms



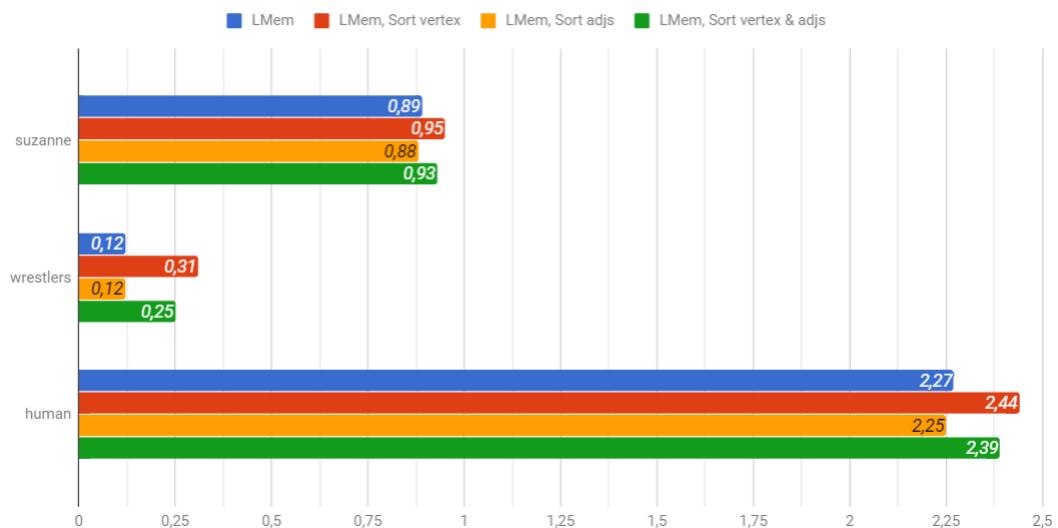
Kernel "smooth" - SMOOTH PASSES/s



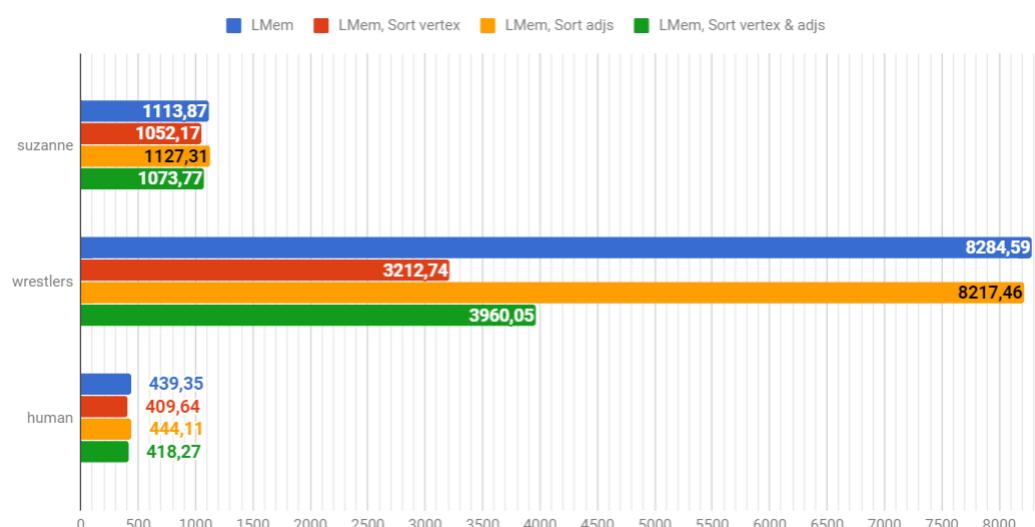
Kernel "smooth" - BANDWIDTH GB/s



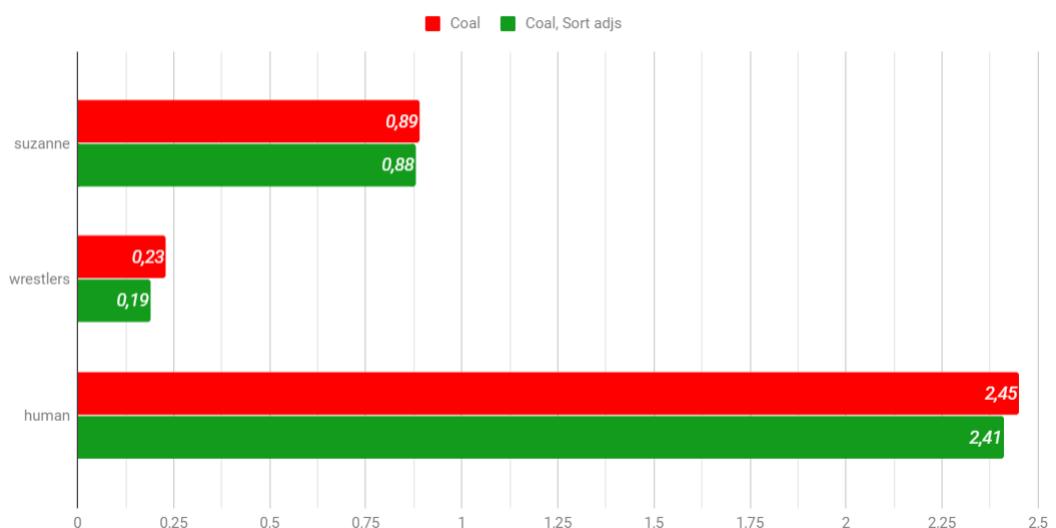
Kernel "smooth_lmem" - SMOOTH PASS RUNTIME ms



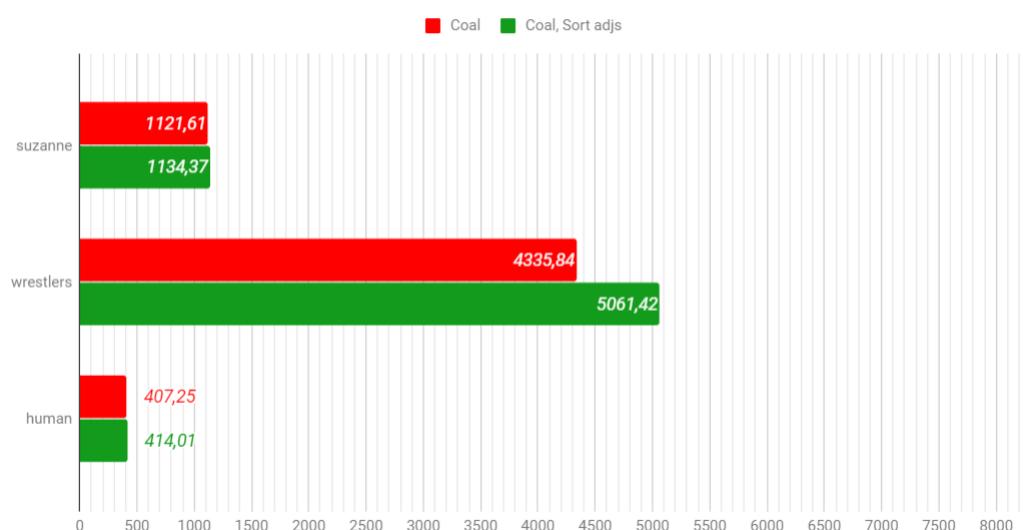
Kernel "smooth_lmem" - SMOOTH PASSES/s



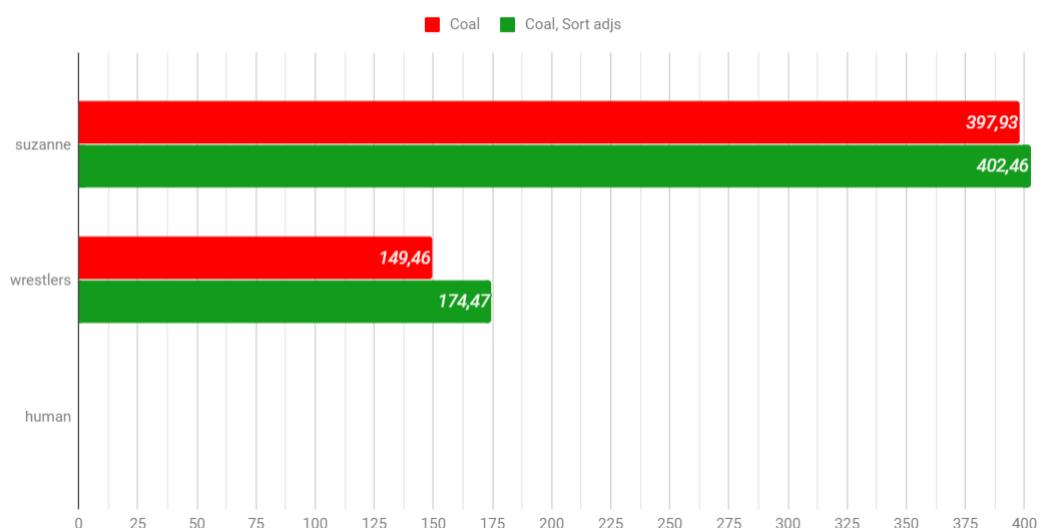
Kernel "smooth_coalescence" - SMOOTH PASS RUNTIME ms



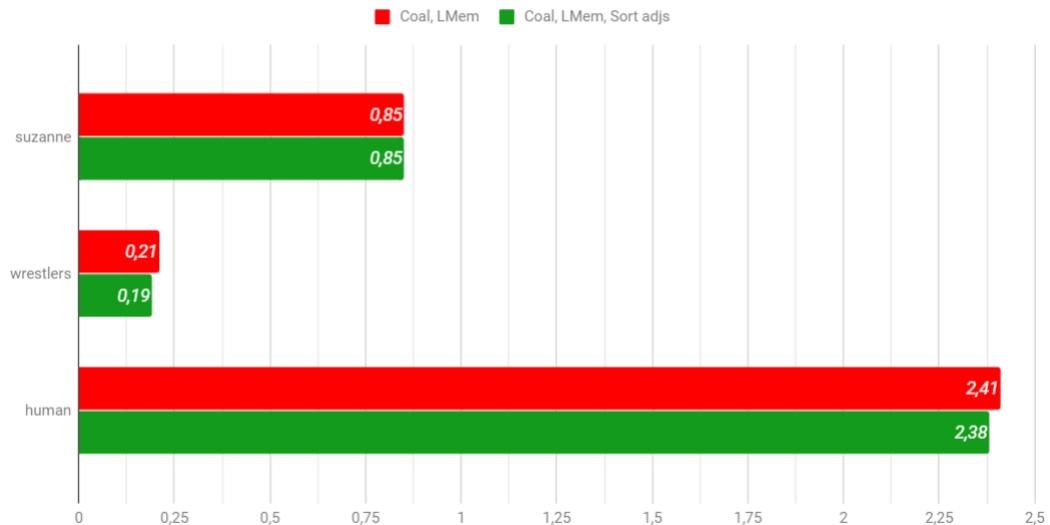
Kernel "smooth_coalescence" - SMOOTH PASSES/s



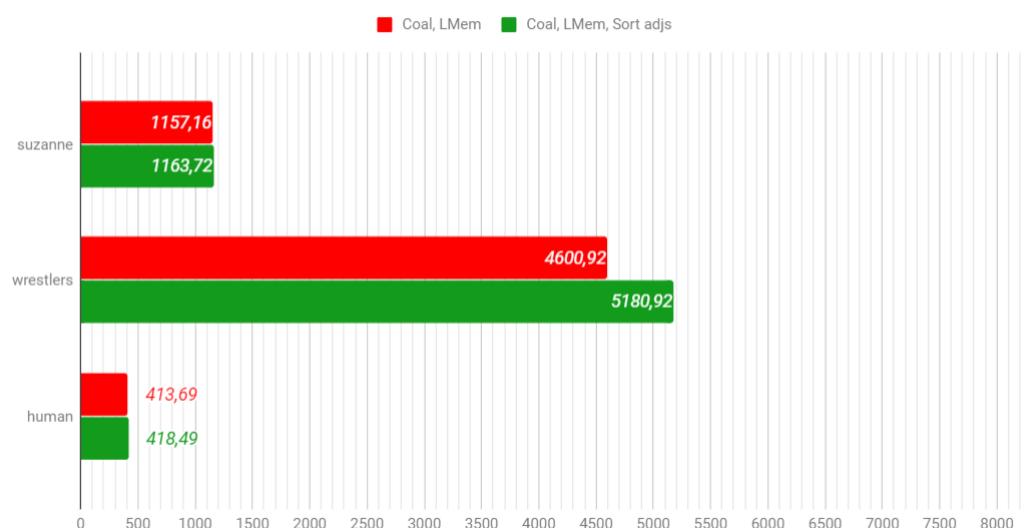
Kernel "smooth_coalescence" - BANDWIDTH GB/s



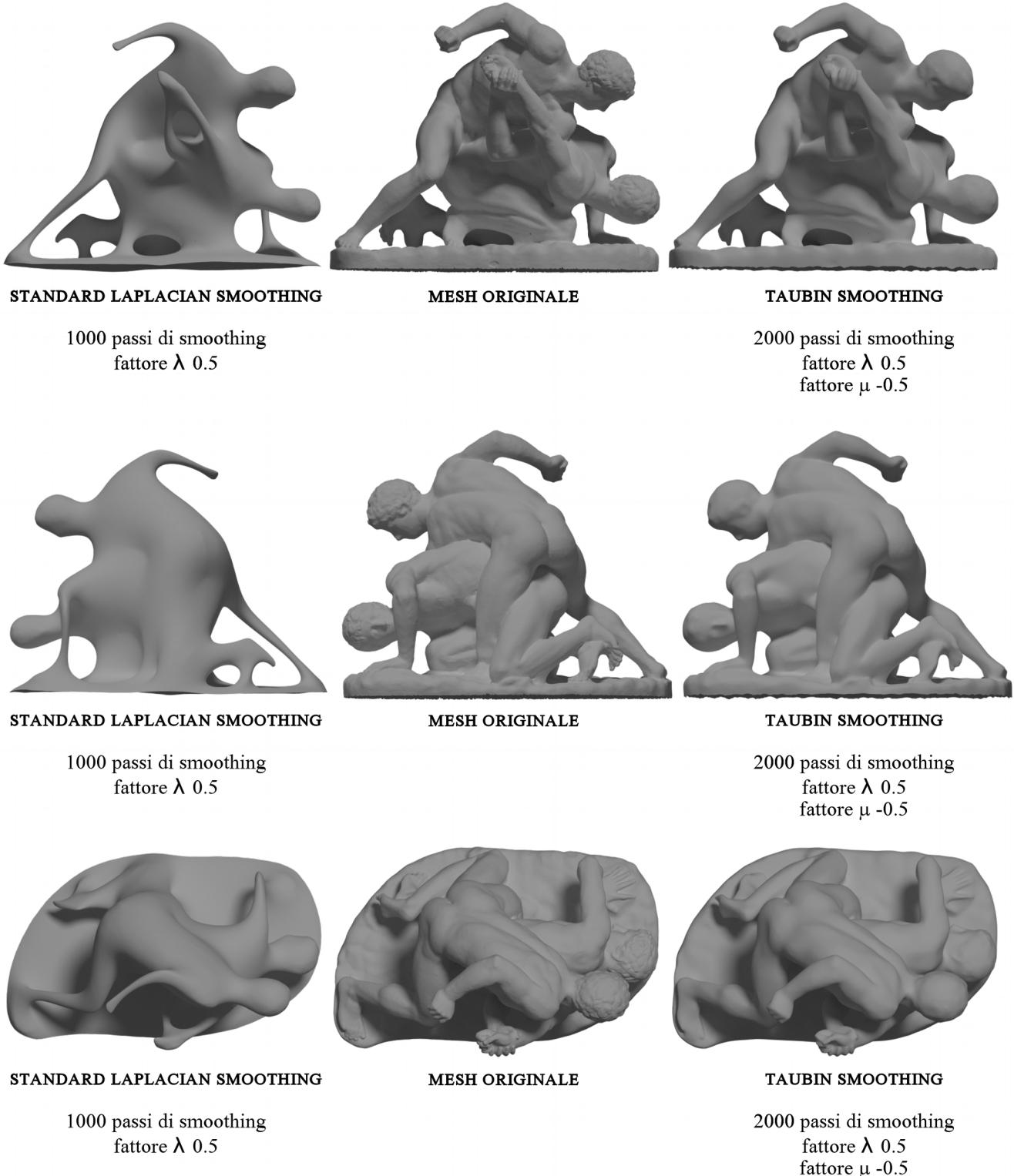
Kernel "smooth_coalescence_lmem" - SMOOTH PASS RUNTIME ms



Kernel "smooth_coalescence_lmem" - SMOOTH PASSES/s



Per brevità riportiamo i risultati di una sola mesh, ottenuti applicando l'algoritmo di smoothing implementato. Ciascuna riga mostra una diversa prospettiva della mesh "Two wrestlers in combat". La colonna centrale è la mesh originale prima dell'applicazione dello smoothing. La colonna sinistra mostra il risultato di 1000 passi di smoothing senza l'accorgimento introdotto da Taubin. Si nota infatti uno stiramento distruttivo della mesh. La colonna destra illustra invece il risultato dell'applicazione dello smoothing come descritto da Taubin: 2000 passi di smoothing di cui 1000 con fattore positivo e 1000 con fattore negativo. Si noti che molti dettagli sono stati conservati.



5. Conclusioni

Dai risultati dei test effettuati possiamo innanzitutto trarne che, come da iniziali aspettative, le performance di ciascun kernel dipendono fortemente dalla topologia della mesh che viene presa in input dal kernel di calcolo.

E' interessante notare che nonostante la banda passante dichiarata per la scheda Nvidia GTX 970 sia 224 GB/s, i nostri test riportano nella quasi totalità, una banda passante teorica superiore che, nel caso migliore, arriva quasi a raddoppiare la banda dichiarata.

La scelta di ordinare gli indici nel buffer “*cl_adjs_array*” tramite l’opzione “-o *sortAdjs*” risulta essere sempre fortemente consigliata in quanto ha avuto risultati migliori di qualsiasi altra opzione in tutti i kernel sviluppati, a fronte di un costo praticamente nullo in fase di pre-elaborazione. Il miglioramento, sembrerebbe quindi confermare le precedenti ipotesi riguardo l’accesso in lettura in memoria di un insieme ristretto di dati e l’aumento della possibilità di trovare in cache dati precedentemente letti.

L’opzione “-o *sortVertex*”, ovvero la scelta di ordinare i vertici in base al numero di adiacenti è invece fortemente sconsigliata, per motivazioni opposte alle precedenti, ordinare i vertici aumenta la percentuale di cache-miss. Ordinando i vertici, all’interno di ciascun work-group, si formano con molta probabilità insiemi di vertici che non hanno molta relazione tra loro.

Di conseguenza, i kernel “*smooth_coalescence*” e “*smooth_coalescence_Imem*”, che richiedono l’ordinamento dei vertici per numero di vertici adiacenti, inglobano tutti gli aspetti negativi appena discussi.

L’utilizzo della local memory ha invece portato sempre benefici se utilizzato in combinazione con gli accessi in coalescenza (riduzione fino a 8%), ma non abbastanza da migliorare i risultati rispetto al kernel “*smooth*” che non fa uso di memoria locale e accesso in coalescenza.

In alcuni casi isolati, la local memory ha anche migliorato le performance della versione principale (riduzione fino a 2%). Grazie al kernel “*smooth_Imem*” possiamo inoltre affermare con più certezza che l’ordinamento dei vertici per numero di adiacenti, è affetto dal problema della perdita di “località” tra vertici vicini tra loro, infatti, la versione principale che fa uso della local memory con l’opzione “-o *sortVertex*”, molto spesso non trae alcun vantaggio dall’uso della memoria locale dando come risultato delle pessime performance.

In conclusione possiamo quindi affermare che il kernel che nella maggior parte dei casi ha avuto le performance migliori è il kernel “*smooth*” con l’aggiunta dell’opzione “-o *sortAdjs*” per il vantaggio di poter sfruttare al massimo la relazione tra tutti i dati all’interno di ogni work-group.