ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων

Διπλωματική Εργασία

ΥΛΟΠΟΙΗΣΗ ΤΟΥ AVS VIDEO STANDARD ΣΕ ΕΝΑΝ MASSIVELY PARALLEL ΠΟΛΥΕΠΕΞΕΡΓΑΣΤΗ .

MAPPING AVS VIDEO STANDARD ON A MASSIVELY PARALLEL MULTIPROCESSOR.

Νεφέλη Παπαπέτρου – Λαμπράκη

Επιβλέποντες Καθηγητές : Νικόλαος Μπέλλας *Αναπληρωτής Καθηγητής*

Χρήστος Δ. Αντωνόπουλος *Επίκουρος Καθηγητής*

Βόλος, Ιούλιος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων

Διπλωματική Εργασία

ΥΛΟΠΟΙΗΣΗ ΤΟΥ AVS VIDEO STANDARD ΣΕ ΕΝΑΝ MASSIVELY PARALLEL ΠΟΛΥΕΠΕΞΕΡΓΑΣΤΗ .

Νεφέλη Παπαπέτρου – Λαμπράκη

Επιβλέποντες Καθηγητές : Νικόλαος Μπέλλας *Αναπληρωτής Καθηγητής*

Χρήστος Δ. Αντωνόπουλος *Επίκουρος Καθηγητής*

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 12η Ιουλίου του 2010.

.....	**.....**
Ν.Μπέλλας	**Χ.Δ.Αντωνόπουλος**
Αναπληρωτής Καθηγητής	**Επίκουρος Καθηγητής**

Διπλωματική εργασία για την απόκτηση του διπλώματος του Μηχανικού Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας, στα πλαίσια του Προγράμματος προπτυχιακών σπουδών του Τμήματος Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας.

.....

Παπαπέτρου – Λαμπράκη Νεφέλη

Διπλωματούχος Μηχανικός Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων Πανεπιστημίου Θεσσαλίας

Στην Βασιλεία και στον Νικήτα

Ευχαριστίες

Με την περάτωση αυτής της εργασίας ένα ταξίδι 6 χρόνων στην υπέροχη πόλη του Βόλου φτάνει στο τέλος του.

Θα ήθελα να ευχαριστήσω θερμά τους κύριους επιβλέποντες της διπλωματικής μου εργασίας, κ. Νικόλαο Μπέλλα και κ. Χρήστο Αντωνόπουλο τόσο για την συνεργασία , τις συμβουλές και την καθοδήγησή τους που διευκόλυναν την ολοκλήρωση της παρούσας εργασίας, όσο και για την σημαντική συμβολή τους στην συνέχεια της ακαδημαϊκής μου πορείας. Επίσης θα ήθελα να ευχαριστήσω τον πρόεδρο της σχολής μας κ. Γεώργιο Σταμούλη για την βοήθεια και τις συμβουλές που προσέφερε καθόλη την διάρκεια των σπουδών μου.

Επίσης θα ήθελα να ευχαριστήσω τον φίλο και συμφοιτητή Κωσταντή Νταλούκα για την άριστη συνεργασία και την βοήθειά που μου προσέφερε καθόλη την διάρκεια της φοιτητικής μου πορείας.

Δεν θα ήθελα να παραλείψω να ευχαριστήσω τους φίλους και συμφοιτητές μου Άρτεμις Τσουφλίδου, Νικόλαο Φλιτρή και Χριστόφορο Κρόνη για την συνεργασία τους κατά την διάρκεια των σπουδών μας, καθώς και όλους τους υπόλοιπους φίλους που στάθηκαν υπομονετικά δίπλα μου.

Επιπρόσθετα ένα μεγάλο ευχαριστώ στον φίλο, συμφοιτητή και δημιουργό του ιστότοπου liveplace.gr Αναστάσιο Μ. Χιδερίδη που επιμελήθηκε της αισθητικής, συντακτικής και λεκτικής ανάλυσης της παρούσας εργασίας.

Τέλος ευχαριστώ θερμά την οικογένεια και τους γονείς μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν τόσο καθόλη την διάρκεια των σπουδών μου όσο και κατά την εκπόνηση της διπλωματικής μου εργασίας. Με την συμπαράστασή τους, την ψυχολογική υποστήριξη και όντας δίπλα μου όλα αυτά τα χρόνια συνέβαλαν στην επιτυχή ολοκλήρωση τόσο αυτής της εργασίας όσο και του κύκλου των προπτυχιακών μου σπουδών.

Παπαπέτρου Λαμπράκη Νεφέλη
Βόλος, 2010

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Περιγραφή του προβλήματος και συμβολή της εργασίας

Τα τελευταία χρόνια παρατηρείται μία στροφή προς τον παράλληλο προγραμματισμό και την έρευνα πάνω σε μη συμβατικές πλατφόρμες πολυεπεξεργαστών. Μία από αυτές τις πλατφόρμες είναι ο επεξεργαστής των καρτών γραφικών (GPU). Επίσης παρατηρούμε και μία συνεχόμενη αύξηση στην χρήση συσκευών που ασχολούνται αποκλειστικά με εφαρμογές πολυμέσων. Ένας μέσος καταναλωτής σήμερα έχει στην κατοχή του από ένα κινητό τηλέφωνο 3^{ης} γενιάς και μία ψηφιακή τηλεόραση. Αυξάνεται λοιπόν και η χρήση των εφαρμογών που ασχολούνται με κωδικοποίηση και αποκωδικοποίηση video.

Στην εργασία αυτήν ασχολούμαστε με την μεταφορά του AVS video decoder στην GPU. Το AVS video standard είναι ένα νέο πρότυπο αποκωδικοποίησης video που επιτυγχάνει video decoding σε απόδοση 7frames/sec. Για να έχουμε real time video decoding, δηλαδή ο παραλήπτης να αποκωδικοποιεί το βίντεο με τέτοια ταχύτητα έτσι ώστε να μην φαίνονται από το μάτι του παρατηρητή “παγώματα” και ασυνέπειες του video, πρέπει ο αποκωδικοποιητής μας να έχει απόδοση 30frames/sec.

Αποτελέσματα της διπλωματικής εργασίας

Με την εργασία αυτή έγινε ένα πολύ σημαντικό πρώτο βήμα, όσον αφορά την μεταφορά του AVS στην GPU. Ένα σημαντικό κομμάτι του κώδικα έχει ξαναγραφτεί έτσι ώστε να τρέχει στην κάρτα γραφικών παράγοντας ένα σωστά αποκωδικοποιημένο video. Επίσης, έχουν εφαρμοστεί κάποιες βελτιστοποιήσεις που συμβάλουν στην ταχύτερη εκτέλεση του κώδικα. Τέλος έχουν μελετηθεί πιθανές δυνατότητες βελτιστοποίησης τόσο όσον αφορά το γενικό επίπεδο παραλληλισμού του αποκωδικοποιητή όσο και ειδικότερα τον βέλτιστο τρόπο προγραμματισμού της GPU.

Διάρθρωση της διπλωματικής εργασίας

Στο κεφάλαιο 2 γίνεται μία γενική περιγραφή της αρχιτεκτονικής και των χαρακτηριστικών των καρτών γραφικών και της γλώσσας προγραμματισμού CUDA. Έπειτα στο κεφάλαιο 3 περιγράφεται η διαδικασία της αποκωδικοποίησης του AVS video standard καθώς και οι θεωρητικές δυνατότητες παραλληλισμού του κώδικα. Έπειτα στο κεφάλαιο 4 περιγράφονται τα στάδια που ακολουθήθηκαν για την μεταφορά του κώδικα και τέλος στο κεφάλαιο 5 έχουμε την μελέτη της απόδοσης της εφαρμογής μας. Στον επίλογο αναφέρονται δυνατότητες για μελλοντική εργασία και επέκταση της παρούσας εργασίας.

Κεφάλαιο 2

Γενική Επισκόπηση των GPUs και της CUDA.

2.1 Εισαγωγή

Από το 1970 η τεχνολογία των GPUs άρχισε να αναπτύσσεται. Αρχικά με το ANTIC chip, έναν επεξεργαστή ειδικού σκοπού για μεταφορά δεδομένων κειμένου και γραφικών στην έξοδο του βίντεο. Το chip αυτό χρησιμοποιήθηκε στους Atari 8-bit computers για την επίτευξη του hardware ελέγχου διαφόρων γραφικών και κειμένου. Στην πορεία πιο πολύπλοκες κάρτες γραφικών άρχισαν να αναπτύσσονται, οι οποίες ήταν αρχικά απρόσιτες στο κοινό λόγω της υψηλής τιμής τους αλλά και του εξειδικευμένου σκοπού τους. Από το 1990 οι κάρτες γραφικών άρχισαν να γίνονται πιο δημοφιλείς στην αγορά και αυτό οδήγησε σε μία αυξημένη ζήτηση για hardware – accelerated 3D γραφικά. Τότε δημιουργήθηκε και η OpenGL, μία βιβλιοθήκη που επέτρεψε στους προγραμματιστές την δημιουργία εφαρμογών που παρήγαγαν 2D και 3D γραφικά. Αμέσως μετά ακολούθησε και η Microsoft με το DirectX το οποίο έγινε δημοφιλές ανάμεσα στους κατασκευαστές παιχνιδιών για Windows, αλλά συχνά παρέμενε μία γενιά πίσω από την OpenGL. Και έτσι ο προγραμματισμός των GPUs,γνωστός ως GPGPU (General Purpose Computing on GPU) άρχισε να αναπτύσσεται.

Μέχρι το 2006, ο GPGPU αποδείχθηκε αρκετά δύσκολος διότι οι προγραμματιστές έπρεπε για να αποκτήσουν πρόσβαση στους πυρήνες των καρτών γραφικών, να χρησιμοποιήσουν τα προγραμματιστικά περιβάλλοντα (APIs) της OpenGL ή του DirectX. Παρά του υψηλού επιπέδου τους, αυτά τα APIs περιόριζαν τα είδη των εφαρμογών που κάποιος μπορούσε να γράψει για τα chips των GPUs. Έτσι πολύ λίγοι άνθρωποι είχαν τα προσόντα που απαιτούνταν για να καταφέρουν να τα χρησιμοποιήσουν, και ο προγραμματισμός των GPUs δεν εξελίχθηκε σε μαζικό κίνημα.

Από το 2007 όμως με την εμφάνιση της CUDA από την NVIDIA όλα άλλαξαν. Από την σειρά 8 και μετά η NVIDIA αφιέρωσε μία περιοχή στο υλικό ειδικά για να διευκολύνει τον παράλληλο προγραμματισμό. Προστέθηκε δηλαδή επιπλέον hardware και δεν έγιναν μόνο αλλαγές στο software. Οι προγραμματιστές τώρα δεν χρειάζεται να χρησιμοποιήσουν το interface ειδικού σκοπού των γραφικών, αλλά ένα καινούργιο interface σχεδιασμένο για τον προγραμματισμό εφαρμογών όλων των ειδών. Επίσης, έγιναν αλλαγές στα επίπεδα του software έτσι ώστε να μπορεί να γίνει ουσιώδης επαναπρογραμματισμός της GPU.

2.2 Parallel Programming and GPUs

Πως ξεκίνησε λοιπόν αυτό το ενδιαφέρον για τις κάρτες γραφικών ;

Για περισσότερες από 2 δεκαετίες, οι μικροεπεξεργαστές που ήταν βασισμένοι σε μια βασική μονάδα επεξεργασίας (CPU) όπως για παράδειγμα οι Pentium της intel, παρείχαν συνεχόμενες και μεγάλες αυξήσεις στην απόδοση, σε συνδυασμό με μείωση του κόστους. Αυτή η συνεχόμενη αύξηση της απόδοσης, έδινε νέες δυνατότητες στους προγραμματιστές εφαρμογών, και τους οδήγησε να εναποφύονται στις εξελίξεις του hardware για πιο γρήγορη εκτέλεση των προγραμμάτων τους. Αυτό ήταν αναμενόμενο αφού κάθε νέα γενιά επεξεργαστών ήταν πιο γρήγορη και παρείχε περισσότερες δυνατότητες από την προηγούμενη[7].

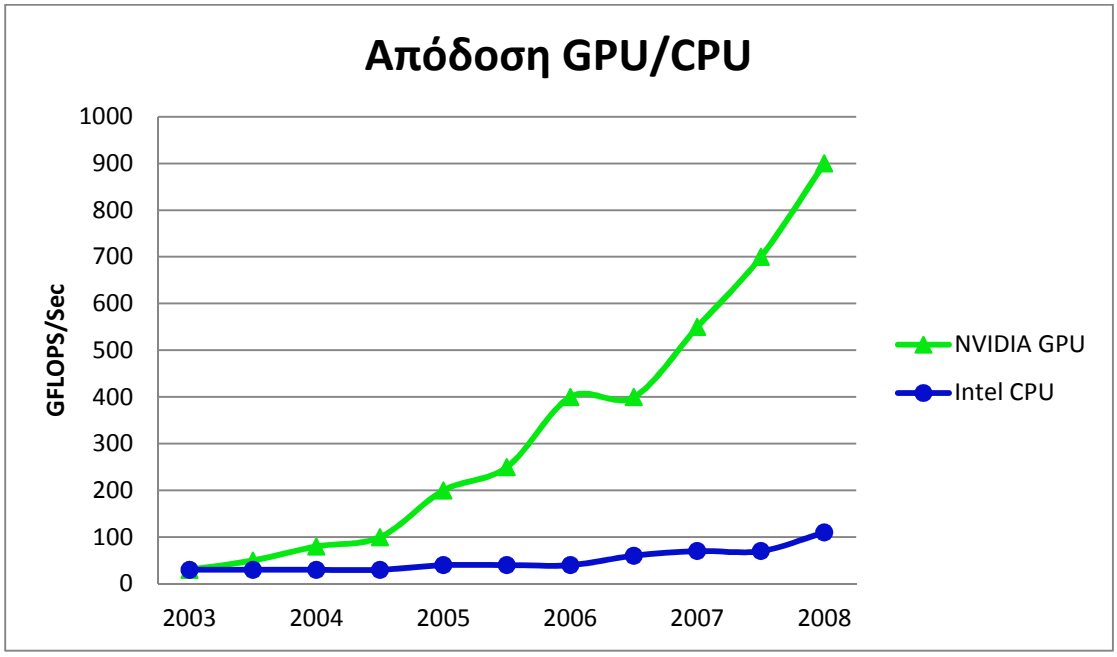
Όμως από το 2003 η αυξανόμενη αυτή βελτίωση σταμάτησε διότι έφτασε σε ένα όριο. Λόγω διαφόρων θεμάτων που αφορούσαν την κατανάλωση ενέργειας, δεν ήταν δυνατό να αυξηθεί άλλο η συχνότητα του ρολογιού μίας μοναδικής CPU. Από τότε οι κατασκευαστές έχουν στραφεί σε πολυπύρηνα μοντέλα, όπου κάθε πυρήνας έχει την δική του μονάδα επεξεργασίας και μπορεί να εκτελεί πράξεις ανεξάρτητα, και απολύτως παράλληλα με τους υπολοίπους. Αυτό όμως προϋποθέτει κατάλληλη συνεργασία μεταξύ του προγράμματος και του υλικού, έτσι ώστε να μπορεί να κατανεμηθεί κατάλληλα ο φόρτος επεξεργασίας. Συνεπώς η αλλαγή αυτή είχε τεράστιο αντίκτυπο στην νοοτροπία του σχεδιασμού προγραμμάτων[7].

Για να μπορέσουν οι προγραμματιστές να εκμεταλλευτούν τις αυξημένες δυνατότητες που προσφέρουν οι πολυπύρηνοι επεξεργαστές και να προσθέσουν πιο εξελιγμένες δυνατότητες στα προγράμματά τους πρέπει να κάνουν την μετάβαση από τα σειριακά προγράμματα στα πολύνηματικά. Επίσης πρέπει να γίνει τροποποίηση των ήδη υπαρχόντων προγραμμάτων. Ένα παλιό σειριακό πρόγραμμα που θα εγκατασταθεί σε ένα καινούργιο υπολογιστή με έναν πολυπύρηνο επεξεργαστή θα τρέχει μόνο σε έναν πυρήνα. Αυτό θα επιφέρει σημαντική επιβράδυνση, διότι ο κάθε πυρήνας μπορεί να έχει λιγότερες επεξεργαστικές δυνατότητες από τις ανάγκες του προγράμματος, αφού πλέον δίνεται έμφαση στον αριθμό των πυρήνων και στην συνεργασία τους και όχι στην χρήση πυρήνων με μεγάλη επεξεργαστική ισχύ. Βέβαια η λογική του παράλληλου προγραμματισμού δεν είναι καινούργια. Για δεκαετίες αναπτύσσονταν παράλληλα προγράμματα τα οποία όμως έτρεχαν γενικά σε ακριβούς υπολογιστές. Μόνο μερικές πολύ ειδικές εφαρμογές μπορούσαν να δικαιολογήσουν την χρήση τόσο ακριβών μηχανημάτων.

Όμως ακόμα και με την παραλληλοποίηση των προγραμμάτων, όσο η επεξεργασία γίνεται στην CPU μπορούμε να φτάσουμε μέχρι ένα όριο απόδοσης. Αυτό γίνεται αφενός μεν διότι δεν μπορούν να χωρέσουν άπειροι πυρήνες μέσα σε ένα chip επεξεργαστή αλλά και λόγω του τρόπου σχεδιασμού των CPUs. Εάν θέλουμε να πετύχουμε μεγαλύτερη απόδοση στα προγράμματά μας πρέπει να στραφούμε στον προγραμματισμό των GPUs. Οι CPUs είναι σημαντικά πιο αργές από τις GPUs λόγω της διαφορετικής φιλοσοφίας σχεδιασμού της κάθε μίας η οποία οφείλεται στον διαφορετικό σκοπό για τον οποίο δημιουργήθηκαν.

2.3 GPUs versus CPUs

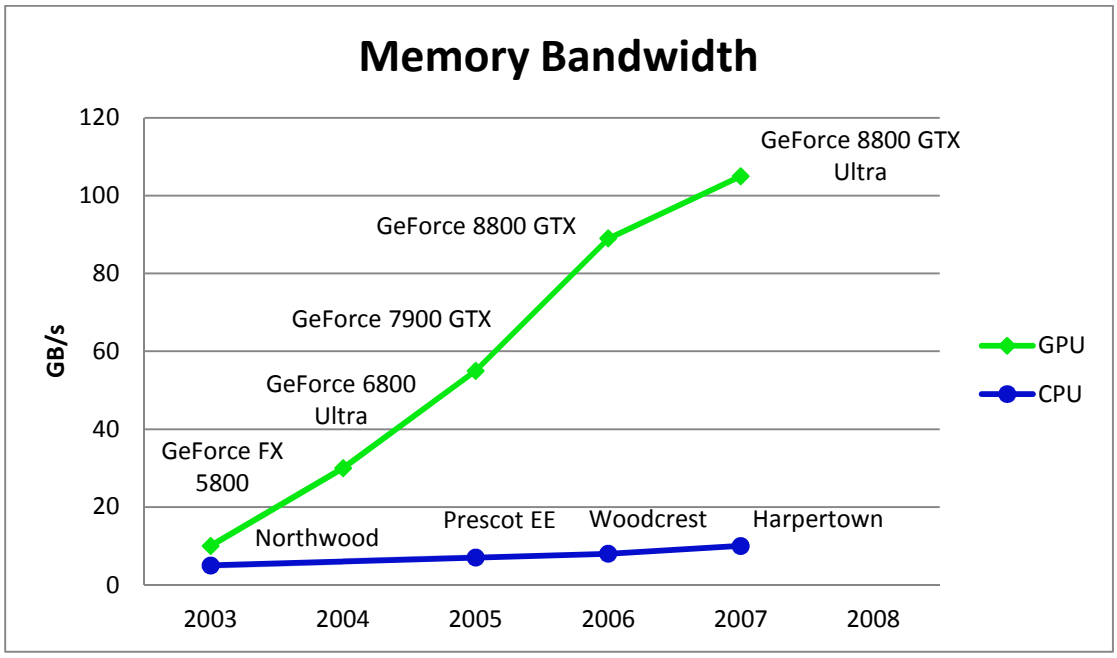
Από το 2003 οι GPUs έρχονται πρώτες στον αγώνα της απόδοσης όσον αφορά τις πράξεις κινητής υποδιαστολής. Όσο η βελτίωση της απόδοσης των επεξεργαστών γενικού σκοπού επιβραδύνεται, η απόδοση των επεξεργαστών των καρτών γραφικών, βελτιώνεται συνεχώς, όπως φαίνεται και στο παρακάτω σχήμα. Αυτή η μεγάλη διαφορά απόδοσης έχει οδηγήσει πολλούς προγραμματιστές εφαρμογών, να μεταφέρουν ορισμένα τμήματα του κώδικά τους που απαιτούν μεγαλύτερη επεξεργαστική ισχύ στην GPU. Αυτό είναι λογικό διότι όσο περισσότερη δουλειά υπάρχει για να γίνει, τόσο περισσότερό αυξάνονται τα περιθώρια που έχουμε για βελτιστοποίηση.



Η διαφορά αυτή στην απόδοση, όπως προαναφέρθηκε, οφείλεται στις θεμελιώδεις σχεδιαστικές φιλοσοφίες με τις οποίες σχεδιάστηκαν οι GPUs και οι CPUs. Οι επεξεργαστές γενικού σκοπού έχουν σχεδιαστεί και βελτιστοποιηθεί για την απόδοση σειριακού κώδικα. Δίνουν δηλαδή έμφαση στο flow control και στο memory caching. Υπάρχει μια πολύπλοκη μονάδα ελέγχου η οποία φροντίζει έτσι ώστε εντολές από ένα μόνο νήμα να μπορούν να εκτελεστούν παράλληλα ή ακόμα και εκτός σειράς. Επίσης υπάρχουν μεγάλης χωρητικότητας μνήμες cache οι οποίες χρησιμοποιούνται για να μειώσουν τον χρόνο που χρειάζεται για να έρθουν δεδομένα και εντολές στην μνήμη, το οποίο είναι σημαντικό σε πολύπλοκες εφαρμογές[7].

Όμως κανένα από τα δύο προαναφερθέντα χαρακτηριστικά δεν συμβάλει σε ταχύτερο χρόνο επεξεργασίας. Οπότε ακόμα και οι καινούριοι πλέον πολυπύρηνοι επεξεργαστές γενικού σκοπού, που έχουν την δυνατότητα παράλληλης εκτέλεσης πολλών εντολών, αποτελούνται από έναν αριθμό πυρήνων που είναι σχεδιασμένοι για την ταχεία εκτέλεση σειριακών μόνο προγραμμάτων. Από την άλλη πλευρά οι κάρτες γραφικών είναι σε θέση να παρέχουν πολλά πλεονεκτήματα στην παράλληλη επεξεργασία δεδομένων.

Όσον αφορά την μνήμη, οι GPUs λειτουργούν με περίπου το 10 πλάσιο memory bandwidth απ' ότι οι CPUs. Χρησιμοποιώντας τα πιο απλά μοντέλα μνήμης, χωρίς ιδιαίτερες αρχιτεκτονικές βελτιστοποιήσεις, οι GPUs είναι ικανές να μεταφέρουν από 80GB/s έως 100GB/s. Ενώ οι CPUs προβλέπεται ότι λόγω των άλλων απαιτήσεων που πρέπει να ικανοποιήσουν παράλληλα (παλιά προγράμματα και παλιά λειτουργικά συστήματα), για τα επόμενα 3 χρόνια δεν προβλέπεται να ανεβούν πάνω από τα 20GB/s.



Η γενική φιλοσοφία των καρτών γραφικών είναι να βελτιστοποιήσουν την εκτέλεση για έναν πολύ μεγάλο αριθμό νημάτων, έτσι ώστε να μπορεί να γίνει παράλληλη επεξεργασία πολλών τμημάτων της οθόνης και το αποτέλεσμα να βγαίνει σε realtime χρόνο σε όλες τις περιοχές. Η GPU δηλαδή είναι σχεδιασμένη για να παρέχει έντονο και παράλληλο υπολογισμό και άρα πολύ περισσότερα τρανζίστορ είναι αφιερωμένα στην επεξεργασία δεδομένων παρά στο caching και στο flow control.

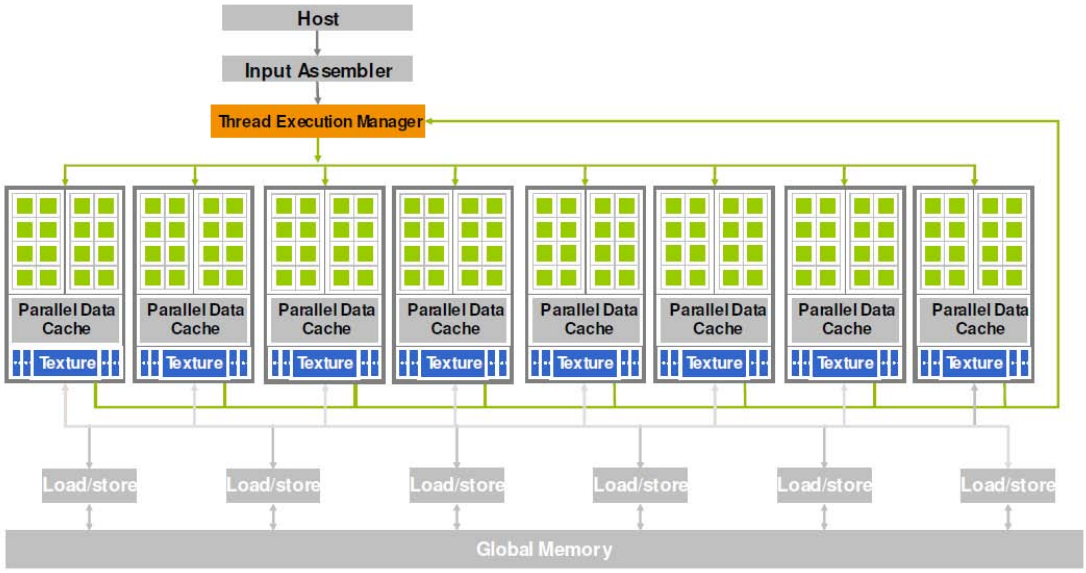
Πιο συγκεκριμένα στην GPU μπορεί ο ίδιος κώδικας να εκτελείται απολύτως παράλληλα με κάθε νήμα να χρησιμοποιεί απλά διαφορετικές θέσεις μνήμης. Άρα πολύ εύκολα μπορεί να αντιμετωπιστούν προβλήματα και αλγόριθμοι που μπορούν να εκφραστούν ως παράλληλος υπολογισμός δεδομένων. Επειδή λοιπόν εκτελείται το ίδιο πρόγραμμα για κάθε στοιχείο, δεν χρειάζεται να έχουμε πολύπλοκο flow control.

Επίσης, το hardware είναι σε θέση να εκμεταλλευτεί τον μεγάλο αριθμό νημάτων που είναι διαθέσιμος για εκτέλεση, έτσι ώστε όταν μερικά νήματα περιμένουν για αργές προσπελάσεις μνήμης, η GPU να μην μένει αδρανής. Πολλές εφαρμογές οι οποίες επεξεργάζονται μεγάλα κομμάτια δεδομένων (όπως η επεξεργασία εικόνας και βίντεο) μπορούν να επωφεληθούν από αυτό το χαρακτηριστικό. Επομένως δεν χρειαζόμαστε μεγάλες μνήμες cache, επειδή ο χρόνος που χρειάζονται οι μεταφορές δεδομένων μπορεί να επικαλυφθεί από τον υπολογισμό δεδομένων. Στην GPU υπάρχουν μικρές μνήμες cache για κάθε νήμα, έτσι ώστε όταν όλα τα νήματα χρειάζονται να χρησιμοποιήσουν την ίδια περιοχή μνήμης, να μην πηγαίνουν όλα στην DRAM προκαλώντας συμφόρηση. Αυτό έχει ως αποτέλεσμα, μεγαλύτερη περιοχή του chip να είναι διαθέσιμη για υπολογισμούς κινητής υποδιαστολής. Αυτό γίνεται ιδιαίτερα κατανοητό με το παρακάτω σχήμα[7].

Σήμερα οι GPUs έχουν ξεκινήσει να αντιμάχονται τις CPUs όσον αφορά την επεξεργαστική δύναμη και ο προγραμματισμός των GPUs έχει επεκταθεί σε πολλούς τομείς. Η πλατφόρμα CUDA της NVIDIA αποτελεί πλέον το πιο ευρέως χρησιμοποιούμενο μοντέλο προγραμματισμού, με την OpenCL επίσης να προσφέρεται ως ανοιχτό πρότυπο. Επίσης λόγω των αυξανόμενων απαιτήσεων για 3D real time γραφικά από την αγορά, ο επεξεργαστής της κάρτας γραφικών GPU (Graphic Processor Unit), έχει εξελιχθεί σε έναν πολυπύρηνο, πολυνηματικό, παράλληλο επεξεργαστή με μεγάλη επεξεργαστική ισχύ, και πολύ μεγάλο εύρος ζώνης μνήμης (memory bandwith).

2.4 Αρχιτεκτονικά χαρακτηριστικά μιας GPU

Η αρχιτεκτονική μιας τυπικής GPU φαίνεται στο παρακάτω σχήμα.

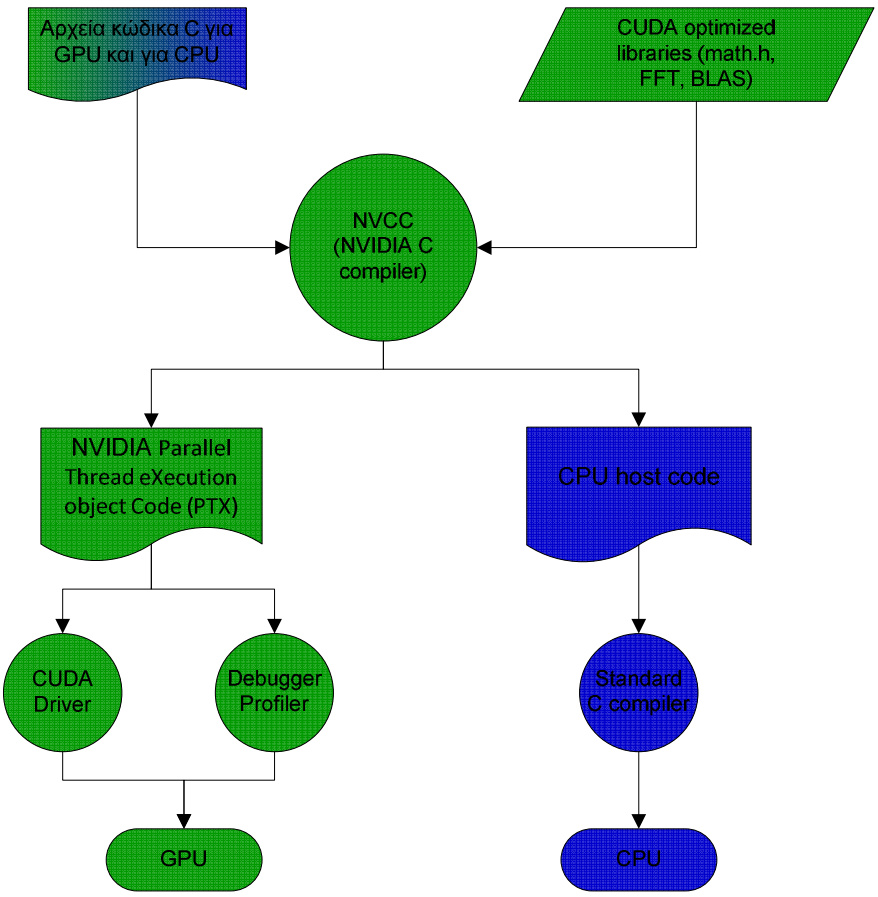


Παρατηρούμε ότι αποτελείται από 16 παράλληλους μικροεπεξεργαστές SMs (highly threaded Streaming Multiprocessors). Κάθε SM αποτελείται από 8 επεξεργαστές SPs (Streaming Processors), οπότε συνολικά έχουμε 128 SPs. Κάθε SP έχει μία μονάδα αθροιστή (MAD unit) και μια μονάδα για πολλαπλασιασμό (MUL unit). Οπότε κάθε unit τρέχει στα 1,35 GHz. Επίσης υπάρχουν ειδικές μονάδες που εκτελούν εντολές κινητής υποδιαστολής όπως ο υπολογισμός της τετραγωνικής ρίζας. Το μέγεθος της Global Memory (DRAM) είναι συνήθως 1,5 MB . Η DRAM αυτή διαφέρει από την DRAM των CPUs και είναι αυτή που συνήθως χρησιμοποιείται για να φορτώνονται HD εικόνες και texture πληροφορίες που χρειάζονται σε 3D παιχνίδια. Όταν χρησιμοποιείται από εφαρμογές δουλεύει ως εξωτερική cache με πολύ υψηλό memory bandwidth αλλά με κάπως μεγάλο latency. Όμως εάν το chip προγραμματιστεί κατάλληλα, το υψηλό αυτό latency μπορεί να επικαλυφτεί από το πολύ υψηλό bandwidth.

Μετάφραση ενός προγράμματος C σε CUDA C

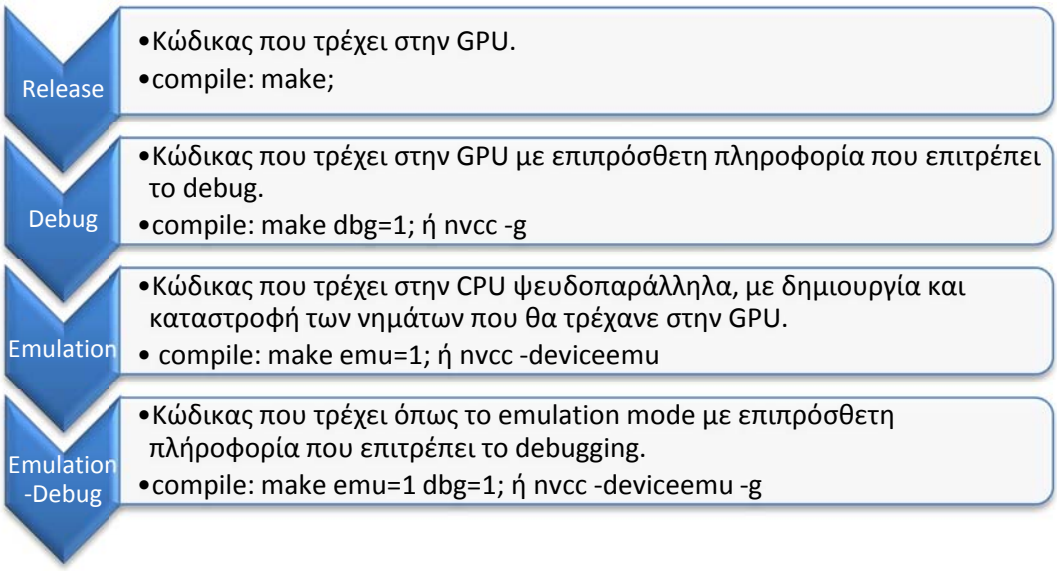
Η CUDA είναι ένα ετερογενές περιβάλλον προγραμματισμού όπου η CPU και η GPU χρησιμοποιούνται από μία εφαρμογή. Ο προγραμματιστής δηλαδή φτιάχνει

ένα αρχείο που περιέχει και κώδικα για την CPU και κώδικα για την GPU. Ο κώδικας της CPU μπορεί να επωφεληθεί χρησιμοποιώντας κάποιες ειδικές GPU συναρτήσεις που περιέχονται στις βιβλιοθήκες FFT και BLAS.



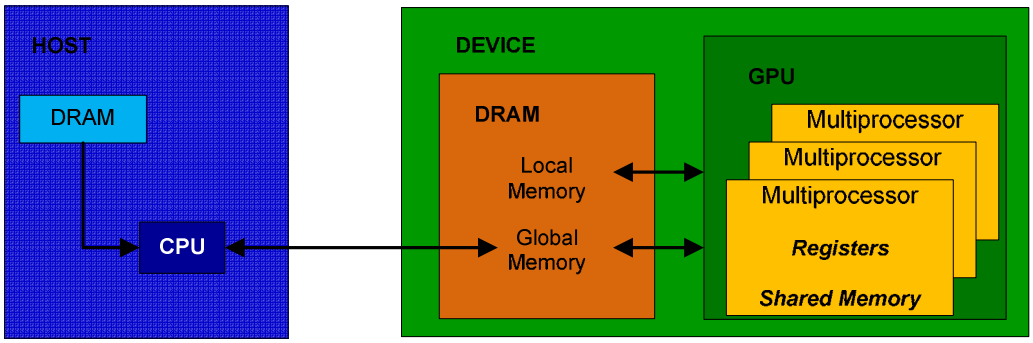
Αφού λοιπόν γράψουμε τον κώδικα έπειτα πρέπει να γίνει compile με τον NVIDIA C Compiler (NVCC) ο οποίος είναι υπεύθυνος για να διαχωρίζει τον κώδικα της GPU (device code) από τον κώδικα για την CPU (host code). Έπειτα ο host code γίνεται compile από τον compiler του συστήματος, ενώ ο device code, μετατρέπεται σε PTX code (Parallel Thread eXecution object code) . Ο NVCC είναι στην ουσία ένας compiler driver ο οποίος περιέχει ρουτίνες για την μεταγλώττιση και host και device κώδικα. Ο PTX κώδικας που παράγεται είναι στην ουσία μια ενδιάμεση αναπαράσταση που μετά πρέπει να μεταγλωττιστεί και να εφαρμοστεί στην συγκεκριμένη κάρτα όπου θα τρέξει το πρόγραμμα. Τα εκτελέσιμα της CUDA περιέχουν τον PTX κώδικα σε συνδυασμό με τον targed code, έτσι ώστε καινούργιος target κώδικας να μπορεί να δημιουργηθεί άμεσα (on the fly) εάν το εκτελέσιμο τρέξει σε κάποια διαφορετική κάρτα.

Ανάλογα με τα flag που θα θέσουμε όταν κάνουμε compile, μπορούμε να μεταφράσουμε το πρόγραμμα μας έτσι ώστε να ΄τρέχει΄ σε release mode,σε debug mode, σε emulation mode ή σε debug-emulation mode.



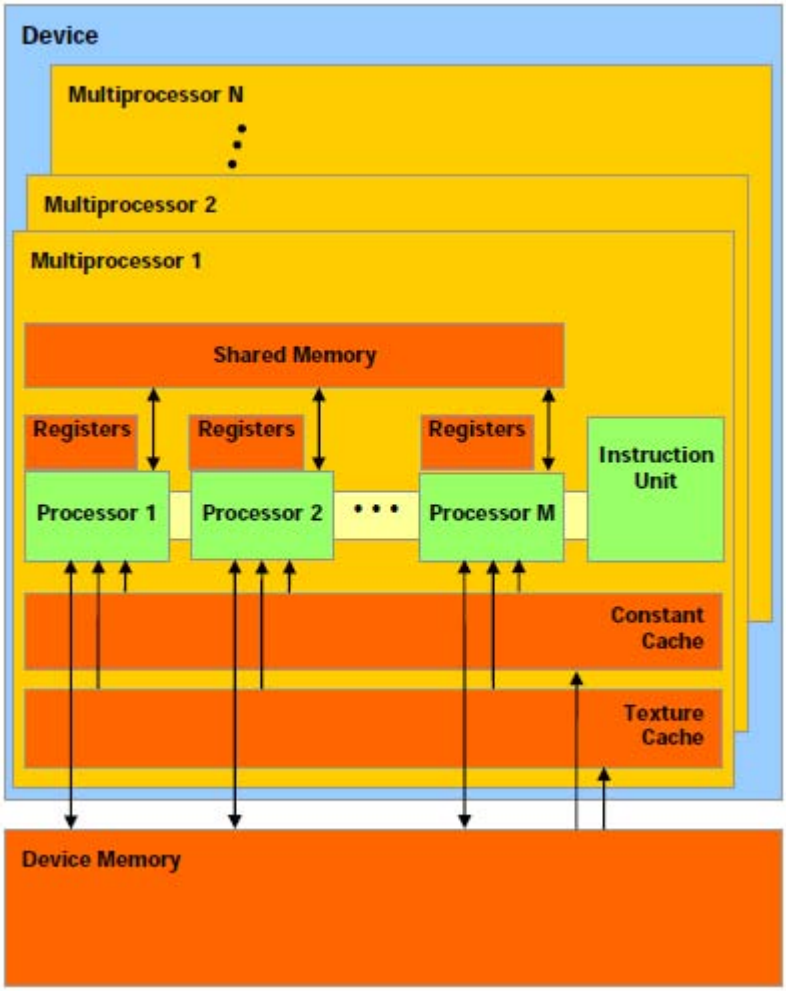
Η μνήμη στην GPU

Η CPU και η GPU έχουν η κάθε μία τον δικό της χώρο διευθύνσεων αλλά και οι δύο ελέγχονται από τον κώδικα που τρέχει στην CPU. Για να εκτελέσει ένα πρόγραμμα στην device πρέπει να γίνει δέσμευση της απαραίτητης μνήμης, να μεταφερθούν τα δεδομένα από την μνήμη host στην μνήμη device, και μετά την εκτέλεση του προγράμματος να μεταφερθούν τα αποτελέσματα πίσω στην μνήμη host και να ελευθερωθεί η δεσμευμένη μνήμη. Για να αναφερθεί η CPU στον χώρο διευθύνσεων της GPU (device memory) ενεργεί περίπου όπως θα ενεργούσε και στον δικό της χώρο μνήμης (host memory). Υπάρχουν ειδικές συναρτήσεις οι οποίες δίνουν την δυνατότητα στον κώδικα host να δεσμεύσει, να αποδεσμεύσει device μνήμη καθώς και να αντιγράφει δεδομένα από την μία μνήμη στην άλλη. Για παράδειγμα για δέσμευση θέσης καλείται η cudaMalloc η οποία λειτουργεί όπως και η malloc, αλλά επιστρέφει έναν pointer ο οποίος αναφέρεται στη device memory. Κατ' αντιστοιχία , για μεταφορά δεδομένων υπάρχει η cudaMemcpy, και για αποδέσμευση της μνήμης η cudaFree.



Στην μνήμη device τα δεδομένα μπορούν να αναπαρασταθούν γραμμικά, η σαν πίνακες CUDA. Υπάρχουν διάφορες συναρτήσεις (cudaMallocPitch,cudaMemcpy3D) που δίνουν στον προγραμματιστή την δυνατότητα να επιλέξει τον τρόπο αναπαράστασης που επιθυμεί. Οι συναρτήσεις αυτές δεσμεύουν τον ζητούμενο χώρο με κατάλληλο τρόπο ώστε να γίνεται βελτιστοποίηση του χρόνου πρόσβασης των δεδομένων.

Η device η αλλιώς global memory δεν είναι η μοναδική μνήμη που υπάρχει στην GPU. Υπάρχει επίσης η constant memory, η texture memory, η shared memory, η local memory, και οι καταχωρητές όπως φαίνεται και στο σχήμα που ακολουθεί. ***Η constant memory*** είναι cached και επιτρέπει μόνο πράξεις ανάγνωσης και όχι εγγραφής, και γι αυτό προσφέρει πιο γρήγορη και πιο παράλληλη πρόσβαση από την global memory. Η ανάγνωση από την costant memory κοστίζει όσο η ανάγνωση από έναν καταχωρητή, εάν όλα τα νήματα του κώδικα διαβάζουν την ίδια θέση μνήμης. ***Η texture memory,*** είναι επίσης cached από την global memory και προσφέρει μικρότερο bandwidth demand αλλά ίδιο fetch latency. Υποστηρίζει πιο γρήγορες μεταφορές ακόμα και σε random memory accesses, αλλά δεν είναι συνεπής εάν θέλουμε να διαβάσουμε μία τιμή η οποία έχει αλλάξει από ένα άλλο νήμα του προγράμματός μας διότι η τιμή της ανανεώνεται μόνο σε κάθε νέο kernel call***. Η shared memory,*** είναι πολύ πιο γρήγορη από τις προηγούμενες επειδή βρίσκεται επάνω στο chip και όχι την DRAM. Είναι χωρισμένη σε κομμάτια (banks) από τα οποία μπορεί να υπάρξει ταυτόχρονη πρόσβαση δεδομένων ανάμεσα στα νήματα ενος kernel. ***Η local memory,*** είναι τμήμα της device memory και δεν είναι cached. Οπότε η ταχύτητα ανάγνωσης από εκεί είναι εφάμιλλη με αυτήν της global memory. Στην local memory αποθηκεύονται μεταβλητές που αφορουν το κάθε νήμα ξεχωριστά και δεν χωράνε στους καταχωρητές. ***Οι registers,*** είναι το σημείο στο οποίο αποθηκεύονται μεταβλητές που χρησιμοποιούνται συχνά από τα νήματα, και είναι περιορισμένης χωρητικότητας. Η πρόσβαση στους registers έχει σχεδόν μηδενικό κόστος και κάθε νήμα μπορεί να δει μόνο τους δικούς του registers.



Συναρτήσεις πυρήνα και νήματα

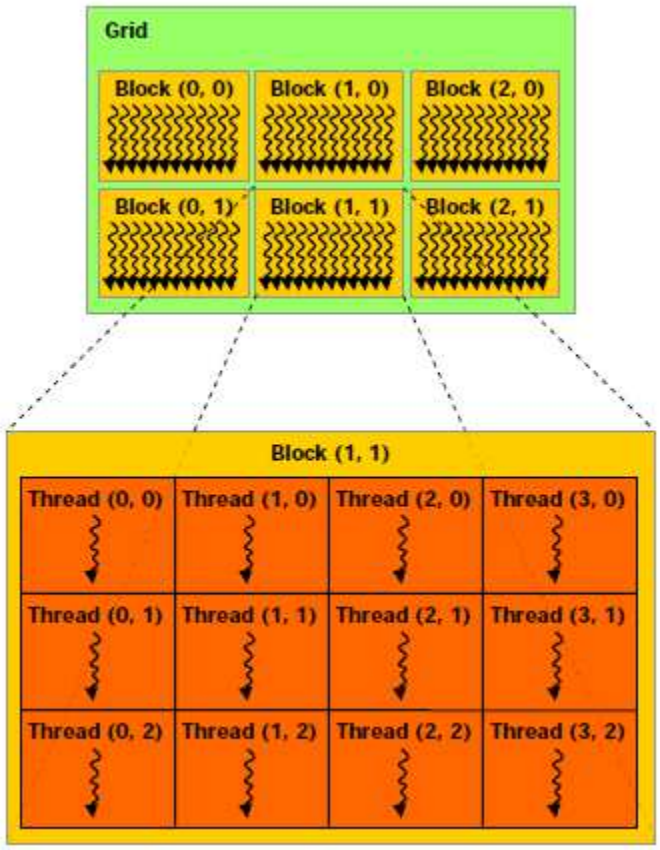
Η CUDA επεκτείνει την C επιτρέποντας στον προγραμματιστή να ορίζει συναρτήσεις C (kernels) οι οποίες όταν κληθούν θα εκτελεστούν N φορές παράλληλα από N διαφορετικά νήματα (threads) της GPU. Οι kernels δηλαδή είναι στην ουσία συναρτήσεις C με κάποιους περιορισμούς. Οι kernels δεν μπορούν να έχουν πρόσβαση στην μνήμη της CPU και δεν μπορούν να επιστρέφουν τιμές (void return type). Επίσης δεν μπορεί να έχουν μεταβλητό αριθμό ορισμάτων ή στατικές μεταβλητές, και δεν μπορεί να είναι αναδρομικές συναρτήσεις. Τα ορίσματα των kernels μεταφέρονται αυτόματα στην device memory με την κλήση της συνάρτησης. Εάν ένα όρισμα είναι δείκτης, τότε μεταφέρεται απλά η τιμή του δείκτη και όχι το περιεχόμενο στο οποίο δείχνει. Για να είναι έγκυρη αυτή η τιμή πρέπει πιο πριν ο προγραμματιστής να έχει δεσμεύσει μία θέση μνήμης στην device memory με την cudaMalloc, να έχει αντιγράψει τα δεδομένα που χρειάζονται με την cudaMemcpy, και έπειτα να περάσει την διεύθυνση που παρήγαγε η cudaMalloc, σαν όρισμα στην συνάρτηση kernel.

Επειδή η CUDA υποστηρίζει και συναρτήσεις που τρέχουν στην GPU αλλά και συναρτήσεις που τρέχουν στην CPU, υπάρχουν κάποια αναγνωριστικά που δηλώνουν την φύση της κάθε συνάρτησης. Οι συναρτήσεις που δεν φέρουν αναγνωριστικό ή καλούνται με το αναγνωριστικό ***__host__*** είναι συναρτήσεις που καλούνται από την CPU και εκτελούνται στην CPU. Οι συναρτήσεις με το αναγνωριστικό ***__global__*** είναι συναρτήσεις που καλούνται από την CPU αλλά εκτελούνται στην GPU, ενώ αυτές με το αναγνωριστικό ***__device__*** καλούνται και εκτελούνται στην GPU και δεν μπορούν να κληθούν από την CPU. Οι kernels είναι οι συναρτήσεις που φέρουν το αναγνωριστικό __global__ και καλούνται από την CPU με την σύνταξη που φαίνεται στον παρακάτω κώδικα.

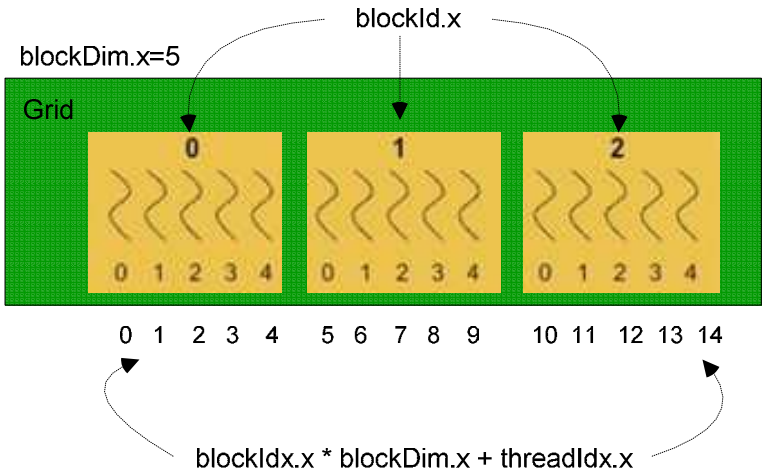
```
__global__ void my_kernel(float A[N][N], float B[N][N],float C[N][N])
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        C[i][j] = A[i][j] + B[i][j];
}
int main()
{
        ...
        dim3 dimGrid(X, Y);
        dim3 dimBlock(N, N);
        my_kernel<<< dimGrid, dimBlock>>>(A, B, C);
        ...
}
```

Η μόνη διαφορά με την κλασική κλήση συνάρτησης στην C είναι το τμήμα με τις τρείς αγκύλες (‘<<<…>>>’) που παρεμβάλλεται ανάμεσα στο όνομα της συνάρτησης και στα ορίσματα. Τα ορίσματα που μπαίνουν μέσα στις αγκύλες, ορίζουν τις διαστάσεις του grid και του block που χρησιμοποιούνται για την εκτέλεση της συνάρτησης στην GPU.

Ο λόγος ύπαρξης του grid και του block είναι στην ουσία η καλύτερη οργάνωση του κώδικα μας όπως γίνεται κατανοητό από το παρακάτω σχήμα. Αντί να ορίζουμε έναν αριθμό threads ανάλογα με τις φορές που θέλουμε να τρέξουμε τον κώδικα, μπορούμε να οργανώσουμε τα thread μας καλύτερα έτσι ώστε να μπορούμε να χειριστούμε αποδοτικά και την πρόσβαση στην μνήμη. Αν για παράδειγμα έχουμε έναν πίνακα 5x10 και θέλουμε κάθε νήμα να έχει πρόσβαση σε ένα στοιχείο του πίνακα, δημιουργούμε ένα block αντιστοίχων διαστάσεων.



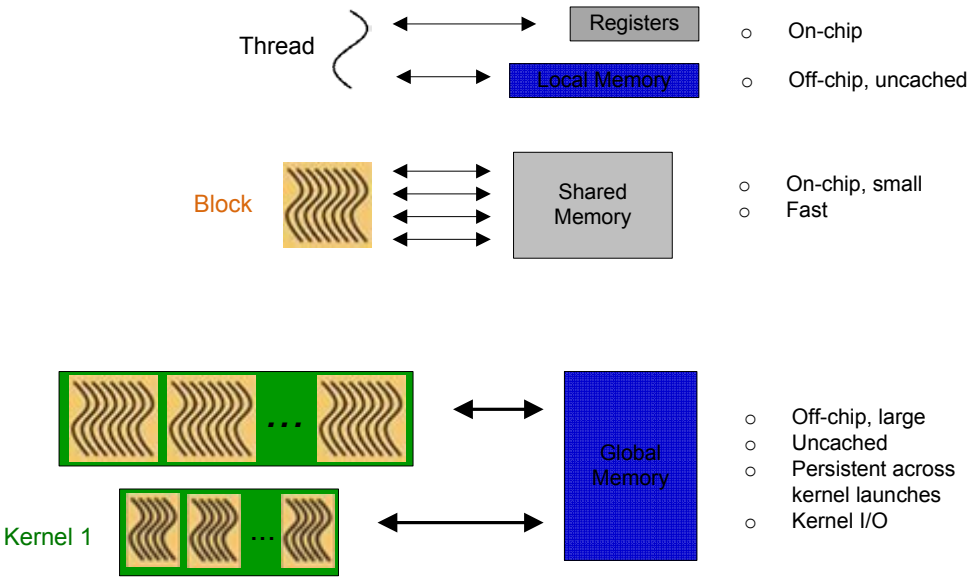
Παρατηρούμε ότι τα IDs των threads αντιστοιχούν στις θέσεις του πίνακα. Με το που δημιουργείται ο kernel, κάθε νήμα αυτόματα έχει πρόσβαση σε κάποιες μεταβλητές που αφορούν τα χαρακτηριστικά του. Οι μεταβλητές αυτές περιέχουν το μέγεθος του grid (dim3 gridDim) και το μέγεθος του block (dim3 blockDim) που είχαν οριστεί από τον προγραμματιστή κατά την κλήση του kernel, καθώς και το ID του block στο οποίο βρίσκεται το συγκεκριμένο νήμα (dim3 blockIdx) και το ID του ίδιου του νήματος (dim3 threadIdx). Στο παρακάτω παράδειγμα βλέπουμε την χρήση των παραπάνω μεταβλητών σε ένα grid με 3 blocks με 5 threads το καθένα.



Το hardware έχει την ελευθερία να ορίσει σε ποιον πυρήνα θα τρέξουν τα διαφορετικά blocks και σε αυτό οφείλεται η δυνατότητα που έχουμε να εκτελούμε το ίδιο πρόγραμμα σε διαφορετικές κάρτες γραφικών. Εάν για παράδειγμα έχουμε έναν kernel με 8 blocks και μία κάρτα με 4 πολυεπεξεργαστές το υλικό θα ορίσει κάθε πολυεπεξεργαστής να τρέξει 4 blocks. Ενώ αν είχαμε 2 πολυεπεξεργαστές, θα έπρεπε ο κάθε ένας να τρέξει 4 blocks. Όλα αυτά γίνονται χωρίς καμία αλλαγή στον πηγαίο κώδικα. Οπότε ένα πρόγραμμα με έναν μεγαλύτερο αριθμό kernels μπορεί να τρέξει σε ένα laptop και σε έναν supercomputer χωρίς καμία αλλαγή στον κώδικά του.

Πιο συγκεκριμένα μια κάρτα γραφικών από την σειρά 10 της NVIDIA, περιέχει 240 thread processors οι οποίοι εκτελούνε kernel threads. Αυτοί οι επεξεργαστές είναι οργανωμένοι σε 30 πολυεπεξεργαστές, ο καθένας από τους οποίους περιέχει 8 thread processors, μια μονάδα διπλής ακρίβειας και μια shared memory η οποία υποστηρίζει την συνεργασία των blocks. Κάθε multiprocessor εκτελεί τους υπολογισμούς του σε SIMT (single instruction multiple thread fashion) το οποίο είναι παρόμοιο με το SIMD με την διαφορά ότι το SIMT είναι πιο ευέλικτο επιτρέποντας σε κάθε thread να εκτελείται ανεξάρτητα με την δικό του χώρο μνήμης και τους δικούς του καταχωρητές. Το SIMT επιτρέπει στους προγραμματιστές να φτιάξουν παράλληλο κώδικα για τελείως ανεξάρτητα νήματα ή κώδικα που επωφελείται και από την συνεργασία των νημάτων.

Στο παρακάτω σχήμα φαίνονται τα είδη της μνήμης που αναφέρθηκαν στο προηγούμενο κεφάλαιο, και το πως αυτά χρησιμοποιούνται από τα threads, τα blocks και τις διαφορετικές κλήσεις kernel.



Τέλος όσον αφορά τον συγχρονισμό μεταξύ CPU και GPU, όλες οι κλήσεις kernel είναι ασύγχρονες και ο έλεγχος επιστρέφει στην CPU με το που θα κληθούν. Όμως οι kernel θα τρέξουν σειριακά στην GPU με την σειρά που κλήθηκαν. Συναρτήσεις όμως όπως η cudaMemcpy όμως είναι σύγχρονες και επιστρέφουν μόνο όταν η πράξη αντιγραφής έχει τερματιστεί. Επίσης ο προγραμματιστής έχει και τη δυνατότητα να τρέξει ασύγχρονες συναρτήσεις μεταφοράς δεδομένων για να επιτύχει κάποιες βελτιστοποιήσεις στο πρόγραμμά του. Σε αυτήν την περίπτωση μπορεί να χρησιμοποιηθεί η συνάρτηση cudaThreadSynchronise η οποία μπλοκάρει τον κώδικα του host μέχρι να ολοκληρωθούν όλες οι συναρτήσεις που τρέχουν στην GPU. Ακόμα ο προγραμματιστής έχει την δυνατότητα να ελέγξει και τον παραλληλισμό των νημάτων μέσα σε ένα block με τον barrier __syncthreads(). Κανένα νήμα δεν μπορεί να ξεπεράσει τον barrier αυτόν μέχρι όλα τα νήματα να έχουν φτάσει εκεί. Αυτός ο barrier χρειάζεται διότι η σειρά με την ο οποία τα νήματα εκτελούνται δεν είναι προκαθορισμένη, για να αποφύγουμε προβλήματα ανάγνωσης μετά από εγγραφή όταν τα νήματα χρησιμοποιούν την shared memory.

2.5 GPU και video decoding

Από τα παραπάνω λοιπόν συμπεραίνουμε ότι η GPU εάν προγραμματιστεί σωστά, έχει να μας προσφέρει μεγάλες δυνατότητες παραλληλισμού. Υπάρχουν πολλές εφαρμογές υποψήφιες για μεταφορά στην GPU. Μία από αυτές είναι οι αποκωδικοποίηση video.

Είναι γεγονός ότι στην καθημερινότητά μας χρησιμοποιούμε συχνά εφαρμογές video οι οποίες έχουν την ανάγκη για κωδικοποίηση και αποκωδικοποίηση αυτού. Δύο διαδικασίες χρονοβόρες που προσθέτουν μεγάλο υπολογιστικό φόρτο στο κάθε σύστημα. Η διαδικασία κωδικοποίησης όμως, αν και απαιτητική, δεν εκτελείται τόσες φορές όσες η αποκωδικοποίηση, και συνήθως εκτελείται σε εξειδικευμένα μηχανήματα. Για παράδειγμα για την μετάδοση μίας ταινίας, ένας κεντρικός server εκτελεί την κωδικοποίηση του σήματος, το στέλνει πάνω από το δίκτυο, και έπειτα εκατομμύρια δέκτες πρέπει να την αποκωδικοποιήσουν, σε real time χρόνο, για να προσφέρουν την απαιτούμενη από τον χρήστη ποιότητα.

Οι περισσότερες GPU που κατασκευάστηκαν από το 1995 υποστηρίζουν το YUV χρωματικό πρότυπο και διαθέτουν και διάφορα harware χαρακτηριστικά (overlays) που είναι σημαντικά για την αναπαραγωγή ψηφιακού βίντεο ενώ πολλές GPUs που κατασκευάστηκαν από το 2000 υποστηρίζουν βασικά στοιχεία του MPEG

όπως το motion compensation και το iDCT. Αυτή η διαδικασία του harware accelerated video decoding όπου τμήματα της διαδικασίας της αποκωδικοποίησης βίντεο και της προ-επεξεργασίας του βίντεο, μεταφέρονται στην GPU, είναι γνωστή ως ‘GPU hardware assisted video decoding’.

Κεφάλαιο 3

Περιγραφή του AVS video standard.

3.1 Εισαγωγή.

Τα τελευταία χρόνια με την διάδοση των πολυμέσων στην ζωή μας, όλο και πιο πολλά συστήματα χρειάζεται να χρησιμοποιήσουν εφαρμογές συμπίεσης και αποσυμπίεσης ήχου και εικόνας. Η ανάγκη αυτή παρουσιάζεται από τα πιο καθημερινά συστήματα (τηλεόραση ,dvd players) μέχρι και στα πιο εξειδικευμένα (κινητά 3^{ης} γενιάς, laptops) , και έχει γίνει επιτακτική τον τελευταίο καιρό λόγω της συνεχούς αυξανόμενης ζήτησης για υψηλή ποιότητα και ανάλυση βίντεο από την αγορά. Είναι προφανές ότι όσο αυξάνεται η ποιότητα του βίντεο τόσο αυξάνεται και η πληροφορία που αυτό περιέχει, και κατά συνέπεια το μέγεθός του. Επομένως , για να μεταδοθεί αυτός ο μεγάλος όγκος πληροφορίας πάνω από το δίκτυο ή να αποθηκευτεί σε κάποιο μέσο αποθήκευσης, πρέπει πρώτα να συμπιεστεί και έπειτα για να δούμε το αποτέλεσμα στην οθόνη μας, πρέπει με κάποιον τόπο να την αποσυμπιέσουμε. Οι πράξεις αυτές συμπίεσης και αποσυμπίεσης βίντεο πραγματοποιούνται από τις εφαρμογές video encoder και decoder αντίστοιχα. Οι εφαρμογές αυτές, λόγω της πολύπλοκης δουλειάς που πρέπει να εκτελέσουν, μέσα σε μικρό χρόνο, προσθέτουν μεγάλο φόρτο εργασίας στο σύστημα.

Το AVS video standard είναι ένα πρότυπο συμπίεσης ήχου και εικόνας, το οποίο δημιουργήθηκε το 2005 από την Λαϊκή Δημοκρατία της Κίνας με σκοπό να την αποδεσμεύσει από την αγορά αδειών για τα εμπορικά MPEG standard αλλά και να οδηγήσει την βιομηχανία της σε μία αναγνώριση από τις υπόλοιπες βιομηχανίες του δυτικού κόσμου όπου ακόμα θεωρείται σαν μία διέξοδος για μαζική παραγωγή με περιορισμένες δυνατότητες σχεδιασμού. Το AVS αντιμάχεται το ανοικτό πρότυπο H.264 για την αντικατάσταση του MPEG-2 ενώ το κλειστό πρότυπο MPEG-4 είναι εκτός συναγωνισμού λόγω της ακριβής τιμής των αδειών του[1].

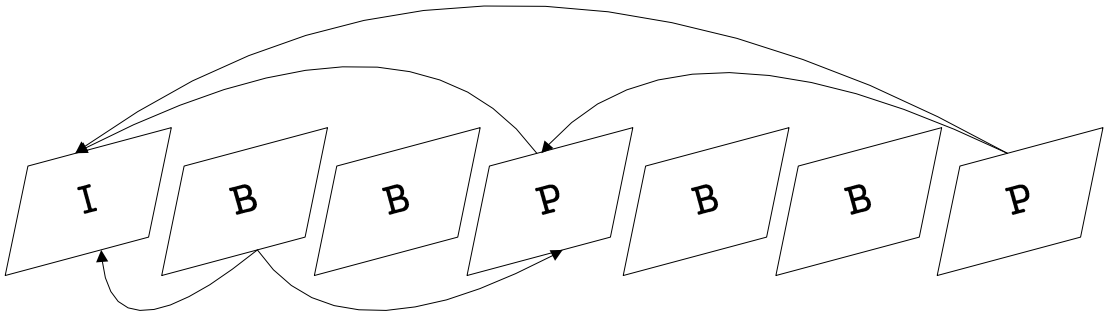
Εμείς στην εργασία αυτή θα ασχοληθούμε με την μελέτη του αποκωδικοποιητή του προτύπου AVS (AVS decoder).

3.2 Γενικές πληροφορίες.

Η διαδικασία που ακολουθεί ο AVS decoder για την αποκωδικοποίηση ενός βίντεο, είναι παρόμοια με αυτήν που ακολουθούν και τα υπόλοιπα πρότυπα όπως το MPEG-4 και το H.264. Η ικανότητα συμπίεσης του AVS είναι ίδιας τάξης με αυτήν του H.264 ενώ οι απαιτήσεις του σε υπολογιστική ισχύ και μνήμη είναι μικρότερες[4].

Ο αλγόριθμος αποκωδικοποίησης τρέχει για κάθε ξεχωριστή εικόνα του βίντεο (frame). Το πρότυπο ορίζει τριών ειδών frames όπως φαίνεται και στο παρακάτω σχήμα. Επίσης τα frames δεν έρχονται με τυχαία σειρά στο bitstream, αλλά επαναλαμβάνεται η αλληλουχία "I-B-B-P-B-B-P" (Group Of Pictures).

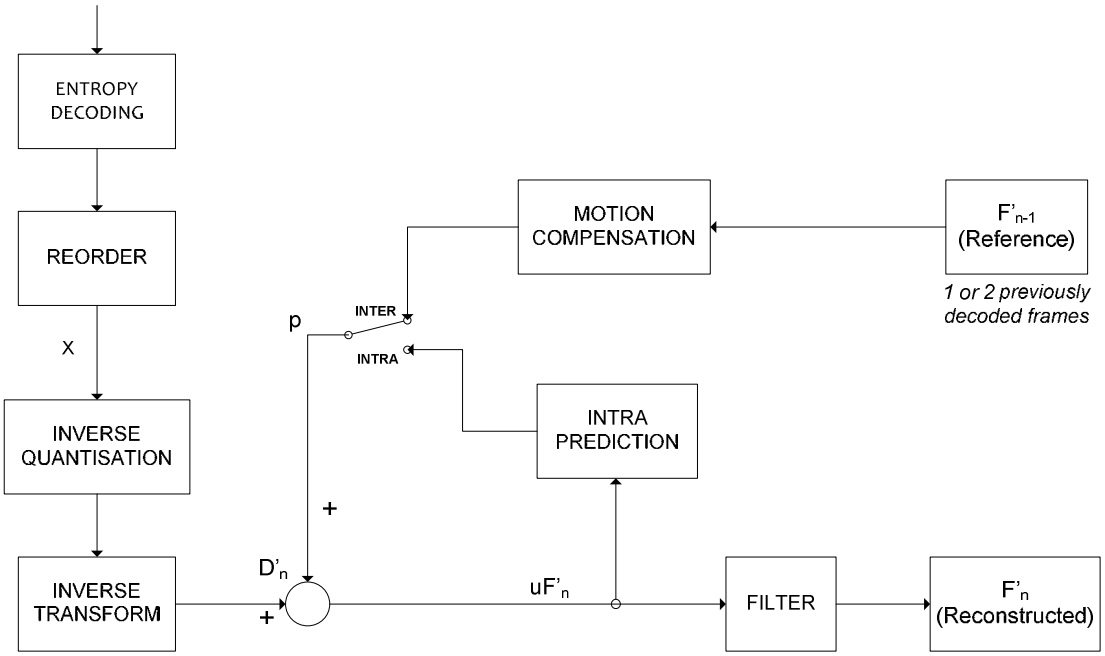
- Τα I-frames (intra coded picture), για την αποκωδικοποίηση των οποίων δεν χρειάζεται να αναφερθούμε σε άλλα frames.
- Τα P-frames (Predictive inter-coded picture), τα οποία εξαρτώνται το πολύ από 2 προηγούμενα frames.
- Τα B-frames (Bidirectional inter-coded picture), τα οποία εξαρτώνται από ένα προηγούμενο αλλά και ένα επόμενο frame. [5]



Τα βήματα που ακολουθεί ο AVS decoder, όπως φαίνονται στο παρακάτω σχήμα, είναι τα εξής. Αρχικά ο decoder λαμβάνει μία συμπιεσμένη αλληλουχία από bits (bitstream) και εκτελεί ***entropy decoding*** σε αυτήν έτσι ώστε να παράγει το κατάλληλο σετ από κβαντοποιημένους συντελεστές (quantinised coefficients). Το ***entropy decoding*** αρχικά μετατρέπει το bit stream σε διακριτές τιμές και έπειτα από αυτές τις τιμές παράγει τα quantinized DCT coefficients[2].

Έπειτα ακολουθεί το ενδεχόμενο ***reordering.*** Εάν στο bitstream δεν υπάρχουν καθόλου B-frames, τότε η διαδικασία της αποκωδικοποίησης θα χρησιμοποιήσει τα frames με την σειρά που τα πήρε. Στην αντίθετη περίπτωση όμως, σύμφωνα με τον αλγόριθμο του AVS η σειρά των frames είναι διαφορετική

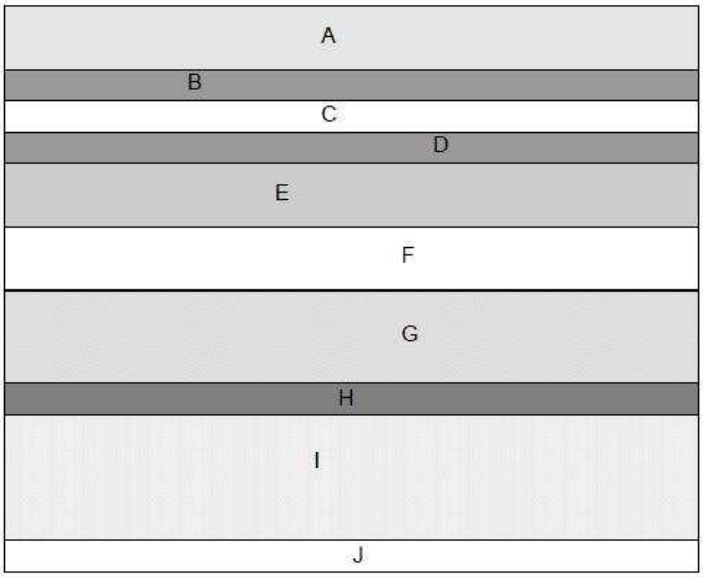
ανάμεσα στο bitstream, την διαδικασία αποκωδικοποίησης και την σειρά με την οποία θα προβληθούν στην οθόνη[5].



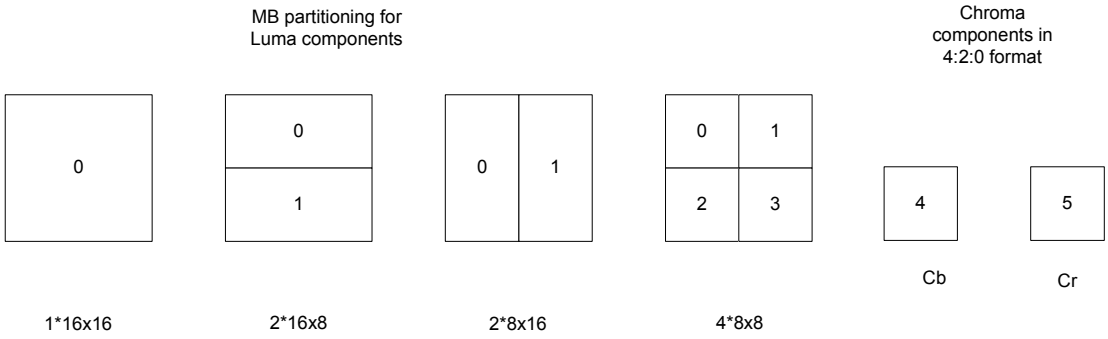
Όπως παρατηρούμε τα επόμενα βήματα στην διαδικασία της αποκωδικοποίησης είναι το ***inverse quantisation*** με το οποίο πολλαπλασιάζουμε όλα τα coefficients με τον αριθμό Q (ο οποίος είχε χρησιμοποιηθεί από τον encoder για να διαιρέσει τους αριθμούς μειώνοντας έτσι τον όγκο της πληροφορίας) και το ***inverse transform***, όπου μετατρέπουμε τα coefficients από την περιοχή των συχνοτήτων (frequency domain), στην περιοχή των pixels.

Ακολουθεί η διαδικασία του ***motion compensation*** για όλα τα Inter macroblocks και του ***intra prediction*** για όλα τα intra macroblocks. Σε αυτό το σημείο πρέπει να αναφέρουμε κάποια πράγματα για την οργάνωση του κάθε frame.

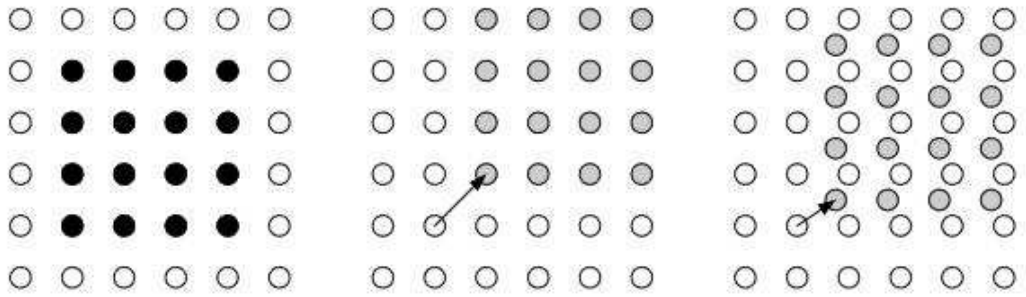
Κάθε frame είναι χωρισμένο σε παράλληλες περιοχές , όπως φαίνεται και στο επόμενο σχήμα, που ονομάζονται slices, των οποίων η διαδικασία αποκωδικοποίησης είναι τελείως ανεξάρτητη.



Επίσης το κάθε frame χωρίζεται σε κομμάτια των 16x16 pixels που ονομάζονται macroblocks(MB). Τα πάνω αριστερά όρια των MB δεν πρέπει να βρίσκονται έξω από το όριο του frame. Παρακάτω φαίνεται ο τρόπος με τον οποίο χωρίζονται τα lyminance MB για να χρησιμοποιηθούν στην διαδικασία του motion compensation. Ο τόπος χωρισμού (partitioning) του MB σε blocks κυμαίνεται από το μεγάλο partitioning size που αποτελείται από ένα block 16x16, έως το μικρό partitioning size που αποτελείται από 4 block των 8x8.



Το ***motion compensation*** είναι ένας βασικός μετασχηματισμός στην διαδικασία αποκωδικοποίησης βίντεο, και εκμεταλλεύεται το γεγονός ότι στις περισσότερες περιπτώσεις δεν υπάρχουν πολλές διαφορές μεταξύ δύο διαδοχικών frames[3]. Εάν για παράδειγμα το βίντεο μας προέρχεται από μία εφαρμογή video call, το background θα μένει το ίδιο και το μόνο που θα αλλάζει είναι η περιοχή του προσώπου του ομιλητή. Οπότε ο αλγόριθμος χωρίζει το frame σε κομμάτια 16x16 pixel (MB) και υπολογίζει την διαφορά που έχει το κάθε MB ανάμεσα σε διαδοχικά frames. Έτσι δεν χρειάζεται για κάθε frame να μεταφέρεται πληροφορία που αφορά όλο το frame. Αρκεί να μεταφέρονται μόνο οι διαφορές στις τιμές των MB. Η πληροφορία αυτή υπάρχει στα motion vectors.



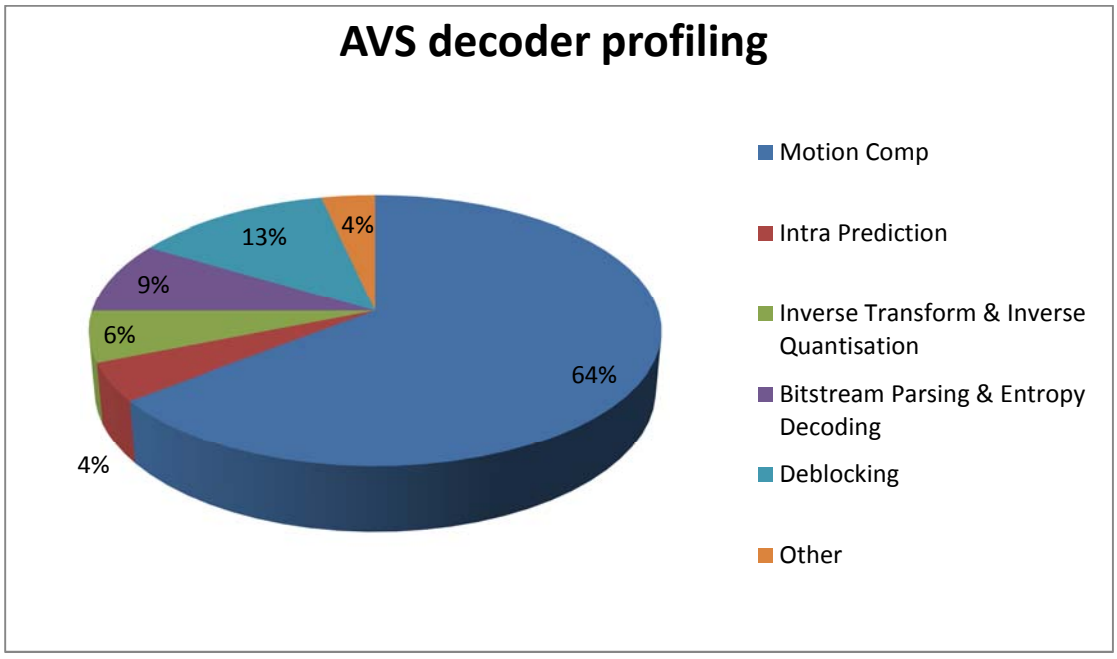
Χρειαζόμαστε ένα ξεχωριστό motion vector για κάθε υποδιαίρεση του κάθε MB. Κάθε motion vector πρέπει να κωδικοποιηθεί στο bitstream, το οποίο πρέπει επίσης να υποδεικνύει τον τρόπο υποδιαίρεσης των MB. Εάν επιλέξουμε τον πρώτο τρόπο partitioning όπως αναφέρθηκε παραπάνω, τότε θα χρειαστούμε μικρότερο αριθμό bits για να περιγράψουμε την επιλογή των motion vectors, αλλά κάθε motion vector πιθανότατα θα έχει περισσότερη πληροφορία. Εάν επιλέξουμε τον τελευταίο τρόπο partitioning, τότε θα χρειαστούμε περισσότερα bits για να περιγράψουμε την επιλογή των motion vectors, αλλά το κάθε ένα θα περιέχει λιγότερη πληροφορία. Γενικά η επιλογή του τρόπου partitioning είναι σημαντική και έχει αποτελέσματα στην απόδοση. Προτιμάται το μικρό partition size για περιοχές με πολλές λεπτομέρειες και το μεγάλο partition size για ομογενείς περιοχές[2]. Παραδείγματος χάριν στο παράδειγμά μας με το video call θα επιλέγαμε για το backround το μεγάλο partition size και για το πρόσωπο του ομιλητή το μικρό partition size.

Ο decoder χρησιμοποιεί τα motion vectors που έχει δημιουργήσει ο encoder για να δημιουργήσει την περιοχή πρόβλεψης και να αποκωδικοποιήσει το υπόλοιπο (residual) block, έπειτα το προσθέτει σ αυτά που είχε, και ανασυγκροτεί μία έκδοση του αρχικού block[2]. Όμως τα luma και τα chroma components δεν υπάρχουν στα frames αναφοράς, οπότε τα δημιουργούμε από κοντινές, ήδη κωδικοποιημένες περιοχές (interpolation process).

Τα προαναφερθέντα γίνονται για όλα τα MB τα οποία έχουν χαρακτηριστεί από το encoding ως inter. Για τα intra MB, ακολουθείται η διαδικασία του ***intra prediction***. Κατά την διαδικασία αυτή δεν χρησιμοποιούμε motion vectors, αλλά pixels από γειτονικά intra MB επειδή είναι συνήθως παρόμοια. Εάν δεν υπάρχουν intra MB, εφαρμόζουμε έναν αλγόριθμο επέκτασης των ορίων του MB, και γι'αυτό μετά την διαδικασία της αποκωδικοποίησης πρέπει να προσθέσουμε λίγο “θόρυβο” στα άκρα των MB για να έχουμε ένα εξομαλυμένο frame (Deblocking process).

3.3 Φόρτος εργασίας και παραλληλοποίηση.

Στο προηγούμενο κεφάλαιο λοιπόν, είδαμε περιληπτικά τα βασικά κομμάτια του αποκωδικοποιητή του προτύπου AVS. Εύκολα συμπεραίνουμε ότι τον μεγαλύτερο φόρτο εργασίας τον έχει η πολύπλοκη διαδικασία του motion compensation. Αυτό φαίνεται και από το παρακάτω σχήμα.



Παρατηρούμε λοιπόν ότι για να βελτιώσουμε την απόδοση του AVS decoder, μεταφέροντας κάποια κομμάτια του κώδικά του στην GPU, πρέπει να αρχίσουμε από την διαδικασία του motion compensation, όπου έχουμε και τα περισσότερα περιθώρια βελτιστοποίησης.

Το motion compensation έχει πολλές δυνατότητες παραλληλισμού.

Frame level

Τα frame όπως περιγράψαμε και προηγουμένως, δεν εξαρτώνται όλα από προηγούμενα και επόμενα frame. Τα I-frame, αποκωδικοποιούνται ανεξάρτητα και τα P, B frame εξαρτώνται από το πολύ 2 άλλα frame. Επίσης αναφέρθηκε παραπάνω η επαναλαμβανόμενη αλληλουχία “$I-(B-B-P)_n$” (GOP). Ανάμεσα σε κάθε GOP, μπορούμε να έχουμε τελείως παράλληλους υπολογισμούς, αφού είναι ανεξάρτητα μεταξύ τους. Επίσης όλα τα I frames θα μπορούσαν να υπολογιστούν στην αρχή εφόσον δεν εξαρτώνται από κανένα άλλο frame.

Αυτό το επίπεδο παραλληλοποίησης όμως έχει μεγάλες απαιτήσεις σε μνήμη, ειδικά για video πολύς υψηλής ανάλυσης (UHD) διότι πρέπει να μπορούμε να διατηρούμε στην μνήμη της GPU 3 frames την φορά. Αυτό δεν μας απασχολεί όμως, διότι οι καινούργιες GPU έρχονται με μέγεθος Device Memory, 512 MegaBytes στα οποία μπορούμε άνετα να αποθηκεύσουμε την πληροφορία που χρειαζόμαστε. Ένα ακόμα θέμα είναι το memory latency που χρειάζεται για να μεταφερθούν τα δεδομένα από την device memory στην local memory έτσι ώστε να γίνει ο υπολογισμός από τα threads. Όμως όπως προαναφέραμε η GPU έχει την δυνατότητα να κρύψει τις μεταφορές μνήμης με τον υπολογισμό. Άρα πάλι με τον κατάλληλο προγραμματισμό, μπορούμε να έχουμε μεγάλες βελτιώσεις στην απόδοση.

Slice Level

Αναφέραμε παραπάνω ότι το frame είναι χωρισμένο σε slices των οποίων ο υπολογισμός είναι ανεξάρτητος. Οπότε όσα περισσότερα slices έχουμε, τόσο αυξάνει η δυνατότητα παραλληλισμού. Δυστυχώς όμως, συνήθως ο αριθμός των slices είναι περιορισμένος, διότι παρόλο που ο μεγάλος αριθμός slices θα είχε θετικά αποτελέσματα σε streaming πάνω από αναξιόπιστα δίκτυα, είναι αντιστρόφως ανάλογος με την δυνατότητα συμπίεσης του video από τον encoder. Οπότε συνηθίζεται κάθε frame να έχει και ένα μόνο slice.

Macroblock level

Μία ακόμα προοπτική είναι ο παραλληλισμός σε επίπεδο Macroblock. Σε ένα frame υπάρχουν χιλιάδες MB, τα οποία μπορούν να αποκωδικοποιηθούν παράλληλα. Βέβαια πρέπει να προσέξουμε τις εξαρτήσεις μεταξύ των intra MB , τα οποία χρησιμοποιούν γειτονικά MB για την αποκωδικοποίηση τους. Επίσης εάν εκτελούμε όλον τον υπολογισμό σε επίπεδο MB πρέπει να προσέξουμε και τις εξαρτήσεις που υπάρχουν και στο Deblocking, οι οποίες όμως είναι ίδιες με αυτές που υπάρχουν για το intra-prediction, με ανάγκη περισσότερων block μέσα σε ένα MB. Επομένως πρέπει να λάβουμε υπόψη μας τις εξαρτήσεις που υπάρχουν μέσα στο frame, αφού ορισμένα MB πρέπει να έχουν τελειώσει τον υπολογισμό τους πριν από κάποια άλλα.

Κεφάλαιο 4

Υλοποίηση του AVS σε μία CUDA capable GPU.

4.1 Compile

Για να πετύχουμε τον σκοπό αυτής της εργασίας, ξεκινήσαμε προσπαθώντας να μεταφράσουμε τον κώδικα του AVS χωρίς κάποιον CUDA kernel, με τον NVCC compiler. Για να γίνει αυτό έπρεπε να γίνουν κάποιες αλλαγές στο makefile του avs. Όπως φαίνεται και στο παράρτημα Α, το makefile του αρχικού κώδικα διαφέρει αρκετά από αυτό που χρειάζεται για τα τρέξει στην GPU.

Τα header files που βρισκόταν στα «HEADERS» του αρχικού makefile τώρα ορίζονται ως «C_DEPS». Τα αρχεία C που περιέχει το πρόγραμμά μας βρίσκονται κάτω από την ετικέτα «CFILES» αντί για την ετικέτα «SOURCES» και τα αρχεία CUDA τοποθετούνται στην ετικέτα «CUFILES». Επίσης πρέπει να οριστεί μία ετικέτα "ROOTDIR" κάτω από την οποία θα βρίσκεται ένα μονοπάτι (path) που θα δείχνει στον υποκατάλογο που έχει εγκατασταθεί το CUDA SDK, στο σημείο όπου υπάρχουν τα πηγαία αρχεία της C (ΠΧ*:/home/nepapape/NVIDIA_GPU_Computing_SDK/C/src*). Εναλλακτικά θα πρέπει να μεταφέρουμε τον κώδικά μας σε εκείνον τον φάκελο. Ακόμα πρέπει να γίνει include το αρχείο common.mk το οποίο περιέχει τις απαραίτητες πληροφορίες για να μεταφραστεί το πρόγραμμά μας από τον NVCC. Τέλος πρέπει να οριστεί μία ετικέτα «BINDIR» η οποία θα είναι ίση με «./» το οποίο υποδεικνύει στον compiler τα το που να τοποθετήσει τα εκτελέσιμα που θα παράγει.

Επόμενο βήμα είναι να κάνουμε compile τον κώδικα. Όπως αναφέρθηκε και παραπάνω εάν θέλουμε να εκτελέσουμε τον κώδικά μας στην κάρτα γραφικών , η εντολή που πρέπει να πληκτρολογήσουμε είναι "**make clean; make**". Η εντολή αυτή θα αφαιρέσει πιθανά προηγούμενα εκτελέσιμα, θα μεταφράσει τον κώδικά μας και θα τοποθετήσει το εκτελέσιμο σε έναν καινούργιο φάκελο «Release» που θα δημιουργήσει μέσα στον φάκελό που εργαζόμαστε. Όταν τρέχουμε το πρόγραμμά μας σε release mode, πρέπει να σεβόμαστε όλες τις προδιαγραφές που ορίζει η γλώσσα CUDA. Δηλαδή δεν μπορούμε να καλέσουμε μία συνάρτηση C από έναν kernel (πχ printf) ,δεν μπορούμε να προσπελάσουμε μεταβλητές που είναι δεσμευμένες στην GPU από συναρτήσεις C, και επομένως το debugging του κώδικα είναι προβληματικό.

4.2 Run

Το επόμενο βήμα είναι το τρέξιμο του προγράμματος μας. Αφού εντοπίσουμε το εκτελέσιμο που έχει δημιουργήσει το στάδιο της μεταγλώττισης, πρέπει να προσθέσουμε στον φάκελο ορισμένα αρχεία που δέχεται ο AVS decoder ως είσοδο. Κατ αρχάς πρέπει να υπάρχει στον φάκελο το βίντεο "video.avs" που θα αποκωδικοποιήσουμε. Επίσης πρέπει να υπάρχει το αρχείο παραμέτρων "decoder.cfg" στο οποίο ορίζουμε το όνομα του video που θα χρησιμοποιήσουμε καθώς και το όνομα του αρχείου yuv ("video.yuv") στο οποίο θα αποθηκευτούν τα αποτελέσματα της αποκωδικοποίησης.

Η εντολή εκτέλεσης του προγράμματος μας είναι " **./ldecode.exe decoder.cfg**". Έχουμε την δυνατότητα, βάζοντας σε σχόλια τα defines των OUTPUT_ENABLE και WRITE_ENABLE να απενεργοποιήσουμε την έξοδο πληροφοριών για κάθε frame που εμφανίζονται στην κονσόλα, και την παραγωγή των αρχείων εξόδου που δημιουργούνται στον φάκελο εκτέλεσης. Αυτό είναι απαραίτητο έτσι ώστε οι μετρήσεις μας να μην περιλαμβάνουν πρόσθετες καθυστερήσεις που δεν αφορούν την αποκωδικοποίηση του video μας.

4.3 Debugging

Για να κάνουμε αποσφαλμάτωση του κώδικά μας, μπορούμε να μεταφράσουμε το πρόγραμμα μας σε debug ή σε emulation mode. Η emulation mode χρησιμοποιείται και όταν θέλουμε να τρέξουμε το πρόγραμμά μας σε έναν υπολογιστή ο οποίος δεν έχει μία CUDA capable κάρτα γραφικών. Η εντολή μεταγλώττισης είναι ***"make clean emu=1; make emu=1;"*** και το εκτελέσιμο τοποθετείται στον φάκελο «emurelease». Πρέπει να τονίσουμε ότι το τρέξιμο του κώδικα σε emulation mode είναι σημαντικά πιο αργό, διότι για να μπορέσει το σύστημα να προσομοιώσει τα kernel threads της GPU, δημιουργεί και καταστρέφει ένα ένα τα αντίστοιχα νήματα στην CPU.

Επίσης μπορούμε να χρησιμοποιήσουμε και τα προγράμματα GDB ή και valgrind για την αποσφαλμάτωση των προγραμμάτων CUDA. Για να γίνει αυτό πρέπει να μεταγλωττίσουμε το πρόγραμμα με την εντολή είναι *"**make clean emu=1 dbg=1; make emu=1 dbg=1**;"* και έπειτα να χρησιμοποιήσουμε το εκτελέσιμο που θα βρίσκεται στον φάκελο "emudebug" κατάλληλα.

Ένα από τα προβλήματα που αντιμετωπίσαμε μεταφέροντας τον κώδικα του AVS στην GPU ήταν το χαρακτηριστικό του αρχικού κώδικα να δηλώνει μεταβλητές μέσα σε header files. Όταν ένα header λοιπόν γινόταν include και από ένα αρχείο C

και από ένα αρχείο CU, αντιμετωπίζαμε multiple definitions errors. Εν τέλη μετατρέψαμε όλες τις δηλώσεις μεταβλητών σε extern στα header files και τις ενσωματώσαμε στο αρχείο “variables.c” το οποίο πλέον γίνεται compile μαζί με τα υπόλοιπα.

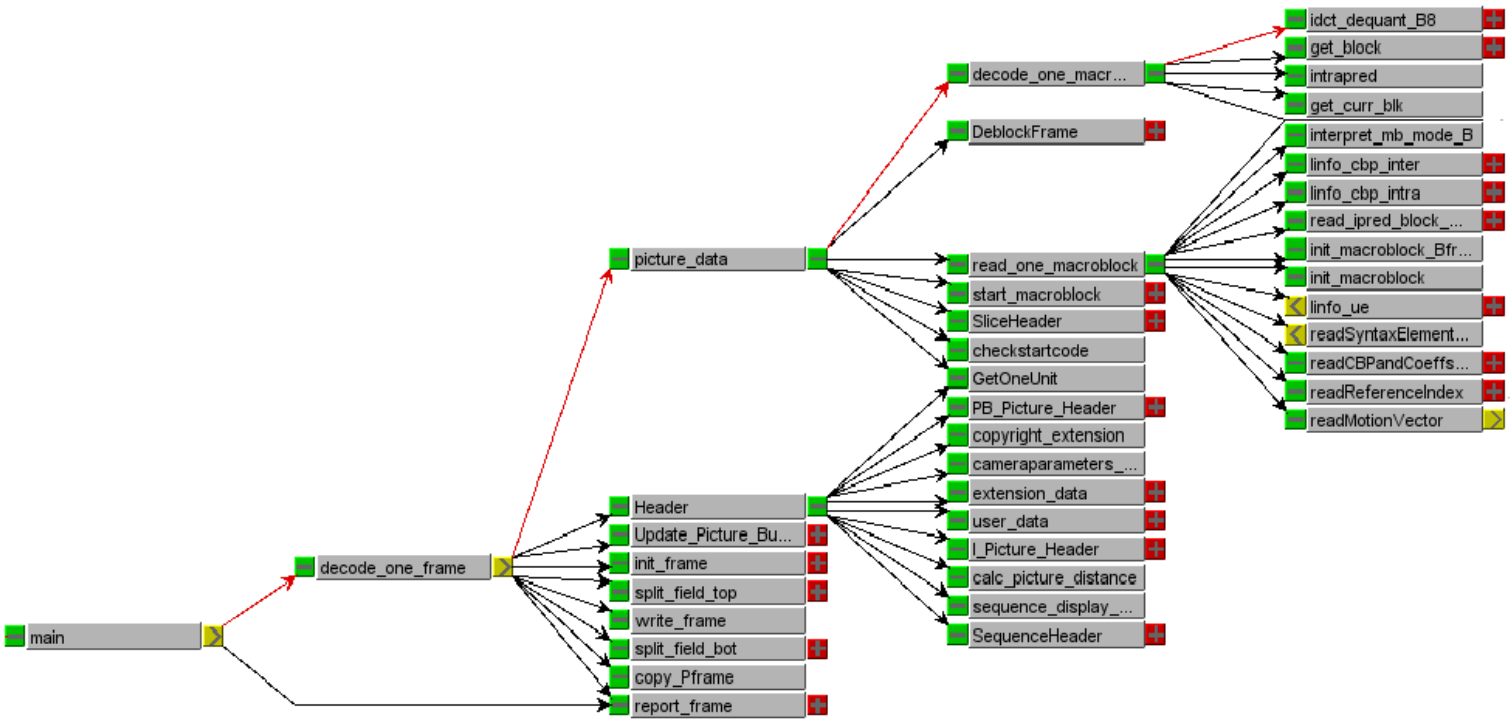
4.4 Υλοποίηση

Για την ανάλυση του αρχικού μας προγράμματος χρησιμοποιήθηκε το εργαλείο Vtune Performance Analyzer. Το εργαλείο αυτό δημιουργήθηκε από την Intel και παρέχει μια ολοκληρωμένη ανάλυση της απόδοσης μιας εφαρμογής για ορισμένες αρχιτεκτονικές . Τρέχοντας τον αρχικό κώδικα στο Vtune, παίρνουμε τον πίνακα 1 που βρίσκεται στο Παράρτημα Σχημάτων. Από τον πίνακα αυτόν βλέπουμε ότι οι συναρτήσεις οι οποίες απασχολούν περισσότερο την CPU είναι η “Inverse_transform_B8”, η “idct_dequant_B8”, η “decode_one_macroblock” και η “get_block”.

Στα πλαίσια της εργασίας αυτής υλοποιήσαμε την “Inverse_transform_B8” την “idct_dequant_B8” και την “get_block” στην γλώσσα CUDA. Από αυτές καταφέραμε να συνδυάσουμε επιτυχημένα με τον υπόλοιπο κώδικα C την “get_block”. Δηλαδή το Interpolation mode του motion compensation.

> *void get_block_c (int ref_frame,int x_pos, int y_pos, struct img_par *img, int block[8][8], unsigned char **ref_pic)*

Το διάγραμμα εκτέλεσης του προγράμματός μας μέχρι την κλήση της get_block φαίνεται στο ακόλουθο σχήμα.



Η get_block όπως φαίνεται και στο παράστημα Β, αποτελείται από 17 διαφορετικούς διπλούς βρόγχους επανάληψης (double nested loops). Το κάθε loop διαβάζει τις δικές του θέσεις μνήμης και εκτελεί μεταβλητό αριθμό επαναλήψεων. Όπως φαίνεται και στο παράρτημα Γ, το πρόγραμμα μας αναδιατάσσεται ως εξής για να τρέξει στην GPU.

Δημιουργούμε 3 καινούργιες συναρτήσεις C και 17 cuda kernels. Η συνάρτηση «get_block_c» είναι η αντίστοιχη της αρχικής «get_block», και καλείται από τα ίδια σημεία που καλούταν εκείνη. Μέσα στην «get_block_c» στα σημεία όπου υπήρχαν τα loop, τώρα υπάρχουν οι αντίστοιχες κλήσεις των kernels. Σε κάθε κλήση kernel, περνάμε στις παραμέτρους τις μεταβλητές που χρειάζεται ο καθένας και θέτουμε κατάλληλα την μεταβλητή που ορίζει το με πόσα νήματα θα εκτελεστεί ο kernel αυτός. Τον αριθμό των νημάτων τον ορίζουμε κάνοντας full loop unrolling. Κάθε φορά που θα εκτελεστεί η get_block_c θα τρέξει ένας μόνο kernel ο οποίος αφού εκτελέσει τους υπολογισμούς του, θα γράψει την θέση μνήμης που του αντιστοιχεί.

Οι μεταβλητές που χρησιμοποιούνται από τους kernel είναι 3 read-only πίνακες και 3 πίνακες όπου αποθηκεύονται τα αποτελέσματα των υπολογισμών μας.

Αρχική προσέγγιση

Με κάθε κλήση της get_block_c αρχικά δεσμεύεται χώρος στην GPU για τις μεταβλητές που θα χρησιμοποιηθούν, αντιγράφονται τα δεδομένα, και έπειτα καλείται ο κατάλληλος kernel. Μετά την κλήση του kernel, τα δεδομένα μεταφέρονται πίσω στην CPU και οι θέσεις μνήμης που είχαν δεσμευτεί αποδεσμεύονται.

Η αποκωδικοποίηση του video είναι επιτυχής, αλλά ο κώδικάς μας έχει αρκετή καθυστέρηση λόγω της μεγάλης επικοινωνίας της CPU με την GPU για μεταφορές μνήμης και δέσμευση θέσεων. Το πρόγραμμα μας τώρα τρέχει στα 1,4 sec/frame.

Πρώτη Βελτιστοποίηση

Δημιουργούμε δύο συναρτήσεις που αναλαμβάνουν την δέσμευση θέσεων μνήμης στην GPU και την αποδέσμευση αυτών. Οι συναρτήσεις αυτές περιγράφονται στο παράρτημα Γ και είναι η initGPU και η freeGPU. Για να εξαλείψουμε την άσκοπη δέσμευση και αποδέσμευση μεταβλητών, οι κλήσεις των συναρτήσεων αυτών τοποθετούνται πριν και μετά την κλήση της decode_one_frame.

```
initGPU();
  while (decode_one_frame(img, input, snr) != EOS);
freeGPU();
```

Η συνάρτηση decode_one_frame καλείται από την συνάρτηση main του κώδικά μας, και εκτελεί τους ίδιους υπολογισμούς για κάθε frame του video. Άρα τώρα οι συναρτήσεις που δεσμεύουν και αποδεσμεύουν θέσεις μνήμης στην GPU, καλούνται μόνο μία φορά. Τώρα η αποκωδικοποίηση του video γίνεται σε 1,3 sec/frame.

Δεύτερη βελτιστοποίηση

Το πρόγραμμά μας ακόμα έχει συχνή επικοινωνία με την GPU λόγω των πολλών αντιγραφών δεδομένων για κάθε frame. Παρατηρούμε ότι οι 2 από τους 3 πίνακες που χρησιμοποιούνται από τους kernel μπορούν να αρχικοποιηθούν μαζί με την δέσμευση θέσεων, άρα μπορούμε να μειώσουμε τις αντιγραφές στην μνήμη της GPU κατα N= αριθμός των frames. Οπότε στην initGPU, προστίθενται και οι κατάλληλες cudaMemcpy.

```
cutilSafeCall(cudaMemcpy(&dev_COEF_HALF[0],(int *)&COEF_HALF[0],
size2 , cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(&dev_COEF_QUART[0], (int *)&COEF_QUART[0],
size3 , cudaMemcpyHostToDevice));
```

Τώρα το πρόγραμμά μας τρέχει σε 1sec/frame.

Τρίτη βελτιστοποίηση

Παρατηρούμε ότι για όλες τις διαδικασίες εγγραφής και ανάγνωσης χρησιμοποιείται η device memory. Για κάθε ανάγνωση/ εγγραφή από την device memory, πληρώνουμε περίπου 400 κύκλους μηχανής. Οπότε μεταφέρουμε 2 από

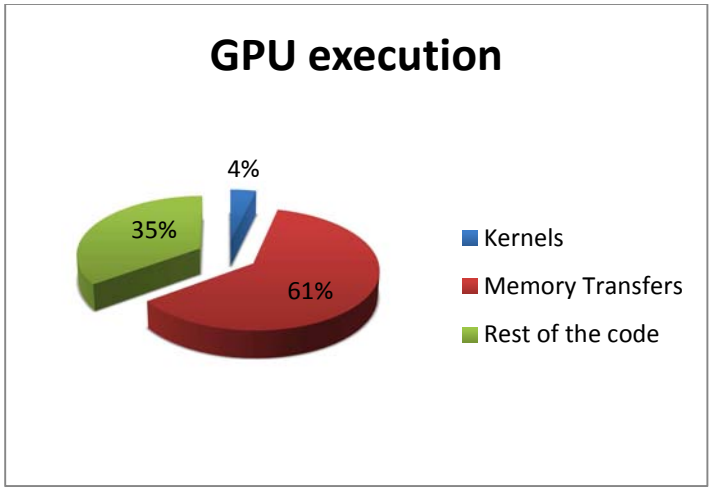
τους πίνακές μας στην texture memory. Παρατηρούμε μικρή βελτίωση και τώρα το πρόγραμμά μας τρέχει σε 0,9 sec/frame.

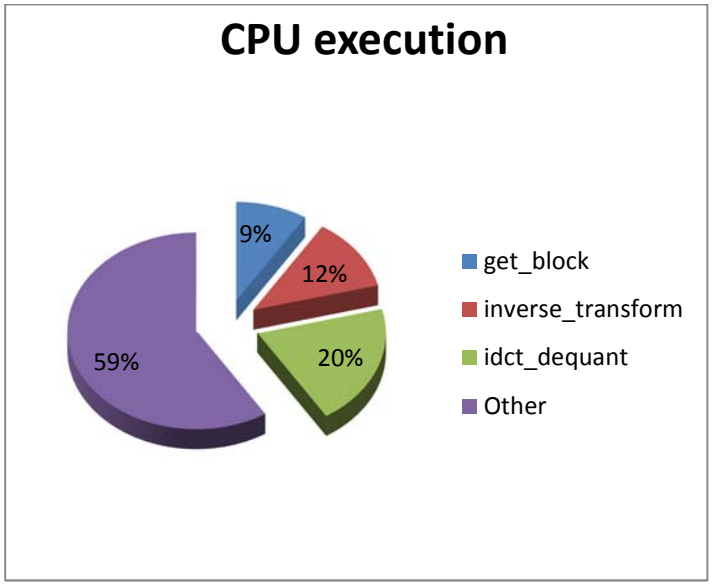
Κεφάλαιο 5

Μελέτη της Απόδοσης της εφαρμογής

Ο αρχικός κώδικας του AVS decoder, αποκωδικοποιούσε ένα video ανάλυσης 720x480 με 256 frames, σε 8 seconds. Είχε δηλαδή απόδοση 32 frames /sec. Ο κώδικάς μας μετά από τις βελτιστοποιήσεις που εφαρμόσαμε έφτασε σε minimum ταχύτητα 0.9 sec/frame. Πού οφείλεται όμως αυτή η επιβράδυνση;

Προσθέτοντας κάποιους μετρητές στον κώδικά μας παρατηρούμε ότι είχαμε 1.772.131 kernel launces. Ο συνολικός χρόνος εκτέλεσης των kernels είναι 157 seconds. Ο συνολικός χρόνος εκτέλεσης των kernel χωρίς τα memory transfers είναι 9 seconds.





Παρατηρούμε λοιπόν ότι το ποσοστό εκτέλεσης του Interpolation σε σχέση με τον υπόλοιπο κώδικα, μειώθηκε κατα 5% .

Γεννιέται λοιπόν το ερώτημα γιατί δεν έχουμε σημαντική επιτάχυνση του κώδικα παρά τις δυνατότητες που προσφέρει η GPU. Η απάντηση βρίσκεται στο ποσοστό χρησιμοποίησης της GPU από τους kernel που έχουμε δημιουργήσει.

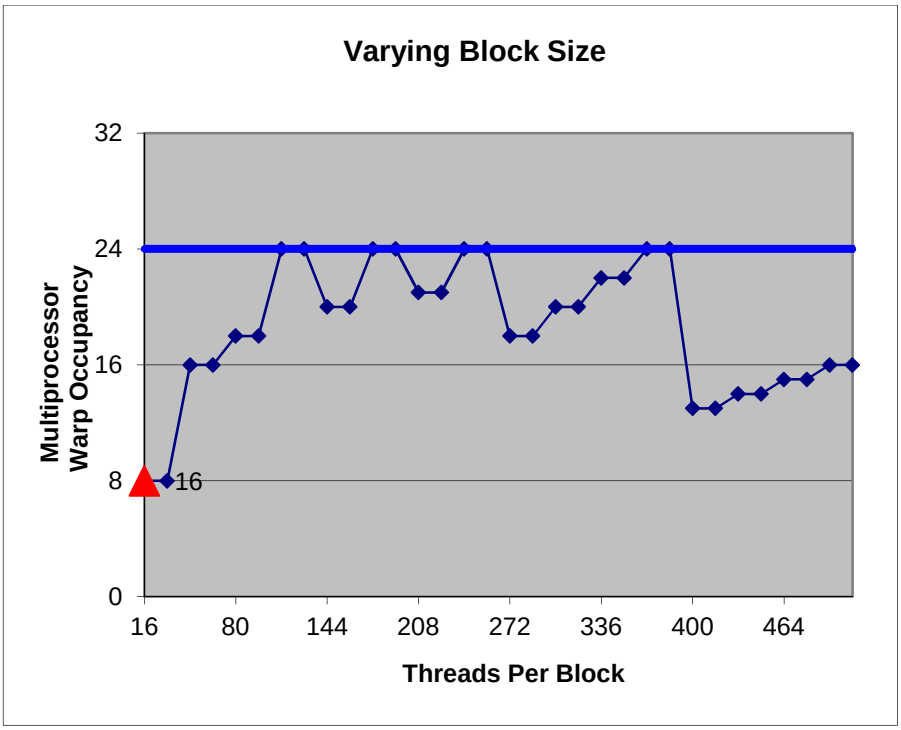
Occupancy Calculator

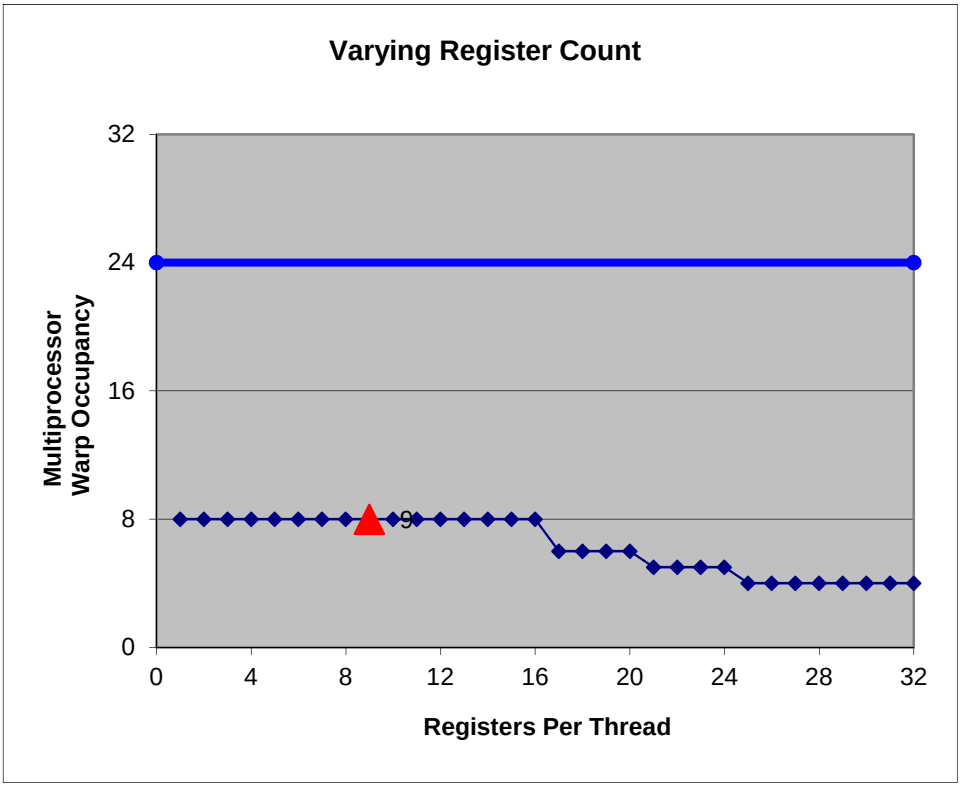
Ο occupancy calculator είναι ένα εργαλείο που προσφέρει η NVIDIA για να μπορεί ο προγραμματιστής να υπολογίσει το ποσοστό χρησιμοποίησης των πολυεπεξεργαστών της GPU (multiprocessor occupancy) από έναν CUDA kernel. Η multiprocessor occupancy προκύπτει από τον λόγο των ενεργών warps, με τον μέγιστο αριθμό warps που προσφέρονται από την συγκεκριμένη κάρτα γραφικών. Το να έχουμε μέγιστη occupancy βοηθάει στο να καλύπτονται οι καθυστερήσεις των μεταφορών από την μνήμη (memory latency), από τον υπολογισμό. Αναφέραμε και παραπάνω ότι ένα χαρακτηριστικό της GPU είναι το να μπορεί να αναδιατάσσει την σειρά εκτέλεσης των threads, έτσι ώστε όταν κάποια χρειάζονται δεδομένα από την μνήμη, να μην μένει αδρανής η GPU στον ενδιάμεσο χρόνο και να εκτελεί υπολογισμούς με τα υπόλοιπα threads που δεν χρειάζονται δεδομένα από την μνήμη.

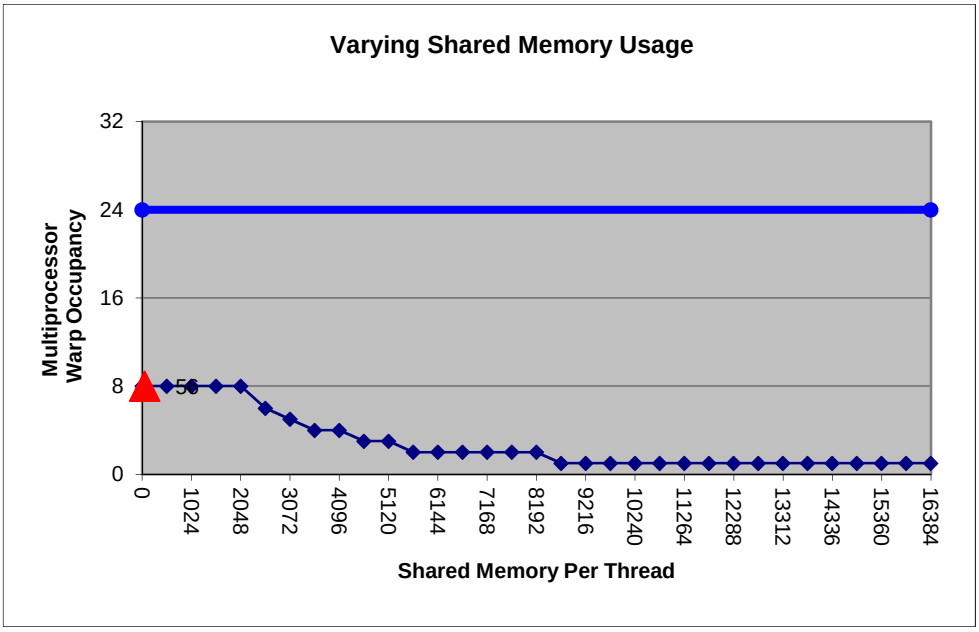
Για να χρησιμοποιήσουμε τον occupancy calculator, πρέπει να κάνουμε compile το αρχείο cu όπου έχουμε τους kernel μας, με την εντολή:

```
nvcc –ptxas-options=-v –I/home/user/NVIDIA_sdk/C/common/inc
 –L/home/user/NVIDIA_sdk/C/lib kernels.cu –lcutil
```

Στο output θα δούμε ορισμένες πληροφορίες για τους kernel μας, όπως ο αριθμός των καταχωρητών και το πόση shared memory χρησιμοποιούνε. Το output για τους δικούς μας kernel φαίνεται στο παράρτημα Δ. Έπειτα εισάγουμε τις πληροφορίες που πήραμε στο excel spreadsheet του occupancy calculator και έχουμε το εξής αποτέλεσμα.







Παρατηρούμε δηλαδή ότι δεν κάνουμε μέγιστη χρήση της υπολογιστικής ισχύος της GPU και αυτό έχει αποτελέσματα στην απόδοση.

Κεφάλαιο 6

Επίλογος

Κλείνοντας αυτήν την εργασία θα ήθελα να αναφερθώ στα συμπεράσματα που καταλήξαμε με την ενασχόλησή μας, καθώς και σε στις δυνατότητες περαιτέρω ανάπτυξης αυτού του project.

Από τις αναλύσεις απόδοσης συμπεραίνουμε ότι για να μπορέσουμε να εκμεταλευτούμε το επεξεργαστικό speed up που μπορεί να μας προσφέρει η GPU, πρέπει να μεταφέρουμε το επίπεδο παραλληλοποίησης του κώδικά μας λίγο ψηλότερα. Για παράδειγμα, πολλά υποσχόμενη είναι η παραλληλοποίηση σε επίπεδο Macroblock, όπως αναφέρθηκε και στο κεφάλαιο 3.3. Εάν προσέξουμε τις εξαρτήσεις που υπάρχουν μεταξύ των intra mackroblocks δυνητικά μπορούμε να έχουμε ένα νήμα, να εκτελεί το motion compensation για κάθε macroblock. Έτσι το μεγάλο latency που έχουμε τώρα, θα μπορέσει να επικαλυφθεί με υπολογισμό.

Στην εργασία αυτή επιχειρήσαμε την μεταφορά ενός σύνθετου προτύπου αποκωδικοποίησης βίντεο στην GPU. Αντιμετωπίσαμε διάφορα προβλήματα που αφορούσαν την δομή του κώδικα και την σύνταξή του, καθώς και προβλήματα που αφορούσαν τις ιδιαιτερότητες της GPU. Η πιο σημαντική όμως από τις δυσκολίες στις οποίες έπρεπε να ανταπεξέλθουμε ήταν η ελλιπής υποστήριξη της γλώσσας προγραμματισμού CUDA. Λόγω της νεαρής ηλικίας της η CUDA έχει πολύ μικρή κοινότητα υποστήριξης και οι προγραμματιστές που ασχολούνται με CUDA programming είναι περιορισμένοι σε αριθμό.

Ήταν μία άριστη ευκαιρία να έρθω σε επαφή με την αρχιτεκτονική της GPU η οποία αποδείχθηκε εξαιρετικά ενδιαφέρουσα και πολλά υποσχόμενη, και να μάθω ορισμένα πράγματα όσον αφορά την αποκωδικοποίηση του video. Επίσης επειδή είχα να κάνω με έναν άγνωστο κώδικα αυξημένης πολυπλοκότητας και χωρίς τεχνική υποστήριξη έμαθα να αντιμετωπίζω τέτοιας φύσεως προβλήματα τα οποία σαφώς θα εμφανιστούν και στην συνέχεια της ακαδημαϊκής μου πορείας. Τέλος ανέπτυξα συνδιαστικές ικανότητες για την περάτωση μίας εργασίας βασισμένη σε έναν άγνωστο κώδικα και σε μία άγνωστη αρχιτεκτονική.

Παράρτημα Α

Makefiles

AVS Original Makefile

```
CC=gcc
EXECUTABLE=ldecode.exe

LIBS=-lm -lpthread
FLAGS= -O3 -fno-builtin -msse3 -ffloat-store -Wall -I$(INCDIR) -
I$(ADDINCDIR)

INCDIR=./
ADDINCDIR=./

SOURCES=b_frame.c bitstream.c block.c block_const.c golomb_dec.c
header.c image.c ldecod.c loopfilter.c macroblock.c memalloc.c output.c
vlc.c frame_expansion.c
HEADERS=annexb.h avs.h b_frame.h block.h contributors.h defines.h
elements.h global.h golomb_dec.h header.h macroblock.h mbuffer.h
memalloc.h


$(EXECUTABLE): $(SOURCES) $(HEADERS)
	$(CC) $(FLAGS) -o $(EXECUTABLE) $(SOURCES) $(LIBS)

.PHONY: clean

clean:
	rm $(EXECUTABLE)
```

GPU Modified Makefile

```
LIB +=-lm -lpthread
CFLAGS += -O3 -fno-builtin -msse3 -ffloat-store -Wall -I$(INCDIR) -
I$(ADDINCDIR)

INCDIR:=./
ADDINCDIR:=./

C_DEPS :=annexb.h avs.h b_frame.h block.h contributors.h defines.h elements.h
global.h golomb_dec.h header.h macroblock.h mbuffer.h memalloc.h

ROOTDIR:=/home/nepapape/NVIDIA_GPU_Computing_SDK/C/src
BINDIR:= ./

# Add source files here
EXECUTABLE:= ldecode.exe
# Cuda source files (compiled with cudacc)
CUFILES:= get_block.cu
# C/C++ source files (compiled with gcc / c++)
CFILES          := \
        variables.c \
        b_frame.c \
        bitstream.c \
        block.c \
        block_const.c \
        golomb_dec.c \
        header.c \
        image.c \
        ldecod.c \
        loopfilter.c \
        macroblock.c \
        memalloc.c \
        output.c \
        vlc.c \
        frame_expansion.c

include $(ROOTDIR)/../common/common.mk
```

Παράρτημα Β

Ο αρχικός κώδικας της get_block.

```
/************************************************************************
* Function:Interpolation of 1/4 subpixel
* Input:
* Output:
* Return:
* Attention:
************************************************************************/

void get_block(int ref_frame,int x_pos, int y_pos, struct img_par *img, int block[8][8],
unsigned char **ref_pic)
{
  int dx, dy;
  int x, y;
  int i, j;
  int maxold_x,maxold_y;
  int ext_pos_x,ext_pos_y,ext_width,ext_height;
  int result;
  int tmp_res[26][32];
  int tmp_res_2[26][32];
  static const int COEF_HALF[4] = { -1, 5, 5, -1};
  static const int COEF_QUART[4] = { 1, 7, 7, 1 };
  short lookup_tableX[4][4]={{0,0,0,0},{2,1,2,1},{1,1,1,1},{2,1,2,1}};
  short lookup_tableY[4][4]={{0,2,1,2},{0,1,1,1},{0,2,1,2},{0,1,1,1}};
  short lookup_table_ext_width[4][4]  =
{ {8,8,8,8},  {13,11,13,11}, {11,11,13,11},  {13,11,13,11}};
  short lookup_table_ext_height[4][4] =
{ {8,13,11,13},{8,11,11,11}, {8,13,11,13},  {8,11,11,11}};
  unsigned char scratch_mem[13][16];

  dx = x_pos&3;
  dy = y_pos&3;

  x_pos = (x_pos-dx)/4;
  y_pos = (y_pos-dy)/4;
  maxold_x = img->width-1;
  maxold_y = img->height-1;

  ext_pos_x=x_pos-lookup_tableX[dx][dy];
  ext_pos_y=y_pos-lookup_tableY[dx][dy];
  ext_width=lookup_table_ext_width[dx][dy];
  ext_height=lookup_table_ext_height[dx][dy];

  if(ext_pos_x>=img->width) ext_pos_x=maxold_x;
  if(ext_pos_x<-16) ext_pos_x=-16;
  if(ext_pos_y>=img->height) ext_pos_y=maxold_y;
  if(ext_pos_y<-16) ext_pos_y=-16;
```

```
//memcpy to scratch_mem
 for(j=0;j<ext_height;j++){
   memcpy(scratch_mem[j],&ref_pic[ext_pos_y+j+20][ext_pos_x+20],sizeof(unsigned
char)*ext_width);
 }

 if (dx == 0 && dy == 0) {  //fullpel position: A
  for (j = 0; j < B8_SIZE; j++)
          for (i = 0; i < B8_SIZE; i++)
                  block[i][j] = scratch_mem[j][i];
 }
 else
 { /* other positions */
    if((dx==2) && (dy==0)){//horizonal 1/2 position: 1
      for (j = 0; j < B8_SIZE; j++){
       for (i = 0; i < B8_SIZE; i++) {
         for (result = 0, x = -1; x < 3; x++)
                  result += scratch_mem[j][i+x+1]*COEF_HALF[x+1];

        block[i][j] = max(0, min(255, (result+4)/8));
       }
      }
    }
    else if(((dx==1) || (dx==3)) && (dy==0)){   //horizonal 1/4 position: a and b
      for (j = 0; j < B8_SIZE; j++) {
        for (i = -1; i < B8_SIZE+1; i++) {
         for (result = 0, x = -1; x < 3; x++)
                  result += scratch_mem[j][i+x+2]*COEF_HALF[x+1];

        tmp_res[j][2*(i+1)] = result;
        tmp_res[j][2*(i+1)+1] = scratch_mem[j][2+i+1]*8;
       }
     }
      for (j = 0; j < B8_SIZE; j++) {
        for (i = 0; i < B8_SIZE; i++) {
         for (result = 0, x = -1; x < 3; x++)
          if(dx==1)//a
            result += tmp_res[j][2*i+x+1]*COEF_QUART[x+1];
          else//b
            result += tmp_res[j][2*i+x+2]*COEF_QUART[x+1];

        block[i][j] = max(0, min(255, (result+64)/128));
       }
      }
    }

    else if((dy==2) && (dx==0)){  //vertical 1/2 position: 2
      for (j = 0; j < B8_SIZE; j++) {
       for (i = 0; i < B8_SIZE; i++) {
         for (result = 0, y = -1; y < 3; y++)
                  result += scratch_mem[j+y+1][i]*COEF_HALF[y+1];
```

```
      block[i][j] = max(0, min(255, (result+4)/8));
    }
   }
}
else if(((dy==1) || (dy==3)) && (dx==0)){ //vertical 1/4 position: c and j
   for (j = -1; j < B8_SIZE+1; j++) {
    for (i = 0; i < B8_SIZE; i++) {
     for (result = 0, y = -1; y < 3; y++)
      result += scratch_mem[2+j+y][i]*COEF_HALF[y+1];

     tmp_res[2*(j+1)][i] = result;
     tmp_res[2*(j+1)+1][i] = scratch_mem[j+3][i]*8;
    }
   }
   for (j = 0; j < B8_SIZE; j++) {
    for (i = 0; i < B8_SIZE; i++) {
     for (result = 0, y = -1; y < 3; y++)
       if(dy==1)//c
         result += tmp_res[2*j+y+1][i]*COEF_QUART[y+1];
       else//j
         result += tmp_res[2*j+y+2][i]*COEF_QUART[y+1];

     block[i][j] = max(0, min(255, (result+64)/128));
    }
   }
}
else if((dx==2) && (dy==2)){ //horizonal and vertical 1/2 position: 3
   for (j = -1; j < B8_SIZE+2; j++) {
    for (i = 0; i < B8_SIZE; i++) {
     for (result = 0, x = -1; x < 3; x++)
      result += scratch_mem[1+j][1+i+x]*COEF_HALF[x+1];

     tmp_res[j+1][i] = result;
    }
   }
   for (j = 0; j < B8_SIZE; j++) {
    for (i = 0; i < B8_SIZE; i++) {
     for (result = 0, y = -1; y < 3; y++)
      result += tmp_res[j+y+1][i]*COEF_HALF[y+1];

     block[i][j] = max(0, min(255, (result+32)/64));
    }
   }
}
else if(((dx==1) || (dx==3)) && dy==2){ //horizonal and vertical 1/4 position: h and i

   for (j = 0; j < B8_SIZE; j++) {
    for (i = -2; i < B8_SIZE+3; i++) {
     for (result = 0, y = -1; y < 3; y++)
      result += scratch_mem[1+j+y][2+i]*COEF_HALF[y+1];

     tmp_res[j][i+2] = result;
    }
```

```
    }
    for (j = 0; j < B8_SIZE; j++) {
      for (i = 0; i < B8_SIZE+2; i++) {
        for (result = 0, x = -1; x < 3; x++)
          result += tmp_res[j][i+1+x]*COEF_HALF[x+1];

        tmp_res_2[j][2*i] = result;
        tmp_res_2[j][2*i+1] = tmp_res[j][i+2]*8;
      }
    }
    for (j = 0; j < B8_SIZE; j++) {
      for (i = 0; i < B8_SIZE; i++) {
        for (result = 0, x = -1; x < 3; x++)
          if(dx==1)//h
            result += tmp_res_2[j][2*i+x+1]*COEF_QUART[x+1];
          else
            result += tmp_res_2[j][2*i+x+2]*COEF_QUART[x+1];

        block[i][j] = max(0, min(255, (result+512)/1024));
      }
    }
  }
  else if(((dy==1) || (dy==3)) && (dx==2)){ //vertical and horizonal 1/4 position: e and l
    for (j = -2; j < B8_SIZE+3; j++) {
      for (i = 0; i < B8_SIZE; i++) {
        for (result = 0, x = -1; x < 3; x++)
          result += scratch_mem[2+j][1+i+x]*COEF_HALF[x+1];

        tmp_res[j+2][i] =result;
      }
    }
    for (j = 0; j < B8_SIZE+2; j++) {
      for (i = 0; i < B8_SIZE; i++) {
        for (result = 0, y = -1; y < 3; y++)
          result += tmp_res[j+y+1][i]*COEF_HALF[y+1];

        tmp_res_2[2*j][i] = result;
        tmp_res_2[2*j+1][i] =  tmp_res[j+2][i]*8;
      }
    }



    for (j = 0; j < B8_SIZE; j++) {
      for (i = 0; i < B8_SIZE; i++) {
        for (result = 0, y = -1; y < 3; y++)
          if(dy==1)//e
            result += tmp_res_2[2*j+y+1][i]*COEF_QUART[y+1];
          else//l
            result += tmp_res_2[2*j+y+2][i]*COEF_QUART[y+1];

        block[i][j] = max(0, min(255, (result+512)/1024));
      }
    }
```

```
    }
    else{ //Diagonal 1/4 position : d, f, k and m
      for (j = -1; j < B8_SIZE+2; j++) {
       for (i = 0; i < B8_SIZE; i++) {
         for (result = 0, x = -1; x < 3; x++)
           result += scratch_mem[1+j][1+i+x]*COEF_HALF[x+1];

         tmp_res[j+1][i] = result;
       }
      }
      for (j = 0; j < B8_SIZE; j++) {
       for (i = 0; i < B8_SIZE; i++) {
         for (result = 0, y = -1; y < 3; y++)
           result += tmp_res[j+y+1][i]*COEF_HALF[y+1];

         tmp_res_2[j][i] = result;
       }
      }
      for (j = 0; j < B8_SIZE; j++) {
       for (i = 0; i < B8_SIZE; i++) {
          if((dx==1) && (dy==1))//d
             result = tmp_res_2[j][i]+scratch_mem[1+j][i+1]*64;
          else if((dx==3) && (dy==1))//f
             result = tmp_res_2[j][i]+scratch_mem[1+j][i+2]*64;
          else if((dx==1) && (dy==3))//k
             result = tmp_res_2[j][i]+scratch_mem[j+2][i+1]*64;
          else if((dx==3) && (dy==3))//m
             result = tmp_res_2[j][i]+scratch_mem[j+2][i+2]*64;
        block[i][j] = max(0, min(255, (result+64)/128));
       }
      }
    }
  }
}
```

Παραρτημα Γ

Κώδικας “get_block.cu”

```
#include <cutil_inline.h>
#include "get_block.cuh"
#include "get_block_Kernel.cu"

#include "defines.h"
#include "contributors.h"
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <sys/timeb.h>
#include <string.h>
#include <assert.h>
#include "global.h"
#include "mbuffer.h"
#include "header.h"
#include "annexb.h"
#include "vlc.h"
#include "memalloc.h"

extern "C"
{
size_t pitch_sm,pitch_b,pitch_tr,pitch_tr2;
int size_tr,size_tr2,size_b,size_sm;
int * dev_COEF_HALF;
unsigned char * dev_scratch_mem;
int * dev_COEF_QUART;
int *dev_tmp_res;
int *dev_tmp_res_2;
int *dev_block;
int size1= 16*sizeof(unsigned char);
int size2=4*sizeof(int);
int size3=4*sizeof(int);
int size4=32*sizeof(int);
int size5=8*sizeof(int);

void get_block_c(int ref_frame,int x_pos, int y_pos, struct img_par *img, int block[8][8],
unsigned char **ref_pic){

int dx, dy;
int i,j,k,l;
int flag_ok=1;
int m_count=0;
int mistakes[8][8];
int block1[8][8];
int maxold_x,maxold_y;
int ext_pos_x,ext_pos_y,ext_width,ext_height;
short lookup_tableX[4][4]={{0,0,0,0},{2,1,2,1},{1,1,1,1},{2,1,2,1}};
short lookup_tableY[4][4]={{0,2,1,2},{0,1,1,1},{0,2,1,2},{0,1,1,1}};
unsigned char scratch_mem[13][16];
```

```
short lookup_table_ext_width[4][4]  =
{ {8,8,8,8},  {13,11,13,11}, {11,11,13,11},  {13,11,13,11}};
short lookup_table_ext_height[4][4] =
{ {8,13,11,13},{8,11,11,11}, {8,13,11,13},  {8,11,11,11}};

dx = x_pos&3;
dy = y_pos&3;

x_pos = (x_pos-dx)/4;
y_pos = (y_pos-dy)/4;
maxold_x = img->width-1;
maxold_y = img->height-1;

ext_pos_x=x_pos-lookup_tableX[dx][dy];
ext_pos_y=y_pos-lookup_tableY[dx][dy];
ext_width=lookup_table_ext_width[dx][dy];
ext_height=lookup_table_ext_height[dx][dy];

if(ext_pos_x>=img->width) ext_pos_x=maxold_x;
if(ext_pos_x<-16) ext_pos_x=-16;
if(ext_pos_y>=img->height) ext_pos_y=maxold_y;
if(ext_pos_y<-16) ext_pos_y=-16;

//memcpy to scratch_mem
for(j=0;j<ext_height;j++){
memcpy(scratch_mem[j],&ref_pic[ext_pos_y+j+20][ext_pos_x+20],sizeof(unsigned
char)*ext_width);
}
//cuda copy variables
size_b=pitch_b/sizeof(int);
size_tr=pitch_tr/sizeof(int);
size_tr2=pitch_tr2/sizeof(int);
size_sm=pitch_sm/sizeof(unsigned char);
cutilSafeCall(cudaMemcpy2D(dev_scratch_mem, pitch_sm,(unsigned char *)
&scratch_mem[0][0], size1, 16, 13, cudaMemcpyHostToDevice));


if (dx == 0 && dy == 0) {  //fullpel position: A
//kernel D1 8*8
dim3 blocksz1(8,8);
get_block_D1<<< 1 , blocksz1>>>( dev_scratch_mem, dev_block, size_b, size_sm);
cutilCheckMsg("D1 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5,(int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
}
else
{ // other positions
if((dx==2) && (dy==0)){ //horizonal 1/2 position: 1
//kernel D2 8*8
dim3 blocksz2(8,8);
get_block_D2<<< 1 , blocksz2>>>( dev_scratch_mem, dev_COEF_HALF,
dev_block,size_b,size_sm);
cutilCheckMsg("D2 Kernel  execution failed");
```

```
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
}
else if(((dx==1) || (dx==3)) && (dy==0)){ //horizonal 1/4 position: a and b
//kernel D4 8*10
dim3 blocksz4(10,8);
get_block_D4<<< 1 , blocksz4>>>( dev_scratch_mem, dev_COEF_HALF,
dev_tmp_res,size_sm,size_tr);
cutilCheckMsg("D4 Kernel  execution failed");
//export tmp_res
//kernel D5 8*8
dim3 blocksz5(8,8);
get_block_D5<<< 1 , blocksz5>>>(dx, dev_tmp_res, dev_COEF_QUART,
dev_block,size_b,size_tr);
cutilCheckMsg("D5 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
}
else if((dy==2) && (dx==0)){ //vertical 1/2 position: 2
//kernel D3 8*8
dim3 blocksz3(8,8);
get_block_D3<<< 1 , blocksz3>>>( dev_scratch_mem, dev_COEF_HALF,
dev_block,size_b,size_sm);
cutilCheckMsg("D3 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
}
else if(((dy==1) || (dy==3)) && (dx==0)){ //vertical 1/4 position: c and j
//kernel D6 10*8
dim3 blocksz6(8,10);
get_block_D6<<< 1 , blocksz6>>>( &dev_scratch_mem[0], &dev_COEF_HALF[0],
&dev_tmp_res[0],size_tr,size_sm);
cutilCheckMsg("D6 Kernel  execution failed");
//export tmp_res
//kernel D7 8*8
dim3 blocksz7(8,8);
get_block_D7<<< 1 , blocksz7>>>(dy, &dev_tmp_res[0], &dev_COEF_QUART[0],
&dev_block[0],size_b,size_tr);
cutilCheckMsg("D7 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
}
else if((dx==2) && (dy==2)){ //horizonal and vertical 1/2 position: 3
//kernel D8 11*8
dim3 blocksz8(11,8);
get_block_D8<<< 1 , blocksz8>>>( dev_scratch_mem, dev_COEF_HALF,
dev_tmp_res,size_tr,size_sm);
cutilCheckMsg("D8 Kernel  execution failed");
//kernel D9 8*8
dim3 blocksz9(8,8);
get_block_D9<<< 1 , blocksz9>>>( dev_tmp_res, dev_COEF_HALF, dev_block, size_b,size_tr);
cutilCheckMsg("D9 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
```

```
}
else if(((dx==1) || (dx==3)) && dy==2){ //horizonal and vertical 1/4 position: h and i
//kernel D10 8*13
dim3 blocksz10(13,8);
get_block_D10<<< 1 , blocksz10>>>( dev_scratch_mem, dev_COEF_HALF, dev_tmp_res,
size_tr, size_sm);
cutilCheckMsg("D10 Kernel  execution failed");
//kernel D11 8*10
dim3 blocksz11(10,8);
get_block_D11<<< 1 , blocksz11>>>( dev_tmp_res, dev_COEF_HALF,
dev_tmp_res_2,size_tr,size_tr2);
cutilCheckMsg("D11 Kernel  execution failed");
//Kernel  D12 8*8
dim3 blocksz12(8,8);
get_block_D12<<< 1 , blocksz12>>>(dx, dev_tmp_res_2, dev_COEF_QUART, dev_block,
size_b,size_tr2);
cutilCheckMsg("D12 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
}
else if(((dy==1) || (dy==3)) && (dx==2)){  //vertical and horizonal 1/4 position: e and l
//kernel D13 13*8
dim3 blocksz13(8,13);
get_block_D13<<< 1 , blocksz13>>>( dev_scratch_mem, dev_COEF_HALF,
dev_tmp_res,size_tr,size_sm);
cutilCheckMsg("D13 Kernel  execution failed");
//kernel D14 10*8
dim3 blocksz14(8,10);
get_block_D14<<< 1 , blocksz14>>>( dev_tmp_res, dev_COEF_HALF,
dev_tmp_res_2,size_tr,size_tr2);
cutilCheckMsg("D14 Kernel  execution failed");
//kernel D15 8*8
dim3 blocksz15(8,8);
get_block_D15<<< 1 , blocksz15>>>(dy, dev_tmp_res_2, dev_COEF_QUART, dev_block,
size_b,size_tr2);
cutilCheckMsg("D15 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));
}
else{ //Diagonal 1/4 position : d, f, k and m
//kernel D16 11*8
dim3 blocksz16(8,11);
get_block_D16<<< 1 , blocksz16>>>( dev_scratch_mem, dev_COEF_HALF,
dev_tmp_res,size_tr,size_sm);
cutilCheckMsg("D16 Kernel  execution failed");
//kernel D17 8*8
dim3 blocksz17(8,8);
get_block_D17<<< 1 , blocksz17>>>(dx, dy, dev_tmp_res, dev_COEF_HALF,
dev_scratch_mem, dev_tmp_res_2, dev_block, size_b, size_tr,size_tr2,size_sm);
cutilCheckMsg("D17 Kernel  execution failed");
cutilSafeCall(cudaMemcpy2D((int *)&block[0][0], size5, (int *)&dev_block[0], pitch_b,
size5,8, cudaMemcpyDeviceToHost));}
}
}
```

```
void initGPU(){

static const int COEF_HALF[4] = {-1, 5, 5, -1};
static const int COEF_QUART[4] = {1, 7, 7, 1};

//scratch_mem
cutilSafeCall(cudaMallocPitch((void **) &dev_scratch_mem, &pitch_sm, size1, 13));
if(dev_scratch_mem==NULL) {printf("\nMemAllocationError!\n");exit(0);}

//coef_half
cutilSafeCall(cudaMalloc((void **) &dev_COEF_HALF, size2));
if(dev_COEF_HALF==NULL) {printf("\nMemAllocationError!\n");exit(0);}

//coef_quart
cutilSafeCall(cudaMalloc((void **) &dev_COEF_QUART, size3));
if(dev_COEF_QUART==NULL) {printf("\nMemAllocationError!\n");exit(0);}

//tmp_res - tmp_res_2
cutilSafeCall(cudaMallocPitch((void **) &dev_tmp_res, &pitch_tr, size4,26));
if(dev_tmp_res==NULL) {printf("\nMemAllocationError!\n");exit(0);}
cutilSafeCall(cudaMallocPitch((void **) &dev_tmp_res_2, &pitch_tr2, size4,26));
if(dev_tmp_res_2==NULL) {printf("\nMemAllocationError!\n");exit(0);}

//block
cutilSafeCall(cudaMallocPitch((void **) &dev_block, &pitch_b, size5, 8));
if(dev_block==NULL) {printf("\nMemAllocationError!\n");exit(0);}

cutilSafeCall(cudaMemcpy(&dev_COEF_HALF[0],(int *)&COEF_HALF[0], size2 ,
cudaMemcpyHostToDevice));
cutilSafeCall(cudaMemcpy(&dev_COEF_QUART[0], (int *)&COEF_QUART[0], size3 ,
cudaMemcpyHostToDevice));
}


void freeGPU(){

cutilSafeCall(cudaFree(dev_scratch_mem));
cutilSafeCall(cudaFree(dev_block));
cutilSafeCall(cudaFree(dev_tmp_res_2));
cutilSafeCall(cudaFree(dev_tmp_res));
cutilSafeCall(cudaFree(dev_COEF_QUART));
cutilSafeCall(cudaFree(dev_COEF_HALF));

}

}//extern C
```

Κώδικας "get_block_Kernel.cu"

```
#ifndef GET_BLOCK_H_
#define GET_BLOCK_H_

__global__ void get_block_D1(unsigned char *scratch_mem, int *block, int pitch_b, int
pitch_sm)
{
	int j=threadIdx.x;
	int i=threadIdx.y;
	block[i*pitch_b+j] = scratch_mem[j*pitch_sm+i];
}

__global__ void get_block_D2(unsigned char *scratch_mem, int * COEF_HALF, int *block,int
pitch_b, int pitch_sm)
{
	int j=threadIdx.x;
	int i=threadIdx.y;
	int result,x;
	for (result = 0, x = -1; x < 3; x++)
		result += scratch_mem[j*pitch_sm+(i+x+1)]*COEF_HALF[x+1];

	block[i*pitch_b+j] = max(0, min(255, (result+4)/8));
}

__global__ void get_block_D3(unsigned char *scratch_mem, int *COEF_HALF, int *block,int
pitch_b,int pitch_sm)
{
	int j=threadIdx.x;
	int i=threadIdx.y;
	int result,y;
	for (result = 0, y = -1; y < 3; y++)
		result += scratch_mem[(j+y+1)*pitch_sm+i]*COEF_HALF[y+1];

	block[i*pitch_b+j] = max(0, min(255, (result+4)/8));
}

__global__ void get_block_D4(unsigned char *scratch_mem, int *COEF_HALF, int *tmp_res,
int pitch_sm,int pitch_tr)
{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,x;
	i=i-1;
	for (result = 0, x = -1; x < 3; x++)
		result += scratch_mem[j* pitch_sm+(i+x+2)]*COEF_HALF[x+1];

	tmp_res[j*pitch_tr+(2*(i+1))] = result;
	tmp_res[j*pitch_tr+(2*(i+1)+1)] = scratch_mem[j*pitch_sm+(2+i+1)]*8;
}
```

```
__global__ void get_block_D5(int dx, int *tmp_res, int *COEF_QUART, int* block,int pitch_b,
int pitch_tr)
{
        int j=threadIdx.y;
        int i=threadIdx.x;
        int result,x;
        for (result = 0, x = -1; x < 3; x++){
                if(dx==1)//a
                result += tmp_res[j*pitch_tr+(2*i+x+1)]*COEF_QUART[x+1];
                else//b
                result += tmp_res[j*pitch_tr+(2*i+x+2)]*COEF_QUART[x+1];
        }
        block[i*pitch_b+j] = max(0, min(255, (result+64)/128));
}

__global__ void get_block_D6(unsigned char *scratch_mem, int *COEF_HALF, int
*tmp_res,int pitch_tr, int pitch_sm)
{
        int j=threadIdx.y;
        int i=threadIdx.x;
        int result,y;
        j=j-1;
        for (result = 0, y = -1; y < 3; y++)
                result += scratch_mem[(2+j+y)*pitch_sm+i]*COEF_HALF[y+1];

        tmp_res[2*(j+1)*pitch_tr+i] = result;
        tmp_res[(2*(j+1)+1)*pitch_tr+i] = scratch_mem[(j+3)*pitch_sm+i]*8;
}


__global__ void get_block_D7(int dy, int *tmp_res, int *COEF_QUART, int* block,int
pitch_b,int pitch_tr)
{
        int j=threadIdx.y;
        int i=threadIdx.x;
        int result,y;
        for (result = 0, y = -1; y < 3; y++){
                if(dy==1)//c
                        result += tmp_res[(2*j+y+1)*pitch_tr+i]*COEF_QUART[y+1];
                else//j
                        result += tmp_res[(2*j+y+2)*pitch_tr+i]*COEF_QUART[y+1];
        }
        block[i*pitch_b+j] = max(0, min(255, (result+64)/128));
}
```

```
__global__ void get_block_D8(unsigned char *scratch_mem, int *COEF_HALF, int
*tmp_res,int pitch_tr,int pitch_sm)

{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,x;
	j=j-1;
	for (result = 0, x = -1; x < 3; x++)
		result += scratch_mem[(1+j)*pitch_sm+(1+i+x)]*COEF_HALF[x+1];

	tmp_res[(j+1)*pitch_tr+i] = result;
}


__global__ void get_block_D9(int *tmp_res, int *COEF_HALF, int* block,int pitch_b,int
pitch_tr)
{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,y;
	for (result = 0, y = -1; y < 3; y++)
		result += tmp_res[(j+y+1)*pitch_tr+i]*COEF_HALF[y+1];

	block[i*pitch_b+j] = max(0, min(255, (result+32)/64));
}

__global__ void get_block_D10(unsigned char *scratch_mem, int *COEF_HALF, int
*tmp_res,int pitch_tr,int pitch_sm)
{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,y,k,l;
	i=i-2;
	for (result = 0, y = -1; y < 3; y++)
		result += scratch_mem[(1+j+y)*pitch_sm+(2+i)]*COEF_HALF[y+1];

	tmp_res[j*pitch_tr+(i+2)] = result;
}

__global__ void get_block_D11(int *tmp_res, int *COEF_HALF, int *tmp_res_2, int pitch_tr,int
pitch_tr2)
{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,x;
	for (result = 0, x = -1; x < 3; x++)
		result += tmp_res[j*pitch_tr+(i+1+x)]*COEF_HALF[x+1];

	tmp_res_2[j*pitch_tr2+2*i] = result;
	tmp_res_2[j*pitch_tr2+2*i+1] = tmp_res[j*pitch_tr+(i+2)]*8;
}
```

```
__global__ void get_block_D12(int dx, int *tmp_res_2, int *COEF_QUART, int* block,int
pitch_b,int pitch_tr2)
{
        int j=threadIdx.y;
        int i=threadIdx.x;
        int result,x;

        for (result = 0, x = -1; x < 3; x++){
                if(dx==1)//h
                        result += tmp_res_2[j*pitch_tr2+(2*i+x+1)]*COEF_QUART[x+1];
                else
                        result += tmp_res_2[j*pitch_tr2+(2*i+x+2)]*COEF_QUART[x+1];
        }
        block[i*pitch_b+j] = max(0, min(255, (result+512)/1024));
}

__global__ void get_block_D13(unsigned char * scratch_mem, int *COEF_HALF, int
*tmp_res,int pitch_tr, int pitch_sm)
{
        int j=threadIdx.y;
        int i=threadIdx.x;
        int result,x;
        j=j-2;
        for (result = 0, x = -1; x < 3; x++)
                result += scratch_mem[(2+j)*pitch_sm+(1+i+x)]*COEF_HALF[x+1];

        tmp_res[(j+2)*pitch_tr+i] =result;
}

__global__ void get_block_D14(int *tmp_res, int *COEF_HALF, int *tmp_res_2, int pitch_tr,
int pitch_tr2)
{
        int j=threadIdx.y;
        int i=threadIdx.x;
        int result,y;
        for (result = 0, y = -1; y < 3; y++)
                result += tmp_res[(j+y+1)*pitch_tr+i]*COEF_HALF[y+1];

        tmp_res_2[2*j*pitch_tr2+i] = result;
        tmp_res_2[(2*j+1)*pitch_tr2+i] = tmp_res[(j+2)*pitch_tr+i]*8;
}
```

```
__global__ void get_block_D15(int dy, int *tmp_res_2, int *COEF_QUART, int* block,int
pitch_b,int pitch_tr2)
{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,y;
	for (result = 0, y = -1; y < 3; y++)
		if(dy==1)//e
		result += tmp_res_2[(2*j+y+1)*pitch_tr2+i]*COEF_QUART[y+1];
		else//l
		result += tmp_res_2[(2*j+y+2)*pitch_tr2+i]*COEF_QUART[y+1];

	block[i*pitch_b+j] = max(0, min(255, (result+512)/1024));
}

__global__ void get_block_D16(unsigned char *scratch_mem, int *COEF_HALF, int
*tmp_res,int pitch_tr,int pitch_sm)
{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,x;
	j=j-1;
	for (result = 0, x = -1; x < 3; x++)
		result += scratch_mem[(1+j)*pitch_sm+(1+i+x)]*COEF_HALF[x+1];

	tmp_res[(j+1)*pitch_tr+i] = result;
}

__global__ void get_block_D17(int dx, int dy, int *tmp_res, int *COEF_HALF, unsigned char
*scratch_mem, int *tmp_res_2, int *block,int pitch_b,int pitch_tr,int pitch_tr2,int pitch_sm)
{
	int j=threadIdx.y;
	int i=threadIdx.x;
	int result,y;
	for (result = 0, y = -1; y < 3; y++)
		result += tmp_res[(j+y+1)*pitch_tr+i]*COEF_HALF[y+1];

	tmp_res_2[j*pitch_tr2+i] = result;

	__syncthreads();

	if((dx==1) && (dy==1))//d
		result = tmp_res_2[j*pitch_tr2+i]+scratch_mem[(1+j)*pitch_sm+(i+1)]*64;
	else if((dx==3) && (dy==1))//f
		result = tmp_res_2[j*pitch_tr2+i]+scratch_mem[(1+j)*pitch_sm+(i+2)]*64;
	else if((dx==1) && (dy==3))//k
		result = tmp_res_2[j*pitch_tr2+i]+scratch_mem[(j+2)*pitch_sm+(i+1)]*64;
	else if((dx==3) && (dy==3))//m
		result = tmp_res_2[j*pitch_tr2+i]+scratch_mem[(j+2)*pitch_sm+(i+2)]*64;

	block[i*pitch_b+j] = max(0, min(255, (result+64)/128));
}

#endif
```

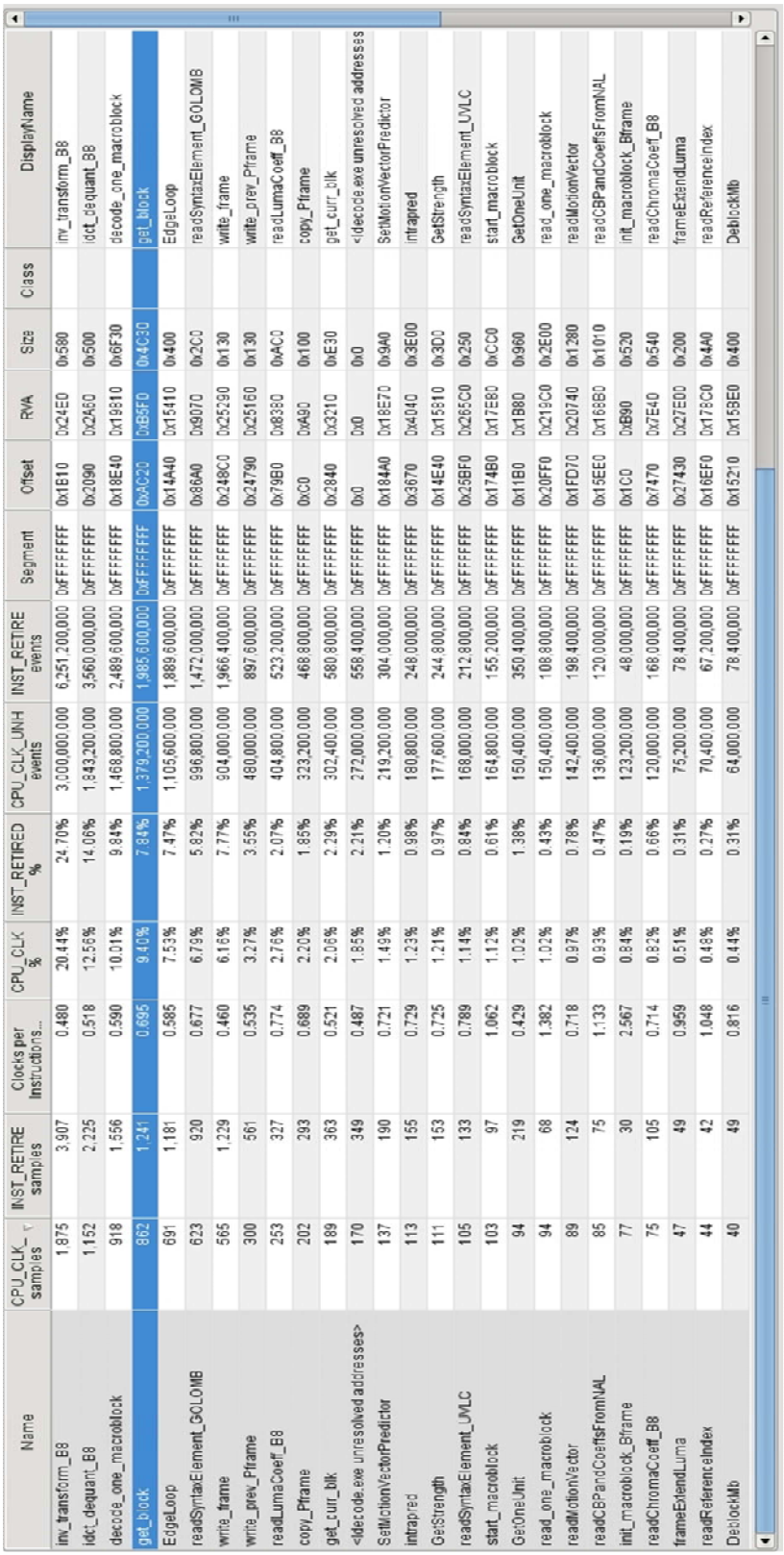
Παράρτημα Δ

Ptxas info

ptxas info : Compiling entry function '_Z13get_block_D17iiPiPhS_S_iiii'
ptxas info : Used 9 registers, 40+16 bytes smem, 24 bytes cmem[1]
ptxas info : Compiling entry function '_Z13get_block_D16PhPiii'
ptxas info : Used 8 registers, 16+16 bytes smem, 8 bytes cmem[1]
ptxas info : Compiling entry function '_Z13get_block_D15iPiS_ii'
ptxas info : Used 9 registers, 20+16 bytes smem, 16 bytes cmem[1]
ptxas info : Compiling entry function '_Z13get_block_D14PiS_ii'
ptxas info : Used 9 registers, 16+16 bytes smem, 8 bytes cmem[1]
ptxas info : Compiling entry function '_Z13get_block_D13PhPiii'
ptxas info : Used 8 registers, 16+16 bytes smem, 8 bytes cmem[1]
ptxas info : Compiling entry function '_Z13get_block_D12iPiS_ii'
ptxas info : Used 9 registers, 20+16 bytes smem, 16 bytes cmem[1]
ptxas info : Compiling entry function '_Z13get_block_D11PiS_ii'
ptxas info : Used 10 registers, 16+16 bytes smem, 4 bytes cmem[1]
ptxas info : Compiling entry function '_Z13get_block_D10PhPiii'
ptxas info : Used 8 registers, 16+16 bytes smem, 8 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D9PiS_ii'
ptxas info : Used 9 registers, 16+16 bytes smem, 16 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D8PhPiii'
ptxas info : Used 8 registers, 16+16 bytes smem, 8 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D7iPiS_ii'
ptxas info : Used 9 registers, 20+16 bytes smem, 20 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D6PhPiii'
ptxas info : Used 9 registers, 16+16 bytes smem, 8 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D5iPiS_ii'
ptxas info : Used 9 registers, 20+16 bytes smem, 20 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D4PhPiii'
ptxas info : Used 9 registers, 16+16 bytes smem, 8 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D3PhPiii'
ptxas info : Used 8 registers, 16+16 bytes smem, 16 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D2PhPiii'
ptxas info : Used 8 registers, 16+16 bytes smem, 16 bytes cmem[1]
ptxas info : Compiling entry function '_Z12get_block_D1PhPiii'
ptxas info : Used 6 registers, 16+16 bytes smem, 4 bytes cmem[1]

Παράρτημα Σχημάτων

Πίνακας 1



Name	CPU_CLK_... samples	INST_RETIRE samples	Clocks per Instructions...	CPU_CLK %	INST_RETIRED %	CPU_CLK_UNH events	INST_RETIRE events	Segment	Offset	RVA	Size	Class	DisplayName
inv_transform_B8	1,875	3,907	0.480	20.44%	24.70%	3,000,000,000	6,251,200,000	0xFFFFFFFF	0x1E10	0x24E0	0x580		inv_transform_B8
idct_dequant_B8	1,152	2,225	0.518	12.56%	14.06%	1,843,200,000	3,560,000,000	0xFFFFFFFF	0x2090	0x2A60	0x500		idct_dequant_B8
decode_one_macroblock	918	1,556	0.590	10.01%	9.84%	1,468,800,000	2,489,600,000	0xFFFFFFFF	0x18E40	0x19810	0x6F30		decode_one_macroblock
get_block	862	1,241	0.695	9.40%	7.84%	1,379,200,000	1,985,600,000	0xFFFFFFFF	0xAC20	0xB5F0	0x4C30		get_block
EdgeLoop	691	1,181	0.585	7.53%	7.47%	1,105,600,000	1,889,600,000	0xFFFFFFFF	0x14A40	0x15410	0x400		EdgeLoop
readSyntaxElement_GOLOMB	623	920	0.677	6.79%	5.82%	996,800,000	1,472,000,000	0xFFFFFFFF	0x86A0	0x9070	0x2C0		readSyntaxElement_GOLOMB
write_frame	565	1,229	0.460	6.16%	7.77%	904,000,000	1,966,400,000	0xFFFFFFFF	0x248C0	0x25290	0x130		write_frame
write_prev_Pframe	300	561	0.535	3.27%	3.55%	480,000,000	897,600,000	0xFFFFFFFF	0x24790	0x25160	0x130		write_prev_Pframe
readLumaCoeff_B8	253	327	0.774	2.76%	2.07%	404,800,000	523,200,000	0xFFFFFFFF	0x79B0	0x8380	0xACC		readLumaCoeff_B8
copy_Pframe	202	293	0.689	2.20%	1.85%	323,200,000	468,800,000	0xFFFFFFFF	0xC0	0xA90	0x100		copy_Pframe
get_curr_blk	189	363	0.521	2.06%	2.29%	302,400,000	580,800,000	0xFFFFFFFF	0x2840	0x3210	0xE30		get_curr_blk
<ldecode.exe unresolved addresses>	170	349	0.487	1.85%	2.21%	272,000,000	558,400,000	0xFFFFFFFF	0x0	0x0	0x0		<ldecode.exe unresolved addresses
SetMotionVectorPredictor	137	190	0.721	1.49%	1.20%	219,200,000	304,000,000	0xFFFFFFFF	0x18440	0x18E70	0x940		SetMotionVectorPredictor
intrapred	113	155	0.729	1.23%	0.98%	180,800,000	248,000,000	0xFFFFFFFF	0x3670	0x4040	0x3E00		intrapred
GetStrength	111	153	0.725	1.21%	0.97%	177,600,000	244,800,000	0xFFFFFFFF	0x14E40	0x15810	0x3D0		GetStrength
readSyntaxElement_UVLC	105	133	0.789	1.14%	0.84%	168,000,000	212,800,000	0xFFFFFFFF	0x26BF0	0x265C0	0x250		readSyntaxElement_UVLC
start_macroblock	103	97	1.062	1.12%	0.61%	164,800,000	155,200,000	0xFFFFFFFF	0x17480	0x17E60	0xCC0		start_macroblock
GetOneUnit	94	219	0.429	1.02%	1.38%	150,400,000	350,400,000	0xFFFFFFFF	0x11B0	0x1B80	0x960		GetOneUnit
read_one_macroblock	94	68	1.382	1.02%	0.43%	150,400,000	108,800,000	0xFFFFFFFF	0x20FF0	0x219C0	0x2E00		read_one_macroblock
readMotionVector	89	124	0.718	0.97%	0.78%	142,400,000	198,400,000	0xFFFFFFFF	0x1FD70	0x20740	0x1280		readMotionVector
readCBPandCoeffsFromNAL	85	75	1.133	0.93%	0.47%	136,000,000	120,000,000	0xFFFFFFFF	0x15EE0	0x168B0	0x1010		readCBPandCoeffsFromNAL
init_macroblock_Bframe	77	30	2.567	0.84%	0.19%	123,200,000	48,000,000	0xFFFFFFFF	0x1C0	0xB90	0x520		init_macroblock_Bframe
readChromaCoeff_B8	75	105	0.714	0.82%	0.66%	120,000,000	168,000,000	0xFFFFFFFF	0x7470	0x7E40	0x540		readChromaCoeff_B8
frameExtendLuma	47	49	0.959	0.51%	0.31%	75,200,000	78,400,000	0xFFFFFFFF	0x27430	0x27E00	0x200		frameExtendLuma
readReferenceIndex	44	42	1.048	0.48%	0.27%	70,400,000	67,200,000	0xFFFFFFFF	0x16EF0	0x178C0	0x440		readReferenceIndex
DeblockMb	40	49	0.816	0.44%	0.31%	64,000,000	78,400,000	0xFFFFFFFF	0x15210	0x15BE0	0x400		DeblockMb

Βιβλιογραφία

[1] AVS official site (http://www.avs.org.cn)

[2] Iain E.G. Richardson "H2.64 and MPEG-4 video compression"

[3]Muhsen Owaida, Christos D Antonopoulos, Nikolaos Bellas, Konstantis Daloukas, Konstantinos Krommidas, Georgios Tsoublekas " Implementation and performance comparison of the motion compensation kernel of the AVS video decoder on FPGA, CELL and multicore processors"

[4] Konstantinos Krommidas, Georgios Tsoublekas, C.D. Antonopoulos, Nikolaos Bellas, "Mapping and Optimization of the AVS Video Decoder on a High Performance Chip Multiprocessor," International Conference on Multimedia and Expo (ICME), July 19-23, 2010, Singapore.

[5] AVS Standard, Released by Bureau of Commodity Quality Inspection and Quarantine

[6] NVIDIA CUDA Programming Guide

[7] David B. Kirk, Wen-mei W. Hwu "Programming Massively Parallel Processors, A Hands-on Approach"