UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA

ELABORAZIONE DATI SU GRAPHIC PROCESSING UNIT

RELATORE: Ch.mo Prof. Gianfranco Bilardi

LAUREANDO: *Francesco De Cassai*

Padova, 9 marzo 2010

A mio fratello Alessandro

Definizione di Lord Kelvin

Si dice *matematico* colui per il quale

è ovvio che $\int_{-\infty}^{+\infty} e^{-x^2}\,dx = \sqrt{\pi}$.

Corollario

Si dice *informatico* colui per il quale

il suddetto integrale è banale

Indice

Sommario

L'elaborato tratta l'utilizzo dei processori delle schede video (*GPU*) come elaboratori paralleli. Verranno trattate le varie tecniche e le indicazioni delle varie famiglie di linguaggi, trattate secondo il loro ordine cronologico.

Considerata la vastità dell'argomento si è ritenuto opportuno limitare le nozioni base e dare per assodato che il lettore sia pratico dei concetti del calcolo parallelo tradizionale. L'elaborato si propone quindi come un'introduzione completa e sufficientemente variegata per un'utente che voglia approcciare il calcolo su GPGPU (*General Purpose Graphic Processing Unit*).

Introduzione

Nel corso degli anni il parallelismo si è presentato nell'ambito informatico in maniera sempre più prevalente. Le ultime generazioni di CPU hanno sempre più puntato sulla presenza di più processori cooperanti in sincrono piuttosto che sulle prestazioni del singolo.

Questo trend è iniziato per una serie di motivi economico-tecnologici. L'andamento del rapporto costo/area del processore segue un andamento quadratico, rendendo sempre più difficoltoso l'aumento di potenza.

Le schede video contengono al loro interno un processore, denominato GPU (*Graphics Processing Unit)* che si è evoluto durante gli anni secondo le specifiche esigenze dell'elaborazione grafica.

Le applicazioni video seguono dei requisiti totalmente differenti rispetto alla programmazione ordinaria e hanno le seguenti caratteristiche

1. **Grande Carico Computazionale** il render real time richiede miliardi di pixel al secondo e ogni pixel può richiedere migliaia di operazioni a sua volta. Le GPU devono elaborare, in maniera continuativa, un'enormità di dati per soddisfare i requisiti per cui sono state progettate;

2. **Forte presenza del parallelismo** il render video è per sua natura parallelo. Ogni pixel di un dato frame può esser calcolato senza considerare i pixel del medesimo frame;

3. **Il Throughput è più importante della Latenza** poiché non ci sono vincoli tra i frame, ma è importante che i dati giungano elaborati con frequenza costante; le GPU non si pongono il problema del tempo di elaborazione che passa tra l'arrivo dei dati e quello di uscita. Questo perchè l'apparato visivo umano ha una sensibilità del millesimo di secondo, mentre un ciclo di elaborazione del processore elabora al nano secondo, ovvero 6 ordini di grandezza inferiore. Ne segue che non è importante se il singolo dato viene reso disponibile in un tempo di anche un paio di ordini di grandezza superiore rispetto all'elaborazione di una CPU.

Fino al 2000 le GPU erano composte da una pipeline grafica che eccelleva nelle elaborazioni tridimensionali ma non era adatta a nient'altro. Dopo il 2000 si è iniziato a rendere le GPU programmabili e a dotarle di API.

Il risultato di questo processo è un sistema con enormi potenzialità di calcolo (una NVIDIA GeForce 8800 GTX ha una potenza di calcolo di circa 330 Gigaflops al secondo in virgola mobile) e dei limiti di banda molto superiori alle CPU equivalenti (80 GB/s)

Nel primo capitolo discuteremo dell'organizzazione hardware storica, ovvero dell'utilizzo storico delle GPU. Il secondo capitolo proporrà il modello a stream, utilizzato nelle schede video. Nel terzo capitolo si illustreranno gli *shading language*, ovvero i linguaggi che si sono sviluppati come tool grafici e sono stati riutilizzati per elaborazioni non grafiche. Nel quarto analizzeremo alla fine il linguaggio di programmazione GPGPU attualmente più diffuso, ovvero il CUDA con il supporto di alcuni algoritmi.

Capitolo 1

Hardware delle GPU

L'obiettivo delle schede video 3D è quello di creare immagini realistiche con un alto numero di frame secondo (60 per le applicazioni realtime). Queste scene contengono delle primitive geometriche che devono esser elaborate al fine di simulare la posizione dell'osservatore, delle luci, delle ombre e dei riflessi.

Tradizionalmente le GPU sono state create come processori per la sintesi di immagini caratterizzate da una pipeline a stage molto specializzate. In questo capitolo l'obiettivo è quello di illustrare la sequenza di operazioni nel loro funzionamento di massima e nelle loro iterazioni, soffermandosi su alcuni algoritmi presenti in letteratura. Successivamente di mostrerà come questa teoria viene applicata ad una scheda video moderna.

1.1 Introduzione - Componenti Scheda Video

Nelle schede video moderne la GPU comunica con la CPU tramite uno slot AGP o PCI Express. Per evitare colli di bottiglia nella trasmissione dei dati anche questi slot si sono andati evolvendo, passando dai 264 Mb/sec della prima versione AGP agli oltre 4 GB/sec dello standard PCI.

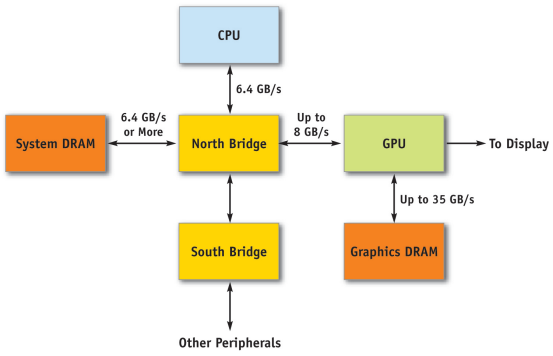

Figura 1.1: Modello a blocchi di un elaboratore(14)

Seguendo le evoluzioni del mercato si è andato a formare quindi un componente con una banda molto elevata (una Nvidia 6600 ha una banda all'interno della GPU di circa 35 GB/sec, contro 6,4 GB/sec dei processori equivalenti)(14). Nella figura 1.1 si mostra una modellizzazione della struttura di un computer.

Possiamo notare che c'è un divario sostanziale di banda tra i vari componenti. Questo ha portato lo sviluppo di alcune tecniche per ottimizzare la trasmissione dei dati. Un primo approccio è quello di cercare l'utilizzo di algoritmi in-place, ovvero che non abbiano bisogno di accedere alla memoria esterna o secondaria, limitando il numero di trasferimenti. Esistono poi dei meccanismi di caching (nelle CPU ci sono le cache di primo e di secondo livello, rispettivamente L1 e L2, mentre nella GPU esiste solamente la cache di primo livello), che permettono di riutilizzare dati appena elaborati, sfruttando il principio di località. In combinazione con i primi due si sono sviluppati invii di dati compressi, con algoritmi di compressione on-fly.

L'elaborazione è per sua natura parallela, dato che gli n pixel appartenenti

ad un ciclo di refresh devono esser visualizzati all'unisono. In questo capitolo verranno presentate le componenti di una GPU, escludendone la componente programmabile.

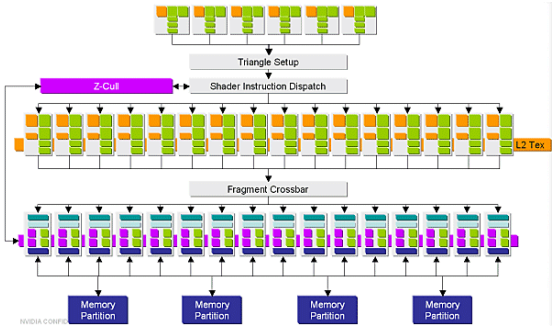

Figura 1.2: Modello a blocchi della Geforce 6800

La figura 1.2 rappresenta lo schema a blocchi di una moderna scheda video.

1.2 Graphic Pipeline

Il dominio delle applicazioni della grafica 3D ha diverse peculiarità che la differenziano dal domini tradizionali. In particolare la grafica iterativa 3D richiede un *rate* computazionale molto elevato e manifesta un sostanziale paralellismo, dovuto all'indipendenza tra frame successivi.

Modellando l'*hardware* al fine di trarre vantaggio dal parallellismo nativo permette un notevole incremento delle prestazioni.

Il modello attivo per le schede video è chiamato **pipeline grafica** (*graphic pipeline*), oppure *render pipeline*. Questo modello è strutturato in modo da

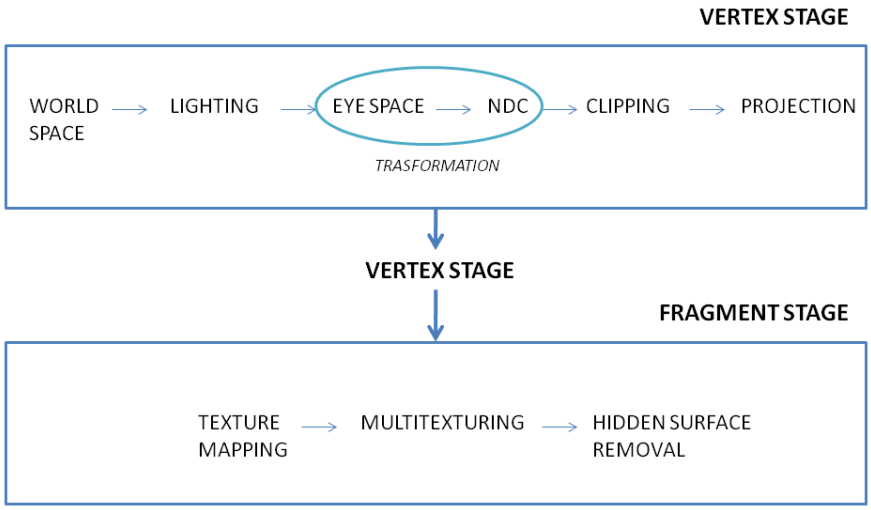

Figura 1.3: I passaggi della pipeline

permettere un elevato *rate* computazionale in base ad esecuzioni parallele.

Esistono altri paradigmi e modelli con cui è possibile effettuare l'elaborazione, tra cui il più noto è il *ray-tracing*. Questi paradigmi non sono attualmente implementati in soluzioni commerciali e quindi non verranno prese in esame. É possibile, tuttavia, implementare questi approcci nella render pipeline

I dati all'inizio della pipeline rappresentano dei punti (vertici) tridimensionali e devono esser trasformati come proiezioni sullo schermo bidimensionale. Tutti i dati devono esser elaborati dei seguenti passaggi:

1. **vertex operation**: ogni primitiva è composta da una serie di vertici. Questi vertici devono esser modellati nello spazio tridimensionale e successiva-

mente elaborati per il calcolo delle ombre (*shading*).

Poiché una scena tipica ha centinaia di migliaia di vertici, che andranno elaborati in parallelo, questo stage si presta molto bene per le elaborazioni parallele. Nello specifico le operazioni sui vertici sono composte da:

- **Modifica del sistema di riferimento in coordinate Globali**: solitamente gli oggetti passati hanno le coordinate espresse in più sistemi di riferimento locali.
- **Calcolo dell'illuminazione sui vertici**
- **Modifica del punto d'osservazione**: la scena viene ruotata per allinearsi con l'osservatore.
- **Applicazione dell'effetto prospettico**
- **Clipping**: se degli oggetti risultano fuori dal cono visivo devono esser eliminati o tagliati per evitare l'elaborazione delle componenti non visualizzabili.
- **3D Screen Space** le coordinate vengono portate in NDC (Normalized Device Coordinate)

2. **Primitive assembly**: i vertici vengono raccolti in triangoli; il triangolo è la primitiva di riferimento per la semplicità di alcuni algoritmi.

3. **Rasterization**: una volta creati i triangoli la GPU li elabora e determina quale area dello schermo sarà coperta dai vari triangoli.

   Il rasterizer compie i seguenti passaggi:

   - Ogni triangolo genera una o più primitive chiamate frammenti (*fragment*). I frammenti sono segmenti (in realtà rettangoli) di larghezza unitaria.

- **Proiezione sul piano immagine**: ogni oggetto, che è intrisicamente tridimensionale viene proiettato sul piano immagine (lo schermo)

Poiché molti frammenti si andranno a sovvrapporre ne risulta che ci saranno, se non controllati, diversi pixel calcolati più volte. La gestione di questo overhead verrà trattata nel capitoli successivi.

4. **Fragment Operations**: elaborando i colori dei vertici e attingendo ad eventuali dati presenti nella memoria globale (solitamente nella forma di texture, ovvero di immagini che varranno mappate sulle superfici) ogni frammento viene ombreggianto, al fine di calcolarne il colore finale. Ovviamente ogni frammento può esser elaborato in parallelo. Questo è il passaggio che solitamente richiede il maggior numero di elaborazioni.

5. **Composizione dell'immagine finale**: viene eseguita una proiezione dell'insieme dei frammenti sul piano immagine dell'osservatore, assegnando, per esempio, ad ogni pixel il valore del frammento ad esso più vicino e gestendo i conflitti che potrebbero insorgere (frammenti equidistanti, oggetti che collidono...)

Nello specifico le operazioni sopra elencate si possono vedere riassunte in tre stage:

1. **vertex stage**: esegue ortotrasformazioni sui vertici e ne calcola il colore e la proiezione sullo schermo

2. **rasterizer stage**: calcola l'area di ogni triangolo passato ad input e ne effettua il fill-in con i frammenti

3. **fragment stage**: elabora i frammenti, calcolandone le ombre e interpolando i colori dei vertici elaborati nella *vertex stage*

Al livello fisico tutti i passaggi sono elementi hardware separati sul processore. Questo permette che, nello stesso istante, ci siano le varie stage che lavorino su subset di dati differenti.

1.3 Vertex Stage

Prima delle stage della pipeline, la Vertex Stage riceve dalla CPU i comandi, le texture e i vertici da elaborare. Ogni vertice ha come parametri le sue coordinate, la sua normale, e la posizione all'interno della texture. Queste informazioni possono risiedere in buffer condivisi (nella RAM) o in buffer locali (come i banchi DRAM della scheda).

I Vertex Processor eseguono le seguenti operazioni tramite i Vertex Shader (il programma adibito, o nelle nuove schede video le subroutine)

1. acquisizione dei dati
2. scrittura dei dati nella memoria
3. elaborazione
4. scatter dei dati

I dati in ingresso sono solitamente espressi in coordinate locali per i singoli oggetti. La vertex stage li deve quindi convertire in un sistema di riferimento comune. Per esser sicuri che i triangoli non vengano distorti o deformati, le trasformazioni sono limitate a semplici trasformazioni affini come rotazioni, traslazioni e ridimensionamenti. L'utilizzo delle coordinate omogenee permette di effettuare tutta la catena di trasformazioni con una singola moltiplicazione di matrice.

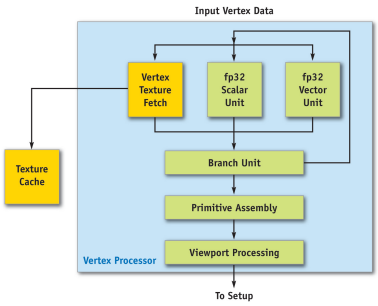

Figura 1.4: Lo schema della Vertex Stage nVidia(14)

L'output finale di questa stage è una serie di triangoli, espressi in coordinate omogenee rispetto il medesimo sistema di riferimento, con l'osservatore posto sull'asse z

La *vertex stage* passerà i dati al *rasterizer*, il cui compito è quello di interporarli; Poichè i dati arrivano indipendentemente gli uni dagli altri l'unica funzionalità richiesta è quella di scatter, e non di gather, dato che le letture avvengono all'esterno. Le GeForce 6 Series hanno introdotto la possibilità di gather, chiamandola vertex texture fetch (VTF).

Questa *stage* accetta in imput un numero limitato di quaterne di vettori *floating-point* a 32 bit. Tali vettori rappresentano le coordinate x, y, z, w in spazio proiettato e i 4 colori RGBA (red, green, blu, alpha) dove alpha è l'opacità.

1.3.1 Trasformazione in World Space

Le coordinate dei vertici sono espresse nel sistema di riferimento dell'oggetto (*object space*). Per poter elaborare coerentemente la scena occorre portare tutte le coordinate nel sistema di riferimento globale (*world space*).

I dati di input sono le coordinate locali A e il sistema di riferimento V. La scheda grafica invece ha per parametro il sistema di riferimento U. Al termine di

questa elaborazione si dovrà disporre delle coordinate nel sistema di riferimento globale B.

In generale un sistema di riferimento in $\Re^3$ è definito da un punto (origine) p_0 e da una base vettoriale v_0, v_1, v_2. Se la base non è degenere si può quindi esprimere ogni vettore come combinazione lineare di v_i. Possiamo esprimere il punto p nei vari sistemi di riferimento come

$$p = \begin{bmatrix} a_0 & a_1 & a_2 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ 0 \end{bmatrix} = \begin{bmatrix} b_0 & b_1 & b_2 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ 0 \end{bmatrix}$$

per passare da un sistema all'altro bisogna calcolare la matrice di cambio base $\gamma = AVU^T$ ed eseguire la moltiplicazione

$$B = \gamma \cdot A$$

1.3.2 Lighting

Lo stadio di lighting ha come scopo la determinazione del modello di illuminazione, ovvero la computazione dell'equazione del trasporto dell'energia luminosa (equazione di illuminazione). Ci sono due passaggi in questo passaggio: il Lighting, ovvero il calcolo del bilancio luminoso e lo Shading. Con lo Shading si calcola il colore di ogni pixel dell'immagine.

Modello di Phong

Trattiamo ora il modello di Phong, sebbene proposto nel 1973 il modello di Phong è attualmente il modello più diffuso, considerando anche le sue derivazioni, per l'ottimo rapporto tra risultato e complessità computazione.

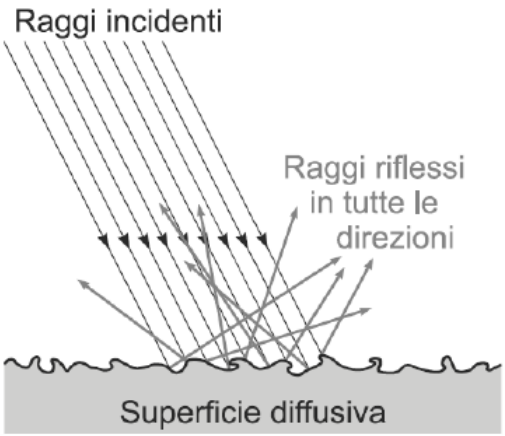

Figura 1.5: Una superficie lambertiana, modello valido per gli oggetti opachi

Il modello di Phong è un modello di riflessione delle luci, in esso si considera la luce come formata da una serie di contributi.

$$\text{luce} = \text{ambiente} + \text{emissione} + \text{riflessione}$$

ambiente: è una costante additiva applicata senza controlli. Si assume che un po' di luce arrivi ovunque nella scena. É a sua volta suddiviso in due parametri, *luce ambientale dell'ambiente* e *luce ambientale del materiale*, non necessariamente uguali tra di loro.

emissione: nel caso si voglia renderizzare un oggetto che emette luce si interviene su questo fattore, visto come una costante addittiva che modifica solamente l'oggetto in questione, senza modificare i vicini

riflessione: a seconda del materiale di cui è composto l'oggetto abbiamo due casi:

- *riflessione lambertiana o diffusa*: utilizzata per oggetti opachi. Materiali molto opachi (es. gesso e legno) hanno una superficie che, a livello microsco-

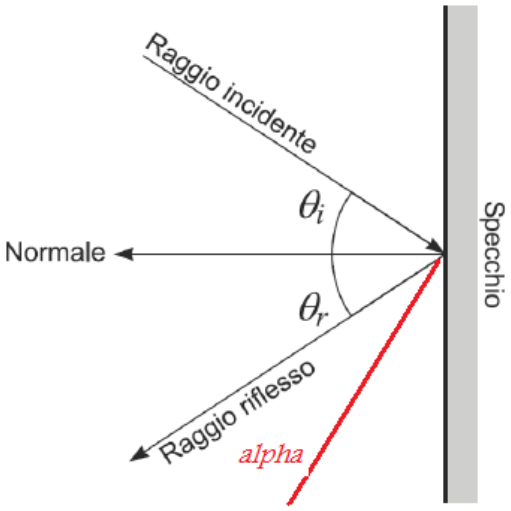

Figura 1.6: La legge di Fresnel

pico, ha piccole sfaccettature che riflettono la luce in una direzione casuale. L'intensità si può quindi modelizzare come

$$I_{diff} \begin{cases} I_{luce} \cdot K_{materiale} \cdot \cos\theta & \text{se } 0 \leq \theta \leq \frac{\pi}{2}, \\ 0 & \text{altrimenti.} \end{cases}$$

- *riflessione speculare*: utilizzabile per materiali lucidi o lisci, segue la **Legge di Fresnel**, secondo cui un raggio incidente viene riflesso con il medesimo angolo. Questo comporta che una sorgente luminosa viene riflessa con intensità differente secondo il punto di osservazione.

$$I_{rifl} = I_{luce} \cdot K_{materiale} \cdot \cos\alpha$$

dove α è l'angolo tra il raggio riflesso e l'osservatore. Esiste una modifica alla formula atta ad avere riflessi di area minore e più accesi ovvero

$$I_{rifl} = I_{luce} \cdot K_{materiale} \cdot \cos^n\alpha$$

Così facendo sia $K_{materiale}$ sia n diventano parametri di ogni singolo materiale, o più genericamente, di ogni oggetto.

Modello di Blinn-Phong

É una modifica del modello di Phong, atta a ridurre i calcoli. Dati i vettori della figura 1.7 si introduce il vettore H, detto *half-way vector* definito come

$$H = \frac{L + V}{|L + V|}$$

questo permette di calcolare l'intensità come

$$I_{diff} = I_{luce} \cdot K_{materiale} \cdot (H \cdot N)^n$$

ottenendo un risultato molto simile al modello originale.

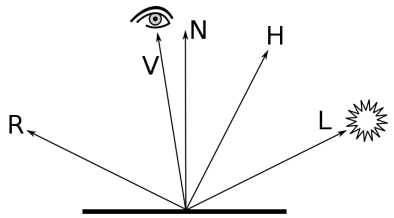

Figura 1.7: Lo schema dei Vettori

1.3.3 Shading

Lo scopo dello shading è quello di individuare le zone dove calcolare l'illuminazione, tenendo presente che ci sono dei vincoli temporali e dei framerate da rispettare.

Shading Costante

Per ogni faccia (triangolo) ne calcolo la normale ed elaboro la I_{luce} secondo un modello d'illuminazione una sola volta per ogni faccia. Così facendo ottengo dei colori piatti per ogni faccia.

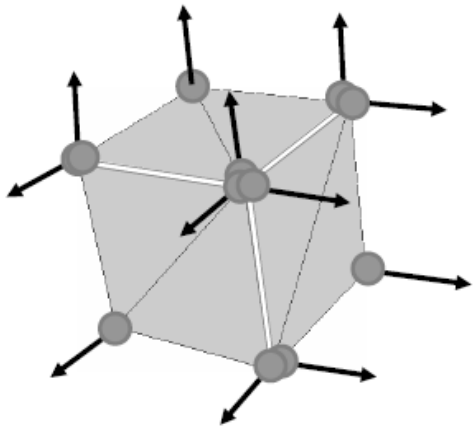

Figura 1.8: Un cubo soggetto allo shading di Gourard

Gouraud Shading

Sfruttando la linearità dello spazio RGB si calcola l'illuminazione solo in certi punti specifici (i vertici) e poi si procede per interpolazione. Nelle schede video l'interpolazione viene eseguita dal processo di rasterizzazione (vedi dopo) e quindi il calcolo delle zone interne, e degli spigoli, viene eseguito in un secondo momento.

Questa tecnica ha dei risultati buoni quando si applica l'algritmo di Phong. I vertici vengono considerati avente una normale pari alla media delle normali delle faccie che insistono su di essi. Ovvero

$$N_v = \frac{\sum_i N_i}{|\sum_i N_i|}$$

Per gli spigoli vivi invece si deve aver l'accortezza di spezzare lo spigolo nelle facce e memorizzarne normali diverse. Questo algoritmo fornisce ottimi risultati nel caso di un basso tasso di specularità. In tal caso difatti le luci tendono ad estendersi eccessivamente per interpolazione, mentre se presenti sugli spigoli si perdono subito.

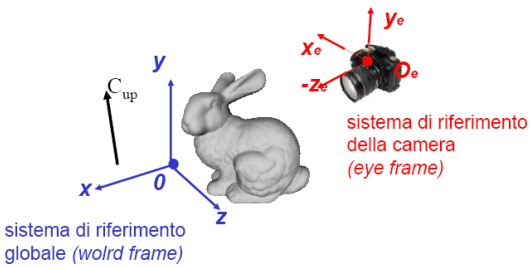

Figura 1.9: I vettori per il cambio di coordinate

Phong Shading

É particolarmente adatto per superfici speculari. Questo perchè si interpolano le normali della superficie e si calcola ogni pixel per la normale ad esso più vicina.

1.3.4 Trasformazioni

Dopo avere calcolato il lighting ed eventualmente lo shading dei vertici si effettuano delle operazioni sulle coordinate. Tali operazioni vengono eseguite da hardware specializzato sulla scheda.

Eye space

Con la trasformazione in *eye space* ci si porta nel punto di vista dell'osservatore.

Il ragionamento è molto simile a quello per la trasformazione in *world space*. Prendiamo in riferimento la figura 1.9 per le nomenclature dei vettori. In questo caso abbiamo come input la posizione della camera C_{pos}, la direzione di vista C_{dir} e il vettore di alto C_{up}. Ponendo come origine del nuovo sistema C_{pos} possiamo

definire una base ortogonale mediante dei prodotti esterni su C_{up} e C_{dir}. Nello specifico si calcola la seguente matrice: [1]

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ p \end{bmatrix} = \begin{bmatrix} C_{dir} \times C_{up} & 0 \\ (C_{dir} \times C_{up}) \times C_{dir} & 0 \\ -C_{dir} & 0 \\ C_{pos} & \end{bmatrix}$$

NDC

Poiché le dimensioni degli schermi e le risoluzioni dell'hardware sono diversi ed eterogenei si portano le coordinate in un formato intermedio, esprimendo le coordinate dei singoli punti come comprese tra [-1,-1] e [1,1]. Tale formato è detto Normalized Device Coordinate.

Dato uno schermo di dimensione x_{world} per y_{world} possiamo mappare il tutto con una proporzione, ottenendo alla fine x_{ndc}, y_{ndc}

$$x_{ndc} = \frac{2}{x_{world}} - 1$$
$$y_{ndc} = \frac{2}{y_{world}} - 1$$

esprimibile come moltiplicazione matriciale dato che la mappatura è lineare

$$\begin{bmatrix} x_{ndc} \\ y_{ndc} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{x_{world}} & 0 & -1 \\ 0 & \frac{2}{y_{world}} & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{world} \\ y_{world} \\ 1 \end{bmatrix}$$

Questa trasformazione viene effettuata da hardware specializzato in operazioni a virgola mobile. La trasformazione da NDC a schermo invece sarà a virgola fissa.

[1] utilizzando un sistema destrorso

1.3.5 Clipping

Una volta impostato il sistema con la correzione prospettica si può effettuare le operazioni di **clipping**.

Si definisce *clipping* le operazioni tramite le quali si modificano gli oggetti in modo da eliminare tutti gli elementi non visibili, perchè esterni al cono visivo dell'osservatore (*viewing frustrum*). Questi elementi possono esser scartati ad esempio perchè esterni al piano immagine proiettato, perchè troppo vicini o lontani o perchè non soddisfano altri vincoli di taglio (*clipping constraint*).

Per motivi di performance vengono eseguite anche alcune operazioni di **culling**, che servono ad ottimizzare il carico di lavoro. Per culling si intende quella famiglia di operazioni che eliminano o marcano come inutili, nella loro interezza, delle primitive. In letteratura si trova anche come Occlusion Culling o Visible Surface Determination. Nello schema a blocchi 1.3 queste operazioni sarebbero incluse nell'ultimo blocco, *visibility*.

Nella figura 1.10 si vede una scena in due angolazioni. L'oggetto giallo e rosso sono soggetti rispettivamente a **view-frustrum culling**, il rosso a **occlusion culling**, ovvero non verranno elaborati perché fuori dal campo visivo e perché nascosti da un altro oggetto. La sfera verde, invece, è soggetta sia a **occlusion clipping** sia **view frustrum clipping**.

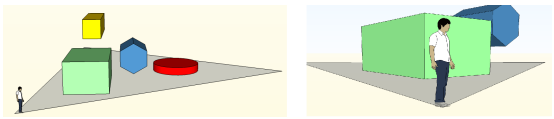

Figura 1.10: Una scena proposta in due diverse angolazioni. Notiamo come l'elemento giallo sia esterno al punto di vista, il rosso sia occluso completamente e il rosso parzialmente dal verde

1.3.6 Algoritmi di Clipping

Analizziamo ora gli algoritmi di clipping. Ne presenteremo due di clipping di segmenti (Algoritmo di Cohen-Sutherland, Algoritmo di Liang-Barsky) e uno di clipping per poligoni (Algoritmo di Sutherland-Hodgman). Nelle schede video d'interesse comunque si agisce solitamente tramite clipping di segmenti.

Algoritmo Banale

Un segmento è interno se i due estremi sono interni al rettangolo di riferimento (lo schermo). Se i due estremi sono uno interno e uno esterno il segmento interseche rà il rettangolo e quindi sarà oggetto di clipping. Se entrambi gli estremi sono esterni al rettangolo, il segmento può intersecare o meno il rettangolo di clipping e si rende necessaria una analisi più accurata per individuare le eventuali parti interne del segmento.

Per far questo si deve calcolare l'intersezione tra il segmento e i 4 lati del rettangolo. Questo approccio è chiaramente poco efficiente.

Algoritmo di Cohen-Sutherland

Si codifica lo spazio in 9 quadranti, ponendo quello interno coincidente con lo schermo. Per fare questo si utilizza un codice binario a 4 bit, denominato *outcode*.

Il primo bit è posto a 1 per le regioni superiori al bordo superiore, il secondo per quelle inferiori al fondo. Il terzo e quarto indicano, rispettivamente, i bordi destro e sinistro, come da figura 1.11.

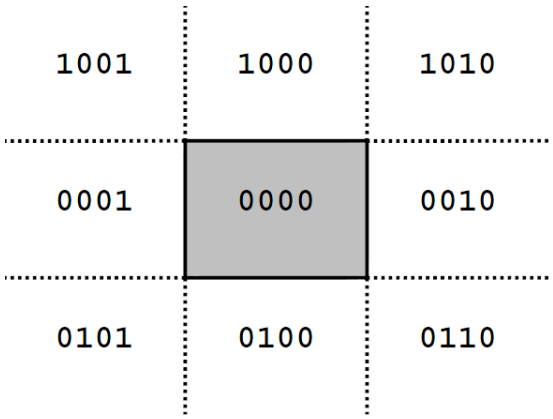

Figura 1.11: Le codifiche per Cohen-Sutherland

Si calcolano quindi gli outcode degli estremi. Se il codice di entrambi gli estremi è 0000 (confronto eseguibile tramite un OR), allora si può dedurre che il segmento è un **segmento interno** al rettangolo di clipping.

Se l'operazione di AND logico tra i codici degli estremi restituisce diverso da 0000 allora il segmento è **totalmente esterno** al rettangolo di clipping. In questo

caso, infatti, gli estremi giacciono in uno stesso semipiano (quello identificato dal bit a 1 del risultato) e quindi il segmento non interseca il rettangolo di clipping.

Se il risultato dell'AND è nullo invece si ha un **segmento che interseca il rettangolo**. Per calcolare il sottosegmento interno si seguono i seguenti passaggi

1. Si identifica il primo bit discordante tra i due outcode
2. Si individua l'intersezione tra il segmento ed la retta relativa
3. L'estremo con bit pari a 1 viene sostituito con l'intersezione (essendo pari a uno vuol dire che è esterno a quella dimensione)
4. Si reitera dal passaggio 1 finchè l'AND è nullo

L'algoritmo rimuove progressivamente le parti esterne; risulta efficiente quando molti dei segmenti da clippare sono completamente esterni al rettangolo di clipping.

La maggior parte delle operazioni sono comunque implementabili via hardware, dato che si basa su logica booleana.

Algoritmo di Liang-Barsky

Ogni punto del segmento $s = (p_0, p_1)$ si può esprimere come combinazione lineare

$$p_i = \lambda p_0 + (1 - \lambda)p_1 \text{ con } \lambda \in [0, 1]$$

esprimendolo in forma parametrica otteniamo

$$\begin{aligned} x = x_0 + (x_1 - x_0)t \qquad & = x_0 + \Delta x t \\ y = y_0 + (y_1 - y_0)t \qquad & = y_0 + \Delta y t \end{aligned}$$

per ogni punto interno al rettangolo (P_{min}, P_{max}) possiamo esprimere i seguenti vincoli

$$v_k t < q_k, \text{ con } k = 1, \ldots, 4$$

dove

$$v_1 = -\Delta x \qquad q_1 = p_{0_x} - x_{min}$$

$$v_2 = \Delta x \qquad q_2 = x_{max} - p_{0_x}$$

$$v_3 = -\Delta y \qquad q_3 = p_{0_y} - y_{min}$$

$$v_4 = \Delta x \qquad q_4 = y_{max} - p_{0_y}$$

Se il generico elemento v_k è positivo allora nel muoversi nel verso da p_0 a p_1 si passa da dentro a fuori rispetto al vincolo. Si sostituiscono i valori calcolati e si ottengono i t calcolando l'uquaglianza nell'equazione 1.3.6.

La parte del segmento interna al rettangolo di clipping è individuata da t_e (entrata) e t_u (uscita) dove: t_e è il massimo tra 0 (valore minimo di t) ed i valori t_k per cui si entra nella regione di clipping ($v_k < 0$) t_u è il minimo tra 1 (valore massimo di t) ed i valori t_k per cui si esce dalla regione di clipping ($v_k > 0$). Se $t_e > t_u$ allora il segmento è esterno al rettangolo di clipping.

Algoritmo di Sutherland-Hodgman

L'approccio dell'algoritmo di Sutherland-Hodgman è divide and conquer. Il passaggio base è il clip del poligono rispetto ad una singola retta infinita. Si reitera quindi il passaggio per tutti e quattro i lati del rettangolo.

In ingresso la sequenza dei vertici $v_1, v_2, \ldots, v_n$ che definiscono gli n spigoli ($n-1$ da v_i a $v_i + 1$, $(i = 1, 2, \ldots, n)$ e lo spigolo da v_n a v_1). Ad ogni iterazione il generico vincolo (ad esempio $x = x_{min}$) individua un semipiano interno (corrispondente alla finestra di clipping) ed uno esterno.

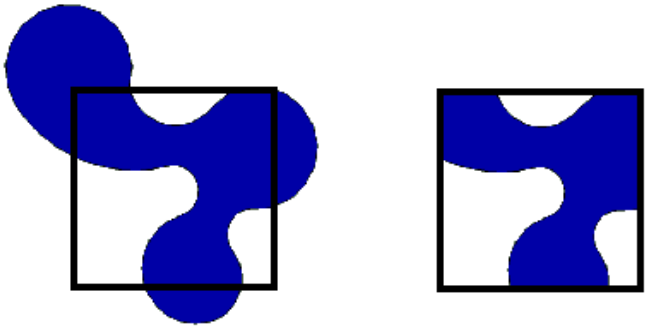

Figura 1.12: Un clipping di poligono

Il confronto si effettua visitando il poligono dal vertice v_1 al vertice v_n e quindi di nuovo a v_1. Ad ogni passo dell'algoritmo si analizza la relazione esistente tra due vertici successivi sul poligono e la retta di clipping (4 casi possibili). Ad ogni iterazione, i vertici inseriti nella lista di output rappresentano i dati in ingresso per l'iterazione successiva.

Ogni confronto ha le seguenti possibilità:

- v_{i-1} e v_i giacciono sul semipiano interno; v_i è aggiunto alla lista di vertici di output;
- v_{i-1} è interno, v_i è esterno; l'intersezione p dello spigolo con il vincolo è aggiunta alla lista di output;
- v_{i-1} e v_i giacciono sul semipiano esterno; nessuna azione;
- v_{i-1} è esterno, v_i è interno; L'intersezione p ed il vertice v_i in output.

1.3.7 2D Screen Space

La modellizzazione finora apportata lavora in uno spazio tridimensionale. Poiché i device sono per loro natura bidimensionali bisogna effettuare una proiezione su di essi. Ci sono due tipi di proiezioni possibili: la proiezione prospettica e quella ortogonale.

Trasformazione Prospettica

Con la proiezione prospettica si deforma la scena per rappresentare la distanza dell'osservatore. Tale operazione non è né reversibile né lineare, e viene quindi eseguita a valle delle altre operazioni.

La matrice di trasformazione prospettica P è definita come

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

da cui

$$P \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} = \begin{bmatrix} \frac{xd}{z} \\ \frac{yd}{z} \\ d \\ 1 \end{bmatrix}$$

l'ultima uguaglianza deriva dalle proprietà delle coordinate omogenee (vedi A) e serve per normalizzare le coordinate.

Il parametro d è detto distanza focale, e nel modello pinhole è la distanza tra l'obiettivo e il piano immagine.

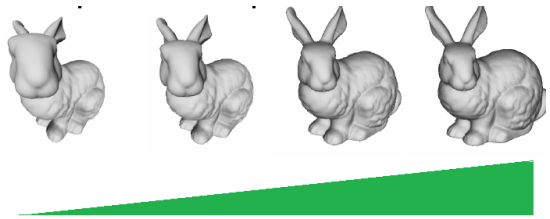

Figura 1.13: Gli effetti di una distorsione con focali diverse

Al variare di d ci sono effetti diversi. Un valore ridotto rende un effetto grandangolare (*fish-eye*) mentre un valore elevato mantiene maggiori prospettive (come da un satellite), come si vede in figura 1.13

La proiezione Ortogonale - da finire

Nella proiezione ortogonale si sostituisce la coordinata z alla distanza focale.

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Così facendo si pone l'osservatore ad una distanza infinita dalla scena.

1.3.8 Confronto tra le proiezioni

Le due proiezioni si equivalgono quando si pone il sistema a distanza infinita.

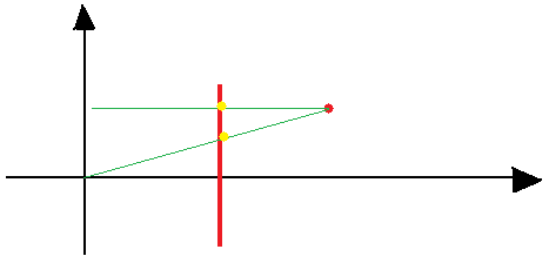

Figura 1.14: Le due proiezioni a confronto

Nella figura 1.14 possiamo vedere come uno stesso punto venga mappato differentemente.

Come si è precedentemente visto le trasformazioni potrebbero avere alterato il nostro ambiente normalizzato. Perciò, sia che la proiezione sia prospettica sia che sia ortogonale occorre ridefinire il volume di vista (*viewing volume*) come il cubo unitario identificato dai semipiani $+1$ e -1 per i tre assi .

Definendo le variabili n la distanza z del piano di clip frontale f quello di fondo, r la coordinata destra del piano immagine, l quella sinistra, t la coordinata di top e b quella del fondo otteniamo, per la proiezione prospettica la seguente matrice

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

La matrice diventa invece per la proiezione ortogonale

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{-(r+l)}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{-(t+b)}{t-b} \\ 0 & 0 & \frac{2}{n-f} & \frac{-(n+f)}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A questo punto si deve effettuare l'ultima trasformazione dei vertici, detta **window to viewport**. La window è la nostra visione della scena, la viewport invece l'area del device in cui dovrà esser visualizzata. L'avere normalizzato la vista ci permette di assumere le coordinate d'interesse nel cubo unitario. Le operazioni che si effettuano in sequenza, di cui si omettono le matrici sono:

1. Cambio del sistema di riferimento in quello del device
2. Scalatura per adottare le proporzioni della viewport
3. Traslazione per avere la stessa posizione della viewport.

Dopo queste operazioni si passa al sottosistema Raster.

1.4 Rasterizer

Il Rasterizer converte triangoli espressi in funzione dei vertici in frammenti. Si utilizzano i triangoli come primitive di riferimento perché sono denotati da una semplicità di trattazione e perchè tutti i poligoni sono riconducibili, anche se in maniera a volte non banale, ad insiemi di triangoli.

Nel Rasterizer si passa dal dominio continuo dell'elaboratore a quello discreto della visualizzazione (codominio). Lo scopo è quindi quello di riuscire a individuare il subset di pixel che sono contenuti nella primitiva d'oggetto.

Oltre alla verifica banale se un punto è o meno contenuto nell'oggetto ci sono le verifiche di visibilità dall'osservatore (**Hidden Surface Removal** o **Visibility Determination**). Un oggetto potrebbe essere nascosto (occluso) totalmente o parzialmente da un altro oggetto. In tal caso bisogna determinare quale dei due oggetti è visibile e quale no. Per oggetti parzialmente trasparenti si parla inoltre di tecniche di **blending**.

Il rasterizer utilizza una serie di buffer, tutti con le medesime dimensioni del frame buffer. C'è lo **z-buffer**, utilizzato per memorizzare la pronfondità dei pixel rasterizzati, l'**alpha buffer**, che memorizza il livello di saturazione dell'opacità e i **color buffer**, che memorizzano le tre componenti di colore dei pixel.

Essendo molto specifico, è l'unica stage della pipeline che allo stato attuale non è programmabile. Ci sono difatti due accortezze da seguire per il rasterizer:

1. **Aumento della taglia dell'elaborazione** l'output è sempre considerevolmente superiore in taglia all'input dato poiché ogni triangolo non degenere crea due o più frammenti in output.

2. **Ordinamento** poichè siamo in un'architettura parallela con controlli minimi e alta richiesta computazionale è necessario rispettare l'ordinamento dell'input (determinato dall'ordine di arrivo) per non rallentare il sistema.

Ne segue che l'implementazione è fondamentale. Segue una breve trattazione teorica del problema del raster, per poi focalizzarsi sulle tecniche utilizzate attualmente

1.4.1 Triangle Setup

Prima di passare alla rasterizzazione e ai suoi algoritmi in senso stretto la pipeline elabora i dati passati dalla Vertex Stage. I dati di cui disponiamo difatti non

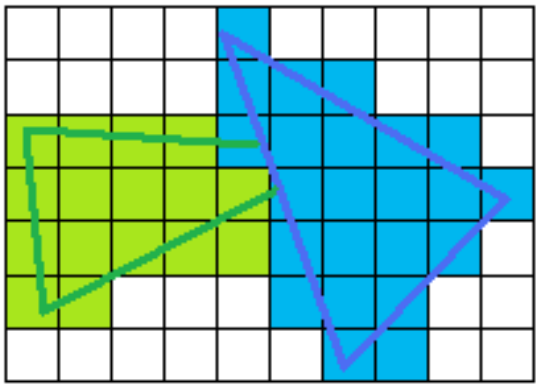

Figura 1.15: Un esempio di rasterizzazione di due triangoli

sono sufficienti per le elaborazioni successive, e quindi devono esser arricchite. I parametri di cui disponiamo difatti sono la posizione dei vertici, la loro normale e come essi formano dei triangoli.In questo passaggio si calcolano invece gli altri elementi, come la pendenza degli spigoli, il gradiente colore, il gradiente delle coordinate texture.

1.4.2 Clipping Rasterization

Un primo approccio di rasterizzazione è quello di effettuare il clipping del piano immagine calcolando le equazioni della retta.

Si potrebbe migliorare il tutto limitando il calcolo ad un rettangolo minimale, avente per vertici (X_{min}, Y_{min}),(X_{max}, Y_{min}),(X_{min}, Y_{max}),(X_{max}, Y_{max}).

Anche così facendo però non miglioriamo eccessivamente le performance, dato che si devono elaborare tutte le celle. Ulteriori raffinamenti ci sono proibiti dato che si inizierebbero ad avere troppi cicli di controllo, andando a degradare le performance. Modificando l'idea si può ottenere la Rasterizzazione a Linea.

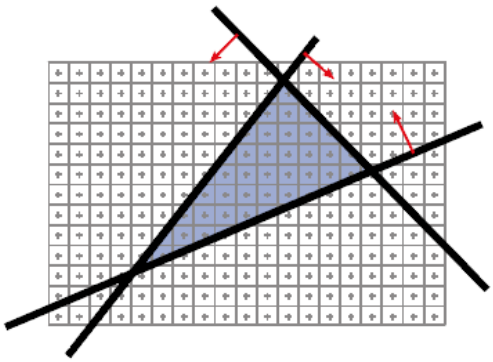

Figura 1.16: Le equazioni di retta con le loro normali

1.4.3 Scanline Rasterization

La Scanline Rasterization è la tecnica storicamente adottata nelle schede video.

L'obiettivo è quello di creare degli *span*, ovvero delle aree di pixel di larghezza unitaria, cercando di ridurne al minimo il numero.

Sia dato un triangolo T con vertici p_0, p_1 e p_2, ordinati per ordinata crescente. Una prima ottimizzazione è quella di dividere il triangolo in due metà T_1 e T_2, imponendo come terzo lato la parallela a p_1 rispetto all'asse y e scartando quelli degeneri. Questa divisione ci permette di tener traccia solamente di due lati al posto di tre e snellisce la parte di controllo del ciclo senza aumentare il numero degli span.

Tipicamente un rasterizer inizia da p_0 e avanza per i due lati sinitro e destro, creando uno span per ogni scanline incontrata (tipicamente per ogni valore intero delle ascisse).

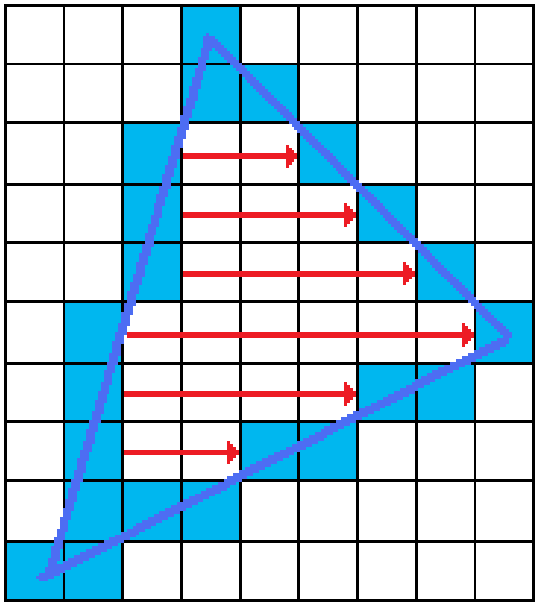

Figura 1.17: L'idea di base della Rasterizzazione a Scansione di Linea

Ricordando che da due punti p_0 e p_1 la retta che li interseca è

$$\begin{aligned} x(y) &= \frac{\Delta y}{\Delta x} y \rightarrow x(y) \\ &= x(y+1) - \frac{\Delta x}{\Delta y} \end{aligned}$$

poichè lo spazio è quantizzato avremo che il valore $x(y+1)$ sarà compreso tra $L_{i+1} = \lfloor x(y+1) \rfloor$ e $H_{i+1} = \lceil x(y+1) \rceil$.

Le distanze di L_{i+1} e H_{i+1} da $x(y+1)$ saranno rispettivamente

$$\begin{aligned} l_{i+1} &= x_{i+1} - L_{i+1} & &= \frac{\Delta y}{\Delta y}(y_i + 1) - x_i \\ h_{i+1} &= H_{i+1} - x_{i+1} & &= x_i + 1 - \frac{\Delta y}{\Delta y}(y_i + 1) \end{aligned}$$

effettuando la differenza $l_{i+1} - h_{i+1}$ otteniamo

$$\begin{aligned} (l-h)_{i+1} &= 2\frac{\Delta y}{\Delta y}(y+1) - 2x_i - 1 \\ \Delta y(l-h)_{i+1} &= 2\Delta x \cdot y_i + 2\Delta x - 2\Delta y \cdot x_i \\ &= 2\Delta x \cdot y_i - 2\Delta y \cdot x_i + d_0 \end{aligned}$$

dove $d_0 = 2\Delta x - \Delta y$.

Così facendo otteniamo un'espressione con una sola moltiplicazione e che dipende dai valori pregressi di y, x.

$$\begin{aligned} d_{i+1} &= \Delta y(h-l)_{i+1} \\ &= 2\Delta x \cdot y_i - 2\Delta y \cdot x_i + d_0 \\ d_i &= \Delta y(h-l)_i \\ &= 2\Delta x \cdot y_{i-1} - 2\Delta y \cdot x_{i-1} + d_0 \end{aligned}$$

nelle nostre applicazioni le y hanno un campionamento omogeno a distanza 1. Se il punto precedentemente selezionato era di tipo L allora avremo $d_{i+1} = 2\Delta x + d_i$, altrimenti $d_{i+1} = 2\Delta x - 2\Delta y + d_i$.

Questo algoritmo è molto funzionale per l'hardware dedicato dato che usa solamente calcoli incrementali per calcolare gli interpolanti (*interpolants*) e riduce al minimo le iterazioni di controllo.

Ottengo così per ogni triangolo una serie di span, definiti da x_{start},x_{end} e dal d_{i+1}

Problemi con la ScanLine Rasterization

Sebbene facile da implementare via hardware essa ha alcuni difetti

- Overhead con i triangoli piccoli: se un trinagolo copre uno spazio ridotto, come ad esempio 1-2 pixel, la preparazione dei delta e i calcoli viene comunque eseguita, anche se inutile.

- Grande numero di interpolanti: per i triangoli grandi si ha un elevato numero di parole di memoria (ogni span ne necessita 2), rischiando di rallentere la trasmissione alle stage successive.

- Interpolanti a metà: se un interpolante è a metà strada da H e L il pixel non viene assegnato. Si potrebbe decidere di impostare un valore di Default, ma anche così non si ha la certezza che triangoli adiacenti coprano lo stesso pixel o che non lo disegnino due volte.

per ovviare almeno all'ultimo dei problemi riportati si può modificare la edge function $E(x, y) = Ax + By + c$ (vedi appendice) per cui un pixel appartiene al trinagolo se

$$E(x,y) > 0 \vee (E(x,y) = 0 \wedge t) \text{ dove } t = \begin{cases} A > 0, & \text{se } x \neq 0, \\ B > 0, & \text{altrimenti} \end{cases}$$

1.4.4 Barycentric Rasterization

[2] Il principale problema dello scanline è l'approccio in caso di triangoli estesi. Questo algoritmo invece permette di spezzare i trinagoli in sotto triangoli fino a quanto si desidera.

Sebbene simile allo Scanline, il Barycentric Rasterization utilizza una differenziazione tra *pixel coverage*, ovvero il calcolo di quali pixel sono interni ad un triangolo e *interpolation*, ovvero il calcolo del colore dei dati pixel.

Ricordando come un punto interno ad un triangolo sia la combinazione lineare dei suoi vertici nello spazio omogeneo

$$p = \lambda_0 p_0 + \lambda_1 p_1 + \lambda_2 p_2$$

Per l'algoritmo quindi dovremmo calcolare le coordinate λ_i per tutte le coordinate (x,y) interne al rettangolo minimale.

$$\lambda_0 = \frac{(y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0}{(y_0 - y_1)x_2 + (x_1 - x_0)y_2 + x_0 y_1 - x_1 y_0}$$

[2] il seguente algoritmo è tratto da (3)

$$\lambda_1 = \frac{(y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2}{(y_2 - y_0)x_1 + (x_0 - x_2)y_1 + x_2y_0 - x_0y_2}$$

$$\lambda_2 = 1 - \lambda_0 - \lambda_1$$

se λ_2 risulta compreso tra 0 e 1 il pixel verrà disegnato, altrimenti scartato.

possiamo inoltre dimostrare come $\lambda_i = \frac{A_i}{\sum_{i=0}^{2} A_i}$ con A_i l'area del triangolo formato dai vertici $p_j, j \neq i$ e p e calcolare con la stessa logica le lambda.

1.4.5 Blending

Quando ci sono degli oggetti trasparenti o non completamente opachi si deve potere vedere attraverso. Per far questo si usa la componente α dell'RGBA. Posta quindi a 1 indica un oggetto completamente opaco, a 0 un oggetto trasparente.

Ad ogni renderizzazione si aggiungono i valori pesati per una costante di opacità k_{add}. Così facendo abbiamo un sistema in media mobile. La componente α viene memorizzata nell'alpha buffer, mentre le componenti RGB nei color buffer.

$$\begin{bmatrix} r_s \\ g_s \\ b_s \\ \alpha_s \end{bmatrix} = k_s \begin{bmatrix} r_s \\ g_s \\ b_s \\ \alpha_s \end{bmatrix} + k_{add} \begin{bmatrix} r_{add} \\ g_{add} \\ b_{add} \\ \alpha_{add} \end{bmatrix}$$

$$k_{add} = 1 - k_s$$

I due approcci possibili sono il **front to back** e il **back to front**. A contrario di quello che si potrebbe pensare l'approccio più funzionale è il secondo. Se si renderizzasse difatti gli elementi partendo da quelli più vicini la pipeline si renderebbe conto che si cerca di disegnare elementi nascosti (per maggiori dettagli vedere la sezione 1.4.6) e li scarterebbe. Disegnando invece dal fondo si applica l'algoritmo del pittore(vedi 1.4.6) che permette di combinare i risultati.

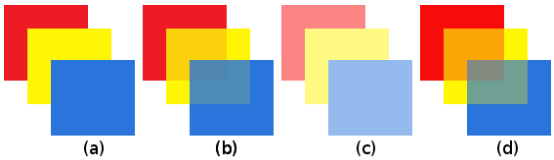

Figura 1.18: La stessa sovvrapposizione front to back (a) back to front con $\alpha_{add} = 1, k_{add} = 1$ (b)$\alpha_{add} = .5, k_{add} = 1$ (c)$\alpha_{add} = .5, k_{add} = .5$ (d)

1.4.6 Hidden Surface Remove

Definiamo queste due classi d'algoritmi:

Definizione 1.1 ***Hidden Surface Removal*** *è l'insieme di operazioni che identificano l'esatta porzione di un poligono visibile.*

Definizione 1.2 *L'**Occlusion Culling** sono i metodi che identificano un sottoinsieme dei poligoni non visibili. Questi algoritmi calcolano quindi il duale del conservative visibility set*

La differenza tra le due classi sono molto labili, dato che spesso algoritmi di HSR contengono al loro interno passaggi di Occlusion Culling e viceversa(4). Per questo motivo da questo punto in poi le due classi verranno utilizzate come sinonimi.

Queste definizioni includono elementi (algoritmi) tra loro molto eterogenei che agiscono in punti diversi della pipeline. Per chiarezza espositiva si è deciso di elencarli tutti insieme, specificando il punto in cui agiscono.

Le operazioni di culling si possono suddividere come

1. **Viewing Frustrum**: se un oggetto è esterno al cono visivo. [3]
2. **Backface Culling**: come si è precedentemente detto le facce nella pipeline sono orientate. Se all'osservatore è visibile un retro ci sono due possibili scenari. Il punto di osservazione è all'interno di un poligono, oppure la faccia dev'esser coperta da qualche altra faccia del poligono (o di qualche altro oggetto). In entrambi i casi la faccia dev'esser scartata.
3. **Contribution Culling**: se l'oggetto non rende contributi utili alla scena (è troppo distante o troppo piccolo) lo si scarta.
4. **Occlusion Culling**: l'oggetto è nascosto, per topologia (ad esempio interno ad un altro) o per punto di vista (è dietro un altro) da un altro oggetto. Questa occlusione può essere totale o parziale.

Backface Culling

Nelle ipotesi che gli oggetti della scena siano rappresentati da poliedri solidi chiusi, che ogni faccia poligonale sia stata modellata in modo tale che la normale ad essa sia diretta verso l'esterno del poliedro di appartenenza allora le facce la cui normale forma angoli inferiori a $\pm\frac{\pi}{2}$ con la direzione di vista certamente non sono visibili.

Per ridurre il carico di lavoro richiesto per la rimozione delle superfici nascoste può essere quindi opportuno eliminare inizialmente tutti le primitive geometriche

[3] poichè gli algoritmi di Viewing Frustrum sono i medesimi del clipping, si rimanda alla sezione 1.3.5 per l'approfondimento

la cui normale è orientata verso il semispazio opposto all'osservatore, non visibile all'osservatore. Indicato con θ l'angolo tra la normale e l'osservatore, la primitiva in esame deve essere rimossa se $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$, ovvero con $\cos\theta$ positivo.

Essendo questa operazione è eseguita in coordinate normalizzate di vista (dopo aver eseguito la proiezione) la determinazione delle facce back-facing si riduce ad un controllo del segno della coordinata z delle normali: ad un segno positivo corrispondono facce front-facing, ad un segno negativo facce back-facing;

Contribution Culling

Il Contribution Culling sono le tecniche con cui si eliminano i contributi trascurabili. Normalmente sono implementate via soglia fisse della scheda. Alcuni di esse sono

- **Z-Killing**: Se il poligono è troppo distante
- **Alpha-Killing**: Se il poligono è troppo trasparente
- **Area-Killing**: Se il poligono ha un'area subpixel

Occlusion Culling

Dato il peso e le complessità delle scene l'Occlusion Culling è la sezione che ha un peso maggiore nel render. Considereremo la divisione proposta da Cohen-Or et all(4). In tale divisione abbiamo

- **Punto contro Regione**: l'algoritmo calcola in funzione del punto dell'osservatore oppure in un intorno di esso.
- **Object-space contro Image-space**: i primi si basano sull'iterazione tra i vari oggetti confrontandoli, i secondi calcolano in base alle proiezioni sullo

schermo. Le prestazioni migliori normalmente si ottengono con algoritmi del secondo tipo.

- **Cell-and-portal contro Scene Generiche**: Il Cell-and-portal è un approccio utilizzato sopprattutto per render di ambienti interni. Si modellizza la scena in ambienti a sè stanti (*rooms*) unite a loro volta da portali (*door* o *portal*). Questi portali sono caratterizzati, come nella realtà, da una finestra che permette la visualizzazione del successivo ambiente.

  Quindi, quando si deve renderizzare una stanza si prendono i portali visibili, e si fa il clipping sulla porzione di stanza collegata. Per gli ambienti esterni questo approccio è complicato, dato che raramente ci sono divisioni che permettono un tale approccio.

  I secondi sono anche detti Potentially Visible Set (PVS Render). In essi si divide la scena in sottoscene convesse e si precalcola la visibilità per ognuna di esse. I vantaggi sono quelli di eliminare la ridondanza di elaborazioni per ogni frame. Come svantaggi abbiamo un possibile rallentamento dovuto alla precomputazione e la necessità di storage addizionale per i dati.

Alcuni algoritmi sono i seguenti:

Algoritmo da Warnock

L'algoritmo adotta un approccio image-space con strategia divide-et-impera basata su una tecnica di suddivisione spaziale. Tale algoritmo viene eseguito durante la rasterizzazione e suddivide il piano immagine in regioni disgiunte. Se per la singola porzione del piano immagine risulta facile decidere le primitive visibili, allora per quella porzione si procede alla restituzione; altrimenti si divide ricorsivamente la porzione in modo da ridurre il numero di primitive incluse. La relazione tra

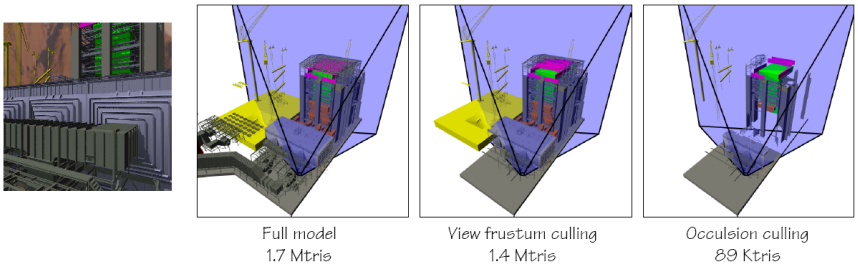

Figura 1.19: Una scena con il numero di triangoli(17)

la proiezione di una primitiva ed la regione può essere contenente,intersecante, contenuta oppure disgiunta come da figura 1.20.

La regione non viene divisa se:

1. Tutte le primitive sono disgiunte rispetto la regione. La regione assume il colore del fondo;

2. Per la regione esiste una sola primitiva intersecante oppure una sola primitiva contenuta. In tal caso la regione assume il colore del fondo e quindi si procede al disegno della parte della primitiva interna ad esso;

3. Per la regione esiste una sola primitiva contenente. La regione assume il

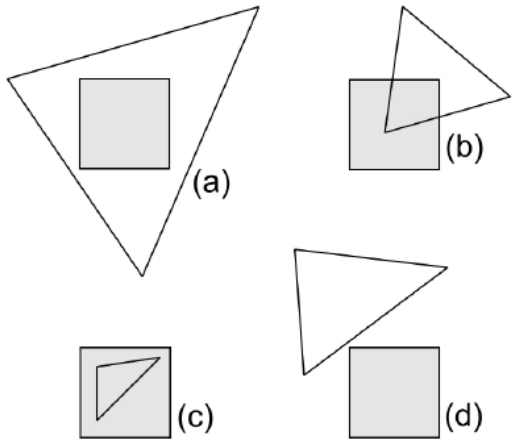

Figura 1.20: Oggetti con una regione contenente (a), intersecante (b), contenuta (c), disgiunta (d)

colore della primitiva;

4. Esistono più primitive in relazione contenente, intersecante o contenente ma solo una è contenente e più vicina delle altre all'osservatore. La regione assume il colore di questa primitiva. Per verificare questa condizione si calcolano le profondità delle singole primitive nei 4 vertici del quadrato. Nel caso di indecidibilità si suddivide e si reiterano i test.

Z-Cull

Durante il processo di raster la pipeline confronta il valore della coordinata z di ogni pixel in elaborazione con quello memorizzato precedentemente nello z-buffer. Ogni elemento dello z-buffer è inizializzato al valore della distanza massima dal centro di proiezione.

Se il nuovo valore è inferiore si elaborano le ombre e si aggiorna la variabile. Poichè il punto di visuale è coicidente con l'osservatore ne otteniamo che la

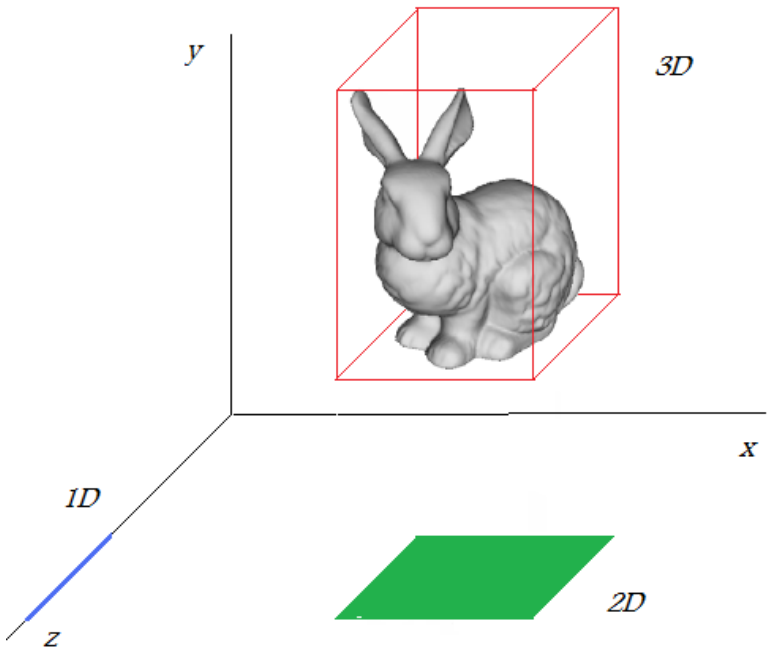

Figura 1.21: Esempio di Extent 1D sull'asse z(blu), Bounding box sul piano xz (verde) e Bounding Volume 3D (rosso)

distanza dall'osservatore è in prima approsimazione il valore dell'asse z.

Nel caso di z-buffer con poca precisione (4 bit) si rischia il fenomeno dello **z-fighting**. Tale fenomeno si verifica quando due primitive sono quasi complanari. I valori per i singoli pixel delle due primitive vincono alternativamente il confronto

Per questo motivo la risoluzione è solitamente portata a 24 bit.

Algoritmo del Pittore

L'algoritmo del pittore è un ordinamento *depth-sort* che viene effettuato prima del calcolo del lighting nella vertex stage. Questo ordinamento ordina per ordine decrescente di distanza dall'osservatore i poligoni. L'algoritmo è nato con l'idea che, come per i pittori, si dipinga prima lo sfondo e poi gli oggetti a mano mano

più vicini. La funzione principale dell'algoritmo, attualmente, è quello di avere un ordinamento degli oggetti e quindi un raggruppamento spaziale (proprietà richiesta da certi algoritmi). Nelle schede video moderne questa funzione è attualmente implementata via hardware(14).

L'algoritmo utilizza i concetti di Extent 1D, bounding box 2D e bounding volume 3D. Questi sono, rispettivamente, il segmento, il rettangolo e il parallelipedo minimali che contengono la proiezione dell'oggetto in esame su di un asse, un piano o uno spazio, come in figura 1.21. L'obiettivo è quello di avere una lista di oggetti ordinati, dove i più distanti precedono i più vicini all'osservatore.

L'algoritmo inizia ordinando gli oggetti prendendo il valore massimo della coordinata Z dei suoi vertici e ordinandoli per ordine decrescente. Per ogni oggetto si calcola l'extent 1D secondo la dimensione z. Se l'extent 1D dell'oggetto A e quello dell'oggetto B non si sovvrapongono allora A precede B. Altrimenti si verifica il bouding box per il piano xy. Se non interferiscono A precede B. Se il test fallisce si verifica se A soddisfa la edge function di B, ovvero se è interna al semipiano più lontano dall'osservatore (figura 1.22). Altrimenti, se tutti i vertici di B si trovano dalla stessa parte dell'osservatore rispetto al piano individuato da A, allora A precede B.

Altrimenti, se le proiezioni di A e B sul piano immagine non interferiscono, allora A precede B. Se tutti i test falliscono si esegue uno scambio tra A e B e si ripete il test. Oggetti che si sovvrappongono ciclicamente come da figura 1.23 darebbero luogo a loop infiniti.

La soluzione esatta prevederebbe la suddivisione come da figura 1.23 Alcune schede video prevedono un risultato approssimato dell'algoritmo, amettendo questi casi degeneri, che vengono trattati con altri controlli. É da dire che è un caso limite, dato che si suppone, normalmente, che la nuvola di punti passati dalla

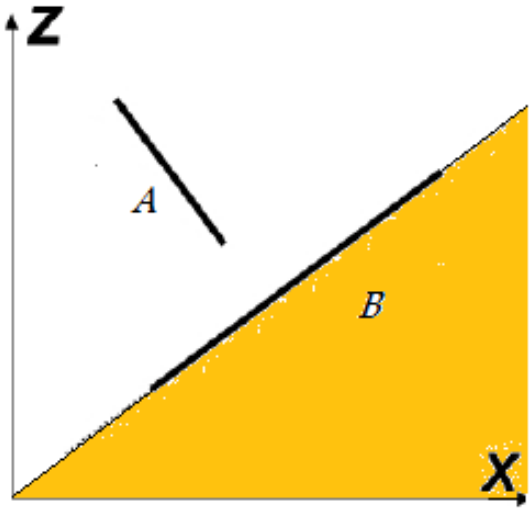

Figura 1.22: Un'edge function per l'algoritmo del pittore

CPU sia sufficientemente densa da evitare queste situazioni.

Scan-line

L'algoritmo è una miglioria dello z-buffer, agisce in fase di raster e riconduce il problema della visibilità tra oggetti a un problema di visibilità tra segmenti. In questo algoritmo sono necessarie delle strutture dati, ovvero una tabella di contatori delle dimensioni del frame-buffer, una tabella con i puntatori alle primitive in oggetto e una con le intersezioni tra le primitive e la scan-line, ordinandole per ascissa crescente e memorizzando le primitive di riferimento.

L'algoritmo cicla ogni riga dello schermo. Per ogni primitiva si calcolano le intersezioni con la scan-line e si incrementa di un'unità i contatori relativo alle celle interne. Se tutti i contatori hanno valore unitario o nullo allora non c'e conflitto. Se invece ci sono degli intervalli di contatori maggiori di uno allora si effettua un test di profondità tra le primitive in oggetto (nella figura è il caso della retta y_c). Se la riga seguente mantiene lo stesso ordine di intersezione della

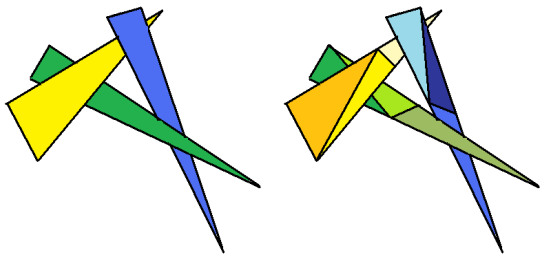

Figura 1.23: Un caso limite e la sua possibile soluzione

precedente allora è sufficiente aggiornare le ascisse dei contatori.

Questo algoritmo ha come pregio un minore numero di confronti, ma ha come svantaggio la necessità di avere delle strutture dati.

1.5 Fragment Stage

La *fragment stage* ha come scopo quello di elaborare tutti i frammenti generati dal rasterizer. Le attività nello specifico sono quelle di interpolare le luminosità dei vertici, di applicare le texture e di rimuovere le superfici nascoste.

La fragment stage ha solitamente una maggiore potenza di calcolo per una serie di motivazioni:

1. ci sono più frammenti che vertici in uno scenario tipico
2. le elaborazioni di una vertex stage devono per forza passare per il rasterizer, rallentando quindi il ciclo di elaborazione, mentre un fragment processor, avendo la possibilita di random access write, può memorizzare le informazioni in memoria, e ricominciare la rielaborazione senza ulteriori passaggi.

L'output risultante può essere visualizzata a schermo oppure scritto su di una ram specifica, detta **texture ram**. Nel primo caso si è in presenza di una operazione *render to monitor*, nella seconda di una *render to texture*. Queste operazioni sono molto importanti perchè invocano rispettivamente il *texture mapping* e il *multitexturing*.

1.5.1 Texture Mapping

Prima operazione della fragment stage, il texture mapping elabora le texture, che sono contenute in una memoria detta **Texture RAM**.

Definizione 1.3 *La **Texture** è una matrice mono o bidimensionale di **texel** (campioni) dello stesso tipo specificata nello spazio parametrico di tessitura* $(u, v) = [0, 1] \times [0, 1]$.

Se si immagina la texture come un'immagine i suoi pixel sono chiamati texel. Un texel può essere di vari tipi. Può essere difatti

- un colore, in tal caso la texture è una color-map
- una componente alpha, in tal caso la texture è una alpha-map
- una normale, in tal caso la texture è una normal-map o bump-map
- un valore di specularità, in tal caso la texture è una shininess-map

A seconda della tipologia di texture si avranno algoritmi differenti. Questi texel non corrisponderanno, come vedremo, necessariamente ad un pixel. [4]

Sia dato in esame il triangolo di vertici $p_i = (x_i, y_i)$ e lo spazio di tessitura (u, v). Per primo passaggio si assegna ad ogni vertice le coordinare di tessitura $p_i \quad (u_i, v_i)$.

Nel caso in cui le mappature siano **esterne allo spazio di tessitura** $(u, v < 0 \vee u, v > 1)$ allora ci sono due approcci possibili. L'approccio **clamp** consste nell'assegnare le coordinate di bordo alle coordinate esterne, mentre l'approccio **repeat** duplica la texture d'oggetto in ogni direzione. Considerando la coordinata u avremo per i due metodi

$$u_i \quad \max(\min(u_i, 1), 0) \tag{1.1}$$

$$u_i \quad u_i \pmod 1 \tag{1.2}$$

Per valutare i punti interni al triangolo occorre tener conto della correzione prospettica. Bisogna quindi fare la prospettiva per ogni punto interno al triangolo. Questa tecnica è detta del **texture mapping perfetto** Se $f(p_i)$ è un attributo del

[4] Per semplicità d'esposizione ora si assumerà una color-map texture bidimensionale applicata al triangoli

vertice $p_i = (x_i, y_i, z_i, w_i)$ come una componente RGBA o l'illuminazione allora avremo per il punto interno $p = c_0 v_0 + c_1 v_1 + c_2 v_2$

$$f(p) = \frac{\sum_i \frac{c_i A_i}{w_i}}{\sum_i \frac{c_i}{w_i}}$$

Come si è detto precedentemente un texel non corrisponderà nella maggior parte dei casi ad un singolo pixel. Nel caso in cui un pixel sia inferiore ad un texel siamo nel caso di **magnification**, il caso opposto è quello della **minification**.

Per la magnification i due approcci più comuni sono il nearest filtering e la bilinear interpolation. Per il **nearest filtering** si assegna al pixel il valore del texel più vicino, mentre per la **bilinear interpolation** si esegue la media pesata dei quattro texel più vicini.

I due metodi hanno dei lati positivi e negativi. Il primo è più veloce, ma mette in risalto le differenze tra pixel adiacenti, rischiando di dar u effetto granulato. Il secondo d'altro canto rende un effetto migliore, ma tende a far perdere i bordi, dando un effetto sfocato.

Per la minification si usa la tecnica del **MIP-mapping**(Multum In Parvo). Questa tecnica, detta anche piramide d'immagine, consiste nel memorizzare la texture a più livelli di dettaglio dimezzando di volta in volta le dimensioni dell'immagine.

Per utilizzare al meglio questa tecnica si introduce un fattore $\rho = \text{texel/pixel}$. Questo fattore viene calcolato nella vertex stage dopo la normalizzazione, e può variare nel triangolo ed essere funzione delle matrici di normalizzazione. Il livello di MIP-mapping che verrà utilizzato sarà $\log_2 \rho$. Il livello 0 corrisponderà al massimo dettaglio.

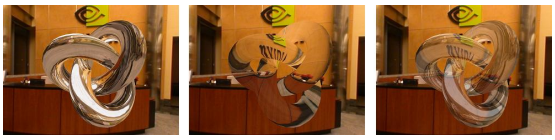

Figura 1.24: Due texture e la loro combinazione (©Nvidia)

MultiTexturing

É possibile applicare più di una texture su di una data primitiva. Le varie texture sono applicate in sequenza usando il risultato del precedente texturing per mixarlo con la texture corrente secondo un proprio texture environment.

Questa tecnica è utile per spargere caratteristiche sull'oggetto diverse dal colore. Ad esempio si può applicare più texture, dette mappe, per definire le varie caratteristiche dell'oggetto come da figura 1.24.

Nella figura si hanno due mappe, la prima di pura riflessione, la seconda di pura rifrazione prese singolarmente. L'immagine a destra è invece il risultato della composizione delle due mappe sulla stessa sorgente.

1.6 Caso Studio: Ge Force GT295

Finora abbiamo analizzato il percorso che una singola primitiva attraversa nella pipeline. Prendendiamo ora in caso una scheda video attualmente in commercio, come la **GeForce GTX295**. Questa scheda è l'ammiraglia nVidia per la prima metà del 2009. Monta al suo interno quattro processori GT200. Adesso passeremo ad analizzare il processore e poi la sua composizione.

1.6.1 GPU: GT295

La prima cosa che si nota è che lo schema da noi adottato nella sezione precedente è diverso rispetto la realtà. Il processore dispone di due modalità: graphic e computing mode.

Nel **Graphic Mode** la struttura è quella della figura 1.25. Le funzioni di vertex e fragment stage sono programmabili. Questo significa che lo sviluppatore implementa le proprie logiche per adattarsi alle esigenze.

L'area di lavoro è divisa quindi in *Texture/Processor Cluster*(TPC) tra loro indipendenti. Ogni TPC ha 3 SM, ogni SMha 24 processori

ROP (raster operations processors) and memory interface units are located at the bottom.

1.7 Confronto Hardware con le CPU

In questa sezione svilupperò le differenze principali tra GPU e CPU a livello hardware. Per far questo si deve però introdurre una differenziazione tra i transistor.

I transistor si possono distinguere all'interno di un chip in base alla funzione che devono soddisfare.

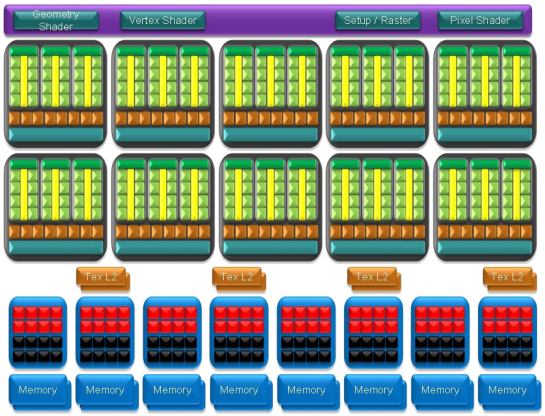

Figura 1.25: Schema a Blocchi del processore GTX295

Un transistor può essere:

1. Control Transistor: dedicato alle funzioni di controllo, come branch prediction, flow etc
2. Datapath Transistor: il transistor funge operazioni di calcolo sui dati
3. Storage Transistor: il transistor memorizza al suo interno i dati, fungendo, a tutti gli effetti da memoria primaria.

Ogni sistema deve avere transistor di tutti e tre i tipi, e la proporzione scelta influirà pesantemente sulle performance del sistema.

Le differenze strutturali tra CPU e GPU sono da imputarsi al loro scopo originario. Le prime difatti hanno sempre elaborato problemi generici e non specifici, con un flusso di carico non omogeneo, mentre le seconde hanno risolto una specifica branchia di problemi, la cui mole di input era costante nel tempo.

Siccome le CPU hanno un modello di sviluppo strettamente sequenziale, la prima differenza che si può notare è che il codice sviluppato per CPU non risalta

il parallelismo intrinseco dei problemi. Avendo quindi un sistema in cui ogni operazione dipende dalla precedente, abbiamo come conseguenza che gli sviluppatori di processori cercheranno di migliorar la latenza a discapito del throuhput. Nelle Gpu, invece, il modello è parallelo, e la latenza viene considerata di secondaria importanza.

Un'altro fattore è che, essendo la programmazione delle CPU strettamente ramificata una buona parte dei transistor son di tipo Control, riducendo il numero di transistor che effettivamente effettuano elaborazioni. Questo significa che a parità di transistor una GPU disporrà di un maggior numero di transistor adibiti al calcolo.

Un'ultima differenza è la specializzazione che hanno le schede video e che ha permesso ai costruttori di montare dell'hardware specializzato al loro interno. Una rasterizzazione di un triangolo (il processo con cui un'area viene segmentata in linee) è indubbiamente più veloce se eseguita su un hardware costruito per svolgere solo questa funzione. Le CPU invece non hanno moduli specializzati in specifici task (ad esclusione di quelli matematici come la ALU) e quindi devono implementare tutto a livello software.

Capitolo 2

Stream Programming Model

In questo capitolo si illustrerà lo *Stream Programming Model* nelle sue caratteristiche generiche, per poi confrontarlo con la graphic pipeline e vedere come i due concetti si debbano unire.

2.1 Stream Programming Model

Nello *Stream Model* i dati vengono visti come un flusso di elaborazioni. I due concetti fondamentali sono lo *stream* e il *kernel*

Definizione 2.1 *Uno* ***Stream*** *o flusso è un insieme ordinati di dati della medesima tipologia.*

Definizione 2.2 *Un* ***Kernel*** *o blocco è una sequenza di operazioni indipendenti e autoconclusive che agiscono su l'intero flusso passato in input e rendono in output un insieme di flussi di cardinalità non nulla.*

Il *kernel* così definito identifica un *Uniform Streaming* poiché elabora tutti gli elementi del flusso. Poiché è sempre possibile ricondursi ad un sistema composto

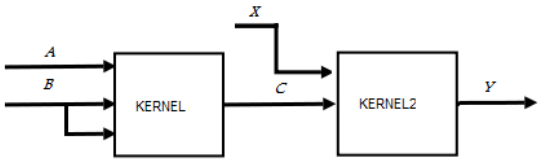

Figura 2.1: un esempio di struttura a blocchi

da soli uniform streaming nell'elaborato consideremo gli stream aventi questa proprietà.

Le due caratteristiche principali del modello sono quelle di esplicitare il parallellismo delle operazioni (secondo il paradigma SIMD, ovvero *single-instruction, multiple-data*) e di separare le fasi computazionali da quelle di caricamento da e per la memoria; questa seconda caratteristica ha come diretta conseguenza una rapida esecuzione dei *kernel*. Difatti, non avendo un accesso alla memoria esterna, le elaborazioni dei singoli kernel dipendono solo da dati locali, e quindi, di conseguenza, solo dai quelli passati a parametro o da alcune costanti del sistema, permettendo un'esecuzione su RAM.

Un esempio di applicazione, utilizzando i Syncronous Data Flow (SDF)(15) può essere quello riportato nella figura 2.1, dove i rettangoli rappresentano i kernel e le frecce i vari stream.

I flussi in ingresso e in uscita dal diagramma effettuano operazioni rispettivamente di **gather** e di **scatter**. Queste operazioni leggono i dati dalla memoria

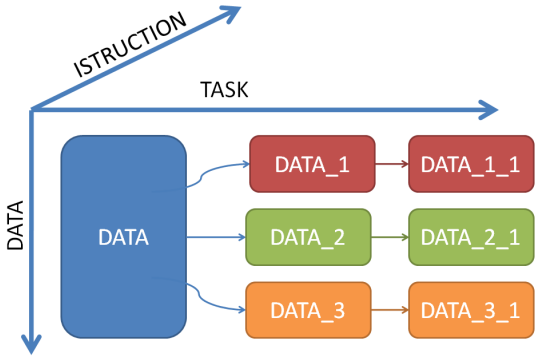

Figura 2.2: I tre livelli di parallelismo

esterna e li scrivono. Separando le operazioni di I/O da quelle elaborative si accentua il parallelismo, poiché, mentre i processore elabora i dati i successivi vengono caricati. Sarà cura del programmatore disporre i dati in ordine sequenziale, in modo che l'accesso alla memoria non sia ad accesso casuale, rallentando il tutto.

La programmazione di un'applicazione deve quindi essere effettuata sia a livello di stream che a livello di kernel, per evitare colli di bottiglia. Nell'immagine 2.1, ad esempio, il kernel2 non potrà essere avviato finchè il kernel che lo precede non abbia completato le elaborazioni. Questa considerazione ci porta a esaminare i tre livelli di parallelismo del modello.

I dati vengono suddivisi in sottoinsiemi tra essi indipendenti; questi flussi vengono eseguiti dai kernel idealmente senza vincoli di dipendenza (*task-parallelism*). Ogni kernel esegue le operazioni vengono eseguite secondo il paradigma SIMD (*data-parallelism*), elaborando quindi più dati con una sola istruzione. Ogni dato, inoltre, se di forma composta, può essere soggetto all'*instruction-parallelism*. Se

ad esempio si stanno trattando punti tridimensionali, le coordinate x,y,z possono essere elaborate separatamente.

Il codice 2.1 mostra una possibile implementazione dell'SDF di figura 2.1 con due kernel in ambiente Java

Listing 2.1: Pseudocodice Java Stream Model

```
1  public static void main (String[] args)
2  {
3          gather(aStream, a, index0, 0, a.length());
4          gather(bStream, b, index1, 0, b.length());
5          kernelCall("Kernel", aStream, bStream, bStream, cStream);
6          gather(xStream, x, index2, 0, x.length());
7          kernelCall("Kernel2",cStream,xStream,yStream);
8          scatter(yStream, y, index3, 0, yStream.length()/2);
9  }
10 public void Kernel(Stream x, Stream y, Stream z,Stream w )
11 {
12         ...       //elaboro gli stream
13         w=...;      //rendo lo stream
14 }
15 public void Kernel2(Stream x, Stream y,Stream z )
16 {
17         ...
18         z=...;      //assegno lo stream
19 }
```

2.1.1 Lo Stream

Secondo la def2.1 lo stream deve contenere dati tra loro omogenei; può essere quindi composto da dati elementari, come interi, double, oppure da strutture dati più complesse come array o matrici, o strutture simili ad oggetti, ma tutti i dati che compongono lo stream devono essere del medesimo tipo.

Possiamo definire la lunghezza del flusso come il numero di *kernel* che lo elaboreranno. Non esiste nessun limite per la lunghezza di ciascuno stream, anche se, come si vedrà successivamente, i flussi più lunghi sono da preferire.

I flussi possono venire suddivisi in substream o riuniti a seconda delle necessità. Queste operazioni vengono effettuate dai kernel. Ogni flusso viene preso in carico da un blocco, che lo elabora.

2.1.2 Kernel

Il kernel può essere visto come una chiamata a funzione di un linguaggio *Object Oriented*. Tuttavia bisogna tenere a mente come un kernel possa elaborare solamente dati locali e non possa, al suo interno, chiamare altri kernel o rendere valori non *void*. Ogni kernel quindi, consterà di una serie di istruzioni (dell'ordine delle centinaia) che potranno essere dei seguenti tipi:

1. **copia**. Il kernel copia dei dati su delle strutture residenti nella memoria locale.

2. **suddivisione in sottoflussi *substream***. Lo stream viene quindi suddiviso in subset secondo una logica booleana. Ad esempio su un'elaborazione agli elementi finiti si potrebbe suddividere un dato stream tra elementi interni e di confine.

3. **elaborazioni dei dati**

L'elaborazione più comune è quella di **mapping**, tramite la quale si moltiplica o si somma un fattore moltiplicativo/additivo al flusso. Altre elaborazioni in uso sono le rotazioni, gli ingrandimenti, le riduzioni e i filtri.

Per ingrandimento si intende quando dato un certo numero di elementi in input se ne produce un numero maggiore (come per un'interpolazione) mentre per riduzione si intende quando si riduce la cardinalità dell'insieme (operazione similare alla decimazione). Il filtro invece rende in output il sottoinsieme dei dati che soddisfano una funzione di valutazione.

2.1.3 Media Application

Il modello a stream è strettamente legato alle applicazioni multimediali per cui sono state ottimizzate le GPU. Difatti abbiamo che

1. al momento dell'elaborazione dispongo di tutti i dati necessari, dato che sono forniti nei flussi di input

2. l'indipendenza delle elaborazioni permette di localizzare le elaborazioni nell'hardware che più si adatta

Da notare come la separazione di caricamento ed elaborazione sia implementata, a livello hardware; esistono difatti delle zone del chip, dette *data-path* destinate all'elaborazione dei dati, e delle zone denominate *stream units* che hanno come funzione l'esecuzione di *gather* e *scatter* verso la memoria esterna. La figura 2.3 indica come la *render pipeline* possa essere espressa in termini di flussi. Per sviluppare un'applicazione GPU si dovrà quindi sviluppare un vertex program, un rasterizator, e cosi via, e collegarli esplicitando la logica del flusso.

Si può notare come lo stream model si adatti bene alla struttura classica delle GPU. Le GPU sono costruite per definizione a stage connesse da flussi dati.

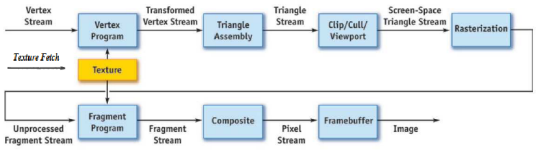

Figura 2.3: Un'applicazione dello Stream applicata alla GPU(22)

Le elaborazioni di un passaggio sono consumate dal successivo, permettendo al sistema di lavorare *in-memory*. Per finire sia il modello sia le GPU dispongono dello stesso dataset di primitive, ragione per cui il carico di lavoro è uniforme lungo il flusso/pipeline.

2.1.4 Performance

Passiamo ora ad analizzare due aspetti fondamentali dell'elaborazione: efficient computation e efficient communication. Lo scopo dei successivi paragrafi è quello di mostrare come la GPU sia efficiente sia per la computazione sia per la comunicazione.

2.1.5 Efficient Computation

Le GPU sono computazionalmente efficienti per la loro stessa struttura. Il motivo principale è che lo stream evidenza in maniera in maniera esplicita il parallelismo.

Poichè i kernel elaborano l'intero stream e la pipeline è composta da più kernel consequenziali, abbiamo due livelli di parallellismo:

1. PARALLELISMO DEI TASK: i vari task, implementati da uno o più kernel, possono esser eseguiti in parallelo, dato che ogni kernel è mappato a livello hardware su unità tra loro indipendenti. Come conseguenza si possono avere quindi anche tutti i kernel che elaborano in parallelo. Ricordando come le Gpu siano composte da hardware altamente specializzato posso inoltre mappare ogni kernel nell'unità che più si adatta alle singole elaborazioni.

2. PARALLELISMO DEI DATI: siccome le unità elaborative all'interno di una GPU sono progettate secondo il paradigma SIMD, più dati vengono elaborati con la stessa istruzione.

Come ultimo elemento avendo elaborazioni elementari e poco strutturate ho una ridotta richiesta di flow control. Questo porta ad una ridotta percentuale di transistor di controllo, incrementando quindi l'area disponibile per il datapath, e quindi per l'elaborazione.

2.1.6 Efficient Communication

Per comunicazione efficiente si intende il minimizzare i costi di trasferimento dei dati da un'unità ad un'altra. Possiamo definire il costo di un trasferimento come

$$tSRC + \frac{nEl * size}{band} + tTGT$$

dove *tSRC* è il tempo in cui il dato è reperibile nella sorgente (access time), *nEl* è il numero di elementi da trasferire, *size* la dimensione media di ogni elemento *band* la banda del canale di comunicazione $tTGT$ il tempo di scrittura sulla memoria di destinazione

I costi di trasferimento on-chip possono esser considerati trascurabili, dato che mediamente una GPU ha una capacità trasmissiva di un ordine superiore a quella del resto della macchina.

I costi che impattano maggiormente sono quelli off-chip, ovvero da e per la GPU. Prendendiamo ad esempio un'hard disk Hitachi Deskstar TK7250 e una Ram DDR3 Samsung. Il primo ha un access time di 12,6 ms, la seconda di 9 ns. Supponiamo inoltre che l'HDD sia correttamente deframmentato, e quindi tutti i dati d'interesse contigui. Se consideriamo le ipotesi di banda della figura 1.1 otteniamo la seguente tabella

Caso	tSRC	Banda	tTGT
da CPU a GPU *(off-chip)*	$1,2 * 10^{-2}[sec]$	6,4 $[Gb/s]$	$9 * 10^{-6}[sec]$
da GPU a GPU *(on-chip)*	$9 * 10^{-6}[sec]$	35 $[Gb/s]$	$9 * 10^{-6}[sec]$

Tabella 2.1: Esempio di trasferimento

Possiamo notare come il trasferimento onchip sia superiore in tutte le casistiche.

Dobbiamo tuttavia considerare che quando si trasferiscono dei dati ci sono due componenti nel costo dell'operazione: un costo fisso che non dipende dalla taglia del trasferimento e una unitaria, che aumenta con il numero di parole trasferite. Trasportando un insieme di dati ovviamente il costo a parola si riduce perchè il costo fisso viene suddiviso tra il numero di parole, raggiungendo, asintoticamente l'upper bound della banda.

Abbiamo inoltre che i dati rimangono sempre on-chip per tutta la durata dell'elaborazione, dato che ogni stage passa i dati alla seguente utilizzando la memoria interna.

Ultimo elemento da considerare è il tempo di trasmissione tra la sorgente e la destinazione, che non è stato espresso nella formula precedente. All'interno di una GPU la banda è superiore, e la distanza fisica è inferiore. Inoltre, per la costruzione delle GPU aree con capacità computazionali simili sono costruite vicino. Per il principio di località temporale abbiamo che esecuzioni tipologicamente simili saranno eseguite a breve distanza l'una dell'altra. Combinando questi due fattori otteniamo che kernel consecutivi saranno, con buona probabilità mappati in unità adiacenti, minimizzando, o quasi, il costo di trasporto.

Abbiamo quindi spiegato come mai le GPU sono efficienti dal punto di vista della comunicazione.

2.2 Flow Control

In questa sezione si riprenderanno i concetti di flow control applicato all'ambiente seriale e successivamente parallelo.

Definizione 2.3 *Il **control flow** è l'individuazione di quali istruzioni e controlli vengono eseguiti all'interno di un programma*

Generalmente le istruzioni di flow control generano uno o più branch che vanno valutati ed eseguiti. In ambiente seriale si cerca di predire quale branch verrà eseguito per tecniche di *caching* e miglioramento di performance.

In ambiente parallelo queste motivazioni si fanno ben più spinte, dato che si lavora sotto il paradigma SIMD. Questo si traduce nel vincolo che ad ogni istruzione i singoli processori possono eseguire solamente una sola istruzione.

Quindi, quando gli shader includono delle branch su un contesto SIMD, la pipeline rischia di non essere più uniforme. Questo problema è noto in letteratura come **branch divergence**.

L'efficacia della pipeline SIMD si basa sull'assunzione che tutti i thread inclusi nello stesso programma eseguano le stesse branch ad ogni flusso. Mentre per la località delle applicazioni grafiche questo è quasi sempre verificato, per le applicazioni general purpose questo assioma non è quasi mai vero.

Ci sono principalmente tre sistemi per il branching di dati sulle moderne GPU:

1. Prediction
2. SIMD branching
3. MIMD branching

Branching Prediction

Architetture che supportano solamente la predizionbe non hanno istruzioni dipendenti dai dati nel senso stretto. Invece, la GPU calcola entrambi i rami decisionali e scarta quello che non soddisfa la condizione booleana di branch. Lo svantaggio della predizione è un overhead computazionale, dato che valutare entrambi i rami potrebbe esser oneroso, ma spesso è l'unica soluzione implementata.

I compilatori per linguaggi di shading ad alto livello come Cg or the OpenGL Shading Language generano in automatico codice macchina con la branch prediction, se la data GPU non supporta altre tipologie di flow control.

SIMD Branching

Nelle architetture Single Instruction Multiple Data (SIMD), ogni processore attivo deve eseguire le stesse istruzioni nello stesso momento per tutti gli elementi che sta elaborando. Ne consegue che in presenza di branch l'unica alternativa è calcolare separatamente le varie branch ed aspettarne l'esecuzione.

Una prima soluzione, chiamata **SIMD Serialization** consiste nell'elaborare i sottoinsiemi di dati che confluiscono in una sola branch, in modo da sfruttare

il parallelismo della macchina. Calcolati tutti gli elementi si passa alla branch successiva. Questa tecnica semplicistica tuttavia degrada in maniera innaccetabile le performance in caso di più istruzioni annidate.

Per questo motivo si adotta anche la tecnica del **SIMD Reconvergence**. Queste tecniche servono per riunificare i flussi e quindi aumentare il parallelismo dell'operazione. L'approccio attualmente utilizzato è quello *immediate post-dominator* che è stato dimostrato essere subottimo(28).

Il concetto base dell'approccio è quello del *post-dominator* così definito

Definizione 2.4 *Una branch X post-domina una branch Y se e solo se tutti i percordi da Y alla terminazione del programma passano per X.*

Definizione 2.5 *Una branch X post-domina immediatamente (immediate post-dominator) una branch Y se e solo se X domina Y e non esiste nessuna branch Z tale che X domina Z e Z domina Y*

.

MIMD Branching

Nelle architetture Multiple Instruction Multiple Data (MIMD) che supportano il branching, processori diversi posso seguire differenti percorsi del programma. Questo avviene perché nelle GPU sono presenti più gruppi di calcolo tra essi indipendenti.

2.2.1 Gestione delle branch lungo la pipeline

Siccome il branc esplicito può degradare le perfomance delle GPU, è utile combinare più tecniche per ridurre il costo del branching. Un'accorgimento è quello di spostare il punto di controllo il prima possibile nella pipeline, al fine di limitare al massimo il numero di elaborazioni che vaerranno scartate.

Static Branch Resolution

Un'analisi della natura del problema può portare ad un'implementazione più performante. Sulle GPU, come per le CPU, evitare branch all'interno di cicli comporta un beneficio in termini di prestazioni. Prendiamo per esempio, quando un problema agli elementi finiti su di una griglia spaziale discreta.

A titolo esemplificativo prendiamo la griglia bidimensionale e composta da n elementi per lato e di indici (i,j). Supponiamo inoltre di avere una subroutine isBoundary(k,m) che renda true se l'elemento è di confine, e false altrimenti Una prima soluzione, posta in logica sequenziale potrebbe essere la seguente

Listing 2.2: Un loop impostato erroneamente a causa della divergence

```
1  public static void main (String[] args)
2  {
3          public static void wrongKernel()
4          { for(i=0;i<n;i++)
5                          for (j=0;j<n;j++)
6                                  if isBoundary(i,j)
7                                          branch1
8                                  else
9                                          branch2
10         }
11 }
```

Questo approccio ha uno svantaggio. Come abbiamo detto in precedenza, la GPU calcolerà ogni singolo branch, giungendo a raddoppiare i calcoli necessari.

Un'implementazione efficiente separerebbe l'elaborazione in più loop, uno per gli elementi interni alla griglia, e uno o più per gli elementi di confine.

Un esempio potrebbe esser il seguente

Listing 2.3: Un loop impostato correttamente

```
1  public static void main (String[] args)
2  {
3          public static void rightKernel()
4          { for(i=1;i<n-1;i++)
5                          for (j=1;j<n-1;j++)
6                                  branch2
7                  for(i=0;i<n;i++)
8                          branch1 con j=0
9                  for(i=0;i<n;i++)
10                         branch1 con j=n-1
11                 for(i=0;i<n;i++)
12                         branch1 con i=0
13                 for(j=0;j<n;j++)
14                         branch1 con i=n-1
15         }
16 }
```

Questa risoluzione dei branch statica produce loop privi di branch. In letteratura questa tecnica trova riscontro nella terminologia dello stream-processing, riferendosi alla suddivisione di stream in sotto-strem (*substream*) Ovviamente il codice precedente non sarebbe implentabile all'interno di una GPU. Questo perché la GPU prenderebbe a parametro per il branch2 un quad disegnato in modo da coprire il buffer d'output con esclusione del bordo, mentre per il branch1 una serie di linee disegnate sul bordo.

Pre-computation

In certi problemi il risultato di un branch risulta costante per un grande numero di iterazioni. In questi casi conviene memorizzare il valore di output per poterlo riutilizzare in futuro. Valuteremo quindi i branch solo quando sappiamo che il risultato è destinato a cambiare, e memorizzandone il valore per tutte le esecuzioni successive.

Questa tecnica è stata utilizzata per precalcolare un ostacolo nella simulazione del fluido di Navier-Stokes fluid nell'esempio delle simulazioni delle SDK dell'NVIDIA NVIDIA SDK [Har05b].

Z-Cull

Si può migliorare ulteriormente il risultato delle *precomputed branch* utilizzando un'altra caratteristica delle GPU, che permette di evitare totalmente i calcoli non necessari.

Nelle GPu moderne esistono dei set d'istruzioni che permettono di non calcolare le ombre per i pixel che non possono esser visualizzati a schermo. Una di queste è Z-cull.

Z-cull è una tecnica gerarchica per confrontare la profondità (Z) di un insieme di frammenti con la profondità del corrispondente blocco di frammenti nello Zbuffer. Se i frammenti in input falliscono tutti il test di profondità (depth test) allora vengono scartati prima che il fragment processor ne calcoli i colori dei rispettivi pixel. Così facendo solo i frammenti che passano il test di profondità vengono effettivamente elaborati, risparmiando lavoro non necessario, e migliorando le performance.

Nella simulazione di fluidi, le celle che rappresentano gli ostacoli bloccati al terreno potrebbero esser escluse assegnandole degli z-valori nulli. In questa manie-

ra il simulatore non calcolerà nessuna elaborazione per quelle celle. Se l'ostacolo è suffientemente ampio si avrà un notevole risparmio di elaborazione.

Cicli Data-Dependent con Query Occlusive

Un'altra feature delle GPU creata per evitar elaborazioni non visibili è hardware occlusion query (OQ). Questa caratteristica riesce a calcolare il numero di pixel modificati da una chiamata di render. Queste interrogazioni vengono inserite nella pipeline, il che significa che forniranno un numero limitato di dati. Poiché le applicazioni GPGPU per la maggior parte delle volte disegnano quad che verranno visualizzati in un'area nota a priori, le QC possono esser utilizzate in combinazione con le funzioni di fragment kill per calcolare il numero di frammenti aggiornati e distrutti.

Questo permette di implementare un sistema di decisioni controllato dalla CPU su elaborazioni della GPU.

Capitolo 3

Shading

Sebbene nel 1990 Cabral et al. avessero utilizzato una scheda video per il motion planning di un robot (16), non ci sono stati linguaggi dedicati all'elaborazione su schede grafiche prima del 2001. In tale data difatti si sono introdotti gradi di liberta dovuti alla creazione di stage programmabili.

Prima di tale data, l'unica via percorribile era l'utilizzo degli **shading language**, obbligando gli sviluppatori a investire più tempo del necessario nell'implementazione a discapito dell'algoritmica.

In questo capitolo si illustreranno le tecniche mediante le quali si possono usare gli shading language utilizzando ad esempio il linguaggio **CG** di nVidia.

3.1 Breve Introduzione

Nella grafica ogni oggetto è caratterizzato dalla forma e dalle sue ombre. Esse dipendono dalla posizione dell'oggetto rispetto alla scena e all'osservatore, e vengono risolte tramite sistemi di equazioni[1].

[1] vedi 1.3.3

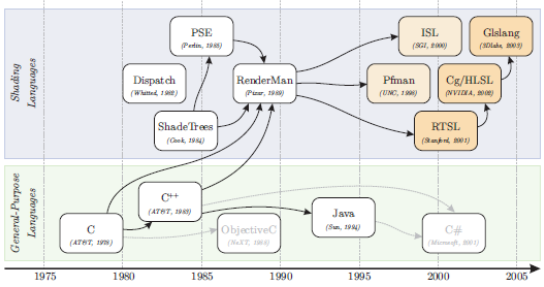

Figura 3.1: Le evoluzioni dei linguaggi di shading

All'inizio gli algoritmi di shading erano integrati a livello hardware nella scheda. Questa soluzione difettava di portabilità e di flessibilità, e quindi si sono iniziati a sviluppare linguaggi per lo shading grafico.

3.1.1 Storia

Il primo progetto comparabile ad un linguaggio di programmazione fu lo **Shades Tree** di Cook, dove una sequenza di equazioni lineari venivano sottoposte a *parsing* ed eseguite in real-time dal sistema di render. Un'ulteriore evoluzione fu opera di Perlin, che sviluppò il *Pixel Stream Editing Language* (PSE) con strutture di alto livello come cicli, salti condizionali e variabili.

Il primo linguaggio di shading nel senso odierno del termine fu il *RenderMan Shading Language*(11). Il linguaggio presenta molte similitudini con il C++, pur non permettendo la definizione di struct o di oggetti. Essendo un linguaggio dedicato alla grafica tuttavia vengono fornite a livello di primitive molte funzionalità come accesso alle texture o trasformazioni ortonormali.

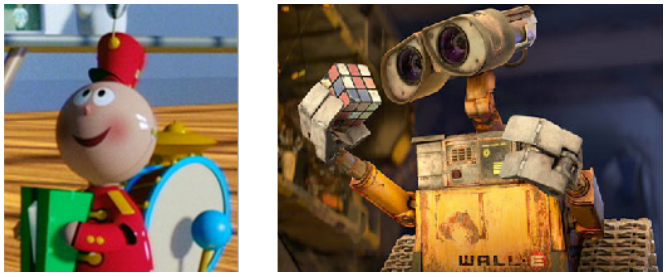

Figura 3.2: Due scene realizzate da Renderman^{TM}rispettivamente nel 1988 e nel 2008

L'evoluzione successiva avvenne nel 1998 quando Olano et al. introdussero **Pfman**, il primo linguaggio di shading realtime. L'approccio era comune a quello della Pixar, ma introduceva alcune limitazioni, limitando le elaborazioni alle operazioni di lighting alle superfici in virgola fissa. Sebbene le ottime prestazioni anche per scene complesse (30 fps), il linguaggio si fermò a livello commerciale a causa dell'alto costo dell'hardware dedicato. Da questa esperienza nacque quindi l'approccio open di **OpenGL**, software che non richiede di hardware specifico e che simula il comportamento di un multiprocessore di tecnologia SIMD.

Con l'introduzione nel 2001 delle vertex unit programmabili si ebbe un ulteriore sviluppo dei linguaggi, passando a quelli che sono stati successivamente definiti *interactive procedural shading*. Il primo di questi linguaggi fu lo **Stanford Real-Time Shading Language** (RTSL)(26).

Lo scopo di RTSL era quello di fornire una pipeline in grado di adeguarsi

al render di scene trattate con approcci differenti (per vertice, per portale, per frammenti).

La CPU, tramite delle simulazioni difatti assegna dei subtask alla pipeline. Così facendo eventuali limitazioni hardware venivano eliminate. Il linguaggio comprendeva anche le cosidette *canned-function*, ovvero chiamate a moduli non ancora implementate via hardware ma che erano verosimilmente sul punto di esserlo. Se l'hardware utilizzato supportava queste chiamate (come il *bump-mapping*) l'esecuzione veniva eseguita via wired, altrimenti tramite subroutine software. RTSL si è sviluppato successivamente in CG e HLSL.

I due linguaggi di shading Cg (*C for Graphics*) e HLSL (*High Level Shading Language*) nacquero nel 2001 dallo sviluppo di un progetto comune tra nVidia e Microsoft.(6) I due linguaggi, sebbene abbiano una semantica e una sintassi comune, si differenziano nella loro concezione in quanto il primo è stato pensato come uno strato sovrastante le API grafiche (OpenGl o DirectX), mentre il secondo non le sfrutta andando ad integrarsi nel framework delle DirectX

L'evoluzione dal predecessore RTSL è nella concezione della macchina. Esiste ancora il concetto di *pipeline programmabile* ma è abbinata alla concezione di una macchina virtualizzata.

3.2 CG

In CG ci sono due profili di linguaggio. Si possono sviluppare difatti Vertex program e Fragment Program, che agiranno, come spiegato nel capitolo 1 i primi sui vertici, i secondi sui frammenti da essi generati. Il listato 3.1 mostra un esempio di vertex program.

Nella firma del metodo ci sono tre tipologie di parametri: i primi sono i parametri d'ingresso, i secondi, identificati dalla parola chiave *out* sono i valori

Listing 3.1: un vertex program in CG

```
1  void vertexProgram(
2                              float4 objectPosition : POSITION,
3                              float4 color : COLOR,
4                              float4 decalCoord : TEXCOORD0,
5                              float4 lightMapCoord : TEXCOORD1,
6                              out float4 clipPosition : POSITION,
7                              out float4 oColor : COLOR,
8                              out float4 oDecalCoord : TEXCOORD0,
9                              out float4 oLightMapCoord : TEXCOORD1,
10                             uniform float brightness,
11                             uniform float4x4 modelViewProjection)
12 {
13         clipPosition = mul(modelViewProjection, objectPosition);
14         oColor = brightness * color;
15         oDecalCoord = decalCoord;
16         oLightMapCoord = lightMapCoord;
17 }
```

resi, mentre gli *uniform* sono equiparabili alle costanti; essi sono difatti delle variabili esterne a CG, come lo stato del device.

CG ha alcune caratteristiche che lo denotano dai normali linguaggi di programmazione. Può ad esempio rendere uno o più oggetti, di tipo eterogeneo (vettori o scalari); inoltre ha dei datatype definiti per l'ambiente grafico, come POSITION per le coordinate tridimensionali di un vertex, o COLOR per lo spazio RGB. Allo stesso modo si possono sviluppare programmi per la fragment stage, come il listato 3.2.

Listing 3.2: un fragment program in CG

```
1 float4 fragmentProgram(
2                         float4 color : COLOR,
3                         float4 decalCoord : TEXCOORD0,
4                         uniform sampler2D decal) : COLOR
5 {
6 float4 d = tex2Dproj(decal, decalCoord);
7 return color * d;
8 }
```

Questa funzione prende un quad identificato da decalCoord, un sample d'interpolazione e il colore. Si prende quindi tramite la chiamata *text2dproj* ad interpolare, e si rende il colore del frammento analizzato.

3.2.1 Differenze rispetto al C

Ci sono alcune notazioni che differiscono sostanzialmente da C. Per l'elaborazioni di vettori ad esempio si possono indicizzare i primi 4 elementi secondo gli assi x, y, z, w. Questo permette un risparmio del codice come da esempio.

Listing 3.3: Codice ottimizzato

```
1 float4 vett1 = float4(4.0, −2.0, 5.0, 3.0);
2 float3 vett3 = vett1.wwx; // vec3 = (3.0, 3.0, 4.0)
```

Sono inoltre presenti una serie di operazioni considerate primitive, come i logaritmi, le radici i moduli; oltre a permettere ovviamente tutti gli accessi a texture. Mancano tuttavia i puntatori o le operazioni di tipo bitwise come pure

oggetti introdotti in C++ come le classi, l'overload degli operatori, le eccezioni e i namespaces.

3.2.2 Elaborazioni su CG

Passiamo ora ad illustrare le tecniche mediante le quali sia possibile eseguire elaborazioni di tipo non grafico con un linguaggio di shading. Tutta la trattazione che segue farà riferimento alle caratteristiche della serie NVIDIA Geforce 6, in quanto sono state le schede che hanno dimostrato migliori performance e versatilità.

La prima cosa da considerare quando si utilizza un linguaggio di shading per le elaborazioni generiche è il pensare i dati disposti in griglie bidimensionali. Questa forzatura è dovuta al fatto che le schede video hanno memorie apposite per gli oggetti a due dimensioni (le texture memory) e quindi possono sfruttarne tutti i benefici che si sono illustrati precedentemente. Qualora i dati fossero disposti in un numero di dimensioni superiori basterebbe effettuare un layout, mentre se si disponessore solamente di array di dati si potrebbe o fare una conversione dell'indirizzo, o effettuare un mapping considerando le località che verrebbero sfruttate dal programma.

Le stage della pipeline grafica si possono riassumere come da figura 3.3. I dati passano quindi la vertex stage, vengono elaborati dal raster e passati poi alla fragment. Poichè il rasterizzatore non è programmabile spesso viene utilizzato solo come interpolatore, e quindi lo ignoreremo dalla trattazione successiva.

Eútile ricordare come la vertex stage possa effettuare solamente gather ma non scatter, mentre la fragment stage possa fare gather ma non scatter.

Ogni stream viene identificato tipicamente da un quad grafico, le cui coordinate corrispondono a tutti gli effetti al dominio dello stream.

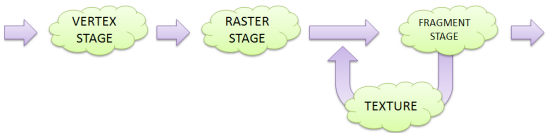

Figura 3.3: Gli stage per l'elaborazione

Questi quattro vertici vengono passati al rasterizer che li interpola, generando i punti di valutazione. A seconda della frequenza di campionamento si potrebbe quindi generare una griglia di naturali, si potrebbe fare un campionamento (ad esempio per fare una riduzione) o un'interpolazione.

I dati passano quindi alla fragment stage, dove avverranno la maggior parte delle esecuzioni. Ogni quad presente in memoria, dopo esser stato rasterizzato viene normalmente colorato nella fragment dallo shading da una serie di kernel successivi. In ambito GPGPU queste chiamate in realtà eseguono le istruzioni del programma, elaborando i dati senza assegnarne significato fisico.

Poiché i cicli (che non sono propri della grafica) sono necessari per qualsiasi programma non triviale si utilizzare il *render-to-texture*. Questo meccanismo in ambito grafico permettte ad un kernel di memorizzare il risultato ottenuto in forma di texture, al fine di riutilizzarne i valori o semplicemente come risultato intermedio di elaborazione. Ponendo di scrivere nella stessa texture in cui risiedono i dati iniziali otteniamo di fatto un meccanismo di looping molto funzionale

ed efficiente. Invocazioni successive dello stesso kernel realizzeranno iterazioni successive di un ciclo. Una volta concluse le elaborazioni il programma effettuerà una *render-to-memory*, salvando i risultati nella memoria globale.

Capitolo 4

CUDA

In questo capitolo si introdurranno le basi del linguaggio di programmazione per schede video, trattando nello specifico il linguaggio CUDA. Dopodiché si prenderanno ad esempio due algoritmi ovvero la somma d'interi e il loro ordinamento.

Per ogni algoritmo ne saranno proposte più versioni: una versione seriale per CPU, e delle versioni per GPU facendo risaltare il differente approccio.

Per finire saranno presentate alcune applicazioni sviluppate in CUDA, allo scopo di fornire una panoramica del linguaggio e delle sue potenzialità

4.1 Introduzione a Cuda

L'architettura CUDA (Compute Unified Device Architecture) si basa sul linguaggio C/C++ per la realizzazione di programmi paralleli di computazione generale in grado di funzionare sui chip grafici.

CUDA è, insieme, un modello architetturale, una Application Programming Interface (API) e una serie di estensioni (C for CUDA) al linguaggio C per descrivere un'applicazione parallela in grado di girare sulle GPU che adottano quel modello.

Il compilatore di CUDA è basato sulla piattaforma Open64, un progetto inizialmente sviluppato da SGI e integrato con una serie di contributi dai gruppi di ricerca nell'ambito dei compilatori, e rilasciato sotto licenza GPL 2.0. Al giorno d'oggi, nonostante gli standard che si vanno ad affermare, come OpenCL e DirectX Compute, CUDA è lo standard de-facto per le applicazioni GPGPU.

Si possono identificare tre generazioni tecnologiche, ognuna delle quali corrispondente ad una release del linguaggio.

- **Prima Generazione**: creata nel 2003, manteneva la possibilità di programmare vertex stage e fragment stage;

- **Seconda Generazione - G80 Series**: è stata introdotta nel Novembre 2006 e ha portato diverse innovazioni, tra cui il supporto al linguaggio C e l'unione di vertex e fragment stage con un'unica stage elaborativa. Questa generazione è stata successivamente rivista con la serie GT200 con migliorie a livello hardware con l'introduzione del concetto di coalescenza (vedi sezione 4.2.1) e la duplicazione della memoria di registro on-chip;

- **Terza Generazione - Fermi**: introdotta nel 2010.

4.1.1 Le basi del modello architetturale di CUDA

Il sistema CUDA lavora concettualmente sia su la CPU che sulla GPU. Il programma viene elaborato sulla CPU (detto *host*), che ne esegue le componenti seriali, e dalla GPU (detto *device*) che ne elabora le componenti parallele. La CPU quindi elabora il programma normalmente, ma demanda le elaborazioni parallele al device.

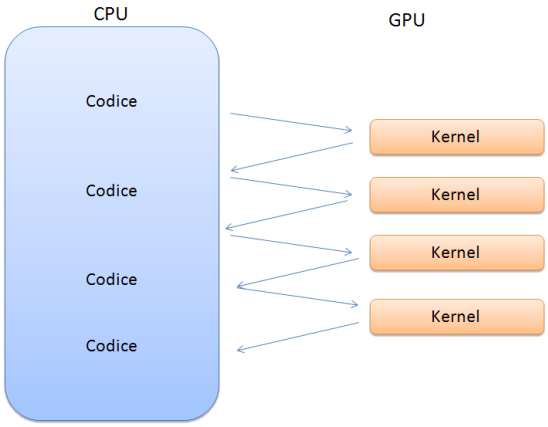

Figura 4.1: La distribuzione del codice in ambiente CUDA

Il device è a sua volta fisicamente suddiviso in **Streaming Multiprocessor** (SM). Queste unità sono tra loro indipendenti, e il loro numero viene solitamente determinato dalla fascia di mercato della scheda.

All'interno dell'SM ci sono gli **Stream Processor**, le unità atomiche CUDA; il numero di processori per SM è fisso durante la generazione architetturale. Questo valore nella serie G80 era fissato ad otto, nella serie Fermi a 32.

A livello software il programma si suddivide anch'esso in tre livelli tra loro inestati: *grid*, *block* e *thread*. Ogni kernel viene suddiviso in blocchi (*blocks*) tra loro indipendenti. Ogni blocco verrà assegnato dallo schedulatore ad un Stream Multiprocessor differente, garantendo un primo livello di parallelismo.

Ogni Stream Multiprocessor a sua volta assegnerà ai suoi processori i diversi thread generati dal programma. Ogni livello di parallelismo viene organizzato a matrice, assegnando quindi a zone attigue indici adiacenti. I blocchi sono considerati elementi di una matrice bidimensionale, mentre i thread di una tridimensionale.

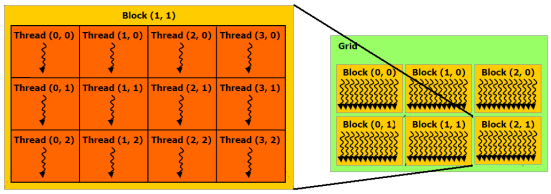

Figura 4.2: La strutture CUDA

Questa organizzazione dimensionale è molto importante per un concetto basilare delle CUDA, ovvero il *thread identifier*. Ogni thread difatti può essere identificato da un indice ottenuto dalla concatenazione di identificatore di blocco e di thread locale. Questa tecnica torna utile nelle componenti parallele, dato che, se opportunamente gestita, permette di assegnare ad ogni thread uno sottospazio computazionale in funzione del proprio indice.

Ne segue che, per come è organizzato il sistema, ogni kernel viene eseguito sequenzialmente, essendo invocato dalla CPU, mentre i blocchi e i thread sono eseguiti logicamente in parallelo, risiedendo sulla GPU. Il numero di thread fisici in esecuzione in parallelo dipende dalla loro organizzazione in blocchi e dalle loro richieste in termini di risorse rispetto alle risorse disponibili nel device.

La strutturazione del programma in blocchi è stata pensata per garantire la scalabilità del programma; l'utente difatti decide la suddivisione in *grids* - *blocks* tramite le parametri delle chiamate a funzioni, ma è l'hardware a decidere, in maniera totalmente autonoma, la distribuzione dei *blocks* sui vari multiproces-

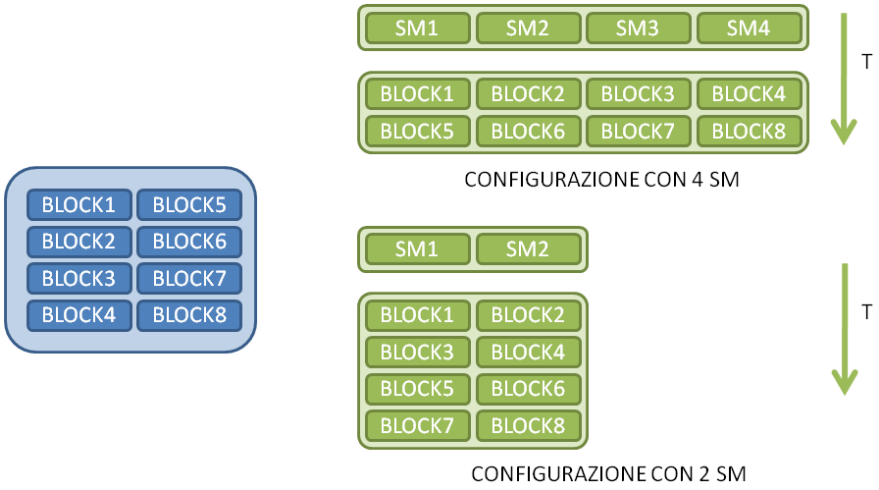

Figura 4.3: Alcune possibili configurazioni di esecuzione

sori come da figura 4.3. Ogni kernel viene scalato quindi sui multiprocessori disponibili. Tale architettura è quindi dotata di *Trasparent Scalability*.

4.1.2 Memorie

La comunicazione tra le memorie residenti nella GPU e quelle del processore avviene tramite le primitive *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost* e *cudaMemcpyDeviceToDevice*. In cui la prima copia i dati dall'host al device, la seconda dal device all'host (per esempio, per leggere i risultati) e la terza i dati in due aree di memoria diverse sul Device. Nelle GPU ci sono differenti memorie a loro volta distinte e sono: *global*, *shared* e *local*.

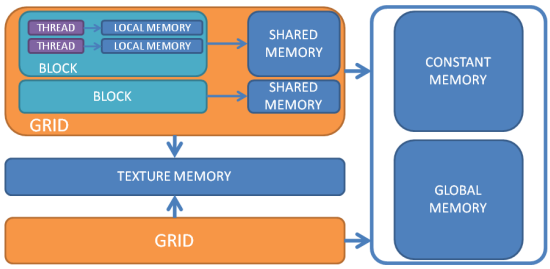

Figura 4.4: Le varie memorie concepite in CUDA

La memoria di tipo **global** è la memoria che permette a device ed host di comunicare; è composta da due memorie, la global e la constant. La differenza ad alto livello è nei permessi di lettura/scrittura.

Mentre la global è accessibile in scrittura dal device, la constant può essere modificata solo dall'host. Ne segue che i dati inviati dall'host e resi disponibili dal device dovranno essere definiti come global e risiederanno in tale memoria, mentre la constant dovrà essere utilizzata per le costanti e i parametri comuni ai thread. Una differenza a livello architetturale è che la constant memory è sottoposta a cache, e quindi è consigliabile il suo utilizzo per accedere allo stesso indice per più thread di un dato warp.

Ogni stream processor inoltre ha una memoria locale dedicata detta *shared memory*, non accessibile dagli altri blocchi. Per finire ogni thread dispone di un quantitativo di memoria di tipo locale per le proprie operazioni interne, a volte denominata *registry memory*. A partire dalle schede video che supportano la tecnologia Fermi è disponibile una miglioria in questo modello, poiché è stato

introdotto un unico spazio d'indirizzi continuo a 40 bit. Così facendo il sistema converte automaticamente gli indirizzi globali in locali, rendendo l'operazione trasparente agli sviluppatori e in stile *C-like*.

Le memorie risiedono fisicamente in banchi separati. Attualmente la global risiede solitamente su DRAM, con una cache abilitata per la componente constant. La memoria shared invece risiede su banchi DDR5, molto più veloci ma inaccessibili dall'host.

Passiamo ora ad approfondire le due memorie che si differenziano maggiormente dal classico modello seriale, ovvero la shared memory e la texture memory:

A - Shared Memory

La memoria di tipo shared è la memoria maggiormente utilizzata nelle applicazioni CUDA. Questo è dovuto ad una serie di fattori che la fanno preferire alla global. Innanzitutto risiede sulla GPU, e quindi sfrutta le potenzialità di banda e la scarsa latenza della scheda grafica (tali fattori si concretizzano in performance di due ordini di grandezza superiori). Inoltre è suddivisa in sottosezioni, chiamati banchi (*banks*) che sono in grado di soddisfare le richieste in parallelo, adeguandosi molto bene all'impostazione parallela del programma. Ha tuttavia alcuni svantaggi; ad esempio le ridotte dimensioni di poche decine di KB ne limitano un un utilizzo indiscriminato, è inoltre soggetta al fenomeno del conflitto di banco così definito.

Definizione 4.1 *Definiamo* ***Bank Conflict*** *il fenomeno che avviene quando due o più indirizzi di memoria contenuti nella medesima richiesta di I/O vengono risolti in indirizzi contenuti nello stesso banco (modulo) di memoria.*

Nel caso in cui in una richiesta ci sia bank conflict il sistema traduce la richiesta in più richieste prive di bank conflict. Questo procedimento abbassa ovviamente il grado di parallelismo (e di conseguenza la banda) dell'elaborazione di un fattore

proporzionale al numero di conflitti presenti nella richiesta. Nel caso peggiore, su una richiesta di n elementi, si possono avere fino a $n-1$ conflitti, che portano ad una lettura seriale dei dati.

Attualmente i banchi sono composti da parole di 32 bit e hanno una banda di 32bit ogni 2 cicli di clock. In caso di compute capability 1.x i 32 thread del warp devono accedere ai 16 banchi della shared memory. Ogni richiesta quindi si traduce come due richieste, una per i primi sedici thread, e la seconda per gli ultimi. Questa limitazione comporta quindi che ogni thread potrà avere bank conflict solamente con altri 15 thread.

Ad esempio se definiamo le seguenti struct

```
1 struct type_no_bank {
2         float x;
3         float y;
4 };
```

Listing 4.1: Struct senza bank conflict.

```
1 struct type_with_bank {
2         float x;
3         char y;
4 };
```

Listing 4.2: Struct con bank conflict.

Nel caso della type_no_bank una struttura da 64 bit (2 bank completi), mentre nella seconda una da 40 bit. Se si accede sequenzialmente ad un array delle due struct la seconda genererà quindi una serie di bank conflict, dato che la maggior parte dei banchi manterranno la fine di un elemento e l'inizio del successivo.

Per finire c'e il meccanismo di **memory broadcasting**. Tramite questo procedimento un indirizzo può essere definito di broadcast (per cui non valgono quindi i bank conflict) permettendogli di servire più richieste con una sola operazione di I/O.

B - Texture Memory

La Texture Memory è la memoria che storicamente ospitava le texture delle schede video[1]. CUDA dispone di alcune delle primitive di texture memory a disposizione delle schede nVidia. Tutte le primitive che eseguono delle letture da questa memoria sono dette di **texture fetch** e puntano a degli oggetti nominati texture reference.

Nelle schede moderne è stata mantenuta separata dalle altre memorie per una serie di caratteristiche che si sono sviluppate nella sua evoluzione e che si rendono utili anche in ambito di general computing.

La texture memory è difatti ottimizzata in cache per la località bidimensionale, permette l'interpolazione a livello hardware fino alle tre dimensioni, e consente l'indirizzamento intero o con coordinata normali.

Queste caratteristiche permettono che in caso di scan o di letture parallele, per il principio della località, i thread dello stesso warp potranno lavorare in sinergia, sfruttanfo la latenza costante della memoria.

È quindi preferibile accedere alla memoria tramite la texture memory nel caso in cui non si seguano i pattern di lettura per la memoria globale o costante[2], se lo stesso dato deve essere letto da più thread, o se si deve effettuare una conversione in floating point *on-fly*.

Questo meccanismo va tuttavia utilizzato con cautela poiché si rischia di ottenere letture inconsistenti (principalmente ghost update). Il motivo è da ricercare nel fatto che memoria globale può esser letta e scritta da più thread, le modifiche non si propagano in automatico nella texture memory, obbligando quindi ad apportare un meccanismo di update della texture memory dopo le scritture sulle zone interessate dalla global.

[1] vedi sez. 1.5.1

[2] Vedi sez. 4.1.2

Dal punto di vista implementativo ogni kernel definisce un texture reference, il quale dev'esser visto come un puntatore ad un'area di memoria, come avviene in C. Non ci sono vincoli alla sovrapposizione delle reference o a puntamenti multipli a singole texture. Ogni texture viene quindi definita con una serie di attributi, tra cui la dimensionalità della texture (fino a tre dimensioni), il tipo dei texel. Ultima considerazione, le texture sono disposte su layout ottimizzati per il fetching monodimensionale.

Le Variabili

Le variabili in Cuda possono esser definite con scope differenti. La sintassi è *scope tipo nome* ad esempio *__device__ __local__ int LocalVar;*. Gli scope possono essere quelli elencati in tabella 4.1

Tabella 4.1: Le possibili dichiarazioni

	Memory	Scope	Lifetime
__device__ __local__	locale	thread	thread
__device__ __shared__	shared	blocco	blocco
__device__	global	grid	applicazione
__device__ __constant__	constant	grid	applicazione

__device__ è opzionale quando seguito da __local__, __shared__, o __constant__. Il qualificatore di default è __local__, quindi, se manca il qualificatore, la variabile risiede sui registri del thread. I puntatori possono esser definiti solamente verso variabili di tipo __global__

4.1.3 Sincronizzazione

L'unica primitiva di sincronizzazione disponibile in CUDA è la *_syncthread()*. Questa primitiva obbliga i singoli thread a fermarsi attendendo che tutti gli altri abbiano raggiunto quel punto dell'elaborazione. Ne segue che questo meccanismo va utilizzato con molta attenzione, rischiando altrimenti accessi non coerenti. Poichè ogni synchthread è indipendente dagli altri non si possono inserire all'interno di una branch di *if*, poichè le due istruzioni allineerebbero solamente i branch al loro interno, rischiando quindi pericolosi loop.

4.2 Struttura di un programma

Possiamo passare ora a vedere come un programma viene strutturato in CUDA. Il codice si divide in componenti sequenziali e parallele, eseguite rispettivamente sull'*host* e sul *device*. La componente seriale comprende le invocazioni dei kernel e la definizione delle variabili e delle strutture dati, mentre l'elaborazione è eseguita in parallelo. É tuttavia possibile effettuare delle elaborazioni in sequenziale, ma è una pratica che degrada le prestazioni, e dovrebbe esser limitata all'elaborazione dei parametri o delle costanti.

4.2.1 Funzioni & kernel

Le funzioni hanno tre identificatori a loro disposizione, a seconda di dove risiede il codice. Essi sono __global__ , __host__, __device__. Il primo definisce un kernel, mentre i seguenti del codice da eseguire localmente sull'host o sul device. Una particolarità di CUDA è che oltre ai parametri normalmente passati ad argomento occorre definire un configurazione d'esecuzione (*execution configuration*) come da listato 4.3.

Listing 4.3: Execution Context

```
1 ...
2 __global__ void KernelFunc(...);
3 dim3 DimGrid(100, 50); // 5000 blocchi
4 dim3 DimBlock(4, 8, 8); // 256 threads a block
5 size_t SharedMemBytes = 64; // 64 bytes di memoria shared
6 KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>
7 ...
```

Inoltre, per le funzioni di tipo __device__ si hanno alcune restrizioni, ovvero:

- non possono esserci ricorsioni
- non sono ammesse variabili statiche dichiarate all'interno della funzione
- non sono ammesse chiamate a funzione con numero variabile di parametri (firme variabili)

Possiamo quindi confrontare i due segmenti di codice 4.4 e 4.5 che mostrano come il calcolo della retta viene modificato nell'architettura CUDA.

Alla chiamata del kernel *retta_p* vengono affiancate le specifiche di blocco e di griglia. Il kernel inoltre si preoccupa di calcolare un indice i (ovvero l'indice di thread) ed elabora nella fattispecie solamente un elemento con quell'indice.

Warp e Scheduling

I blocchi, con i thread a loro connessi vengono assegnati ad uno streaming multiprocessor. Ogni scheda ha dei limiti tecnologici da rispettare per il numero di oggetti concorrenti. Facendo riferimento alla G80, ad esempio, abbiamo che i th-

Listing 4.4: Calcolo della retta seriale.

```
1 void retta_s(int n,float a, float b, float *x, float *y)
2 {
3         for(int i=0; i<n; i++)
4                 y[i]=a*x[i]+b;
5 }
6 ...
7 retta_s(n,2,3,x,y);
```

Listing 4.5: Calcolo della retta parallela.

```
1 _global_ void retta_p(int n,float a, float b, float *x, float *y)
2 {
3         int i=blockIdx.x*blockDim.x +threadIdx.x;
4         if (i<n) y[i]=a*x[i]+b;
5 }
6 ...
7 int n_bloc=(n+255)/256;
8 retta_p<<<n_bloc,256>>>(n,2,3,x,y);
```

read all'interno del blocco possono essere un massimo di $256 \cdot 256$, e che ogni SM supporta al più 758 thread. La distribuzione dei thread a blocco è indifferente.

Ogni blocco esegue i thread raggrupandoli in gruppi detti *warp*. I thread all'interno del warp vengono eseguiti *fisicamente* insieme. Il numero massimo di thread per warp è un parametro tecnologico non modificabile. Ogni Stream Multiprocessor ha uno schedulatore interno di warp, detto *zero-overhead warp scheduling*. La schedulazione sceglie, di volta in volta, un warp tra quelli la cui istruzione successiva ha tutti gli operandi disponibili, secondo una coda a priorità.

Quando un warp viene eseguito tutti i thread al suo interno eseguono la medesima istruzione.

Coalescenza

Un ulteriore grado di parallelismo viene fornito in fase di lettura/scrittura dei dati tramite il meccanismo di coalescenza (**coalescence**). Gli accessi in memoria dei chip grafici compatibili con CUDA devono seguire delle regole ben precise per minimizzare il numero di transazioni verso la memoria esterna (la global), considerando la velocità ridotta di quest'ultima.

L'approccio utilizzato è quello di caricare il massimo numero di dati attigui con il minor numero possibile di transazioni.

Il meccanismo di coalescenza lavora al livello di half-warp, cioè di 16 thread facenti parti dello stesso gruppo di 32 thread tenuti in esecuzione per ciascun ciclo di clock su un multiprocessore. A seconda della compute capability supportata dal dispositivo, cambiano anche le linee guida da seguire nell'organizzare gli accessi di un half warp per ottenere la loro coalescenza.

Per i chip grafici con compute capability 1.0 o 1.1, e cioè le soluzioni derivate da G80 e G92 sopra elencate, gli accessi di un half-warp sono coalescenti se leggono un'area contigua di memoria di diverse dimensioni (64 byte per thread per leggere una word, 128 per una double e 256 per una quad).

Ci sono inoltre una serie di restrizioni supplementari alla semplice adiacenza:

- L'indirizzo iniziale di una regione deve essere multiplo della grandezza della regione;
- Gli accessi devono cioè essere perfettamente allineati tra i thread, ovvero il k-esimo thread di un half-warp deve accedere al k-esimo elemento di un blocco (letto o scritto);

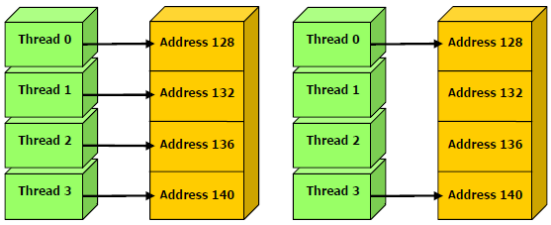

Figura 4.5: Esempi di coalescenza. Per motivi grafici sono mostrati solamente quattro thread

Tuttavia gli accessi rimangono coalescenti se alcuni thread non partecipano (cioè non eseguono la lettura o la scrittura in esame).

Nel caso di compute capability 1.2 e 1.3 (supportata da tutti i chip derivati da GT200), le condizioni per ottenere la coalescenza degli accessi sono più lasche. In particolare, una singola transazione di memoria è eseguita per un half warp se gli accessi di tutti i thread sono compresi all'interno dello stesso segmento di grandezza uguale a:

- 32 byte, se tutti i thread accedono a word di 8 bit;
- 64 byte, se tutti i thread accedono a word di 16 bit;
- 128 byte, se tutti i thread accedono word di 32 o 64 bit.

La coalescenza degli accessi in una singola transazione, se essi risiedono nello stesso segmento come specificato, è ottenuta per tutti gli schemi di indirizzi richiesti dall'half warp, inclusi anche schemi dove più thread accedono allo stesso

indirizzo. Se, invece, un half-warp indirizza parole in segmenti differenti, saranno eseguite tante transazioni quanti sono i segmenti indirizzati.

L'algoritmo che si occupa di lanciare le transazioni, cerca anche di ridurre, se possibile, la loro grandezza. Infatti, se in una transazione da 128 byte sono contenuti solo nella metà alta o bassa del segmento (per esempio, dati tutti a 32 bit allineati, ma anche pattern più complessi ma in un blocco di dimensioni inferiori), allora la grandezza della transazione verrà ridotta a 64 byte.

Lo stesso accade nel caso di transazioni a 64 byte che possono essere contenute in transazioni da 32 byte, anche quando esse siano già il risultato di una riduzione da una transazione a 128 byte. La suddivisione in segmenti parte, come immaginabile, da multipli interi della grandezza del segmento stesso. Dunque, un segmento a 128 byte parte da 0, 128, 256, ... e, nel caso della riduzione della gandezza della transazione a 64 byte, i dati devo rientrare nella sua metà alta o bassa, dunque di fatto partire da multipli interi di 64.

4.3 Algoritmi

In questa sezione passiamo ad analizzare due algoritmi implementati in CUDA: la somma e l'ordinamento bitonico. Per il primo si procederà a spiegare i singoli passaggi, motivando le scelte implementative al lettore in maniera approfondita.

4.3.1 Somma

Introduzione

L'algoritmo di somma è un buon esempio di *parallel reduction*

Definizione 4.2 *Definiamo parallel reduction la famiglia di algoritmi che dato in input un dataset di taglia n ne rendono un sottoinsieme di taglia m con* $m \leq n$ *effettuando dei confronti tra gli elementi con operazioni elementari.*

Tramite questo esempio si vedranno quindi le applicazioni dei pattern di programmazione illustrati nei capitoli precedenti, osservando il loro impatto sulle esecuzioni.

Seriale

La somma di n numeri può essere implementata in maniera banale in seriale. In tal caso si ha complessità computazionale $\Theta(n)$. Il codice sviluppato è riportato in appendice B.0.1, si tratta di un programma che crea in C++ un array di numeri interi casuali e li somma in maniera sequenziale.

CUDA

Per l'implementazione parallela introduciamo invece qualche vincolo. La somma del vettore (denominata anche riduzione vettoriale) verrà utilizzata con elaborazioni in-memory sulla shared memory.

I dati originali risiedono sulla memoria globale, e il risultato finale verrà memorizzato nell'elemento di indice 0. Supponiamo inoltre di effettuare chiamate ricorsive, e che ogni chiamata potrà essere contenuta in una SM distinta.

Un primo approccio può essere quello illustrato nel listato 4.6

Listing 4.6: Kernel con Branch Divergence.

```
1  __global__ void reduce0(int *g_idata, int *g_odata) {
2          extern __shared__ int sdata[];
3          // ogni thread elabora un solo elemento dalla global alla shared
4          unsigned int tid = threadIdx.x;
5          unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
6          sdata[tid] = g_idata[i];
7          __syncthreads();
8          // elaborazione nella shared
9          for(unsigned int s=1; s < blockDim.x; s *= 2) {
10                 if (tid % (2*s) == 0) {
11                         sdata[tid] += sdata[tid + s];
12                 }
13         __syncthreads();
14         }
15         // scrivo i dati nella global
16         if (tid == 0) g_odata[blockIdx.x] = sdata[0];
17 }
```

Questa implementazione ha due svantaggi; il primo è l'utilizzo dell'operatore %, caratterizzato dalle basse performance, il secondo è la creazione di più thread di quelli necessari, come si vedrà in seguito.

Passiamo quindi ad analizzare il primo problema, ovvero la scelta di come effettuare le chiamate. La prima miglioria da introdurre è la modifica dell'ope-

ratore di resto e la clausola di if. Tale clausola va' difatti ottimizzata, dato che obbligherà, almeno metà dei thread creati a non effettuare elaborazioni.

I thread difatti verranno creati con indici crescenti, ma al più metà di essi potranno soddisfarre la condizione (essere multipli di 2^i alla i-esima iterazione).

Introduciamo quindi il codice 4.7 che risolve questi problemi.

Listing 4.7: Kernel con Bank Conflict.

```
1  for (unsigned int s=1; s < blockDim.x; s *= 2) {
2      int index = 2 * s * tid;
3      if (index < blockDim.x) {
4          sdata[index] += sdata[index + s];
5      }
6      __syncthreads();
7  }
```

Il codice è tuttavia ancora affetto da Bank Conflict per le sue prime iterazioni, poichè ogni thread somma ogni numero di indice tid con il numero di indice $tid+2^i$.

Per minimizzare questo fenomeno possiamo, quindi, sostituire il ciclo interno con uno basato sull'indice del thread (listato 4.8). Questa scelta, difatti, permette al programma di accedere a banchi di memoria differenti ogniqualvolta lo *stride* è maggiore della dimensione del blocco.

Listing 4.8: Kernel senza Bank Conflict.

```
1  for (unsigned int s=blockDim.x/2; s>0; s>>=1){
2      if (tid < s){
3          sdata[tid] += sdata[tid + s];
4      }
5      __syncthreads();
6  }
```

Nel listato 4.8 non è ancora ottimizzata per l'utilizzo dei thread. Questo perchè metà dei thread a blocco non vengono sfruttati. Possiamo quindi dimezzare la taglia del blocco ed effettuare una somma al volo durante il caricamento da memoria globale, come da listato 4.9.

Listing 4.9: Somma on-fly.

```
1    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
2    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
```

I warp che non effettueranno elaborazioni verranno chiusi man mano che non sono più richiesti. Per blocchi di 512 thread dopo quattro iterazioni si avrà un solo warp attivo per blocco. Abbiamo quindi comunque un'utilizzo poco efficiente delle risorse, dato che alla quarta iterazione per ogni warp ci sarà solamente un thread attivo. Questo significa che nel caso lo *stride* sia inferiore alla lunghezza del warp allora l'istruzione di sincronizzazione __syncthreads diventa obsoleta come la clausola di *if* $tid < s$, poichè si conosce la risoluzione. Si può quindi applicare l'**unroll** del ciclo, esplicitando le istruzioni come nel listato 4.10

Listing 4.10: Kernel con unroll dell'ultimo warp

```
1   for (unsigned int s=blockDim.x; s>32; s>>=1)
2   {
3       if (tid < s)
4           sdata[tid] += sdata[tid + s];
5       __syncthreads();
6   }
7   if (tid < 32)
8   {
9       sdata[tid] += sdata[tid + 32];
10      sdata[tid] += sdata[tid + 16];
11      sdata[tid] += sdata[tid + 8];
12      sdata[tid] += sdata[tid + 4];
13      sdata[tid] += sdata[tid + 2];
14      sdata[tid] += sdata[tid + 1];
15  }
```

.

Possiamo rendere il tutto parametrico con un template in linguaggio C, rendendo le variabili in fase di compilazione. Per il sorgente di questa opzione riferirsi all'appendice B.1.

Ultima miglioria applicabile è un'estensione del listato 4.9 nel quale si è introdotta la somma al volo di due elementi.

Poichè le riduzioni hanno scarsa intensità aritmetica abbiamo che il collo di bottiglia in questo caso è dovuto ai trasferimenti host-device. Sfruttando quindi il meccanismo di coalescenza[3] possiamo accedere ad un numero di indirizzi molto

[3] sez4.2.1

maggiore, a patto che ognuno di essi sia su di un banco differente (listato 4.11).

Listing 4.11: Kernel Ottimizzato

```
1        while (i < n) {
2                sdata[tid] += g_idata[i] + g_idata[i+blockSize];
3                i += gridSize;
4        }
```

.

Benchmark

Si sono effettuati dei benchmark per i casi seriali e paralleli su di una macchina di fascia media, montante un processore Intel T7500 (dualcore, versione da 2,2 Ghz) e una scheda video G80. Per ogni test si sono effettuate due batterie da 500 test consecutivi, e si è riportato il valore medio.

Tabella 4.2: Benchmark per Kernel Cuda

	Tempo [ms]	Banda [GB/s]	Speed Up
Caso Seriale	23.356	–	
Branch Divergence	8.054	2.083	2.90X
Bank Conflict	3.456	4.854	6.76X
Senza Bank Conflict	1.722	9.741	12.56X
Con somma iniziale	0.965	17.377	34.86X
Unroll del Warp	0.536	31.289	62.75X
Unroll completo	0.381	43.996	88.27X
Versione Finale	0.281	62.671	119.69X

Alcune considerazioni sono innanzitutto notare come anche nel caso peggiore l'utilizzo della GPU permetta un triplicamento delle prestazioni. Ogni passaggio

inoltre porta dei vantaggi nei termini prestazionali. Possiamo inoltre considerare come la banda del device (80 Gb/s) non sia stata completamente utilizzata. Implementazioni più spinte potrebbero utilizzare al pieno la potenza elaborativa.

4.3.2 Algoritmo di Sorting

In questa sezione si tratteranno due algoritmi di ordinamento, per il caso seriale il Randomized Quick Sort, per il parallelo il Bitonic Sort. Entrambi gli algoritmi sono, rispettivamente per il seriale e il parallelo, considerati con le migliori performance

Randomized Quick Sort

Dato un dataset D si n elementi si applica un algoritmo ricorsivo. Si sceglie casualmente un elemento di indice i, con $1 \leq i \leq n$ chiamato *pivot*. Si determinano due dataset, A e B, e, ad ogni iterazione si effettuerà uno scan del dataset e si porranno gli elementi minori di $D[i]$ nel dataset A, e quelli maggiori in B. Poichè non si hanno garanzie che il pivot sia l'elemento mediano normalmente A e B avranno taglie diverse.

L'algoritmo poi viene richiamato ricorsivamente su A e B; tali risultati verranno successivamente concatenati.

L'algoritmo richiede mediamente una complessità temporale di $\Theta(n \log n)$ e una spaziale di $\Theta(1)$.

Bitonic Sorting

Il Bitonic Sorting esegue una serie di confronti tra di loro indipendenti su di una sequenza di $n = 2^m$ elementi.

Definizione 4.3 *Definiamo una* ***sequenza bitonica*** *una sequenza* $a = (a_1, \ldots, a_n)$ *una sequenza per cui è possibile identificare un indice* $k \in 1, \ldots, n$*; per cui* $a_1, \ldots, a_k$ *sia monotona crescente (decrescente) e* $a_k, \ldots, a_n$ *sia monotona decrescente (crescente).*

Si può dimostrare che

Teorema 4.1 *Data una sequenza bitonica* A *di* $2n$ *elementi, le sottosequenze* A_0 *e* A_1 *ottenute come*

$$\begin{aligned} A_0 &= \{\min\{a_0, a_n\}, \ldots, \min\{a_i, a_{i+n}\}, \ldots, \min\{a_n, a_{2n}\}\} \\ A_1 &= \{\max\{a_0, a_n\}, \ldots, \max\{a_i, a_{i+n}\}, \ldots, \max\{a_n, a_{2n}\}\} \end{aligned} \tag{4.1}$$

Sfruttando il teorema 4.1 si può esprimere l'algoritmo in forma ricorsiva in due fasi.

L'idea è quella di ottenere nella prima metà una sequenza monotonica crescente, e nella sequenza una monotonica decrescente. L'unione delle due rende quindi una bitonica. Applicando poi il merge bitonico ricorsivamente otteremo due sequenze bitoniche nelle quali tutti gli elementi della prima sono inferiori a quelli della seconda.

Nella prima fase si applica la ricorsione con un approccio divide-and-conquer, ovvero, dati gli n elementi si separano le due sequenze monotoniche di lunghezza $n/2$ come da codice 4.12. Lo scopo è quello di creare delle sequenze bitoniche di lunghezza man mano crescente

Nella seconda fase, invece, si uniscono i risultati così ottenuti. Per farlo si confrontano gli elementi delle due metà, se sono fuori posto si scambiano e si procede in maniera ricorsica a fondere le due metà.

L'algoritmo risultante ha quindi una complessità temporale di $\Theta(\log(n^2))$, supponendo di avere un grado di parallellismo pari ad $n/2$

Listing 4.12: bitonicSort

```
1  private void bitonicSort(int start, int n_el, boolean dir)
2      {
3          if (n_el>1)
4          {
5              n_el/=2;
6              bitonicSort(start, n_el, ASCENDING);
7              bitonicSort(start+n_el, n_el, DESCENDING);
8              bitonicMerge(start, 2*n_el, dir);
9          }
10     }
```

Implementazione Bitonic Sort

In questo paragrafo si illustrerà l'implementazione parallela sviluppata. L'algoritmo parallelo viene invocato dalle seguenti istruzioni poste nel main. La terza riga chiama il metodo con un unico blocco, specificando che dovrà essere composto da 512 thread al suo interno, forzandone le dimensione a singoli interi.

Il metodo bitonicSort prende in input un array di *value* e contiene il seguente codice

In questo esempio si nota l'approccio differente. Per prima cosa il thread calcola il proprio identificativo, che servirà anche da indice, e lo memorizza in *tid* dall'elemento globale al condiviso. Si aspetta che tutti i thread effettuino la copia dopodichè ogni thread lavora in maniera indipendente dagli altri un'esecuzione alla volta. Dopo ogni ciclo di possibile swap i thread difatti aspettano di sincronizzarsi per procedere.

Listing 4.13: bitonicMerge

```
1  private void bitonicMerge(int start, int n_el, boolean dir)
2      {
3          if (n_el>1)
4          {
5              n_el/=2;
6              for (int i=start; i<start+n_el; i++)
7                  compare(i, i+n_el, dir);
8              bitonicMerge(start, n_el, dir);
9              bitonicMerge(start+n_el, n_el, dir);
10         }
11     }
```

Poiché l'algoritmo non potrà essere contenuto all'interno di un unico blocco, dato che gli indici interrogati sono sparsi lungo tutta la lunghezza dal campo d'indirizzamento si è scelto di lavorare in memoria globale.

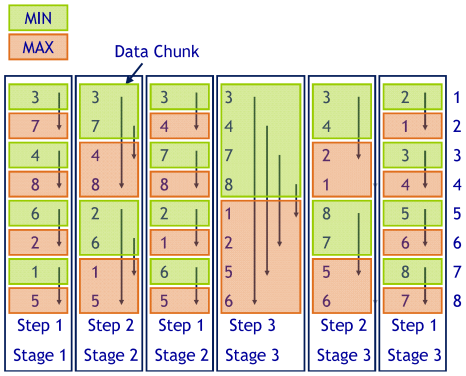

Figura 4.6: I passaggi dell'ordinamento bitonico

Benchmark

Per il confronto tra questi due algoritmi si è deciso di effettuare un'analisi riguardo le prestazioni al crescere alle dimensioni del dataset.

Tabella 4.3: Benchmark per Sorting

DataSet [int]	Tempo Seriale [ms]	Tempo Cuda [ms]	Speed Up
16384	1	0	–
32768	3	0	–
65536	6	1	6X
262144	27	4	7X
524288	46	5	9X
1048576	139	7	20X
2097152	311	10	31X
4194304	720	14	51X

Listing 4.14: chiamata di Bitonic Sort

```
1  ...
2  extern __shared__ int shared[];
3  shared[tid] = values[tid];
4  __syncthreads(); //attendo che tutti abbiano finito
5  for (unsigned int k = 2; k <= NUM; k *= 2)
6  {
7          for (unsigned int j = k / 2; j>0; j /= 2)
8      {
9          unsigned int ixj = tid ^ j;
10        if (ixj > tid)
11          if ((tid & k) == 0)
12                  if (shared[tid] != shared[ixj])
13                  swap(shared[tid], shared[ixj]);
14                  __syncthreads(); //attendo che tutti abbiano finito
15          }
16 }
17 values[tid] = shared[tid];
```

All'aumentare della taglia le performance iniziano a discostarsi come dalla teoria. Non sono stati eseguiti test con più di 4 milioni di elementi perché avrebbero comportato una diversa implementazione CUDA. Esistono inoltre tecniche di mappatura del dataset che permettono di innalzare ulteriomente le performance.

4.3.3 Applicazioni GPU

Crittografia

La crittografia è suddivisa in due famiglie: a chiave simmetrica (AES) e assimmetrica (RSA). Attualmente il protocollo maggiormente utilizzato (e considerato sicuro dal dipartimento di stato americano) è la codifica RSA a 1024 bit.

L'RSA ha delle chiavi pubbliche e private. Le chiavi pubbliche, utilizzate per codificare i messaggi sono note a tutti, le private solo ai destinatari. Esse vengono calcolate prendendo il prodotto $n = pq$ con p e q numeri primi. A questo punto si calcola la funzione ϕ di Eulero ($\phi(pq) = (p-1)(q-1)$). Si sceglie poi un coprimo d minore di $\phi(pq)$ tale che $de = 1(mod\phi(n))$.

Se Bob deve comunicare con Alice nel protocollo RSA il messaggio M si fa comunicare da Alice la coppia di chiavi (n, e) e le il crittografato c calcolato secondo la formula

$$M^e \equiv c(modn)$$

. Alice per decodificare il messaggio dovrà utilizzare la chiave d.

La sicurezza dell'algoritmo è di tipo computazionale, risiede difatti che non esiste un algoritmo noto che calcoli il logaritmo modulare (l'operazione necessaria per decodificare il messaggio senza la chiave d) senza ricorrere al *brute force*.

Con le schede Nvidia si sono sviluppati quindi sia algoritmi di codifica sia di attacco all'algoritmo RSA.

Harrison e Waldron(13) hanno sviluppato una serie di algoritmi di codifica che migliorano di un fattore 4 le performance di codifica come da figura 4.7 Esistono poi lavori esterni alla comunità scientifica di attacchi al protocollo(19) che permettono, nella versione senza dizionario, incrementi un due ordini di grandezza.

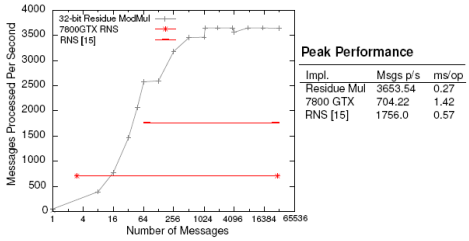

Figura 4.7: Benchmark di codifica RSA

Database

Le GPU possono essere utilizzate in ambito aziendale per la gestione di database di grandi dimensioni. Le operazioni che richiedono mediamente il maggiore utilizzo della macchina sono gli ordinamenti e le operazioni di join.

Entrambe queste operazioni sono parallelizzabili per loro natura, la prima come si è visto con il bitonic sort, la seconda basandosi su un ordinamento e dividendo i risultati tramite dei bucket indicizzati.

Solitamente le elaborazioni su database avvengono sulla memoria esterna. Gli ordinameti esterni si possono dividere in due famiglie principali: quelli *Distribution Based* e quelli *Merge Based*. I primi creano n bucket B in cui ogni elemento b_{B_i} sia minore di ogni elemento $c_{B_{i+1}}$, ordinano i singoli bucket e rendono la concatenazione $B_0, B_1, \cdot, B_n$, i secondi creano n bucket casuali, ordinano ogni bucket e successivamente rendono il merge (ottenibile in $\Theta(n)$).

Per rendere al meglio i bucket devono avere una taglia confrontabile con la taglia della memoria interna.

Le analisi mostrano come il collo di bottiglia di questi algoritmi siano le operazioni di I/O. Nello specifico ci sono tre cause principali che ne degradano le performance.

1. **Cache miss** poichè la memoria interna non è molto vasta bisogna trovare un trade off tra la taglia del bucket e i cache miss che si ottengono nelle cache L1, L2, L3.

2. **Perfomance I/O** gli algoritmi di sorting hanno una grande intensità aritmetica, ma non massimizzano la banda dell'applicazione

3. **Mancato Parallellismo** come si è detto precedentemente le operazioni presentano un forte grado di parallelismo che non viene sfruttato né come numero di processori nè con tecniche di pipeline.

Govindaraju et all (9) hanno sviluppato un motore per database che utilizza la sinergia tra CPU e GPU. Con un approccio molto simile al linguaggio CUDA (parti seriali demandate al processore, elaborazioni alla GPU).

L'algoritmo risultante (basato sul bitonic sort) ha una complessità computazionale di $O(\frac{n \log^2 n}{2})$ e una richiesta di banda di $O(n)$. Questo risultato dimostra come la banda non sia più un collo di bottiglia (nei benchmark si raggiungono i 49 GB/sec, circa il 90% della banda teorica).

Come risultati operativi si è ottenuto che una piattorma di tipo desktop vede le proprie performance raddoppiate e equiparabili a quelle delle piattaforme server di tipo Xeon (vedi fig 4.8).

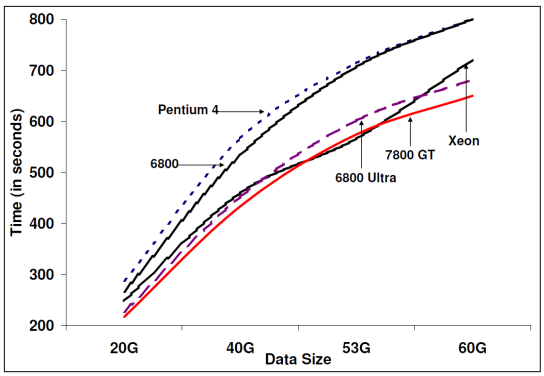

Figura 4.8: Benchmark di GPUTeraSoft

Capitolo 5

Conclusioni

In questo elaborato abbiamo prima trattato la struttura fisica delle schede video e le tecniche che sono implementate via hardware. Abbiamo poi analizzato lo stream programming model, i linguaggi di shading e studiato il linguaggio di ultima generazione CUDA con i relativi benchmark.

Alcuni spunti per futuri sviluppi sono sicuramente basati sulla terza edizione del linguaggio, come pure dall'affermazione di linguaggi non proprietari come OpenGL.

Bibliografia

[1] Jeroen Bedorf. *N-body simulations on graphics processing units II: An implementation in CUDA*. Tesi di Dottorato di Ricerca, Universiteit van Amsterdam, novembre 2007.

[2] David Blythe. Rise of the graphics processor. *IEEE No. 5*, 96:761 – 778, maggio 2008.

[3] Russ Brown. Barycentric coordinates as interpolants. *IEEE*, 1999.

[4] Daniel Cohen-Or, Yiorgos Chrysanthou, Claudio Silva, e Fredo Durand. A survey of visibility for walkthrough applications. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 9(3), luglio 2003.

[5] R. L. Cook. Shade trees. *Computer Graphics , SIGGRAPH 1984 Proceedings*, 18:223/231, Luglio 1984.

[6] R. Fernando e M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003.

[7] S. Fleissner. Gpu-accelerated montgomery exponentiation. In *Computational Science ICCS 2007, 7th International Conference*, maggio 2007.

[8] Peter Geldof. *High performance direct gravitational N-body simulation of Graphica Processing Units.* Tesi di Dottorato di Ricerca, Universiteit van Amsterdam, maggio 2007.

[9] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, e Dinesh Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. Relazione tecnica, Microsoft Technical Report, 2005.

[10] K. Gray. *The Microsoft DirectX 9 Programmable Graphics Pipeline.* Microsoft Press, 2003.

[11] P. Hanrahan e J. Lawson. A language for shading and lighting calculations. In *Computer Graphics SIGGRAPH 1990 Proceedings*, volume 24, p. 289, agosto 1990.

[12] Zhang Hansong. *'Effective Occlusion Culling for the Interactive Display of Arbitrary Models.* Tesi di Dottorato di Ricerca, Dept. of Computer Science, Univ. of North Carolina-Chapel Hill, 1998.

[13] O. Harrison e J. Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. *Africacrypt 2009*, luglio 2009.

[14] E Kilgariff e R Fernando. *GPU Gems 2*, volume The GeForce 6 series GPU architecture. Addison Wesley, 2005.

[15] A. Lee e G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, gennaio 1987.

[16] J. Lengyel, M. Reichert, B. R. Donald, e D. P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics , SIGGRAPH 1990 Proceedings*, 24:327/335, Agosto 1990.

[17] Brandon Lloyd. Slide del corso computer graphics - comp 770 (236). University of North Carolina, 2007.

[18] David Luebke e Greg Humphreys. How gpus work. *Computer Graphics*, 40(2):96 – 100, Febbraio 2007.

[19] Lueg Lukas. Home page del progetto pyrit, dicembre 2009.

[20] Masatoshi Niizeki e Fujio Yamaguchi. Projectively invariant intersection detections for solid modeling. *ACM Transactions on Graphics*, 13(3), 1994.

[21] Marc Olano e Trey Greer. Triangle scan conversion using 2d homogeneous coordinates. *SIGGRAPH 1997 /Eurographics Workshop on Graphics Hardware*, 1995.

[22] John Owens. Streaming architectures and technology trends. *GPU Gems 2*, pp. 457 – 470, marzo 2005.

[23] John Owens. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, Marzo 2007.

[24] John Owens, Mike Houston, David Luebke, Simon Green, John Stone, e James Phillips. Gpu computing. *IEEE No. 5*, 96:879 – 899, 2008.

[25] John D. Owens. *Computer Graphics on a Stream Architecture*. Tesi di Dottorato di Ricerca, University of Standford, 2002.

[26] K. Proudfoot, W. R. Mark, S. Tzvetkov, e P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics SIGGRAPH 2001 Proceedings*, volume 35, p. 159, agosto 2001.

[27] Mark William, Glanville Robert, Akeley Kurt, e Kilgard Mark. Cg: A system for programming graphics hardware in a c-like language. In *Computer Graphics SIGGRAPH 2003 Proceedings*, volume 37, p. 896, luglio 2003.

[28] W. Wilson, Ivan Fung, e George Sham. Dynamic warp formation and scheduling for efficient gpu control flow. In *40th IEEE/ACM International Symposium on Microarchitecture*, dic 2007.

Appendice A

Richiami di Geometria Omogenea

Un **punto** in $\Re^3$, $P = (X, Y, Z)$ può essere rappresentato in coordinate omogenee da un vettore di quattro elementi $p = (x, y, z, w = 1)$. Qualsiasi multiplo non nullo di questo vettore rappresenta il medesimo punto tridimensionale, quindi per convenzione ci si riporta nel caso $w = 1$ Le quadruple di coordinate omogenee con $w = 0$, cioe' della forma $p = (x, y, z, 0)$ identificano punti all'infinito nella direzione del vettore (x, y, z).

Allo stesso modo per lo spazio $\Re^2$ abbiamo le coordinate non omogenee $P = (X, Y)$, e omogenee, $p = (x, y, w)$.

Per convertire da una rappresentazione omogenea ad una non omogenea si divide ogni componente per w, ovvero $P = (X = x/w, Y = y/w)$. Questa operazione è chiamata la proiezione di un punto omogeneo. Viceversa per effettuare la conversione opposta si aggiunge il terzo elemento, ottenendo $p = (X, Y, 1)$.

Una **retta** in $\Re^2$ è definibile da due punti $p_0 = (x_0, y_0, 1)$ e $p_1 = (x_1, y_1, 1)$ come il prodotto

$$
\begin{aligned}
e &= \dot{p_o} \times \dot{p_1} \\
&= \begin{bmatrix} x_0 & y_0 & 1 \end{bmatrix}^T \times \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} && \text{(A.1)} \\
&= \begin{bmatrix} y_0 - y_1 & x_1 - x_0 & x_0 y_1 - x_1 y_0 \end{bmatrix} && \text{(A.2)} \\
&= \begin{bmatrix} A & B & C \end{bmatrix} && \text{(A.3)}
\end{aligned}
$$

Ottenendo $E(x, y) = Ax + By + C$. La **edge function** per un dato punto p_i è soddisfatta se $E(x_i, y_i) > 0$ La retta suddivide quindi il piano in due semipiani, positivo e negativo, dipendentemente se la funzione di linea è soddisfatta o meno.

Un **triangolo** invece può esser definito(20) nello spazio omogeneo come l'insieme dei punti che possono esser espressi come combinazione lineare dei suoi vertici

$$p = \lambda_0 p_0 + \lambda_1 p_1 + \lambda_2 p_2$$

con $\sum_i \lambda_i = 1$. Se tutti i λ_i hanno lo stesso segno si parla di triangolo interno, altrimenti di triangolo esterno [1].

un **poligono** invece viene definito come l'intersezione dei semipiani negativi delle edge function.

Un **piano** in $\Re^3$ può esser espresso in coordinate omogenee come

$$Ax + By + Cz + D = 0 \rightarrow H = (A, B, C, D)$$

Se A,B,C sono normalizzate possiamo verificare come

$$d = H \cdot p = H^T p$$

[1] Olano et all hanno sviluppato approfonditamente i triangoli esterni (21)

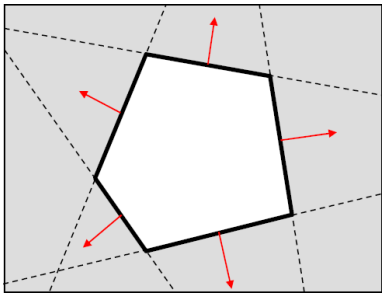

Figura A.1: Un poligono

sia una misura di distanza con segno. Se si vuole sapere se il segmento p, q attraversa il piano H bisognerà verificare se

$$(H^T p) * (H^T q) < 0$$

possiamo notare come queste operazioni siano facilmente implementabili in hardware.

A.1 Trasformazioni Omogenee

Le trasformazioni affini in uno spazio euclideo sono le trasformazioni del tipo

$$x \Rightarrow Ax + b$$

ovvero esprimibili come la composizione di una trasformazione lineare determinata da una matrice A e di una traslazione determinata da un vettore b.

Queste trasformazioni mantengono le proporzioni tra rette ma non necessariamente quelle tra angoli e distanze.

Traslazioni

Per traslare un punto si utilizza il vettore di traslazione(tx, ty, tz). Ovvero nello spazio non omogeneo se $P = (X, Y, Z)$ e' un punto, il punto traslato e' $P' = (X + tx, Y + ty, Z + tz) = (X, Y, Z) + (tx, ty, tz)$.

In coordinate omogenee e forma matriciale:

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

La traslazione inversa corrisponde al vettore di traslazione inverso $(-tx, -ty, -tz)$.

Scalatura

Per scalatura di un oggetto si intende scegliere un punto C che rimane fermo e riposizionare ciascun punto dell'oggetto ricalcolando le sue distanze da C (nella direzione di ciascun asse coordinato) in base ad una nuova unità di misura. Adesso vediamo scalatura con punto fermo l'origine.

Si definiscono i fattori di scala (sx, sy, sz), uno per ogni asse del sistema di riferimento.

Se $P = (X, Y, Z)$ è un punto, il punto scalato è $P' = (sx * X, sy * Y, sz * Z)$.

In coordinate omogenee e forma matriciale

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

La scalatura tenendo fermo un punto generico C diverso dall'origine si ottiene come combinazione di trasformazioni. Se i fattori di scala sono negativi, si ottiene una riflessione per quegli assi.

Rotazione

Per ruotare un oggetto si sceglie un asse di rotazione (una retta r) ed un verso di rotazione, e muovere ciascun punto dell'oggetto solidalmente dello stesso angolo attorno ad r nel verso assegnato (anzichè una retta ed un verso di rotazione, posso dare una retta orientata dato che il verso di rotazione e' quello che appare antiorario guardando nella direzione opposta a quella della retta).

Adesso vediamo rotazione attorno ad uno degli assi coordinati, per esempio l'asse z.

Come parametri si ha l'angolo di rotazione α ($\alpha > 0$ per senso antiorario, $\alpha < 0$ per senso orario).

Se $P = (X, Y, Z)$ e' un punto, il punto ruotato è $P' = (X', Y', Z')$ dove:

$$X' = X * \cos(\alpha) - Y * \sin(\alpha) Y' = X * \sin(\alpha) + Y * \cos(\alpha) Z' = Z$$

In coordinate omogenee e forma matriciale:

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Analogamente le matrici di rotazione per rotazione attorno all'asse x ed y sono uguali all'identita' tranne per le righe e colonne corrispondenti alle coordinate yz e xz.

La matrice di rotazione per rotazione attorno ad una retta generica passante per l'origine ha forma piu' complicata, che non vediamo.

La rotazione attorno ad una retta non passante per l'origine si ottiene come combinazione di trasformazioni (ved. dopo).

A.2 Composizione di trasformazioni

Se si vogliono eseguire più trasformazioni in cascata si può sviluppare il tutto come una serie di moltiplicazioni. Siano A e B due matrici di trasformazioni affini. Si ottiene che

$$P' = AP$$

$$P'' = BP'$$

$$P'' = B(AP) = (BA)P$$

Come la moltiplicazione di matrici, in generale non è commutativa. La composizione di trasformazioni dello stesso tipo è commutativa (traslazioni con traslazioni, scalature con fermo lo stesso punto, rotazioni attorno allo stesso asse).

Appendice B

Codici Sorgenti

B.0.1 Somma - Seriale

```
1  package sum;
2  public class Main {
3      public static final int n_elements=15000000;
4
5      public static void main(String[] args) {
6          int[] data=new int[n_elements];
7          for (int i=0;i<n_elements;i++)
8              data[i]=(int)(Math.random() * 100);
9          double tot=0;
10         for (int i=0;i<n_elements;i++)
11             tot+=data[i];
12         System.out.println(tot);
13     }//main
14 }//class
```

B.0.2 Sorting - Serial

```
#include<process.h>
#include<iostream>
#include<conio.h>
#include<stdlib.h>
#include <time.h>
using namespace std;
int Partition(int low,int high,int arr[]);
void Quick_sort(int low,int high,int arr[]);
void main(){
        int *a,n,low,high,i;
        n=100000000;
        a=new int[n];
        for(i=0;i<n;i++)
                a[i]=rand();
        int clo = clock();
        high=n-1;
        low=0;
        Quick_sort(low,high,a);
        cout << (clock() - clo) << endl;
        getch();
}
int Partition(int low,int high,int arr[]){
        int i,high_vac,low_vac,pivot/*,itr*/;
        pivot=arr[low];
        while(high>low)
        {
                high_vac=arr[high];
```

```
28                  while(pivot<=high_vac)
29                  {
30                          if(high<=low) break;
31                          high--;
32                          high_vac=arr[high];
33                  }
34                  arr[low]=high_vac;
35                  low_vac=arr[low];
36                  while(pivot>low_vac)
37                  {
38                          if(high<=low) break;
39                          low++;
40                          low_vac=arr[low];
41                  }
42                  arr[high]=low_vac;
43          }
44          arr[low]=pivot;
45          return low;
46  }
47  void Quick_sort(int low,int high,int arr[]){
48    int Piv_index,i;
49    if(low<high)
50    {
51     Piv_index=Partition(low,high,arr);
52     Quick_sort(low,Piv_index-1,arr);
53     Quick_sort(Piv_index+1,high,arr);
54    }
55  }
```

B.0.3 Somma - CUDA

Nella sezione è riportato solamente il codice del kernel, il main è omesso

Listing B.1: Kernel di Somma.

```
template <unsigned int blockSize>
__global__ void reduce(int *g_idata, int *g_odata, unsigned int n)
{
        extern __shared__ int sdata[];
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x*(blockSize*2) + tid;
        unsigned int gridSize = blockSize*2*gridDim.x;
        sdata[tid] = 0;
        while (i < n) {
                sdata[tid] += g_idata[i] + g_idata[i+blockSize];
                i += gridSize;
        }
        __syncthreads();
        if (blockSize >= 512) {
                if (tid < 256) {
                        sdata[tid] += sdata[tid + 256];
                }
                __syncthreads();
        }
        if (blockSize >= 256) {
                if (tid < 128) {
                        sdata[tid] += sdata[tid + 128];
                }
                __syncthreads();
```

```
25          }
26          if (blockSize >= 128) {
27                  if (tid < 64) {
28                          sdata[tid] += sdata[tid + 64];
29                  }
30                  __syncthreads();
31          }
32          if (tid < 32) {
33                  if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
34                  if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
35                  if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
36                  if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
37                  if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
38                  if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
39          }
40          if (tid == 0) g_odata[blockIdx.x] = sdata[0];
41  }
```

B.0.4 Sorting - CUDA

bitonic.cu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cutil_inline.h>
4  #include "bitonic_kernel.cu"
5
6  int main(int argc, char** argv)
7  {
```

```
8     if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") )
9                 cutilDeviceInit(argc, argv);
10        else
11                cudaSetDevice( cutGetMaxGflopsDeviceId() );
12    int values[NUM];
13    for(int i = 0; i < NUM; i++)
14        values[i] = rand();
15    int * dvalues;
16    cutilSafeCall(cudaMalloc((void**)&dvalues, sizeof(int) * NUM));
17    cutilSafeCall(cudaMemcpy(dvalues, values, sizeof(int) * NUM,
18                    cudaMemcpyHostToDevice));
19    bitonicSort<<<1, NUM, sizeof(int) * NUM>>>(dvalues);
20    // check for any errors
21    cutilCheckMsg("Kernel execution failed");
22    cutilSafeCall(cudaMemcpy(values, dvalues, sizeof(int) * NUM,
23                     cudaMemcpyDeviceToHost));
24    cutilSafeCall(cudaFree(dvalues));
25    bool passed = true;
26    for(int i = 1; i < NUM; i++)
27        if (values[i-1] > values[i])
28            passed = false;
29    printf( "Test %s\n", passed ? "PASSED" : "FAILED");
30    cudaThreadExit();
31    cutilExit(argc, argv);
32 }
```

bitonic_kernel.cu

```
#ifndef _BITONIC_KERNEL_CU_
#define _BITONIC_KERNEL_CU_
#define NUM 256
__device__ inline void swap(int & a, int & b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

__global__ static void bitonicSort(int * values)
{
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    shared[tid] = values[tid];
    __syncthreads();
   //Algoritmo Parallelo
   for (unsigned int k = 2; k <= NUM; k *= 2)
    {
        //merge:
        for (unsigned int j = k / 2; j>0; j /= 2)
        {
            unsigned int ixj = tid ^ j;
            if (ixj > tid)
            {
                if ((tid & k) == 0)
                {
```

```
28                      if (shared[tid] > shared[ixj])
29                      {
30                          swap(shared[tid], shared[ixj]);
31                      }
32                  }
33                  else
34                  {
35                      if (shared[tid] < shared[ixj])
36                      {
37                          swap(shared[tid], shared[ixj]);
38                      }
39                  }
40              }
41              __syncthreads();
42          }
43      }
44      //copio i valori
45      values[tid] = shared[tid];
46  }
47  #endif // _BITONIC_KERNEL_H_
```

Elenco delle tabelle

Elenco delle figure

Ringraziamenti

Sebbene stia scrivendo all'interno di un cuore
(erano 6 anni che lo volevo dire) rin-grazio di tutto cuore la mia famiglia,
che mi ha sempre supportato in tutte le av-venture più o meno riuscite di questi anni, e che è riuscita a portarmi a questa (sofferta) tesi e a tanti altri piccoli e meno piccoli traguardi, permettendomi di diventare quello che sono. Ringrazio il mio relatore, che ha avuto la pazienza di portarmi in questa tesi, sebbene le tempistiche lavorative abbiano lasciato poco tempo. Uno speciale pensiero poi ad alcune fedeli amiche di vita, come Taurina, Caffeina, Teina e Guaranà, senza le quali dubito avrei potuto produrre quello che ho prodotto. Un ulteriore grazie ai pochi correttori di questo mattone di 150 pagine, ovvero mio papà e la Miriam e all'organizzatrice logistica, ovvero mia mamma. Essendo troppe per essere nominate singolarmente un ringraziamento alla compagnia di Mestre, ai ragazzi dell'università, a quelli di Milano. Uno speciale grazie a tutti quelli che hanno avuto la sfiga di avermi in un qualsivoglia team di sviluppo, per aver sopportato le mie dimenticanze e i miei "ma io pensavo..". Un ringraziamento poi a tutte le persone che sono venute e andate in questi anni, per quello che mi hanno portato e lasciato.

♡

Sono sicuro di essermi dimenticato almeno il doppio di quelli che ho ricordato, quindi tanto vale finire la pagina con una dotta citazione, utilizzabile come perfetto disclaimer per le dimenticanze

Conosco la metà di voi soltanto a metà; e nutro, per meno della metà di voi, metà dell'affetto che meritate

Frodo Baggings