UNIVERSITÀ DEGLI STUDI

DI MODENA E REGGIO EMILIA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Studio e Sviluppo di un metodo di parallelizzazione in ambiente grafico di algoritmi seriali

Rabitti Andrea

Tesi di Laurea

Relatore:

Prof. Riccardo Martoglia

Anno Accademico 2011/2012

INDICE

Introduzione 5
PARTE I – TECNOLOGIE UTILIZZATE 7
CAPITOLO 1 CUDA 7
Glossario 8
1.1) Che cos'è CUDA 10
1.2) Dove usare CUDA 11
1.3) I Primi Passi 12
1.3.1) Gestione della Memoria 12
1.3.2) Griglia, Blocchi e Threads 14
1.4) Come scrivere un Kernel 16
1.4.1) Definizione 16
1.4.2) "Posizione" del Thread 16
1.4.3) Shared Memory 18
1.5) Gli Strumenti e il Software CUDA 20
1.5.1) Il Compilatore e l'Architettura Interna 20
1.5.2) Debugger, Memory Checker e Nsight 20
1.5.3) I Pacchetti CUDA 21
PARTE II – ALGORITMI IMPLEMENTATI 22
CAPITOLO 2 ALGORITMO JPEG 22
2.1) Introduzione dell'Algoritmo 23
2.2) Descrizione dei Passaggi 24
2.2.1) Modifica dello Spazio di Colore 24
2.2.2) DCT 24
2.2.3) Quantizzazione 25
2.2.4) ZigZag 26
2.2.5) Codifica di Huffman 26
2.2.6) Codifica di Huffman nell'algoritmo JPEG 28
2.3) Soluzioni Adottate 29
2.3.1) Passi Preliminari 29
2.3.2) Modifica dello Spazio di Colore 29

2.3.3) DCT 30
2.3.4) Quantizzazione e ZigZag 31
2.3.5) Codifica di Huffman con Tabella Standard 32
2.3.6) Scrittura su File 33
CAPITOLO 3 OCR 35
3.1) Introduzione 36
3.2) Descrizione dei Passaggi 37
3.2.1) Operatore di Sobel 37
3.2.2) Filtro Mediano 37
3.2.3) Filtro Gaussiano 38
3.3) Soluzioni Adottate 39
3.3.1) Operatore di Sobel 39
3.3.2) Filtro Mediano 39
3.3.3) Filtro Gaussiano 40
CAPITOLO 4 TEST E PROVE SPERIMENTALI 41
4.1) Algoritmo JPEG 42
4.2) Filtri OCR 43
PARTE III – FASI FINALI 44
CAPITOLO 5 SVILUPPI FUTURI 44
5.1) Algoritmo JPEG 45
5.2) Filtri OCR 46
CAPITOLO 6 CONCLUSIONI 47
Appendice A 49
Appendice B 51

Introduzione

Col raggiungimento del limite di frequenza dei processori avvenuto qualche anno fa, l'industria ha iniziato a sviluppare soluzioni dotate di più nuclei elaborativi. Questa soluzione però già era stata adottata in un settore parallelo: il settore della computazione grafica. I processori grafici già da vari anni adottano soluzioni con molti nuclei di elaborazione. In origine tali processori sono nati per coadiuvare il nucleo elaborativo centrale nelle operazioni appunto grafiche, ossia di rendering e visualizzazione sullo schermo d'immagini complesse. Inutile dire che l'ambito principale in cui tali processori grafici davano il meglio era il settore video ludico. Negli ultimi anni, però, le aziende produttrici di processori grafici hanno iniziato a far sentire la propria voce anche per quanto riguarda lavori tipicamente da CPU. I processori grafici hanno dalla loro parte un numero di nuclei elaborativi molto superiore a qualsiasi CPU, arrivando in alcuni casi anche a più di 3000 nuclei. D'altro canto i processori grafici sono anche contraddistinti da una frequenza di clock molto minore. Queste due caratteristiche derivano intrinsecamente dal tipo di elaborazione per cui sono nati: elaborare migliaia di pixel nello stesso momento per visualizzarli contemporaneamente. Ebbene, si è già detto come i produttori di processori grafici abbiano manifestato la volontà di allargare i propri orizzonti, e per far questo hanno sviluppato e messo a disposizione ambienti, librerie e per così dire "linguaggi" per permettere agli sviluppatori di scrivere codice atto a sfruttare il gran numero di nuclei elaborativi, senza però che il suddetto codice sia riferito a computazioni grafiche. Il beneficio di ciò è naturalmente una diminuzione dei tempi di elaborazione.

Nella presente tesi si vuole esplorare la possibilità di portare due tipologie di software in ambiente grafico parallelo. I due software sono rispettivamente:

1. Algoritmo di compressione JPEG
2. Pretrattamento di un'immagine raw per il successivo riconoscimento ottico dei caratteri presenti (OCR, optical character recognition)

Il documento sarà suddiviso in tre parti: la prima in cui si esporranno l'ambiente e gli strumenti utilizzati, una seconda e una terza rispettivamente per i due algoritmi affrontati con esposizione delle problematiche, relative soluzioni pensate e adottate e possibili sviluppi futuri. I processori grafici utilizzati sono quelli realizzati dall'azienda NVIDIA. Anche l'ambiente di sviluppo è fornito, in modo totalmente gratuito, direttamente da NVIDIA. È stata scelta questa azienda produttrice poiché gli strumenti forniti, rispetto alla concorrenza, sono più maturi ed efficaci, così come il supporto e la comunità che ci stanno dietro.

PARTE I – TECNOLOGIE UTILIZZATE

CAPITOLO 1 CUDA

Glossario

Termine	Significato
Host	É la parte adibita all'esecuzione del codice principale e che chiamerà la funzione da eseguire sul Device. In generale è il calcolatore, comprensivo di processore e memoria locale, come si intende comunemente.
Device	È la parte adibita all'esecuzione del codice CUDA e che sarà utilizzata dall'Host. In generale è ciò che viene comunemente chiamata scheda grafica, comprensiva di processore grafico e memoria grafica dedicata.
Kernel	Ogni funzione che viene chiamata dall'Host ed eseguita sul Device. Nella chiamata a ogni Kernel, l'Host deve specificare i parametri da passare, come in ogni funzione, e la dimensione della Griglia.
Griglia	La Griglia è un'astrazione di come sono eseguite le operazioni sul Device. La Griglia ha 3 dimensioni ed è suddivisa in Blocchi.
Blocco	Il Blocco ha anch'esso 3 dimensioni, anche se nelle schede più datate la terza dimensione equivale sempre a 1. Ogni Blocco è suddiviso in Thread.
Thread	Il Thread è l'esecuzione effettiva del codice CUDA.
Shared Memory	Quando viene eseguito un Kernel, per ogni Blocco è messa a disposizione un'area di memoria di dimensioni contenute accessibile da ogni Thread del Blocco. Le dimensioni massime sono pochi KB, ma variano a seconda del Device. In questa memoria risiedono anche le variabili locali dei Thread in esecuzione.
Visual Nsight	Strumento di debug che si integra con Visual Studio e che permette di scandire a fondo le variabili di ogni Thread e spostarsi per la Griglia a tempo di esecuzione. Esiste anche

	una versione beta per Eclipse, ma sempre e comunque per ambiente Windows.
Texture	Rappresentazione hardware di come vengono scritti e letti i dati su memoria video.
VRAM	È la memoria del Device, detta anche Video Ram o memoria video.

1.1) Che cos'è CUDA

CUDA (Compute Unified Device Architecture) è un'architettura hardware per l'elaborazione parallela creata da NVIDIA. Di per sé, quindi, il nome CUDA indicherebbe propriamente l'architettura hardware con cui vengono realizzati i processori grafici, ma è utilizzato anche per tutto ciò che ruota attorno allo sviluppo di software parallelo, come per esempio le librerie, il Software Development Kit, fino a definire i vari strumenti messi a disposizione come "Linguaggio CUDA", anche se di vero e proprio linguaggio non si tratta. Il "Linguaggio CUDA" comprende molteplici strumenti, anche se non può, per forza di cose, essere indipendente. Tali strumenti sono:

1) Librerie di varia natura contenenti funzioni ottimizzate per ambiente CUDA tra cui librerie di Algebra Lineare (CUBLAS), librerie che implementano i vari aspetti della Trasformata Fourier (CUFFT) e molte altre.
2) Primitive e variabili d'ambiente per scrivere esplicitamente funzioni che gireranno su GPU, per richiamarle e per la gestione della memoria grafica.
3) Compilatore specifico e proprietario per le parti di codice CUDA
4) Strumenti per il debug (i normali debugger non possono analizzare a fondo una funzione CUDA).

1.2) Dove usare CUDA

Come detto, gli strumenti forniti non sono indipendenti, ma dipendono strettamente dal sistema operativo presente sulla macchina usata sia per sviluppare il codice, sia per eseguirlo. Le API e librerie CUDA sono scritte e pensate per essere usate principalmente in C, anche se negli ultimi tempi stanno espandendosi anche ad altri linguaggi, come il Fortran o il Python. Il compilatore fornito, che si chiama nvcc, si occupa solo della parte di codice che eseguirà su GPU; per la parte restante, l'nvcc si appoggia al compilatore di sistema, in particolar modo al gcc e g++ per ambiente Linux, e il compilatore di Visual Studio per sistemi Windows. Durante lo sviluppo del progetto sono stati usati entrambi gli ambienti; Linux poiché il software finale dovrà girare sotto ambiente Linux, Windows poiché rappresenta la piattaforma primaria su cui sono pubblicati aggiornamenti e strumenti primari per lo sviluppo, primo fra tutti l'utility Visual Nsight.

1.3) I Primi Passi

1.3.1) Gestione della Memoria

I due passi fondamentali per scrivere codice eseguibile sono la dichiarazione, con le opportune parole chiave, del Kernel e la gestione della memoria su Device. Per poter elaborare dati sul Device, infatti, è necessario che questi siano presenti sulla memoria video poiché la memoria dell'Host non è né visibile né tantomeno accessibile dal Kernel. Per fare ciò, CUDA mette a disposizione alcune funzioni basilari per gestire l'allocazione e liberazione della memoria su Device e la copia dei dati tra le due memorie. Tali funzioni sono:

1) cudaMalloc: per allocare memoria sul Device.

cudaError_t cudaMalloc (void **devPtr, size_t size)	Puntatore dove allocare la memoria sul Device Dimensione in byte della richiesta di memoria
Descrizione Valore di ritorno	Allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. cudaMalloc() returns cudaErrorMemoryAllocation in case of failure. cudaSuccess, cudaErrorMemoryAllocation

2) cudaMemcpy: per copiare i dati dalla memoria dell'Host a quella del Device, viceversa e anche per "duplicare" i dati sulla memoria del Device.

cudaError_t cudaMemcpy (void *dst, const void *src, size_t size, enum CudaMemcpyKind kind)	Indirizzo della memoria di destinazione Indirizzo della memoria sorgente Dimensione in byte dei dati da copiare Tipo di trasferimento
Descrizione Valore di ritorno	Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. The memory areas may not overlap. Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.cudaErrorMemoryAllocation in case of failure. cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

3) cudaFree: per liberare la memoria allocata sul Device.

cudaError_t cudaMemcpy (void *devPtr)	Puntatore all'area di memoria del Device da liberare
Descrizione	Frees the memory space pointed to by devPtr, which must have been returned by a previous call to cudaMalloc() or cudaMallocPitch(). Otherwise, or if cudaFree(devPtr) has already been called before, an error is returned. If devPtr is 0, no operation is performed. cudaFree() returns cudaErrorInvalidDevicePointer in case of failure.
Valore di ritorno	cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorInitializationError

Benché sia possibile passare al Kernel un tipo di dato normale, si predilige sempre passare come parametro un puntatore, il quale naturalmente indicherà l'area di memoria sul Device precedentemente allocata e riempita con gli opportuni dati tramite le due funzioni appena descritte. Alla fine di tutta la computazione si dovrà liberare la memoria allocata. Dunque, se volessimo scrivere un primo scheletro di utilizzo di un Kernel, sarebbe:

```
int *someIntPointer_onHost;   //Puntatore ai dati sull'Host
int *someIntPointer_onDevice;   //Puntatore ai dati sul Device
size_t  size=sizeof(int);
cudaError_t  error;
error=cudamalloc((void **) someIntPointer_onDevice, size);   // allocazione della memoria sul Device
if(error!=cudaSuccess)   //Se il valore non è un successo
  printf("There is something wrong in the Malloc!!!\n");   //qualcosa è andato storto
//inizializzazione del puntatore ai dati sull'Host
error=cudaMemcpy(someIntPointer_onDevice, someIntPointer_onHost, size,
cudaMemcpyHostToDevice);   //Copia dei dati da Host a Device. Attenzione!  I parametri vanno passati
                           //prima la destinazione e poi la sorgente, mentre la parola chiave del
                           //quarto parametro indica prima la sorgente e poi la destinazione
if(error!=cudaSuccess)   //Se il valore non è un successo
  printf("There is something wrong in the MemCopy to Device!!!\n");   //qualcosa è andato storto
//Chiamata al kernel
error=cudaMemcpy(someIntPointer_onHost, someIntPointer_onDevice, size,
cudaMemcpyDeviceToHost);   //Se I dati sono stati modificati è necessario ricopiare sull'host il risultato
if(error!=cudaSuccess)   //Se il valore non è un successo
  printf("There is something wrong in the MemCopy back to Host!!!\n");   //qualcosa è andato storto
error=cudaFree(someIntPointer_onDevice);  //liberazione della memoria
if(error!=cudaSuccess)   //Se il valore non è un successo
  printf("There is something wrong in the Free!!!\n");   //qualcosa è andato storto
```

1.3.2) Griglia, Blocchi e Threads

Ora rimane da vedere come scrivere e richiamare un Kernel. L'invocazione di un Kernel avviene in modo similare all'invocazione di una normale funzione, con le uniche eccezioni che la funzione non ritorna mai nulla e oltre ai parametri è necessario specificare le dimensioni della Griglia. La Griglia è il nome che prende il modello di esecuzione parallela del codice su Device. La Griglia è un'entità totalmente astratta tridimensionale suddivisa a sua volta in Blocchi. Ogni Blocco è a sua volta tridimensionale e suddiviso in Threads. Il Thread è l'unità fondamentale in cui risiede il flusso di istruzioni che eseguono. Quando viene invocato un Kernel, dunque, è necessario anche specificare in quanti Blocchi è suddivisa la Griglia e in quanti Threads è suddiviso ogni Blocco. Le dimensioni massime di Griglia e Blocco variano a seconda del Device; la dimensione massima minima della Griglia è 65536x65536x1, con valore massimo totale uguale al prodotto, mentre quella di ogni Blocco è 512x512x64, con, però, numero massimo totale non superiore a 512 (es. 512x1x1=512 ok, 64x64x1= 4096 no!!!). La dimensione di Griglia e Blocchi va specificato fra tre parentesi angolate <<<>>> tra il nome del Kernel e i parametri. Il prototipo di una chiamata diventa dunque:

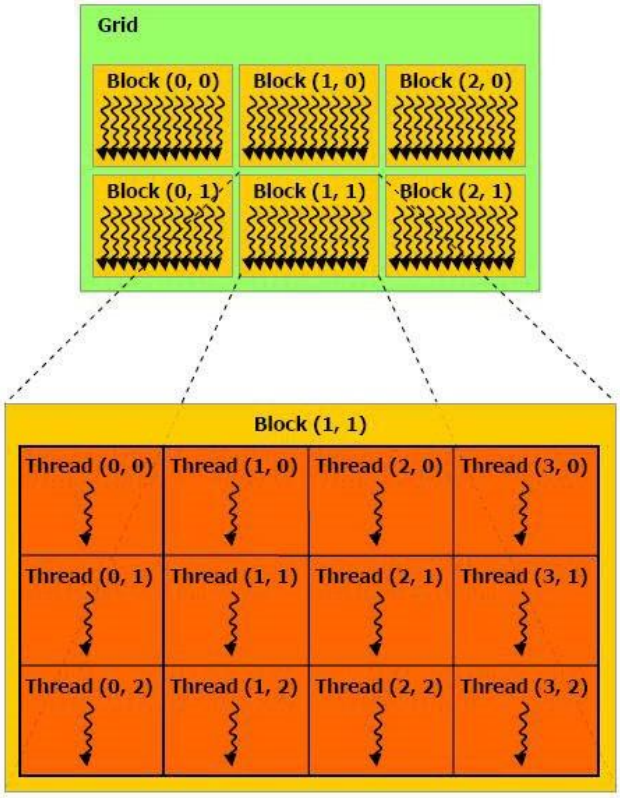


Figura 1 Gestione della Griglia

```
Kernel<<<dim3 grid(x,y,z), dim3 block(x,y,z)>>>(parameter1, paramenter2, ...);
```

Al posto delle variabili dim3 è possibile inserire valori numerici espliciti, anche se naturalmente è sconsigliato. L'invocazione di un Kernel non è bloccante, e quindi l'esecuzione del codice sull'Host continua parallelamente a quella sul Device e non vi è garanzia, al momento di ricopiare indietro i dati, che sia finita l'elaborazione e che quindi tali dati siano consistenti. Per fare in modo che l'Host attenda la fine

dell'esecuzione di un Kernel, esiste la funzione cudaThreadSynchronize() che appunto sospende l'esecuzione del codice finché il Kernel non ha finito l'esecuzione. Ovviamente è possibile eseguire altre operazioni nel parallelamente all'esecuzione del Kernel, ma subito prima di ricopiare i dati sull'Host è sempre necessario invocare tale funzione. Applicando questa informazione al precedente codice scheletro il risultato è il seguente:

```
int *someIntPointer_onHost;
int *someIntPointer_onDevice;
size_t  size=sizeof(int);
cudaError_t  error;
dim3 gridDim(10,10);   //La dimensione della Griglia sarà 10x10x1. Se uno o più valori non vengono
                       // dichiarati, assumeranno il valore di default 1
dim3 blockDim(8,8);   //La dimensione di ogni Blocco sarà 8x8x1=64, ampiamente nei margini!
error=cudamalloc((void **) someIntPointer_onDevice, size);
if(error!=cudaSuccess)
  printf("There is something wrong in the Malloc!!!\n");
//inizializzazione del puntatore ai dati sull'Host
error=cudaMemcpy(someIntPointer_onDevice, someIntPointer_onHost, size,
cudaMemcpyHostToDevice);
if(error!=cudaSuccess)
  printf("There is something wrong in the MemCopy to Device!!!\n");
Kernel<<<gridDim, blockDim>>>( someIntPointer_onDevice);   //come parametro si passa il puntatore o
                                                           // più spesso I puntatori ai dati
cudaThreadSynchronize(); // Si aspetta la fine dell'esecuzione
error=cudaMemcpy(someIntPointer_onHost, someIntPointer_onDevice, size,
cudaMemcpyDeviceToHost);
if(error!=cudaSuccess)
  printf("There is something wrong in the MemCopy back to Host!!!\n");
error=cudaFree(someIntPointer_onDevice);
if(error!=cudaSuccess)   //Se il valore non è un successo
  printf("There is something wrong in the Free!!!\n");
```

Con queste informazioni è possibile scrivere qualsiasi codice che richiami dei Kernel e che gestisce la memoria sul Device.

1.4) Come scrivere un Kernel

1.4.1) Definizione

La definizione di un Kernel CUDA è molto simile alla definizione di una funziona classica. Come già detto, un Kernel non ha mai valore di ritorno, e dunque la funzione sarà sempre void. Per fare in modo che la funzione sia eseguita sul Device e godere dei relativi benefici (che saranno esplicati in seguito), occorre iniziare la definizione con la direttiva __global__ che identifica appunto una funzione che eseguirà su Device. Esistono altre due direttive dello stesso genere; __device__, che identifica una funzione che viene invocata all'interno di un Kernel e che si comporta a tutti gli effetti come una funzione classica, sia come sintassi dell'invocazione, sia come valori di ritorno, che possono essere di qualsiasi tipo, sia come parametri, i quali non è necessario passarli come puntatori con relative allocazioni, e __host__, che identifica una funziona chiamata dall'Host ed eseguita sull'Host. Le funzioni *__device__* e *__host__* possono essere combinate, per esempio per generare codice che funzioni sia sul Device sia sull'Host, sfruttando il meccanismo di overloading degli operatori. Un semplice Kernel che prende in ingresso un puntatore ad un intero e semplicemente gli assegna un valore costante, per esempio, avrà dunque questa forma:

```
__global__ void PrimoKernel(int *val){
  *val=2;
}
```

Naturalmente un Kernel del genere è piuttosto inutile. Considerando che CUDA non permette la comunicazione esplicita tra Thread, è necessario un altro tipo di strumento per rendere utili i Kernel.

1.4.2) "Posizione" del Thread

All'interno di ogni Thread sono definite nativamente quattro costanti, composte a loro volta da tre elementi, che permettono di capire la posizione del Thread all'interno del Blocco, del Blocco all'interno della Griglia, e le dimensioni totali di questi ultimi due elementi. Tali costanti sono:

1) threadIdx.x/y/z: i rispettivi campi x, y, z descrivono la posizione del Thread all'interno di un Blocco. Tali indici partono da 0, dunque il primissimo Thread di ogni Blocco avrà threadIdx.x= threadIdx.y= threadIdx.z=0.
2) blockIdx.x/y/z: analogamente, questa costante permette di ricavare la posizione del Blocco all'interno della Griglia.
3) blockDim.x/y/z: diversamente dalla precedente, questa costante non varia da Thread a Thread in quanto, come si evince dal nome, descrive la dimensione totale di ogni Blocco, espressa naturalmente in numero Thread. In pratica assume il valore della variabile dim3 blockDim che abbiamo utilizzato nell'esempio precedente.

4) gridDim.x/y/z: analogamente, questa costante contiene la dimensione dell'intera Griglia, espressa in numero di Blocchi. In pratica assume il valore della variabile dim3 gridDim dell'esempio.

Combinando queste quattro costanti si può ricavare la posizione assoluta di ogni Thread. In generale, la prima istruzione di ogni Kernel serve a determinare proprio la posizione assoluta di ogni Thread, in modo da differenziare dinamicamente la porzione di dati da elaborare. Un esempio, supponendo di non usare la terza dimensione, che poi è il caso più frequente:

```
__global__ void Posizione(int *pos);{
  const int tid= threadIdx.x+threadIdx.y*blockDim.x;
  //tid assumerà il valore della posizione assoluta all'interno di un Blocco. Il primo Thread (indice 0,0,0)
  //del primo Blocco e il primo Thread del secondo Blocco avranno tid uguale!
  const int bid= blockIdx.x*blockDim.x *blockDim.y + blockIdx.y * blockDim.x*blockDim.y*gridDim.x;
  //bid assumerà il valore assoluto della posizione relative all'intera Griglia. Ogni Thread di uno stesso
  //Blocco avrà bid uguale!
  pos[tid+bid]=tid+bid;
  //Dunque sommando tid e bid, si ottiene semplicemente la posizione assoluta di ogni Thread
}
.
.
.
//Invocazione del kernel all'inetrno del codice classic. Naturalmente pos_d è un puntatore con
//memoria allocata pari a 8*8*10*10*sizeog(int) byte
Posizione<<<(10,10),(8,8)>>>(pos_d);
.
.
.
//Equivale a
for(int i=0;i<8*8*10*10;i++){
  pos_h[i]=i;
//ma eseguito in parallelo!
```

Da notare che un puntatore viene sostanzialmente trattato come un array monodimensionale; ad una prima occhiata, dunque, suddividere l'esecuzione di un Kernel in Blocchi e Griglia multidimensionale potrebbe sembrare un'inutile complicazione. Non è però così per due motivi: il primo, e più naive se vogliamo, è per avere una corrispondenza 1:1 col problema da risolvere (lo vedremo più avanti nei problemi specifici si affronteranno), il secondo invece affonda le sue motivazioni nella natura originale delle schede grafiche e parallelamente alla loro architettura interna e gestione della memoria. La nascita delle GPU è dovuta all'esigenza di elaborare, in pratica, gli elementi di una matrice simultaneamente, e per questo i dati vengono immagazzinati in memoria sottoforma di Texture, una matrice multidimensionale in cui ogni elemento rappresenta idealmente le informazioni del pixel in quella posizione: i dati su memoria video vengono sempre immagazzinati e letti, a basso livello, sotto forma di Texture. Le direttive presentate in precedenza si occupano di tradurre una

classica scrittura e lettura in scrittura e lettura di Texture, aumentando leggermente il tempo d esecuzione. Esistono ovviamente delle funzioni per allocare, scrivere e leggere dalla memoria direttamente sotto forma di Texture, velocizzando così l'esecuzione, ma sono operazioni difficoltose e arzigogolate, e per questo motivo non sono state prese in considerazione in questa sede. Una possibile esplorazione futura.

1.4.3) Shared Memory

A causa dell'architettura Hardware appena descritta, l'accesso alla memoria è più lento rispetto agli accessi eseguiti normalmente su Host, e deve essere presente un metodo di scambio d'informazioni tra i Thread senza dover passare per forza dalle variabili salvate in memoria, anche perché non è detto che siano modificabili. Per questi motivi esiste uno spazio di memoria molto veloce condivisa da ogni Thread di un Blocco, di nome Shared Memory. La Shared Memory ha dimensioni limitate, generalmente di pochi KB con differenze tra Device diversi, e dunque è bene non abusarne poiché non esiste un metodo di "swap": se la Shared Memory è piena, il programma termina immediatamente con un errore! Ogni Blocco ha un'area di memoria condivisa indipendente dagli altri Blocchi: Thread di Blocchi diversi non possono in nessun caso scambiarsi dati al di fuori della memoria del Device. In quest'area di memoria sono salvate le variabili locali di ogni Thread, ma nonostante sia condivisa, un Thread non ha accesso alle variabili locali di un altro. È possibile dichiarare delle variabili condivise, le quali saranno accessibili, sia in lettura che in scrittura, ad ogni Thread del Blocco. Da notare che se per una variabile locale nella Shared Memory sarà occupata memoria pari alla dimensione della variabile moltiplicata per il numero di Thread, una variabile condivisa occuperà memoria solo una volta. Una variabile condivisa viene dichiarata all'esterno del Kernel tramite la direttiva __shared__, ma non viene e non va inizializzata. È possibile anche dichiarare una costante condivisa tramite la direttiva __constant__, naturalmente inizializzandola. Ecco un esempio:

```
__constant__ int costante[]={0,1,2,3,4,5,6,7,8,9};//costante inizializzata e non modificabile
__shared__ int variabile[10];//variabile condivisa non inizializzata accessibile da ogni Thread di un
//Blocco. Se un Thread del Blocco 0,0,0 scrive su variabile[0] un valore, un Thread del Blocco 1,0,0 che
//legge variabile[0] troverà dei dati inconsistenti!
__global__ void SomeKernel(someType *somePointer){
.
.
.
}
```

A questo punto sorge un problema. Non essendoci alcun controllo e nessuna garanzia sull'ordine di esecuzione dei Thread, si rende necessaria una direttiva atta alla sincronizzazione, almeno, dei Thread di uno stesso Blocco. __syncthreads() esegue quest'operazione, mettendo in attesa i Thread che la raggiungono finché tutti quelli del Blocco non vi sono arrivati.

```
__shared__ int variabile[10];
__global__ void SomeKernel(someType *somePointer){
.
.
.
//scrivo un valore su variabile[n]
__syncthreads(); // aspetto che tutti i Thread abbiano inizializzato variabile
//accedo a variabile[m] con m!=n
.
.
.
}
```

1.5) Gli Strumenti e il Software CUDA

1.5.1) Il Compilatore e l'Architettura Interna

Come già detto, tra gli strumenti forniti vi è anche il compilatore proprietario nvcc. Il codice dunque andrà compilato usando questo compilatore, il quale ha la duplice funzione di suddivisione del codice tra codice Host e codice Device e di compilazione effettiva del codice Device. Il codice Host viene passato invece al compilatore di sistema in modo automatico, il quale lavorerà come con un codice normale. In generale, i file contenenti istruzioni CUDA hanno estensione .cu. Le opzioni di nvcc sono molto simili a quelle del gcc, come ad esempio l'istruzione –o che specifica il nome del file eseguibile. Da notare che al momento non esiste un modo per passare al compilatore di sistema tramite nvcc alcuna opzione. Tra le varie opzioni proprie di nvcc ne esiste una che è bene tenere in molta considerazione, l'opzione –arch. Questa opzione permette di specificare al compilatore l'architettura del processore grafico usato, così da generare codice eseguibile ottimizzato e permettere l'accesso ad alcune funzioni disponibili solo per architetture più avanzate. Il tipo di architettura viene specificata tramite sm_xy, dove x è la CUDA Capability Major, e y è la Minor. Se non viene specificata l'opzione –arch, il compilatore assume che l'architettura sia sm_10, ossia la minima possibile. Le architetture più recenti sono compatibili con quelle più vecchie, quindi un codice compilato tramite –arch sm_10 viene eseguito correttamente su tutte le architetture. Al contrario, compilando tramite l'opzione, ad esempio, -arch sm_20 ed eseguendo su un Device che al massimo supporta sm_13, il comportamento del codice non è prevedibile: potrebbe eseguire senza errori con risultati giusti, potrebbe eseguire senza errori con risultati sbagliati, potrebbe crasshare, potrebbero avvenire accessi sbagliati alla memoria senza avviso ecc. ecc. Per conoscere l'architettura di un Device si può ricorrere ad una ricerca sul sito Nvidia, oppure eseguire una utility sempre di Nvidia chiamata DeviceQuery che fornisce informazioni dettagliate sul Device attualmente in uso.

1.5.2) Debugger, Memory Checker e Nsight

Un debugger classico non ha la possibilità di fornire informazioni sulle variabili di un Kernel e sulla memoria del Device. Per questo oltre al compilatore vengono forniti anche due importanti strumenti, il debugger chiamato cudaDbg e un memory checker chiamato cudaMemCheck. Il debugger CUDA, in fin dei conti, lavora esattamente come un debugger classico: fornisce informazioni sulle variabili, permette di modificarne i valori e, funzione importantissima, permette di saltare da un Thread ad un altro specificandone le coordinate. Il memory checker fornisce informazioni accurate sullo stato della memoria del Device, della Shared Memory e soprattutto monitora eventuali letture errate della memoria. Un Thread, infatti, può leggere un'area di memoria fuori dai confini di quella correttamente allocata senza generare errori o far crasshare il

programma ed è quindi indispensabile avere un controllo a tutti gli accessi. Diverso il discorso per la scrittura su aree non allocate, che causa un errore e l'uscita dal Kernel e spesso dall'intero programma. Nsight è l'utility che fornisce un'interfaccia grafica ai due strumenti appena descritti, appoggiandosi ad ambienti già esistenti. Al momento, l'unica versione definitiva è Visual Nsight, che si integra in Visual Studio, e dunque esclusiva dei sistemi Windows. Esiste una versione beta per Eclipse su sistemi Linux e MacOS.

1.5.3) I Pacchetti CUDA

Tutti gli strumenti descritti sono forniti tramite due eseguibili (con la futura release della versione 5.0 di CUDA sarà uno solo) denominati CUDA Toolkit e CUDA SDK, liberamente reperibili dal sito Nvidia. Nsight è un eseguibile a parte. È possibile inoltre installare i driver per la propria scheda video appositamente ottimizzati per l'esecuzione di codice CUDA, visto che normalmente sono installati driver ottimizzati per l'uso comune. I driver per uso comune sono indicati dal nome "display driver", mentre quelli ottimizzati per l'esecuzione di codice CUDA sono denominati "dev driver". I "dev driver" hanno un rilascio più lento rispetto ai driver classici e viene privilegiata la piattaforma Windows: fino a luglio 2012 non esistevano "dev driver" per sistemi diversi da Windows. Infine, è interessante notare come nell'SDK siano presenti molteplici esempi di programmi che sfruttano l'accelerazione grafica sia per renderizzazioni grafiche, sia per l'esecuzione di algoritmi totalmente slegati dall'ambito grafico, tra cui l'utilissimo "DeviceQuery" che fornisce informazioni approfondite sull'hardware video che la macchina monta. Tali esempi sono disponibili sia sottoforma di eseguibili, sia di sorgenti liberamente modificabili, comprensivi, nei sistemi Windows, dei progetti per essere aperti nelle varie versioni di Visual Studio.

PARTE II – ALGORITMI IMPLEMENTATI

CAPITOLO 2 ALGORITMO JPEG

2.1) Introduzione dell'Algoritmo

Il formato JPEG è uno standard per la compressione d'immagini usato sia nelle immagini singole, sia nei filmati. Ciò che caratterizza questo formato è la dimensione molto ridotta che il file finale occupa, rispetto al classico bitmap, dovuta ad una perdita d'informazioni variabile arbitrariamente alla creazione del file. L'algoritmo si compone di vari passaggi che andranno via via a modificare l'immagine per renderla adatta alla compressione finale. Tali passaggi sono:

1) Modifica dello Spazio di Colore
2) Trasformata Discreta Coseno (DCT)
3) Quantizzazione
4) Ordinamento a ZigZag
5) Codifica di Huffman

2.2) Descrizione dei Passaggi

2.2.1) Modifica dello Spazio di Colore

Lo spazio di colore è il modo in cui il colore di ogni pixel viene ricostruito in base ai valori ad esso associati. Le classiche immagini bitmap, in generale, vengono salvate con lo spazio di colore RGB, in cui la combinazione dei tre colori Rosso, Verde e Blu danno il risultato finale. Ogni pixel è rappresentato da 24 bit, 3 byte, uno per ogni colore, avendo così un range di valori da 0 a 255. L'algoritmo di compressione JPEG lavora però su uno spazio diverso, YCbCr, ed è quindi necessario modificare i valori iniziali. Anche i valori di Y, Cb e Cr hanno un range di 256 valori, ma a differenze dell'RGB, questi variano da -128 a +127. Y rappresenta la luminosità del pixel, infatti viene chiamata Luminanza, restituendo in pratica un'immagine in toni di grigio; Cb e Cr sono dette Crominanze, e rappresentano la differenza del colore originale, rispettivamente, dal blu e dal rosso. Esiste una semplice funzione che permette di passare da RGB a YCbCr.

$$Y = K_r R + (1 - K_r - K_b)G + K_b B$$

$$Cb = 0.5\frac{B - Y}{1 - K_b}$$

$$Cr = 0.5\frac{R - Y}{1 - K_r}$$

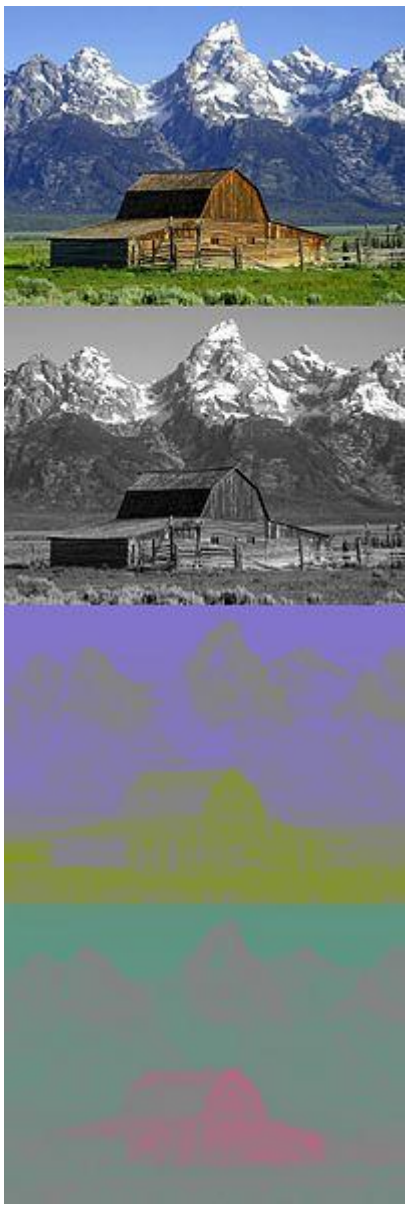


Figura 2 Immagine suddivisa nello spazio di colore Y CB Cr

Dove K_r e K_b sono due coefficienti che definiscono vari possibili sistemi. Ad esempio, lo standard BT.601, che quello più usato, prevede K_r=0.299 e K_b =0.114, mentre nel formato HDTV dei televisori ad alta definizione K_r=0.212 e K_b =0.0722.

2.2.2) DCT

Una volta ottenuto il giusto spazio di colore, inizia l'elaborazione vera e propria dell'immagine. A questo punto si hanno tre matrici di dimensioni pari alle dimensioni dell'immagine, dove ogni elemento assume il valore di Y, Cb e Cr del pixel nella medesima posizione. Ognuna delle matrici viene suddivisa in quadrati 8x8 ai quali sarà

applicata la trasformata. La DCT ha varie definizioni, ma quella che viene usata in questo ambito prende la seguente forma:

$$G_{u,v} = \sum_{x=0}^{7} \sum_{y=0}^{7} \alpha(u)\alpha(v) g_{x,y} \cos\left[\frac{\pi}{8}\left(x+\frac{1}{2}\right)u\right] \cos\left[\frac{\pi}{8}\left(y+\frac{1}{2}\right)v\right] \quad [1]$$

Dove

$\alpha(u) = \begin{cases} \sqrt{\frac{1}{8}}, se\ u = 0 \\ \sqrt{\frac{2}{8}}, altrimenti \end{cases}$ è un fattore di normalizzazione che rende la trasformata ortonormale;

u e v rappresentano lo spazio di frequenza, rispettivamente orizzontale e verticale, per gli interi da 0 a 7 compresi;

$g_{x,y}$ è il valore del pixel alla coordinata (x,y);

$G_{u,v}$ è il coefficiente DCT alle coordinate (u,v).

Alla fine dell'elaborazione, per ogni quadrato 8x8, il valore alla posizione (0,0) si ha la componente denominata DC che rappresenta l'uniformità del quadrato. Nelle altre 63 posizioni si trovano le componenti denominate AC che invece rappresentano le variazioni all'interno del quadrato. Dunque, tanto più i valori all'interno del quadrato sono uniformi, tanto più la componente DC sarà elevata e tanto più le componenti AC saranno basse; viceversa, se nel quadrato sono presenti molte e repentine variazioni, la componente DC avrà valori bassi e le componenti AC risulteranno più pronunciate. La trasformata è reversibile e non introduce alcun tipo di perdita d'informazione, salvo naturalmente a quelle dovute ai limiti di rappresentazione dei dati nel calcolatore.

2.2.3) Quantizzazione

Questo è il passaggio che introduce la perdita d'informazioni e nodo cruciale per determinare la qualità finale dell'immagine. L'occhio umano è più sensibile alle basse frequenza ed è quindi necessario rappresentarle meglio nell'immagine finale. La quantizzazione riduce le ampiezze delle frequenze più elevate e riporta in numeri ad essere degli interi. Le ampiezze trasformate alle frequenze più elevate risultano avere valori più piccoli rispetto alle frequenze basse, sia per la natura dell'immagine che per la quantizzazione. Formalmente, la quantizzazione viene definita in questo modo:

$$O(f_x, f_y) = \left\lfloor \frac{\alpha Y_{f_x,f_y}}{Q_{f_x,f_y}} \right\rfloor$$

Dove $O(f_x, f_y)$ rappresenta il valore quantizzato di indice (f_x,f_y);

Y_{f_x,f_y} è il valore Trasformato di indice (f_x,f_y);

Q_{f_x,f_y} è il corrispettivo valore della tabella di Quantizzazione;

$\alpha \in [0,1]$ è un coefficiente che permette di variare il fattore di compressione dell'immagine, che risulta tanto più elevato quanto più α e basso. La tabella di Quantizzazione è diversa tra le componenti Y e CbCr, e ne esistono di vari tipi, a seconda della qualità base che si vuole dare all'immagine.

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 36 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Tabella di Quantizzazione base della Luminanza, cioè le componenti Y.

$$\begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

Tabella di Quantizzazione base della Crominanza, cioè le componenti Cb e Cr.

2.2.4) ZigZag

Come detto, la quantizzazione ha il compito di ridurre l'ampiezza delle frequenze elevate il che si traduce, a livello pratico, nel rendere molte componenti AC nulle. Per la successiva codifica, è bene avere il maggior numero di zeri consecutivi. Per come sono strutturate le matrici di quantizzazione ed in relazione alle informazioni originariamente rappresentate dalle varie componenti AC, si procede al riordino della matrice 8x8 come mostrato in figura. Il risultato è una stringa monodimensionale di 64 elementi ed è bene notare che la componente DC, che in origine aveva coordinate (0,0), rimane nella prima posizione.

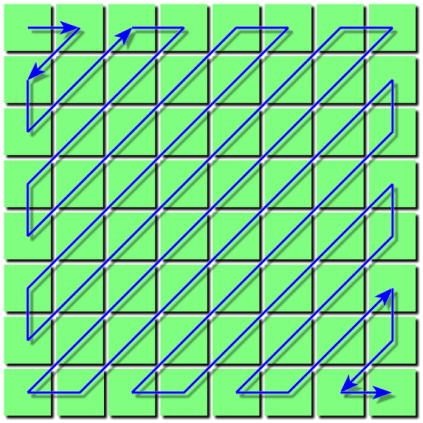


Figura 3 Rappresentazione grafica della scansione a ZigZag

2.2.5) Codifica di Huffman

Questo passaggio è quello che introduce il risparmio di spazio nel file finale. Il leitmotiv della Codifica Huffman è quello di sostituire agli elementi maggiormente

ricorrenti un codice breve, che sarà salvato nel file finale. Naturalmente, in ogni sequenza la ricorrenza dei vari elementi cambia, e per costruire una codifica efficiente dal punto di vista del risparmio di spazio, sono necessari alcuni passaggi preliminari.

1) Conteggio delle ricorrenze: una semplice scansione dell'intera sequenza per arrivare a conoscere, per ogni elemento, quante volte esso compare.
2) Costruzione di un albero binario: ad ogni elemento diverso viene associata la propria ricorrenza, e trattato come una foglia isolata; inizia così un ciclo fino a quando esiste solo un albero. Ad ogni iterazione, viene creato un nodo al quale vengono appese le due foglie con ricorrenza più bassa e reinserito come foglia a sua volta nella sequenza originale con ricorrenza pari alla somma delle due foglie. Così facendo, si arriva alla fine in cui è presente solo un albero le cui foglie contengono gli elementi iniziali.
3) Costruzione della tabella: tramite una visita all'albero appena creato, si ricava una tabella che associa ad ogni elemento il codice a cui andrà sostituito. Partendo dalla radice, si segue il percorso attraverso i nodi fino ad arrivare ad ogni foglia, andando ad inserire nel codice uno 0 se si è preso il nodo di sinistra, 1 per quello di destra.

Arrivati a questo punto, nella tabella si ha ogni elemento col relativo codice.

Ad esempio, la frase "this is an example of a huffman tree" produrrà l'albero seguente, che si traduce nella tabella:

elemento	freq.	codice
spazio	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

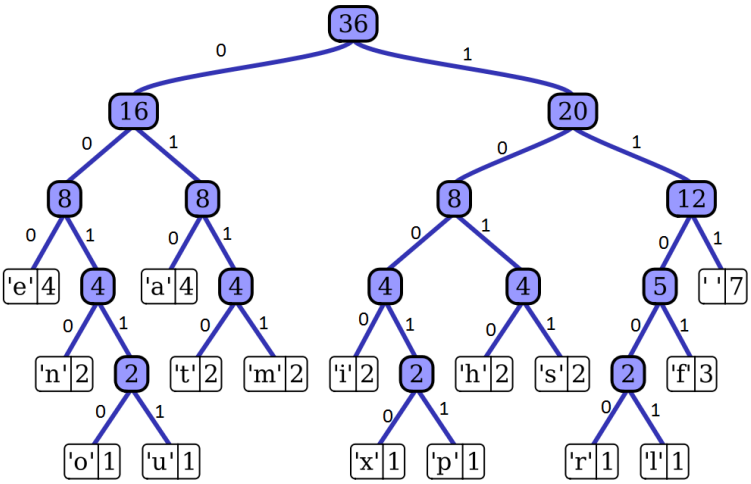


Figura 4 Tabella e Albero risultante dopo la codifica Huffman della frase "this is an example of a huffman tree"

2.2.6) Codifica di Huffman nell'algoritmo JPEG

Questo capitolo è necessario poiché nell'algoritmo JPEG la codifica di Huffman non è implementata esattamente come la definizione vorrebbe. Innanzitutto, esistono quattro tabelle di Huffman standard che vengono utilizzate a seconda del tipo di elemento da codificare. Abbiamo così una tabella per ognuno dei quattro casi: componenti DC della Luminanza, componenti AC della Luminanza, componenti DC delle Crominanze e componenti AC delle Crominanze. Non solo, anche i valori codificati non sono quelli usciti dopo l'ordinamento ZigZag. Per quanto riguarda le componenti DC, detta D_j la componente del quadrato j-esimo, la codifica si attua sul valore $M_j = D_j - D_{j-1}$ con $D_0=0$. Ciò significa che in presenza di blocchi contigui, la differenza sarà minore rispetto ai valori originali DC. Le componenti AC, invece, si codificano in coppie, dove il primo elemento rappresenta il numero di 0 consecutivi, e il secondo elemento rappresenta il valore non nullo che li segue: ecco spiegato il motivo per cui si vuole il maggior numero di 0 consecutivi. Ad esempio, la sequenza di componenti AC = 3,4,0,0,2,0,0,0,1 viene codificata come (0,3)(0,4)(2,2)(3,1). Il massimo numero di 0 consecutivi rappresentabile è 15, dunque in presenza di una sequenza con numero di 0 maggiore, si inserisce una coppia speciale, chiamata ZRL (zero run lenght) che rappresenta i primi 16 0 consecutivi, per poi proseguire il conteggio normalmente. Esiste un'altra coppia speciale, chiama EOB (end of block) che, in teoria, descrive la fine della sequenza, ma che in pratica, un decodificatore JPEG, interpreta come "il resto degli elementi che rimarrebbero per completare la sequenza di 63 componenti AC del blocco sono 0", e che dunque, nel caso in cui l'ultimo elemento del blocco sia diverso da 0, deve essere omesso.

2.3) Soluzioni Adottate

2.3.1) Passi Preliminari

Come detto, l'elaborazione grafica può avvenire solo sui dati presenti nella memoria del Device, ed è quindi necessario allocare e riempire tale memoria coi dati che dovranno essere utilizzati. L'allocazione è un'operazione che richiede un certo tempo e, visto che l'algoritmo dovrà eseguire su molte immagini, quest'operazione è bene farla una volta sola, allocando all'inizio tutto lo spazio di cui si avrà bisogno. Da notare anche che la primissima chiamata ad una funzione CUDA serve oltretutto ad inizializzare il contesto sul Device e richiede molto tempo, anche diversi secondi per ambienti complessi con più Device. Prima di iniziare l'elaborazione è dunque bene vedere quali dati sono necessari su Device. Considerando che ogni pixel dell'immagine è rappresentato da tre colori, si hanno tre matrici di tipo *unsigned char* ognuna contenente i valori di un colore, e quindi serviranno altrettante matrici nella memoria del Device per poterle riempire con quelle presenti sull'Host. Altri dati da portare dall'Host al Device non sono presenti e quindi si procede con la copia delle suddette matrici, utilizzando la funzione descritta nel capitolo precedente, la *cudaMemCpy*.

2.3.2) Modifica dello Spazio di Colore

Dal lato pratico, in questo passaggio vanno semplicemente modificato opportunamente il valore di ogni elemento, in modo tale che nelle tre matrici siano presenti i valori della Luminanza e le due Crominanze. Visto che l'elaborazione di un elemento è completamente indipendente dagli altri, l'idea di base è di creare un numero opportuno di Threads, e conseguentemente Blocchi, per eseguire questo passaggio il più rapidamente possibile senza preoccuparsi di corrispondenze con la dimensione del problema. Questo numero opportuno non è dato saperlo con certezza poiché fortemente dipendente dall'architettura del Device e dal tipo di elaborazione, ed è quindi necessario ricavarlo sperimentalmente. Non si è proceduto con questi test perché per i problemi successivi il numero di Threads e Blocchi, sempre per mantenere una corrispondenza col problema, è fissato e si è utilizzata tale dimensione della Griglia anche per questo passaggio. Per l'esecuzione è stato preso un ulteriore accorgimento: visto che i successivi passaggi lavorano su blocchi 8x8 e anche il salvataggio finale dell'immagine andrà fatto scrivendo un blocco per volta, i dati presenti sul Device sono stati precedentemente elaborati in modo che siano salvati su un array monodimensionale, e che i primi 64 elementi siano il primo blocco 8x8, i successivi 64 il secondo blocco 8x8 e così via, procedendo con una scansione orizzontale, da destra a sinistra dell'immagine. I valori di Y, Cb e Cr variano anch'essi da 0 a 255, ma per la successiva elaborazione è più conveniente salvare i valori in array di tipo reale con segno, sia per mantenere i decimali, sia perché le elaborazioni successive produrranno comunque valori reali. Per questo motivo è necessario, all'inizio, allocare i tre array

reali che conterranno i valori Y, Cb e Cr. Il prototipo del Kernel che esegue questo passaggio è il seguente:

```
__global__ void RGBtoYCbCr(int *r_d, int *g_d, int *b_d, float *y_d, float *cb_d, float *cr_d);
```

2.3.3) DCT

Come accennato in precedenza, da qui in avanti l'elaborazione avverrà sempre sui blocchi 8x8. Il numero di Threads e Blocchi è stato dunque dichiarato come segue:

```
#define BLOCK_SIZE          8   //Dimensione del blocco
#define WIDTH               1600   //Larghezza dell'immagine
#define HEIGHT              1200   //Altezza dell'immagine
dim3 threads(BLOCK_SIZE,BLOCK_SIZE);
dim3 grid(WIDTH/BLOCK_SIZE,HEIGHT/BLOCK_SIZE);
```

Dal punto di vista pratico, la trasformata si traduce in due semplici moltiplicazioni tra matrici. Data la formula [1] infatti, è inutile ricalcolare ad ogni passaggio il valore del coseno poiché costante per una specifica posizione all'interno del blocco 8x8. Si sono quindi precalcolati tali valori e salvati in un array di 64 elementi condiviso nella memoria del Device tramite la parola chiave vista nei precedenti capitoli __constant__.

```
__constant__ float DCTv8matrix[] = {
        0.3535533905932738f, 0.4903926402016152f, 0.4619397662556434f,
0.4157348061512726f, 0.3535533905932738f, 0.2777851165098011f, 0.1913417161825449f,
0.0975451610080642f,
        0.3535533905932738f, 0.4157348061512726f, 0.1913417161825449f, -
0.0975451610080641f, -0.3535533905932737f, -0.4903926402016152f, -0.4619397662556434f, -
0.2777851165098011f,
        0.3535533905932738f, 0.2777851165098011f, -0.1913417161825449f, -
0.4903926402016152f, -0.3535533905932738f, 0.0975451610080642f, 0.4619397662556433f,
0.4157348061512727f,
        0.3535533905932738f, 0.0975451610080642f, -0.4619397662556434f, -
0.2777851165098011f, 0.3535533905932737f, 0.4157348061512727f, -0.1913417161825450f, -
0.4903926402016153f,
        0.3535533905932738f, -0.0975451610080641f, -0.4619397662556434f,
0.2777851165098009f, 0.3535533905932738f, -0.4157348061512726f, -0.1913417161825453f,
0.4903926402016152f,
        0.3535533905932738f, -0.2777851165098010f, -0.1913417161825452f,
0.4903926402016153f, -0.3535533905932733f, -0.0975451610080649f, 0.4619397662556437f, -
0.4157348061512720f,
        0.3535533905932738f, -0.4157348061512727f, 0.1913417161825450f,
0.0975451610080640f, -0.3535533905932736f, 0.4903926402016152f, -0.4619397662556435f,
0.2777851165098022f,
        0.3535533905932738f, -0.4903926402016152f, 0.4619397662556433f, -
0.4157348061512721f, 0.3535533905932733f, -0.2777851165098008f, 0.1913417161825431f, -
0.0975451610080625f
};
```

Una volta calcolati questi coefficienti, i passaggi rimangono due:

$$C \times M = DCT_R \qquad [2]$$

per calcolare la DCT sulle colonne, dove M è la matrice 8x8 e C è la matrice dei coefficienti, e

$$DCT_R \times C^T = DCT_2D \qquad [3]$$

per calcolare la DCT anche sulle righe, ottenendo così il blocco finale 8x8 trasformato. Ogni Thread, in questo modo, si occuperà di calcolare il valore finale di un elemento del blocco 8x8. Per fare ciò deve accedere alla memoria per 16 volte, cioè leggere gli 8 elementi di una riga e gli 8 di una colonna. L'accesso alla memoria, come già visto, è lento ed è quindi bene cercare una soluzione che limiti gli accessi. La soluzione adottata è quella di usare due array di 64 elementi allocati in Shared Memory. All'inizio del Kernel, ogni Thread si occuperà di copiare il valore iniziale dell'elemento che dovrà elaborare nel primo array condiviso, cosicché ogni Thread andrà a leggere gli 8 valori della colonna che gli serve dalla memoria condivisa, e non dalla memoria normale, risparmiando considerevolmente tempo. È naturalmente necessario attendere che tutti i 64 Threads del Blocco abbiano scritto il proprio valore, e quindi, prima di proseguire l'elaborazione, è d'obbligo l'inserimento della direttiva __syncthreads(). Il secondo array condiviso sarà riempito col risultato dell'operazione [2] sempre per lo stesso motivo. Anche qui è necessario attendere che tutti i 64 Threads abbiano calcolato e scritto il proprio risultato. I due array condivisi sono dichiarati come segue:

```
// Temporary blocks
__shared__ float CurBlockLocal1[BLOCK_SIZE*BLOCK_SIZE];
__shared__ float CurBlockLocal2[BLOCK_SIZE*BLOCK_SIZE];
```

Infine, rimane solo da salvare I risultati in memoria. Non essendo necessario il mantenimento dei dati originali, si è deciso di sovrascrivere i valori iniziali coi valori appena calcolati. Il prototipo del Kernel è dunque il seguente:

```
__global__ void CUDAkernel1DCT(float *SrcDst)
```

2.3.4) Quantizzazione e ZigZag

In questo passaggio, i valori del blocco 8x8 sono moltiplicati per un opportuno coefficiente a seconda della loro posizione all'interno del blocco. Il Kernel viene invocato con le stesse dimensioni della DCT, in quanto l'elaborazione avviene sempre sui blocchi 8x8. Le due matrici dei coefficienti, una per la Luminanza e una per le Crominanze, sono dichiarate nella memoria condivisa, come nel caso della DCT. Dato che il passaggio successivo alla Quantizzazione è il riordinamento dei blocchi 8x8 col metodo a ZigZag, si è scelto di incorporare i due passaggi in un solo Kernel: una volta calcolato il valore Quantizzato, invece che riscriverlo nella sua posizione originale, lo si scrive nella posizione che prevede la ZigZag. I valori quantizzati, essendo degli interi, vengono salvati in un array di interi, diverso da quello sorgente. Dato che il

procedimento è uguale sia per la Luminanza sia per le Crominanze, e l'unica cosa che varia è la tabella da usare, come parametro del Kernel viene passato anche la "modalità" appunto per distinguere i due casi. Vi è un ultimo parametro che rappresenta il coefficiente α trattato in precedenza, quello che serve per variare il fattore di compressione. Il prototipo della funzione prende quindi la seguente forma:

```
__global__ void CUDAkernelQuantizationZig(float *Src, int *Dst, float a,int mod)
```

2.3.5) Codifica di Huffman con Tabella Standard

Come detto nel precedente capitolo relativo alla codifica di Huffman, per tradurre i simboli della frase coi rispettivi codici è necessaria una tabella che contenga il codice, la lunghezza del codice e la parola che andrà sostituita. Esistono quattro tabelle standard, una per tipo di valore, che sono utilizzate per l'algoritmo JPEG. Queste tabelle sono dichiarate sempre in memoria condivisa ed ognuna è rappresentata con due array: il primo che contiene il codice effettivo, e il secondo che ne contiene la lunghezza. Il valore che andrà sostituito col codice relativo viene a sua volta codificato con un sistema a bit variabili in cui, detta L il numero di bit utilizzati, con L=1 si rappresentano i 2 valori {-1(0),1(1)}, L=2 rappresenta i 4 valori {-3(00),-2(01),2(10),3(11)}, L=3 rappresenta gli 8 valori {-7(000),-6(001),-5(010),-4(011),4(100),5(101),6(110),7(111)} e così via. In generale con L bit si rappresentano 2^L valori, è più precisamente {-2^L+1(L * 0),…,-2^{L-1}(0 + L-1 * 1), 2^{L-1}(1 + L-1 * 0),…, 2^L-1(L * 1)}. Il valore -4, ad esempio, verrà rappresentato con L=3 011. Il valore di L contiene così l'indice negli array dei codici e delle lunghezze dove trovare appunto il codice relativo da andare a scrivere sul file finale seguito dagli L bit che identificano il valore. Se per esempio il nostro valore L=3 di prima corrispondesse al codice 1011, sul file finale si dovrebbe scrivere 1011011(1011=codice; 011= -4 rappresentato coi 3 bit). Questo per quanto riguarda le componenti DC. I codici delle componenti AC, invece, rappresentano la coppia descritta nel capitolo relativo. Anche i valori diversi da 0 delle componenti AC sono codificati inizialmente col sistema a bit variabili. La coppia (1,3), per esempio, descrive dunque un valore rappresentabile con 3 bit preceduto da uno 0. Il primo valore dell'array, quello di indice 0, è la coppia (0,0) e cioè l'EOB, mentre l'ultimo è la coppia (15,0) (15 zeri seguiti da uno 0) ossia lo ZRL. Gli elementi in mezzo sono idealmente divisi in 16 righe, che descrivono il numero di zeri, da 0 a 15, il primo elemento della coppia, e 10 colonne, che descrivono il valore L, da 1 a 10, il secondo elemento della coppia. Il Kernel che implementa questo passaggio si divide dunque in due parti, la prima che codifica il valore DC del blocco 8x8 e la seconda che codifica le 63 componenti AC componendo le coppie (zeri, valore). Per il calcolo del valore L si è dichiarata una funzione di tipo __device__ che come parametro prende in ingresso il valore da codificare e che restituisce l'intero L.

```
__device__ int CodDif(int val)
```

Da ricordare che per le componenti DC, tale funzione viene invocata passando come valore la differenza con la componente del blocco precedente. Per trovare il codice del valore effettivo da codificare, cioè il codice binario che segue il codice corrispondente al valore L, è necessario sommare ai valori negativi il numero 2^L-1. In questo modo i valori negativi codificati con un determinato L, assumeranno i valori da 0 a 2^{L-1}-1. Questi valori, con la rappresentazione binaria, hanno il bit L-esimo uguale a 0, mentre i valori positivi hanno il bit L-esimo uguale a 1 e dunque non vi sono sovrapposizioni. Sommando il numero di valori compresi tra -2^L+1 e -2^{L-1} e tra 2^{L-1} e 2^L-1 si ottiene infatti 2^L e cioè il numero di valori rappresentabili con 2^L bit. I codici trovati sono salvati in un array della medesima lunghezza di quello sorgente di interi senza segno così da preservarli senza problemi coi segni. La lunghezza dei codici viene salvata in un altro array di interi senza segno, ponendo alla fine delle lunghezze uno 0, descrivendo così la fine dei codici per quel blocco 8x8. Le due matrici di interi senza segno sono, infatti, anch'esse idealmente divise in segmenti di 64 elementi dove il primo contiene rispettivamente il codice e la sua lunghezza della componente DC, e negli altri i codici e le lunghezze delle coppie (zeri, valore) trovate. L'unico modo con cui si rischia di riempire completamente il segmento di 64 elementi è il caso in cui ogni elemento del blocco 8x8 sia diverso da 0: un caso del genere sta a significare che i precedenti passaggi non hanno lavorato correttamente, e non si è quindi inserito nessun controllo (che comunque rallenterebbe l'esecuzione). Anche in questo caso è necessario indicare al Kernel quali tabelle usare passandogli come parametro la modalità. Questo Kernel lavora, a differenza dei precedenti, su un intero blocco 8x8 e non su un singolo elemento. È quindi necessario invocarlo con una dimensione della Griglia differente. In principio, si è provato ad invocarlo con lo stesso numero di Blocchi dei precedenti, ma con un solo Thread per Blocco invece che 64. Questa soluzione comporta però una cattiva gestione dei Threads e impiega moltissimo tempo. Si è quindi deciso di utilizzare 500 Threads per Blocco (500 è il MCD di altezza e larghezza minore del massimo numero di Threads utilizzabili, 512) e i conseguenti Blocchi per arrivare a coprire ogni blocco 8x8. Il prototipo è il seguente:

```
__global__ void Huff(int *Src, unsigned int *Dst, unsigned int *GMat,int mod)
```

L'invocazione invece prende la seguente forma:

```
Huff<<<WIDTH*HEIGHT/(BLOCK_SIZE*BLOCK_SIZE*500),500>>>(cb_d,cbcod_d,cbg_d,COLOR_MOD)
```

2.3.6) Scrittura su File

Rimane un ultimo passaggio per completare l'algoritmo: la scrittura dei dati su File. Sebbene questo passaggio non sia stato implementato in ambiente parallelo, è comunque bene prenderlo in considerazione. A questo punto, occorre ricopiare in locale i dati necessari presenti sulla memoria del Device, ricorrendo all'istruzione cudaMemCpy

invocata due volte: una per l'array coi codici e una per le lunghezze. Ora si devono "comprimere" tutti i codici in una sequenza unica, rimuovendo sia i bit in eccesso da ogni elemento del blocco 8x8 sia i byte inutilizzati di ogni blocco 8x8, in quanto non esistono funzioni che permettono di scrivere un numero arbitrario di bit. Dopo aver "compresso" i codici, è necessaria una scansione per fare lo swap di ogni intero senza segno, poiché essendo il sistema Little Endian, la funzione di scrittura scriverebbe i byte ribaltati. Un'ulteriore scansione è necessaria per l'aggiunta di uno stuff byte, in quanto il decodificatore JPEG interpreta una sequenza di 2 byte che inizia con 0xFF un header che descrive l'inizio di una determinata sequenza di informazioni, mentre se il secondo byte è 0x00, questo viene semplicemente eliminato e il primo 0xFF interpretato come un dato semplice. La struttura di un file JPEG è piuttosto complicata, e, tra gli altri dati, sono anche scritte le tabelle di quantizzazione e di Huffman utilizzate.

CAPITOLO 3 OCR

3.1) Introduzione

L'obiettivo finale dell'algoritmo è quello di riconoscere una targa data l'immagine del transito del veicolo. Il riconoscimento si compone di svariati passaggi, ma per ora si sono studiati solo tre di questi, quelli iniziali:

1) Operatore di Sobel

2) Filtro Mediano

3) Filtro Gaussiano

3.2) Descrizione dei Passaggi

3.2.1) Operatore di Sobel

Questo algoritmo permette di elaborare un'immagine digitale in modo da riconoscerne i contorni (Edge Detection). In termini pratici, l'operatore applica due matrici 3x3 di convoluzione all'immagine originaria per calcolare valori approssimati delle derivate, una in direzione orizzontale, ed una in direzione verticale.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{e} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

G_x e G_y rappresentano rispettivamente i valori approssimati delle derivate in orizzontale e verticale. Si parte sempre dal pixel in alto a sinistra per poi proseguire da sinistra verso destra e dall'alto verso il basso. Una volta trovati questi due valori, si va a settare il pixel originale col valore approssimato del gradiente, dato dalla formula seguente.

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

È anche possibile implementare l'operatore di Sobel via hardware, ma è altresì necessario un pretrattamento dell'immagine per eliminare eventuali rumori che potrebbero inficiare sulla corretta esecuzione dell'algoritmo, motivo per cui si è scelto di implementare via software questo algoritmo.

3.2.2) Filtro Mediano

Come si intuisce dal nome, questo filtro non fa altro che sostituire un determinato valore con la mediana di una serie di valori. Nel caso specifico di un'immagine, la mediana si applica su matrici 3x3, 5x5 o 7x7 centrate nel pixel in esame. L'utilità di questo algoritmo risiede nell'eliminazione del cosiddetto rumore sale e pepe. Nella seguente immagine si può vedere l'immagine originale poi sottoposta a rumore sale e pepe e infine ripulita con il filtro Mediano.

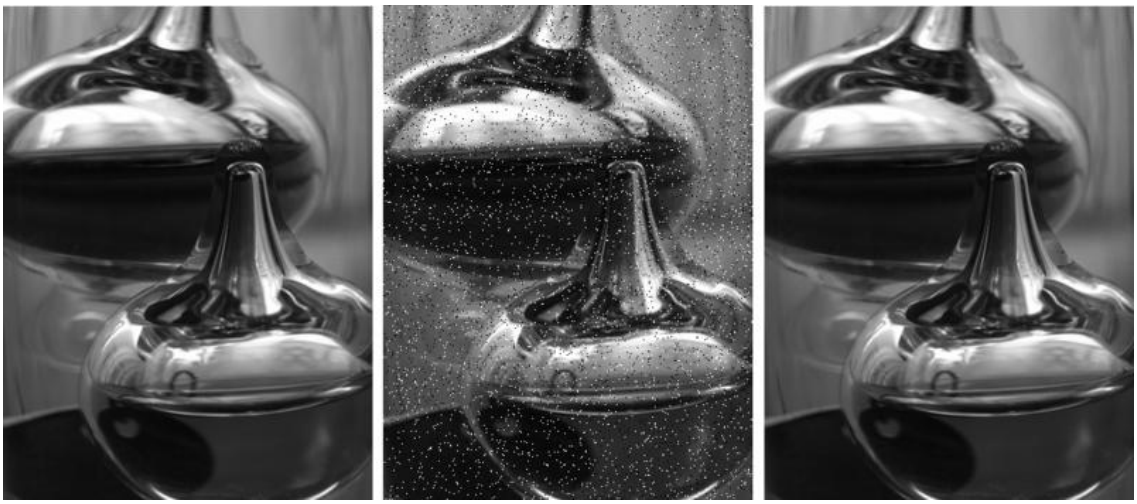


Figura 5 Immagine alla quale è stato applicato del rumore "sale e pepe" e poi ricostruita tramite filtro Mediano

In generale, la matrice usata è quella 3x3 poiché nella maggior parte dei casi è sufficiente ad eliminare efficacemente il rumore. La scelta di matrici più grandi serve in casi in cui l'immagine presenti molto disturbo, a costa naturalmente di una computazione più pesante.

3.2.3) Filtro Gaussiano

Il filtro Gaussiano (chiamato Gaussian Blur in inglese) serve a "sfumare" un'immagine, rendendola più sfocata. Questo permette al successivo riconoscimento dei contorni un'esecuzione più precisa, restituendo, a seconda del livello di "sfocatura" (smoothing), sempre meno contorni, idealmente quelli più importanti. L'algoritmo utilizza una funziona gaussiana per calcolare la trasformazione da applicare al pixel. In due dimensioni, il filtro viene definito come segue:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Dove x è la distanza dall'origine lungo l'asse orizzontale, y è la distanza dall'origine lungo l'asse verticale, e σ è la deviazione standard della distribuzione gaussiana.

3.3) Soluzioni Adottate

3.3.1) Operatore di Sobel

L'operatore dei Sobel, come tutti e tre gli algoritmi studiati, opera su immagini in toni di grigio, e quindi sempre e solo sulla matrice Y ampiamente vista nel capitolo precedente. Il primo punto studiato è stato la dimensione della Griglia con la quale invocare il Kernel. Nonostante l'algoritmo lavori su matrici 3x3, usare 3x3 Threads per ogni blocco non rappresenta una soluzione buona: la matrice 3x3 non è fissa, ma cambia per ogni pixel, e per questo si renderebbero necessarie molteplici letture dalla memoria. La soluzione adottata è stata quindi quella di usare il massimo numero di Threads per blocco divisibile per il numero di pixel di una riga dell'immagine. In un'immagine con larghezza uguale a 1200 pixel, ad esempio, si utilizzano 400 Threads per Blocco. In questo modo, ogni Thread carica in un array nella memoria condivisa lungo 512 elementi (il massimo numero di Threads) e alto 3 (l'altezza della matrice 3x3). Così facendo, ogni Threads esegue 3 letture e salvano questi valori nella memoria condivisa così che gli altri Threads possano accedere ai dati in modo più veloce. Ovviamente, agli elementi della prima e dell'ultima riga non è possibile applicare una matrice 3x3 in quanto si sforerebbero i bordi, ed è stato quindi scelto di non elaborarli facendo partire il Kernel dalla riga 2 e facendolo finire alla penultima: tali valori saranno solamente letti dalle matrici 3x3 della riga inferiore o superiore, rispettivamente. Discorso simile vale per gli elementi della prima ed ultima colonna dell'immagine; per questi viene eseguita solo la prima parte del Kernel, ossia quella della lettura dei 3 valori e la loro scrittura su memoria condivisa, ma non la seconda, ossia quella effettiva del calcolo di G. Il calcolo di G viene eseguito da una funzione di tipo __device__ alla quale vengono passati gli 8 valori che circondano il pixel corrispondente al Thread (il pixel stesso non è necessario in quanto sarebbe sempre moltiplicato per 0) e restituisce il valore G di tipo intero. I prototipi del Kernel e della funzione sono i seguenti:

```
__global__ void Sobel(unsigned char *Src,unsigned char *Dst,float fScale)

__device__ unsigned char ComputeSobel
(unsigned char ul, unsigned char um,  unsigned char ur,unsigned char ml,  unsigned char mm,
unsigned char mr,  unsigned char ll, unsigned char lm,unsigned char lr, float fScale )
```

I valori G così trovati sono salvati su una variabile differente da quella di origine così da non far leggere valori differenti da quelli originali all'elaborazione delle righe successive.

3.3.2) Filtro Mediano

Per il filtro Mediano, la dimensione della Griglia e le istruzioni del Kernel sono esattamente identici all'operatore di Sobel. Si tratta sempre di applicare un'elaborazione al pixel corrispondente del Thread utilizzando anche gli 8 pixel che lo

circondano. Anche il calcolo della mediana viene eseguito da una funzione di tipo __device__ la quale prende stavolta in ingresso tutti e 9 i valori, in quanto anche il pixel preso in considerazione è interessante. Per quanto siano molto simili, i prototipi del Kernel e della funzione sono i seguenti:

```
__global__ void MedianFilter3x3(unsigned char *Src, unsigned char *Dst)

__device__ unsigned char ComputeMedianFilter(unsigned char *list, unsigned int dim)
```

Come nel caso precedente, i valori elaborati sono salvati in una variabile diversa.

3.3.3) Filtro Gaussiano

Anche per il Filtro Gaussiano, la tecnica adottata è la solita. Dal punto di vista pratico, l'algoritmo lavora su matrici 3x3 (molto raramente), 5x5 o 7x7, sempre centrate nel pixel preso in considerazione, dando un peso diverso ad ogni valore a seconda della sua posizione; il pixel centrale avrà il peso maggiore e quelli sul contorno quello minore. Ogni valore viene moltiplicato per il suo peso, sommato a tutti gli altri, e poi eseguita una media, dividendo per la somma totale di tutti i pesi. Esistono vari tipi di matrici dei pesi: alcune la cui somma è pari a 1, e i cui valori quindi sono reali, e rendendo così inutile la media finale, altre, come quella utilizzata, i cui valori sono interi. Visto che l'utilizzo di una matrice 3x3 limita fortemente la bontà dell'algoritmo, si sono implementate due soluzioni che utilizzano una tabella 5x5 e una 7x7. Gli accorgimenti sulla prima ed ultima riga che non devono eseguire e sulla prima ed ultima colonna rimangono validi, con l'unica differenza che per la soluzione 5x5 sono le prime e ultime due colonne e righe da escludere, mentre per la soluzione 7x7 sono le prime ed ultime tre. La tabella coi pesi è salvata in memoria condivisa e l'esecuzione effettiva dell'elaborazione del valore viene effettuata da una funzione su Device, come nei casi precedenti.

```
__global__ void GaussianBlur(unsigned char *Src, unsigned char *Dst)

__device__ unsigned char ComputeGauss(unsigned char *list)
```

CAPITOLO 4 TEST E PROVE SPERIMENTALI

4.1) Algoritmo JPEG

L'obiettivo della parallelizzazione è naturalmente quello di velocizzare l'esecuzione dell'algoritmo ed è quindi utile vedere qualche tempistica sull'esecuzione dei vari Kernel. I tempi presentati sono stati misurati sotto ambiente Linux. L'algoritmo è stato testato invocando i Kernel sia su tutte e tre le matrici Y, Cb e Cr per produrre un'immagine a colori, sia solo sulla matrice Y per produrre un'immagine in toni di grigio.

	Immagine in Toni di Grigio	Immagine a Colori
Allocazione	63.538776 ms	65.073914 ms
MemCopy HtD	7.563212 ms	7.853120 ms
Modifica dello SdC	2.512016 ms	2.522975 ms
DCT	2.486025 ms	7.461004 ms
Quantizzazione e ZigZag	1.204995 ms	3.614683 ms
Codifica di Huffman	8.095178 ms	20.448874 ms
MemCopy DtH	8.487242 ms	25.238743 ms
"Compressione"	31.949245 ms	64.483749 ms
Scrittura su File	2.839108 ms	6.877340 ms
Totale Senza Allocazione	65.137021 ms	142.466122 ms

Il tempo dell'allocazione è stato riportato solo per avere un'idea del tempo medio che richiede dato che, come già detto, la primissima invocazione ad una funzione CUDA inizializza anche il contesto, e per questo può richiedere tempi molto differenti anche su una stessa macchina: in alcune prove sono stati misurati tempi di oltre 500 ms. Poiché l'allocazione richiede tempi molto variabili, e che comunque viene fatta una volta sola all'inizio, non è stata inclusa nel conteggio totale del tempo di esecuzione. La prima MemCopy deve copiare le tre matrici R G B dalla memoria locale a quella del Device, quindi sia per un'immagine a colori sia per una in toni di grigio il tempo non cambia. Stesso discorso per la modifica dello spazio di colore dato che in ogni caso si deve calcolare la matrice Y ricorrendo a tutte e tre le matrici R G B. La DCT, la quantizzazione e la codifica di Huffman, invece, essendo invocate per ognuna delle tre matrici Y Cb Cr, richiedono un tempo triplo per l'immagine a colori, risultato comunque atteso. Anche la copia degli array contenenti i codici e le lunghezze impiega il triplo del tempo per un'immagine a colori, dovendo copiare 6 array rispetto ai 2 riferiti alla sola matrice Y. Aspetto interessante la fase di "compressione", che da sola richiede la metà del tempo totale di tutto l'algoritmo, con "solo" un raddoppio del tempo di esecuzione. Un tempo così lungo è dovuto alle tre scansioni degli array coi codici (una per l'eliminazione dei bit e byte inutilizzati, una per lo swap e una per l'aggiunta dello stuff byte) che, essendo per natura seriali, devono essere eseguite in locale. Il raddoppio del tempo è dovuto al fatto che, una volta eseguita la "compressione", la lunghezza dell'array, e quindi la dimensione dei dati da scrivere su file, non triplica passando dai toni di grigio al colore. Infine, nemmeno la scrittura effettiva su file richiede il triplo del tempo proprio per il motivo appena discusso; sia i dati dell'immagine, sia le altre

informazioni da scrivere su file come le tabelle di Quantizzazione e di Huffman, non sono di dimensioni triple, ma poco più che doppie. Infine, è da tenere presente che i tempi qui riportati sono stati misurati creando matrici R G B completamente casuali, in cui gli elementi sono completamente privi di correlazione. Questo incide moltissimo sulle dimensioni finali del file, e in maniera meno evidente, ma comunque interessante, sui passaggi di "compressione" e scrittura del file; i passaggi precedenti non ne sono influenzati in quanto, in ogni caso, l'elaborazione deve essere effettuata per ogni elemento.

4.2) Filtri OCR

	Dimensione	Elaborazione
Sobel 3x3	512 x 512	0,142082
	1600 x 1200	0,934395
	3200 x 2400	3,230643
Mediana 3x3	512 x 512	1,286980
	1600 x 1200	9,616951
	3200 x 2400	37,978973/16.082735
Gaussian Blur 5x5	512 x 512	0,226595
	1600 x 1200	1,580784
	3200 x 2400	5,840321
Gaussian Blur 7x7	512 x 512	0,375639
	1600 x 1200	2,625038
	3200 x 2400	10,180774

Le prove sono state effettuate su 3 matrici casuali di dimensioni 512x512, 1600x1200 e 3200x2400, per avere un'idea della scalabilità degli algoritmi su un numero di Threads e Blocchi diversi. I tempi però sono lineari con l'aumento del numero di pixel dell'immagine e la dimensione dei Threads e dei Blocchi non influisce sul tempo. Dei tre, quello che impiega il maggior tempo è il filtro Mediano. Questo è dovuto al metodo di calcolo della mediana che ha costo $O(n^2/2)$. È inoltre molto interessante come il passaggio dalla versione 4.2 di CUDA alla versione 5.0 abbia portato ad una drastica diminuzione del tempo d'esecuzione del Filtro Mediano. Non essendo pubblici i cambiamenti precisi avvenuti nel software CUDA, l'unica supposizione è che abbiano migliorato la gestione delle istruzioni cicliche eseguite su Device.

PARTE III – FASI FINALI

CAPITOLO 5 SVILUPPI FUTURI

5.1) Algoritmo JPEG

Per quanto riguarda i primi tre Kernel, l'aspetto principale da studiare in futuro è l'utilizzo delle funzioni per il trattamento dei dati come Texture, tecnica che porterebbe sicuramente ad una ulteriore diminuzione del tempo d'esecuzione. Per la codifica Huffman un caso di studio futuro potrebbe essere la costruzione di una tabella personalizzata invece di quella standard; questo porterebbe sicuramente ad una diminuzione delle dimensioni finali del file ma anche ad un aumento del tempo di esecuzione, in quanto si dovranno eseguire vari passaggi, che comprendono quelli già visti nel capitolo dedicato, come il conteggio degli elementi, la costruzione e la scansione dell'albero. Aspetto sicuramente primario per uno sviluppo futuro è quello di ridurre i tempi del passaggio precedente alla scrittura su file, e cioè l'eliminazione dei bit e byte inutilizzati, lo swap e l'aggiunta dello stuff byte. L'operazione di swap sarebbe estremamente efficace se eseguita su Device, poiché ogni Thread si potrebbe occupare di un singolo intero senza preoccuparsi degli altri. Per fare ciò, con l'algoritmo attuale, si dovrebbe però ricopiare sulla memoria del Device l'array contenente i dati compressi, passaggio che richiederebbe sicuramente molto tempo. Eseguire efficacemente la "compressione" su Device è altresì complicato dato che questo passaggio non solo è intrinsecamente seriale, ma richiede molti accessi alla memoria sia in lettura, ma è risolvibile con la memoria condivisa, sia soprattutto in scrittura, aspetto principale della lentezza di esecuzione. La copia dei dati da Device a Host prima della "compressione" è anch'essa un nodo cruciale per il tempo d'esecuzione, specialmente per l'immagine a colori. Per ora non sono state trovate soluzioni efficaci per ridurre questo tempo; si è provato, ad esempio, ad eseguire la copia dei due array riguardanti la matrice Y mentre è in esecuzione il Kernel sulla matrice Cb, ma senza risultati utili, in quanto la copia occupa tempo di calcolo del Device che rallenta l'esecuzione del Kernel.

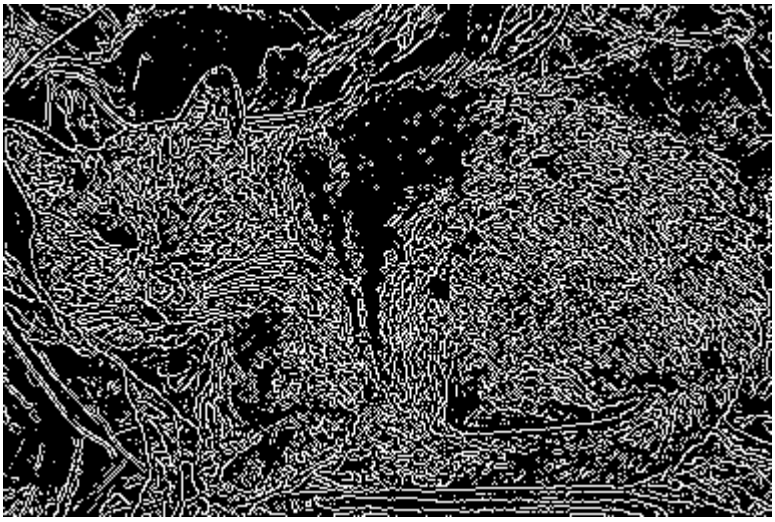
5.2) Filtri OCR

In futuro si studieranno sicuramente gli altri passaggi necessari per arrivare al riconoscimento vero e proprio dei caratteri. Per quanto riguarda i tre algoritmi studiati, i risultati sono estremamente soddisfacenti e poco si può migliorare, magari utilizzando le Texture, come nel caso dell'algoritmo JPEG. Il filtro Mediano è forse quello che potrebbe aver bisogno di ulteriori studi per ridurne il tempo d'esecuzione, anche se per il momento non sembrano esserci spunti interessanti. Si è anche provato ad implementare il filtro Mediano solo su Host, utilizzando le librerie parallele OpenMP senza risultati soddisfacenti.

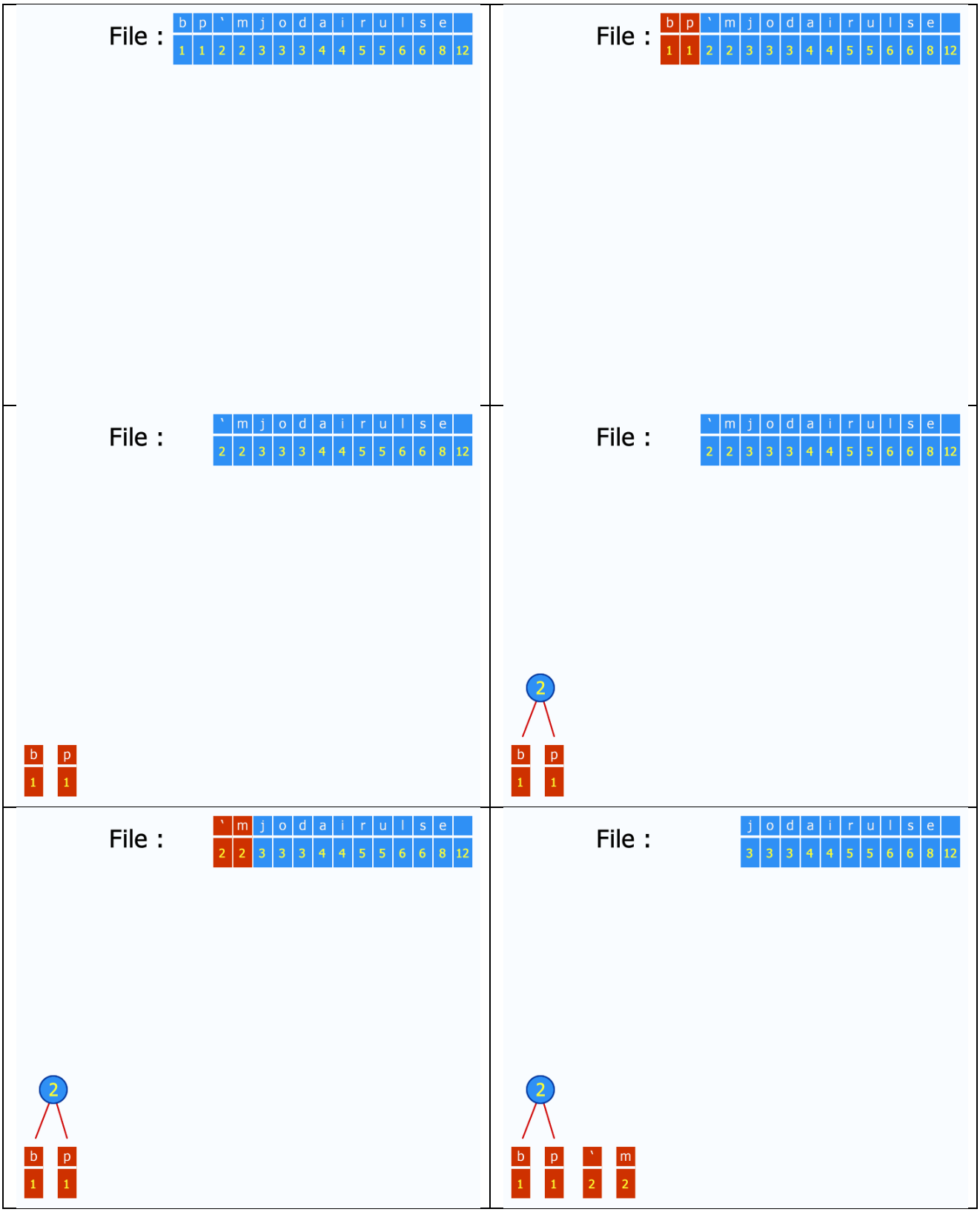
CAPITOLO 6 CONCLUSIONI

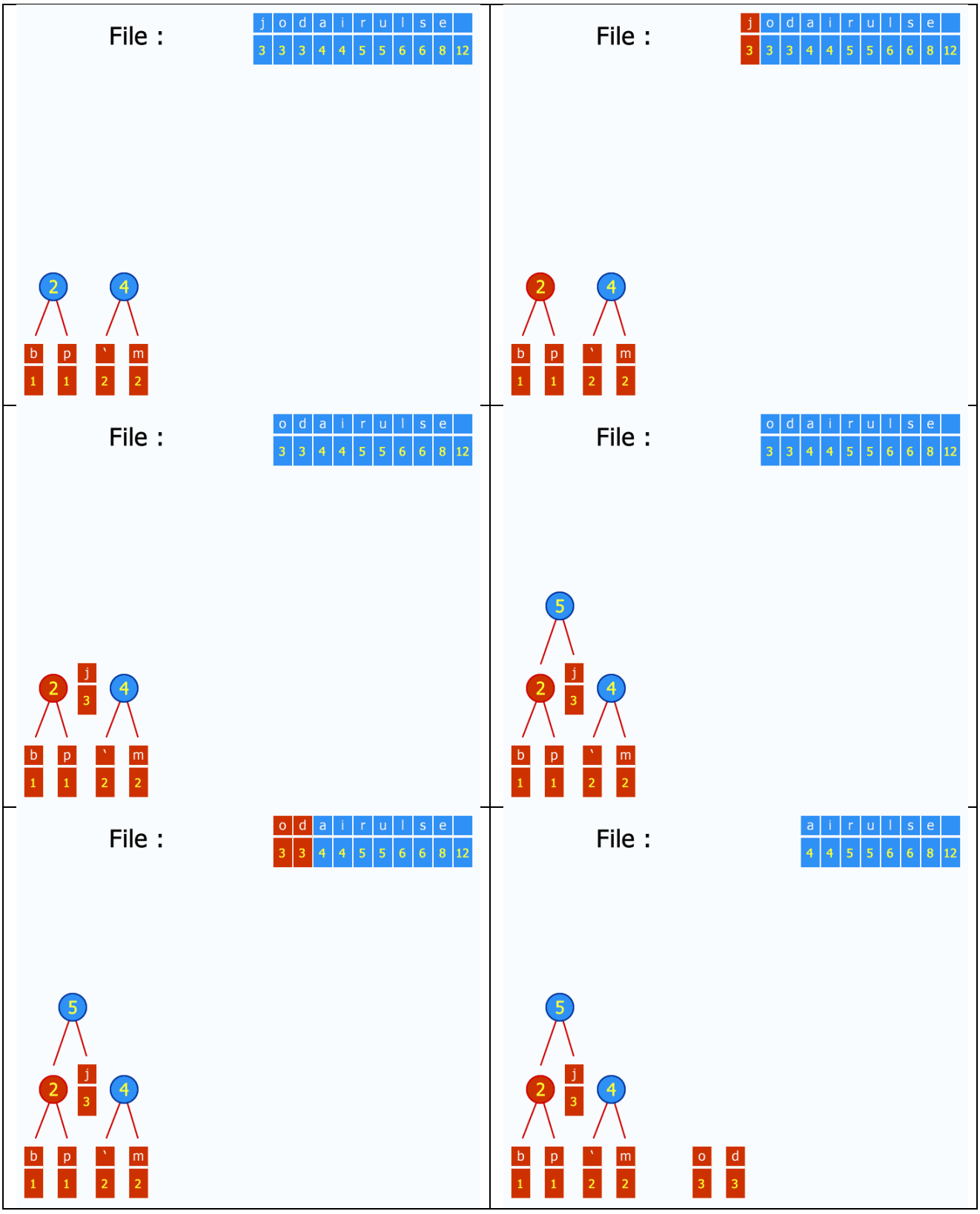
Come primo approccio all'utilizzo di un ambiente grafico per l'esecuzione di software generico, i risultati sono stati particolarmente soddisfacenti. Nonostante non si abbiano dei dati precisi sui tempi d'esecuzione degli algoritmi trattati in ambiente seriale classico, i tempi di riferimento per l'algoritmo JPEG erano circa 180ms per un'immagine in toni di grigio mentre per i tre algoritmo di filtraggio erano di circa 60ms ognuno. Per quanto riguarda l'algoritmo JPEG lo speedup generale è stato di circa 3, un risultato forse un po' sotto le aspettative, ma se si considera esclusivamente la parte parallelizzata, il guadagno è sicuramente maggiore e dunque come risultato iniziale ci si ritiene pienamente soddisfatti. Per i tre algoritmi di filtraggio, quello coi migliori risultati è sicuramente il Sobel con uno speedup di circa 20, ma anche gli altri due con uno speedup di 6 per il filtro Gaussiano e 4 per il Mediano, si sono comportati decisamente bene. In questo studio si è quindi visto come l'adozione di un supporto all'esecuzione classica di algoritmi porti innumerevoli vantaggi, primo fra tutti un minor tempo di esecuzione, ma anche un carico di lavoro minore per la CPU che, una volta demandato il lavoro al Device, è libera di eseguire altre istruzioni. I software che sfruttano questa possibilità sono in aumento, anche se la maggior parte di questi lavora in ambito grafico, come rendering di immagini o encoding video. Anche le tecnologie che permettono l'utilizzo di Device grafici per algoritmi generici si stanno evolvendo. Nvidia con CUDA è stata la prima ad iniziare questo trend, con risultati appunto ottimi, e negli ultimissimi anni vi è un'altra tecnologia che si sta sviluppando e che permette lo sfruttamento delle schede grafiche, ossia OpenCL. Tale tecnologia ha il pregio di essere completamente indipendente dall'architettura hardware su cui gira, permettendo la portabilità su qualsiasi scheda grafica, ma anche su processori multi thread. Il principale sostenitore e fautore dell'OpenCL, come accennato all'inizio, è AMD, ma anche Intel sta investendo su questo campo per far sfruttare appieno i chip grafici che integra nei suoi processori di ultima generazione. OpenCL è anche compatibile con le schede Nvidia, anche se, per ovvi motivi commerciali, non è molto incoraggiato. In conclusione, la GPGPU (**G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits) permette di ottenere risultati interessanti senza la necessità di avere molte macchine molto costose a disposizione e per questo è sicuramente uno dei punti fondamentali dello sviluppo futuro delle Scienze Informatiche.

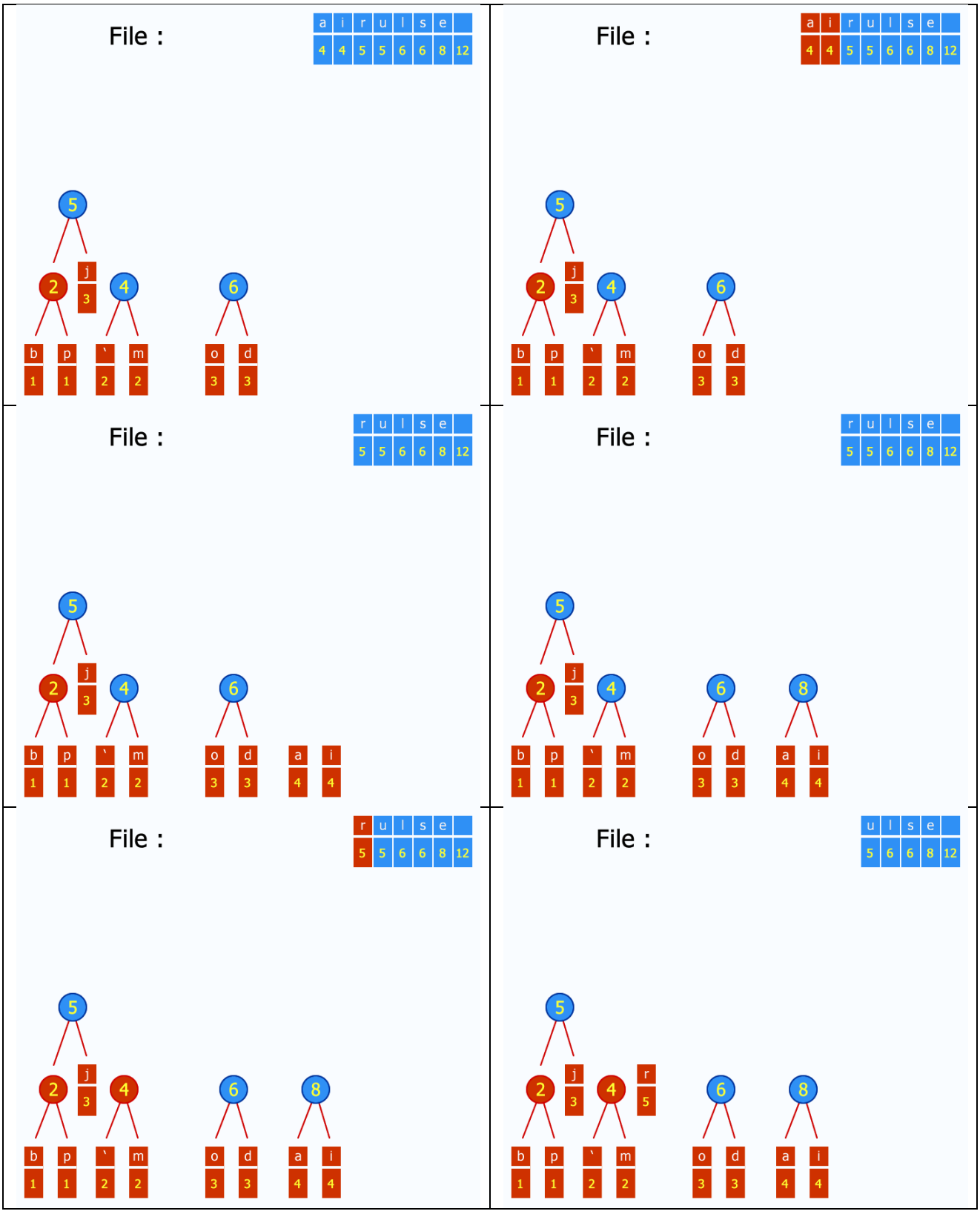
Appendice A – Immagine sottoposta a filtro Sobel dopo il trattamento con filtri Gauss di diversa intensità

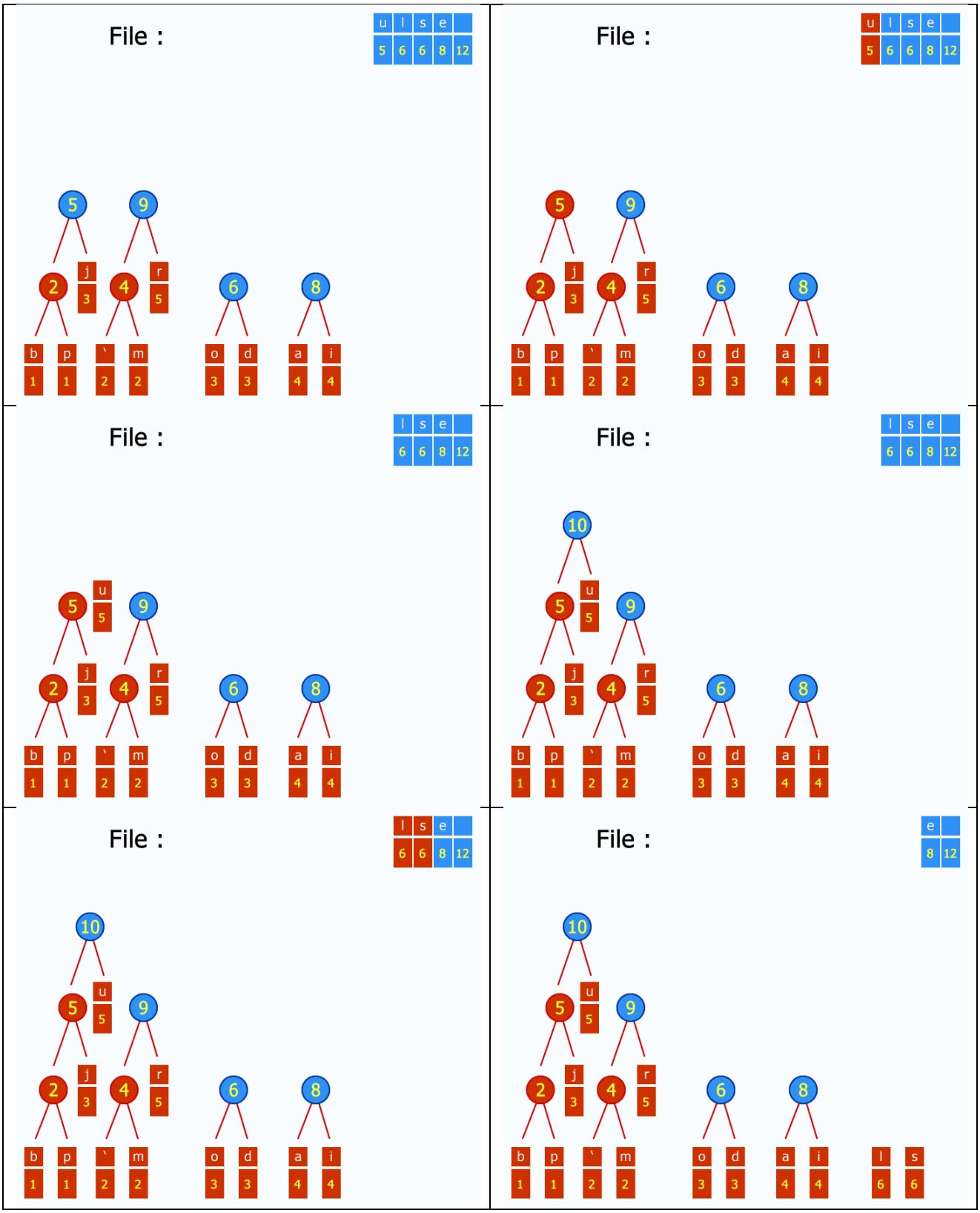


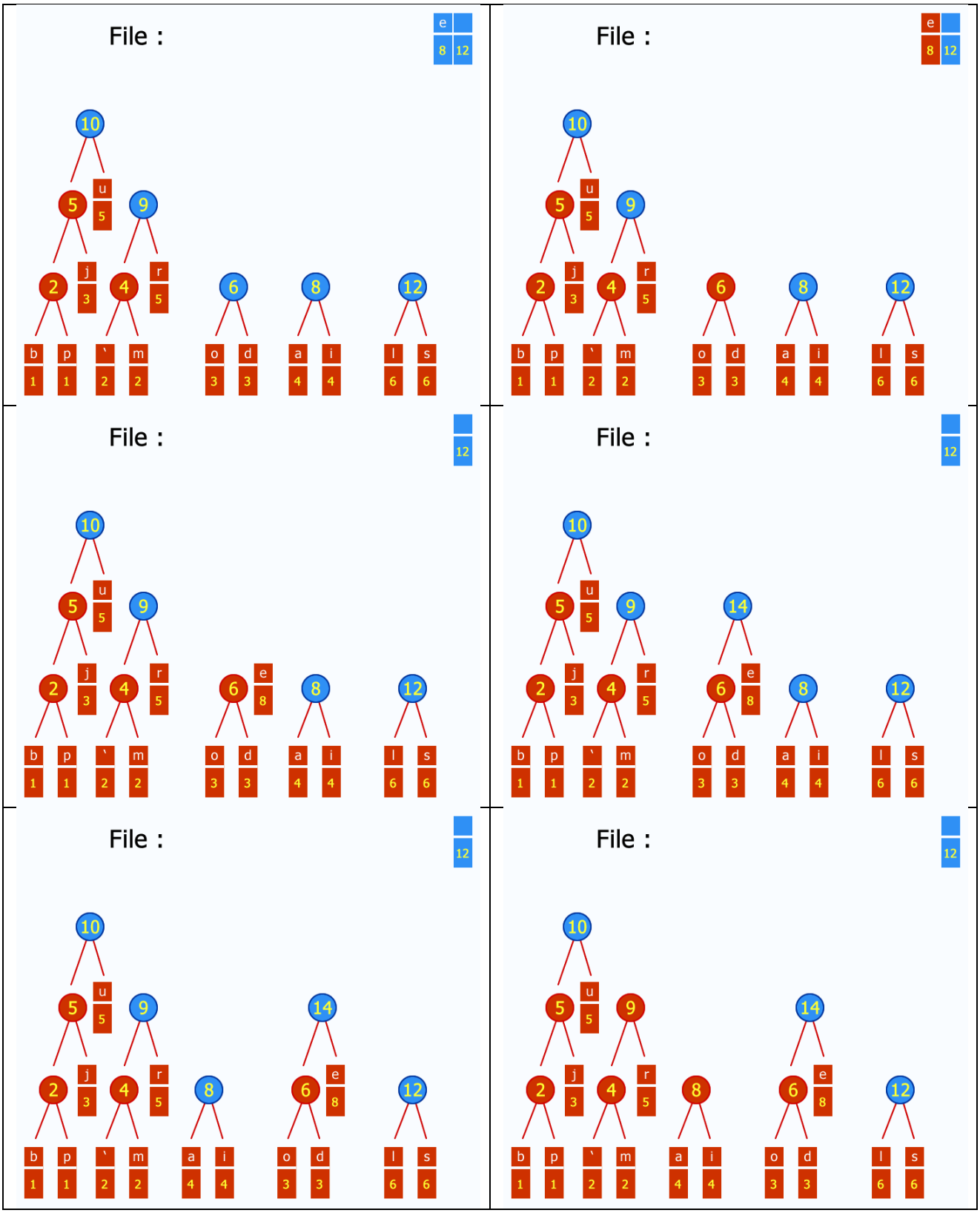
Appendice B – Sviluppo della codifica di Huffman

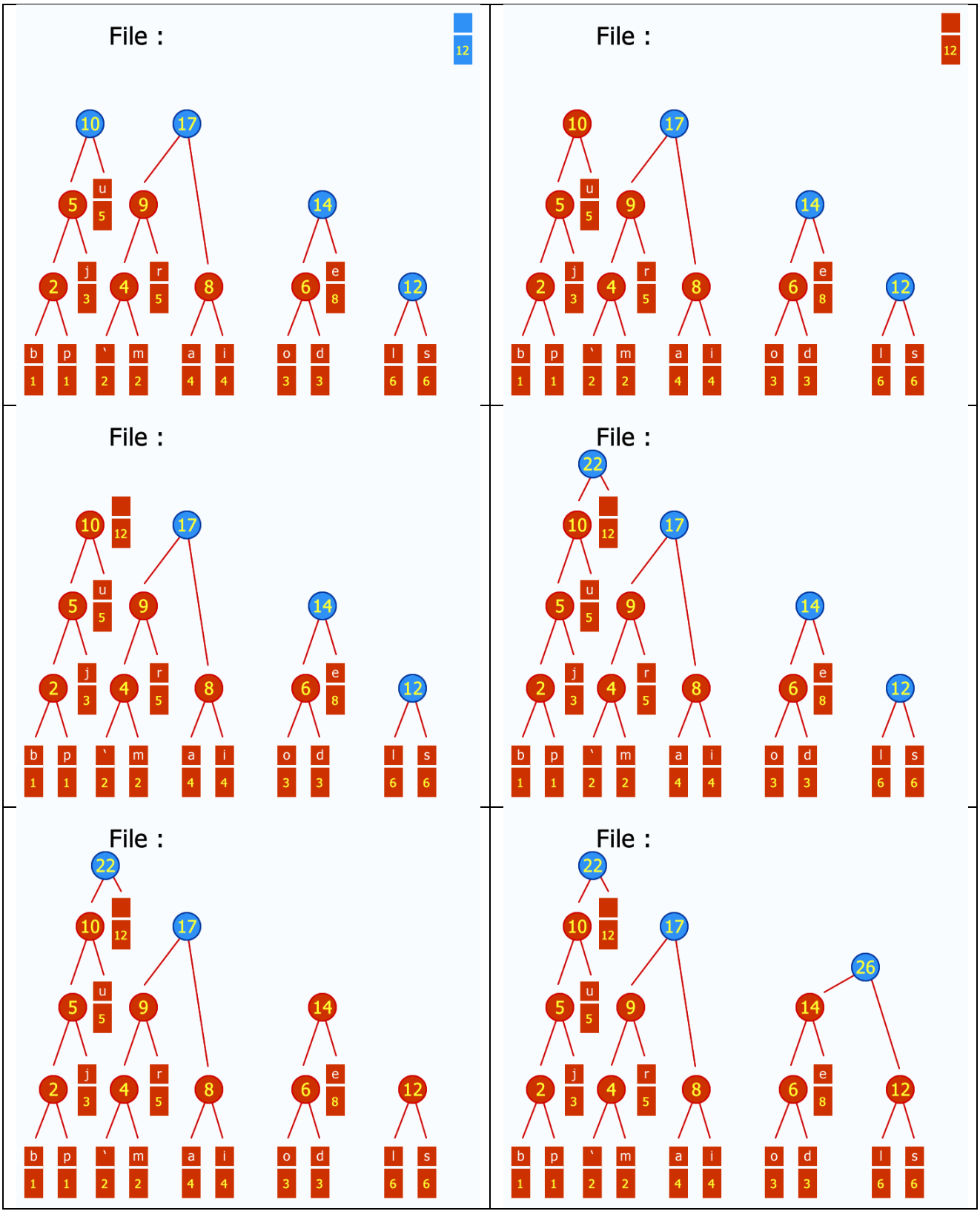


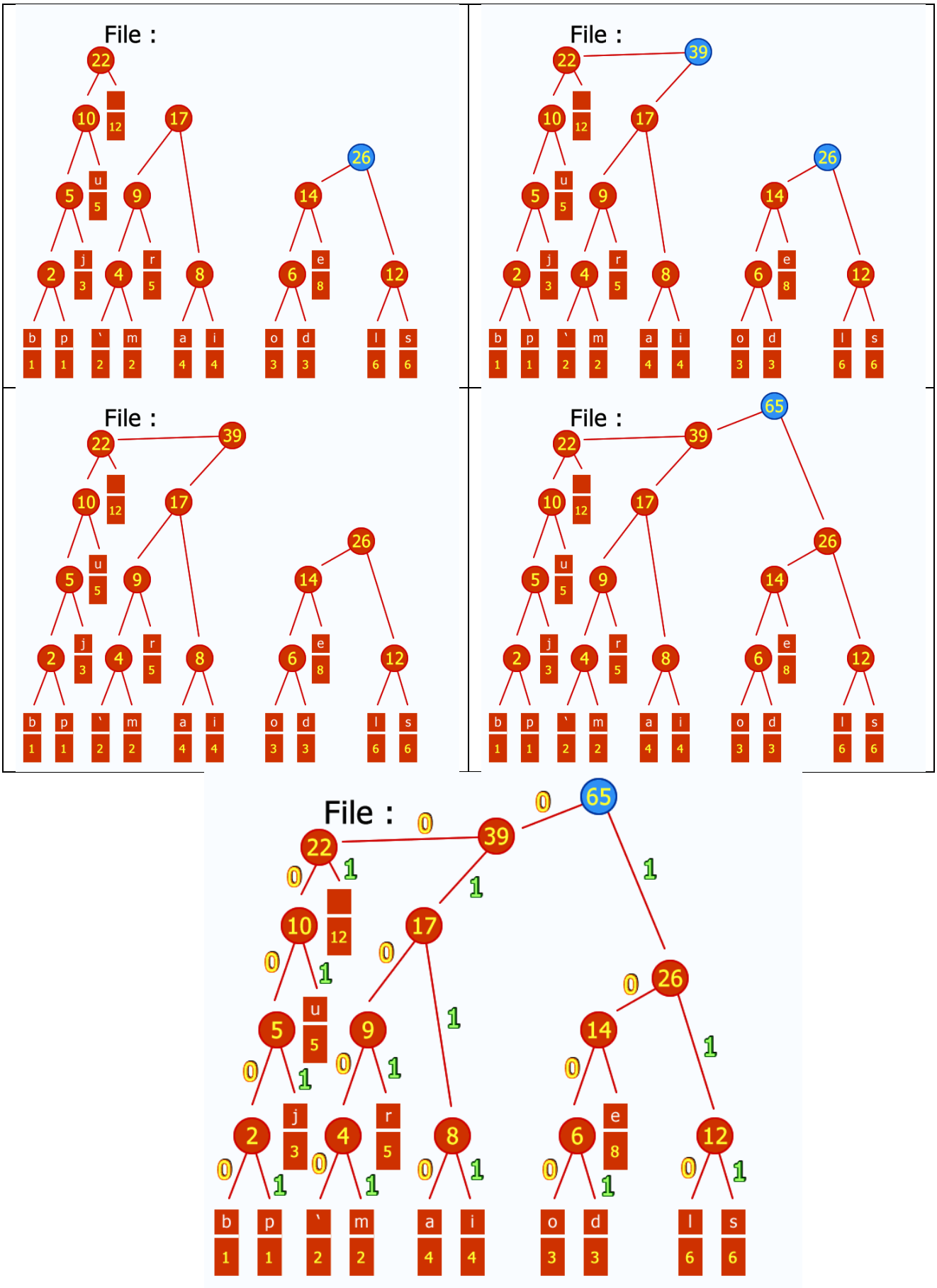

File :
j o d a i r u l s e
3 3 3 4 4 5 5 6 6 8 12
2 4
b p ` m
1 1 2 2
File :
j o d a i r u l s e
3 3 3 4 4 5 5 6 6 8 12
2 4
b p ` m
1 1 2 2
File :
o d a i r u l s e
3 3 4 4 5 5 6 6 8 12
2 j 4
3
b p ` m
1 1 2 2
File :
o d a i r u l s e
3 3 4 4 5 5 6 6 8 12
5
2 j 4
3
b p ` m
1 1 2 2
File :
o d a i r u l s e
3 3 4 4 5 5 6 6 8 12
5
2 j 4
3
b p ` m
1 1 2 2
File :
a i r u l s e
4 4 5 5 6 6 8 12
5
2 j 4
3
b p ` m o d
1 1 2 2 3 3


File :
a i r u l s e
4 4 5 5 6 6 8 12
5
2 j 3 4 6
b p ` m o d
1 1 2 2 3 3
File :
a i r u l s e
4 4 5 5 6 6 8 12
5
2 j 3 4 6
b p ` m o d
1 1 2 2 3 3
File :
r u l s e
5 5 6 6 8 12
5
2 j 3 4 6
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
r u l s e
5 5 6 6 8 12
5
2 j 3 4 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
r u l s e
5 5 6 6 8 12
5
2 j 3 4 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
u l s e
5 6 6 8 12
5
2 j 3 4 r 5 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4


File :
u l s e
5 6 6 8 12
5 9
2 j 3 4 r 5 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
u l s e
5 6 6 8 12
5 9
2 j 3 4 r 5 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
l s e
6 6 8 12
5 u 5 9
2 j 3 4 r 5 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
l s e
6 6 8 12
10
5 u 5 9
2 j 3 4 r 5 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
l s e
6 6 8 12
10
5 u 5 9
2 j 3 4 r 5 6 8
b p ` m o d a i
1 1 2 2 3 3 4 4
File :
e
8 12
10
5 u 5 9
2 j 3 4 r 5 6 8
b p ` m o d a i l s
1 1 2 2 3 3 4 4 6 6


File :
e 8
12
10
5 u 5
9
2 j 3
4 r 5
6
8
12
b 1
p 1
` 2
m 2
o 3
d 3
a 4
i 4
l 6
s 6
File :
e 8
12
10
5 u 5
9
2 j 3
4 r 5
6
8
12
b 1
p 1
` 2
m 2
o 3
d 3
a 4
i 4
l 6
s 6
File :
12
10
5 u 5
9
2 j 3
4 r 5
6 e 8
8
12
b 1
p 1
` 2
m 2
o 3
d 3
a 4
i 4
l 6
s 6
File :
12
10
5 u 5
9
14
2 j 3
4 r 5
6 e 8
8
12
b 1
p 1
` 2
m 2
o 3
d 3
a 4
i 4
l 6
s 6
File :
12
10
5 u 5
9
14
2 j 3
4 r 5
8
6 e 8
12
b 1
p 1
` 2
m 2
a 4
i 4
o 3
d 3
l 6
s 6
File :
12
10
5 u 5
9
14
2 j 3
4 r 5
8
6 e 8
12
b 1
p 1
` 2
m 2
a 4
i 4
o 3
d 3
l 6
s 6


File :
File :
File :
File :
File :
File :


File :
22
10
12
17
5
u
5
9
26
14
2
j
3
4
r
5
8
6
e
8
12
b
1
p
1
`
2
m
2
a
4
i
4
o
3
d
3
l
6
s
6
File :
39
File :
65
File :
0
1

Riferimenti Bibliografici:

1) http://forums.nvidia.com/: per tutte le soluzioni e dubbi riguardanti lo sviluppo di software CUDA
2) http://www.nvidia.com/: per gli strumenti e per la documentazione ufficiale
3) http://en.wikipedia.org/wiki/: per la descrizione di alcuni algoritmi studiati
4) http://www.impulseadventure.com/: per la descrizione approfondita della codifica finale JPEG
5)