ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Dynamic Memory Management exploration σε συστήματα πολλών accelerators με χρήση Vivado HLS

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΓΓΕΛΙΚΗ Α. ΔΑΒΟΥΡΛΗ

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Μάρτιος 2016

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Dynamic Memory Management exploration σε συστήματα πολλών accelerators με χρήση Vivado HLS

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΓΓΕΛΙΚΗ Α. ΔΑΒΟΥΡΛΗ

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 9η Μαρτίου 2016.

....	....	....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γεώργιος Οικονομάκος
Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα, Μάρτιος 2016

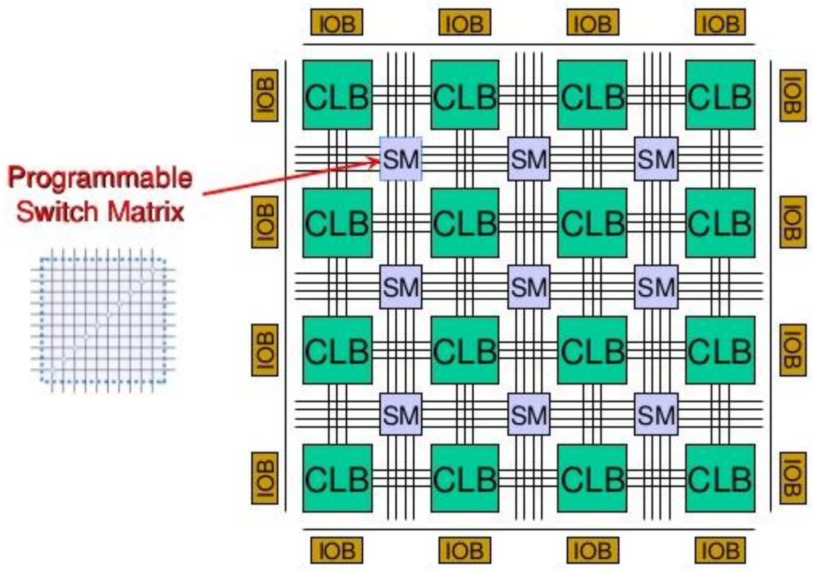
....
Αγγελική Α. Δαβουρλή
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η συστοιχία επιτόπια προγραμματιζόμενων πυλών, από εδώ και στο εξής FPGA (field-programmable gate array), ονομάζεται ένας τύπος ολοκληρωμένου κυκλώματος γενικής χρήσης, το οποίο μπορεί να προγραμματιστεί πολλές φορές μετά την κατασκευή του. Αυτό το βασικό χαρακτηριστικό του αποτελεί τον κυριότερο λόγο για τον οποίο τα FPGAs χρησιμοποιούνται σε πολλούς τομείς της βιομηχανίας όπως την αεροδιαστημική και αμυντική βιομηχανία, την αυτοκινητοβιομηχανία, τις ιατρικές εφαρμογές, την επεξεργασία εικόνων και βίντεο, τις κινητές και σταθερές επικοινωνίες, την πληροφορική υψηλής απόδοσης (HPC) κ.ά. Δεδομένου όμως ότι ο προγραμματισμός του γίνεται με χρήση γλωσσών περιγραφής υλικού (HDL), το FPGA καθίσταται αρκετά δύσκολο για χρήση από τον μέσο προγραμματιστή, που ασχολείται με γλώσσες υψηλού επιπέδου. Για το λόγο αυτό, ταυτόχρονα με την ανάπτυξη των σύγχρονων συσκευών FPGA, αναπτύσονται και εργαλεία που εφαρμόζουν την ιδέα της Σύνθεσης Υψηλού Επιπέδου (High Level Synthesis, HLS). Η ιδέα του HLS αφορά την μετατροπή κώδικα γραμμένου σε κάποια γλώσσα υψηλού επιπέδου (π.χ. C/C++) σε γλώσσα περιγραφής υλικού [1]. Πέρα από την ιδέα του HLS, η ευρεία χρήση των FPGAs σε τεχνολογίες αιχμής έχει οδηγήσει στην ανάπτυξη ενός ακόμη σταδίου κατά τη διάρκεια της διαδικασίας σχεδιασμού λογισμικού, την Εξερεύνηση του Χώρου Λύσεων (Design Space Exploration, DSE). Η ιδέα του DSE αφορά την εξερεύνηση και την εξέταση όλων των δυνατών εναλλακτικών πριν την υλοποίηση ενός αλγορίθμου σε κώδικα [2]. Το DSE είναι ιδιαίτερα χρήσιμο στην περίπτωση των FPGAs, καθώς αυτά περιλαμβάνουν περιορισμένους πόρους σε υλικό (hardware resources).

Στην παρούσα διπλωματική εξετάζουμε με τη μέθοδο του DSE, την οποία αρχικά εφαρμόζουμε στο επίπεδο της διαχείρισης μνήμης, πέντε αλγορίθμους από τη σουίτα Phoenix MapReduce, τους Histogram, MMUL, PCA, Kmeans, String match. Προκειμένου να εφαρμόσουμε την ιδέα του DSE στο επίπεδο της διαχείρησης μνήμης προτείνουμε τη χρήση δυναμικής δέσμευσης μνήμης στο πλαίσιο της μνήμης που διαθέτει ένα FPGA. Στη συνέχεια εφαρμόζουμε τη μέθοδο του DSE στο επίπεδο του κώδικα των εφαρμογών Histogram και PCA, χρησιμοποιώντας ορισμένα HLS directives (loop unroll, loop pipeline, loop merge, loop flatten).

Προκειμένου να αξιολογήσουμε τις λύσεις που μας προσφέρει το DSE αναφορικά είτε με το επίπεδο διαχείρησης μνήμης είτε με το επίπεδο του κώδικα των εφαρμογών, θα πρέπει να εξετάσουμε ορισμένα στοιχεία μιας πλακέτας FPGA.



Σχήμα 1: Δομή ενός FPGA [3]

Τα δομικά στοιχεία ενός FPGA είναι

- Το προγραμματιζόμενο λογικό μπλοκ, στο εξής CLB, το οποίο αποτελείται από πίνακες αναφοράς (Look-up tables, LUTs), φλιπ-φλοπ (Flip flops, FF), καλώδια και υποδοχείς εισόδου-εξόδου.

- Το μπλοκ εισόδου-εξόδου, στο εξής IOB, το οποίο αποτελείται από αντιστάσεις, buffers και FFs.

- Ο προγραμματιζόμενος πίνακας διακοπτών, στο εξής SM, ο οποίος αποτελείται από καλώδια και buffers.

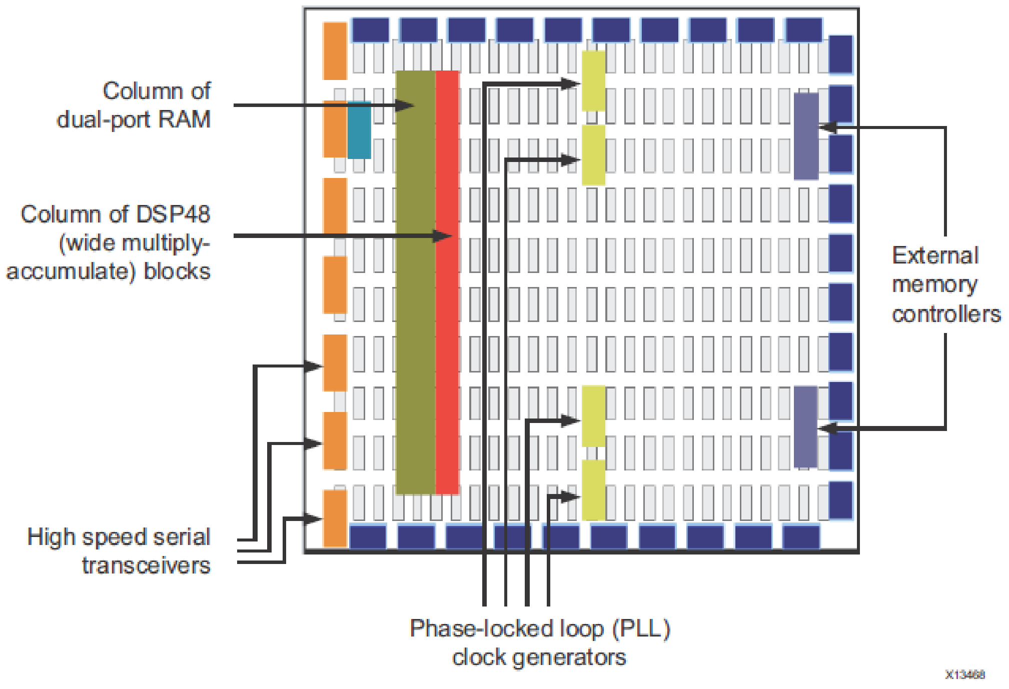
Τα στοιχεία με τα οποία ασχολείται η τύπου DSE ανάλυσή μας και τα οποία αποτελούν τις θεμελιώδεις μονάδες του FPGA είναι

- Οι πίνακες αναφοράς (Look-up tables, LUTs), οι οποίοι χρησιμοποιούνται τόσο για την υλοποίηση λογικών συναρτήσεων όσο και για την αποθήκευση δεδομένων.

- Τα φλιπ-φλοπ (FFs), τα οποία αποτελούν τη βασική μονάδα αποθήκευσης δεδομένων πάνω σε μια πλακέτα FPGA,και συνήθως συνδυάζονται με ένα LUT για το pipelining λογικών διεργασιών και την αποθήκευση δεδομένων.

- Οι επεξεργαστές ψηφιακού σήματος, στο εξής DSP (Digital Signal Processor), οι οποίοι είναι ειδικού τύπου επεξεργαστές και χρησιμεύουν ως αριθμητικές-λογικές μονάδες στις FPGA συσκευές.

- Τα μπλοκ μνήμης RAM, στο εξής BRAM(Block RAM), τα οποία αποτελούν τη μονάδα αποθήκευσης μεγάλου όγκου δεδομένων πάνω στο FPGA.


Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
External memory controllers
Phase-locked loop (PLL) clock generators
X13468

Σχήμα 2: Αρχιτεκτονική μιας πλακέτας FPGA [4]

Προηγουμένως αναφέραμε ότι θα εξετάσουμε με τη μέθοδο του DSE ορισμένες βελτιστοποιήσεις με βάση των κώδικα και με βάση τη δομή της μνήμης.

Σε αυτή τη διπλωματική εξετάζουμε τις ακόλουθες βελτιστοποιήσεις που αφορούν τον κώδικα και πιο συγκεκριμένα τις δομές επανάληψης που περιλαμβάνει

- Το loop unroll, το οποίο αποτελεί μια βελτιστοποίηση σε επίπεδο μεταγλωττιστή και μειώνει το χρόνο εκτέλεσης του προγράμματος δημιουργώντας αντίγραφα της λειτουργίας του βρόγχου επανάληψης, δεδομένου ότι ο κώδικας δεν περιλαμβάνει εξαρτήσεις δεδομένων.

- Το loop pipeline, το οποίο επίσης αποτελεί μια βελτιστοποίηση σε επίπεδο μεταγλωττιστή και μειώνει το χρόνο εκτέλεσης του προγράμματος με την χρονοδρομολόγηση των διάφορων εργασιών κατά τέτοιο τρόπο ώστε οι υπολογιστικοί πόροι της συσκευής να μένουν συνεχώς απασχολημένοι. Είναι προφανές ότι με αυτή τη βελτιστοποίηση υπάρχει μεγάλη επικάλυψη μεταξύ των διάφορων εντολών.

- Το loop merge, το οποίο αποτελεί μια βελτιστοποίηση σε επίπεδο κώδικα και μειώνει το χρόνο εκτέλεσης του προγράμματος με τη συγχώνευση σειριακών βρόγχων επανάληψης σε έναν, υπό συγκεκριμένες προϋποθέσεις. Η μείωση του χρόνου εκτέλεσης προκύπτει από τον περιορισμό των κύκλων μηχανής που απαιτούνται για την είσοδο ή την έξοδο από το βρόγχο επανάληψης.

- Το loop flatten, το οποίο αποτελεί επίσης μια βελτιστοποίση σε επίπεδο κώδικα και μειώνει το χρόνο εκτέλεσης του προγράμματος με τη συγχώνευση «φωλιασμένων» (nested) βρόγχων σε έναν, υπό συγκεκριμένες προϋποθέσεις. Η μείωση του χρόνου εκτέλεσης προκύπτει, όπως και για το loop merge, από τον περιορισμό των κύκλων μηχανής που απαιτούνται για την είσοδο ή την έξοδο από το βρόγχο επανάληψης.

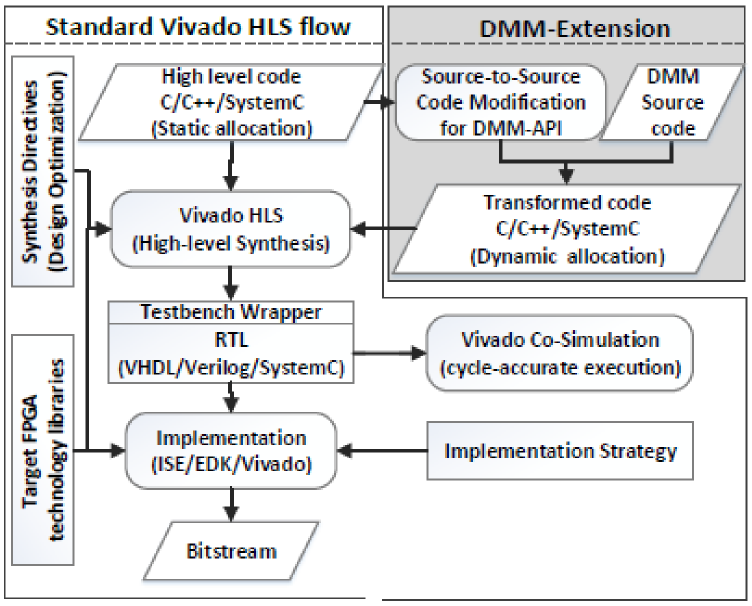
Η ιδέα του DSE δεν μπορεί να εφαρμοστεί σε στατικές δομές μνήμης (καθώς αυτές είναι αμετάβλητες), γι' αυτό και προϋπόθεση για την εξερεύνηση του χώρου λύσεων είναι η ύπαρξη δομής δυναμικής μνήμης και η ρύθμισή της κατά τέτοιον τρόπο ώστε να διευκολύνει την εκτέλεση της συγεκριμένης εφαρμογής που εξετάζουμε κάθε φορά. Συνεπώς, προτού εξετάσουμε με τη μέθοδο του DSE τη βέλτιστη δομή της μνήμης, θα πρέπει να εξηγήσουμε τους λόγους που μας ωθούν στην αναζήτηση δομών δυναμικής μνήμης σε μια πλακέτα FPGA.

Στον τομέα των υπολογιστικών συστημάτων, η τάση που κυριαρχεί είναι η τοποθέτηση ετερογενών αλγοριθμικών μπλοκ (με προσαρμοσμένη αρχιτεκτονική) στο ίδιο τσιπ (heterogeneous SoC). Αυτό γίνεται προκειμένου να επιτευχθεί μείωση της κατανάλωσης ενέργειας (σε σχέση με τα συστήματα πολλών επεξεργαστών), ενώ η προσαρμογή της αρχιτεκτονικής της εκάστοτε πλατφόρμας στις ανάγκες των αλγοριθμικών μπλοκ προσφέρει υψηλή απόδοση με ταυτόχρονη κατανάλωση περιορισμένων υπολογιστικών πόρων. Άρα τα ετερογενή FPGA αποτελούν μια κατάλληλη πλατφόρμα για την υλοποίση συστημάτων συνύπαρξης πολλών αλγορίθμων (many accelerator systems, MA systems).

Η δομή της μνήμης αποτελεί ένα σημαντικό παράγοντα περιορισμού της επίδοσης εξαιτίας του πλήθους και της ποικιλίας των αλγορίθμων που αποτελούν τα MA systems που αναφέραμε. Έχει παρατηρηθεί ότι στα σύγχρονα FPGA συστήματα, η έλλειψη διαθέσιμης μνήμης μπορεί να οδηγήσει σε δέσμευση και εκμετάλλευση αρκετά λιγότερων υπολογιστικών πόρων σε σχέση με τους διαθέσιμους. Επιπλέον η έλλειψη διαθέσιμης μνήμης πάνω στο τσιπ περιορίζει την κλιμάκωση του αριθμού των αλγορίθμων. Τα διαθέσιμα εργαλεία σχεδιασμού και προγραμματισμού των FPGA υποστηρίζουν τη χρήση στατικής μνήμης, η οποία μπορεί να υποστηρίξει μέχρι κάποιο αριθμό αλγορίθμων αλλά δεν επιτρέπει την κλιμάκωση του αριθμού αυτών.

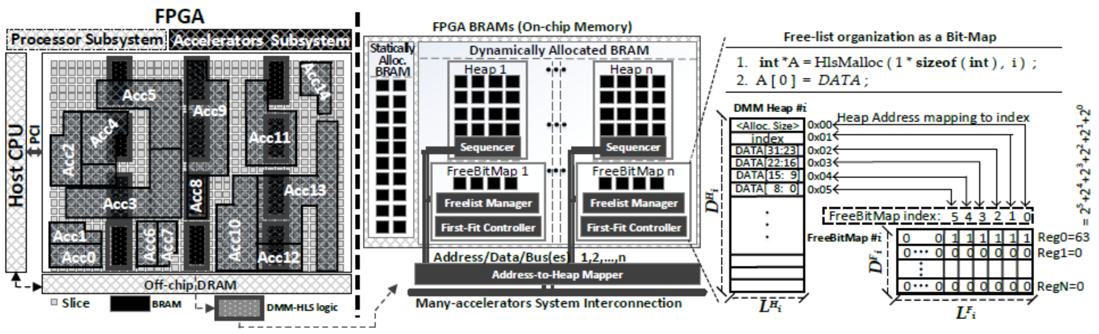
Μετά από ανάλυση πειραματικών δεδομένων παρατηρούμε ότι οι μονάδες της BRAM είναι αυτές που εξαντλούνται γρηγορότερα, συγκριτικά με τα DSPs, LUTs και FFs και οδηγεί στους περιορισμούς που αναφέραμε παραπάνω.

Στην παρούσα διπλωματική προτείνουμε τη χρήση μιας βιβλιοθήκης για δυναμική δέσμευση μνήμης (DMM-HLS library), κατάλληλη για MA systems που υλοποιούνται σε πλατφόρμες FPGA. Η βιβλιοθήκη αυτή βασίζεται στο εργαλείο Vivado HLS, το οποίο χρησιμοποιείται για τη σύνθεση του κώδικα της βιβλιοθήκης σε κώδικα RTL. Πιο συγκεκριμένα, η βιβλιοθήκη αυτή i) επεκτείνει τη διαδικασία του HLS για το σχεδιασμό κάποιου FPGA, ii) παρέχει μια διεπαφή μέσω της οποίας μετατρέπεται η στατική δέσμευση μνήμης (μέσα στον κώδικα του κάθε αλγορίθμου) σε δυναμική, με κλήση συναρτήσεων παρόμοιων με αυτές της βιβλιοθήκης glibc (malloc/free). Η DMM-HLS βιβλιοθήκη υποστηρίζει τη συνύπαρξη στατικής και δυναμικής δέσμευσης μνήμης μέσα στον κώδικα του ίδιου αλγορίθμου. Πριν χρησιμοποιήσουμε την DMM-HLS βιβλιοθήκη, θα πρέπει να κάνουμε τις κατάλληλες τροποποιήσεις στον αρχικό κώδικα ώστε να χρησιμοποιεί δυναμική δέσμευση μνήμης. Στη συνέχεια θα πρέπει να προσθέσουμε στον κώδικα τις κλήσεις των συναρτήσεων για τη δυναμική δέσμευση/αποδέσμευση μνήμης και τελικά να συνθέσουμε τον αλγόριθμο σε κώδικα RTL χρησιμοποιώντας το εργαλείο Vivado HLS.


Standard Vivado HLS flow
DMM-Extension
Synthesis Directives (Design Optimization)
High level code C/C++/SystemC (Static allocation)
Source-to-Source Code Modification for DMM-API
DMM Source code
Transformed code C/C++/SystemC (Dynamic allocation)
Vivado HLS (High-level Synthesis)
Testbench Wrapper
RTL (VHDL/Verilog/SystemC)
Vivado Co-Simulation (cycle-accurate execution)
Target FPGA technology libraries
Implementation (ISE/EDK/Vivado)
Implementation Strategy
Bitstream

Σχήμα 3: Η DMM-HLS βιβλιοθήκη ως επέκταση της λειτουργίας του εργαλείου Vivado HLS [5]

Η βιβλιοθήκη δυναμικής διαχείρισης μνήμης προσφέρει τη δυνατότητα παραλληλοποίησης των δεσμεύσεων/αποδεσμεύσεων μνήμης, χωρίζοντας τις μονάδες BRAM (που περιλαμβάνει το FPGA) σε στοίβες (heaps). Κάθε heap περιλαμμβάνει έναν μηχανισμό κατανομής μνήμης (DM allocator), ο οποίος είναι ικανός να δεσμεύει στον heap διαφορετικού τύπου δεδομένα (int, float, double). Ο DM allocator περιλαμβάνει δύο μέρη: i) τη δομή του freelist και ii) τον αλγόριθμο που ρυθμίζει τον τρόπο εύρεσης του διαθέσιμου τμήματος μνήμης, στο εξής fit allocation algorithm. Ο πίνακας freelist παρακολουθεί τα δεσμευμένα και τα ελεύθερα μπλοκ της μνήμης, ενώ ο fit allocation algorithm ψάχνει πάνω στη δομή του freelist και εντοπίζει το πρώτο μπλοκ μνήμης που ταιριάζει σε μέγεθος με το αίτημα για δέσμευση μνήμης (καθώς αυτός ο αλγόριθμος είναι τύπου first fit).



Σχήμα 4: Αρχιτεκτονικός σχεδιασμός της βιβλιοθήκης δυναμικής δέσμευσης μνήμης και απεικόνιση της δομής του freelist [6]

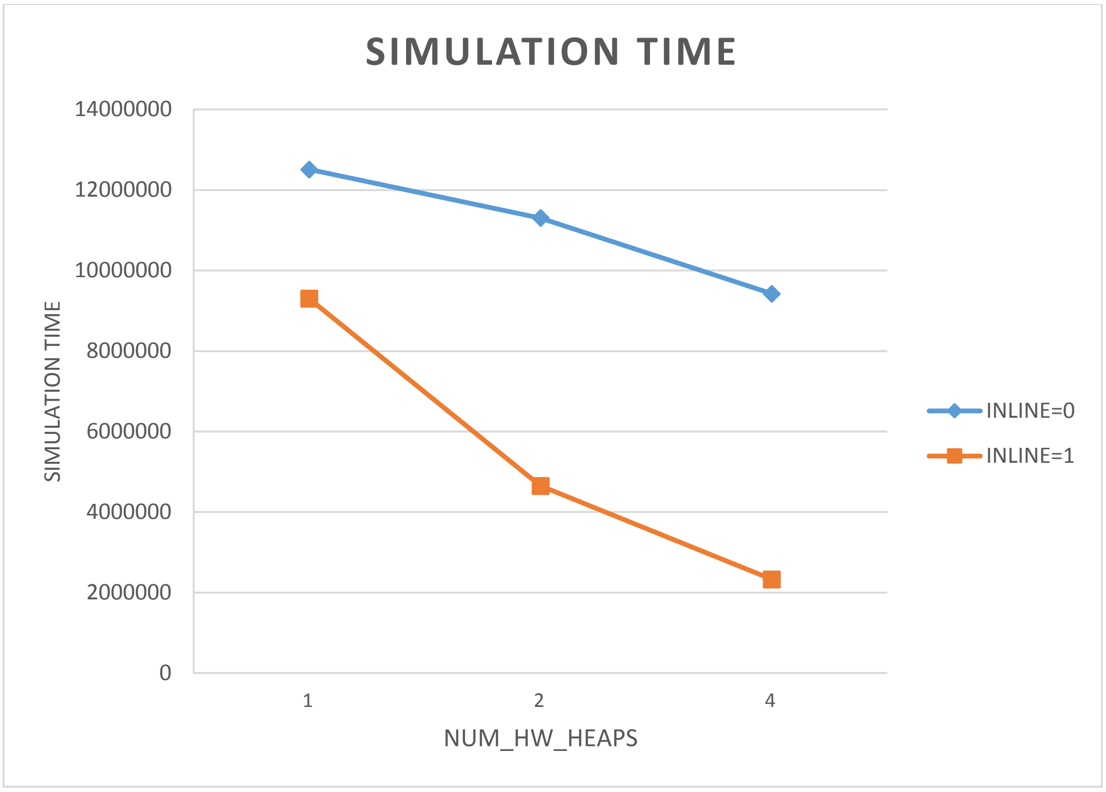
Κάθε ένας από τους heaps του συστήματος μπορεί να εξυπηρετεί τα αιτήματα δέσμευσης/αποδέσμευσης μνήμης για παραπάνω από έναν αλγορίθμους. Άρα αυξάνοντας τον αριθμό των heaps που υλοποιούνται πάνω σε ένα FPGA, αυξάνουμε την παραλληλία στο πλαίσιο της αλληλεπίδρασης μεταξύ αλγορίθμου και μνήμης και αυτό οδηγεί στη μείωση του χρόνου εκτέλεσης του εκάστοτε αλγορίθμου. Το κόστος της υλοποίησης πολλαπλών heaps αφορά τη δέσμευση υπολογιστικών πόρων πάνω στο board, οι οποίοι σε άλλη περίπτωση θα μπρούσαν να χρησιμοποιηθούν για την υλοποίηση περισσότερων αλγορίθμων.

Τα κομμάτια της DMM-HLS στα οποία εν τέλει εφαρμόζουμε DSE είναι

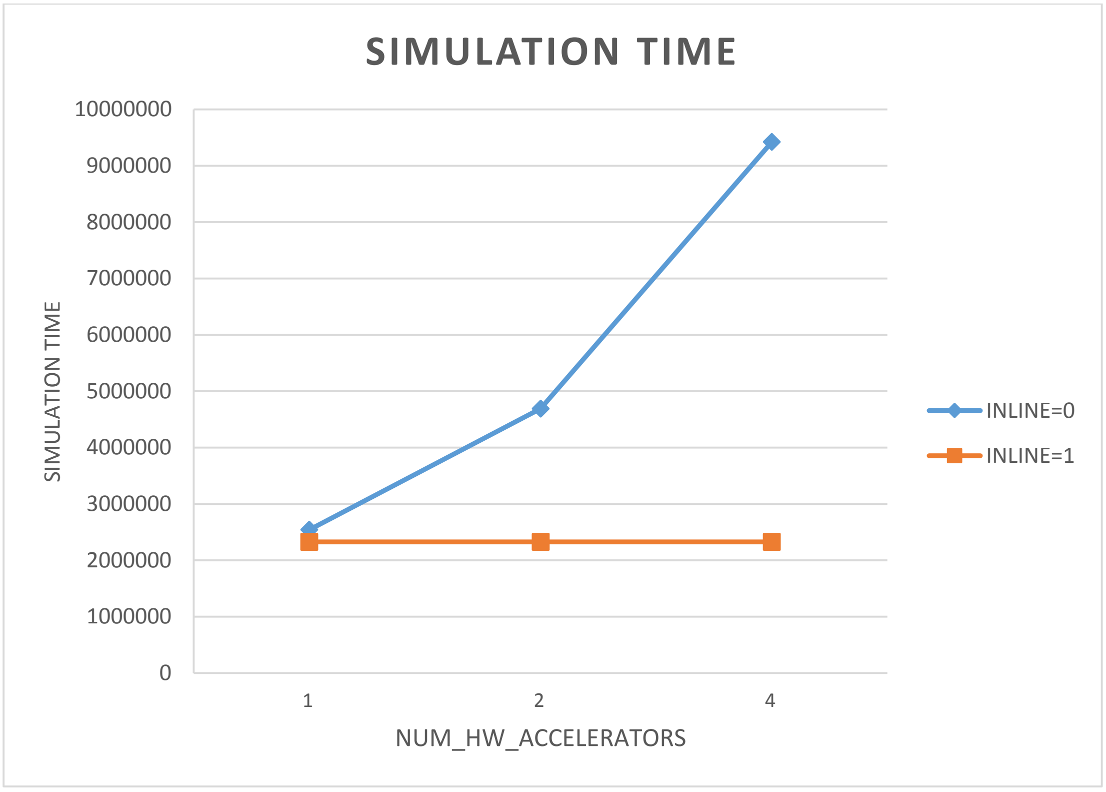
- Το function inlining, το οποίο αφορά στην περαιτέρω παραλληλοποίηση των κλήσεων των συναρτήσεων της βιβλιοθήκης για δέσμευση/αποδέσμευση μνήμης και επιτρέπει την ταυτόχρονη πρόσβαση πολλών αλγορίθμων σε πολλούς heaps.

- Το πλάτος του πίνακα freelist, το οποίο μας επιτρέπει να ελέγχουμε σε κάθε επανάληψη ένα τμήμα του heap. Όσο αυξάνουμε το πλάτος του πίνακα τόσο περισσότερα στοιχεία του heap μπορούμε να ελέγξουμε σε μια επανάληψη. Συνεπώς αναμένουμε ότι σε γενικές γραμμές όσο αυξάνουμε το μέγεθος του πλάτους του πίνακα, τόσο θα μειώνεται ο χρόνος εκτέλεσης του προγράμματος.

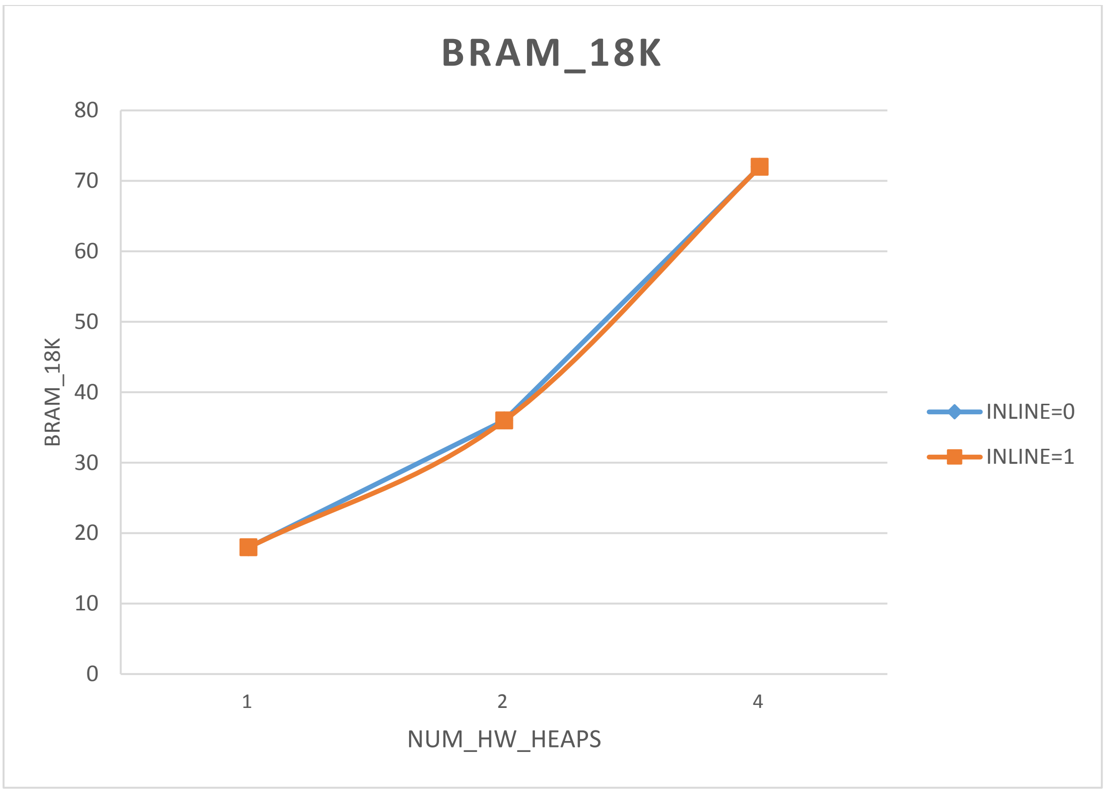
Στη συνέχεια παραθέτουμε τα διαγράμματα της εξερεύνησης του χώρου λύσεων στο επίπεδο της διαχείρισης μνήμης για τον αλγόριθμο Histogram, ο οποίος είναι ενδεικτικός για το σύνολο των αλγορίθμων που μελετήθηκαν. Αξίζει να σημειωθεί ότι οι χρόνοι εκτέλεσης είναι σε ns και οι μονάδες υπολογιστικών πόρων σε απόλυτους αριθμούς. Επιπλέον οι καμπύλες που αφορούν το inline σχεδιάστηκαν με freelist width ίσο με 8bits και οι καμπύλες που αφορούν το freelist width σχεδιάστηκαν με inline ίσο με 0.



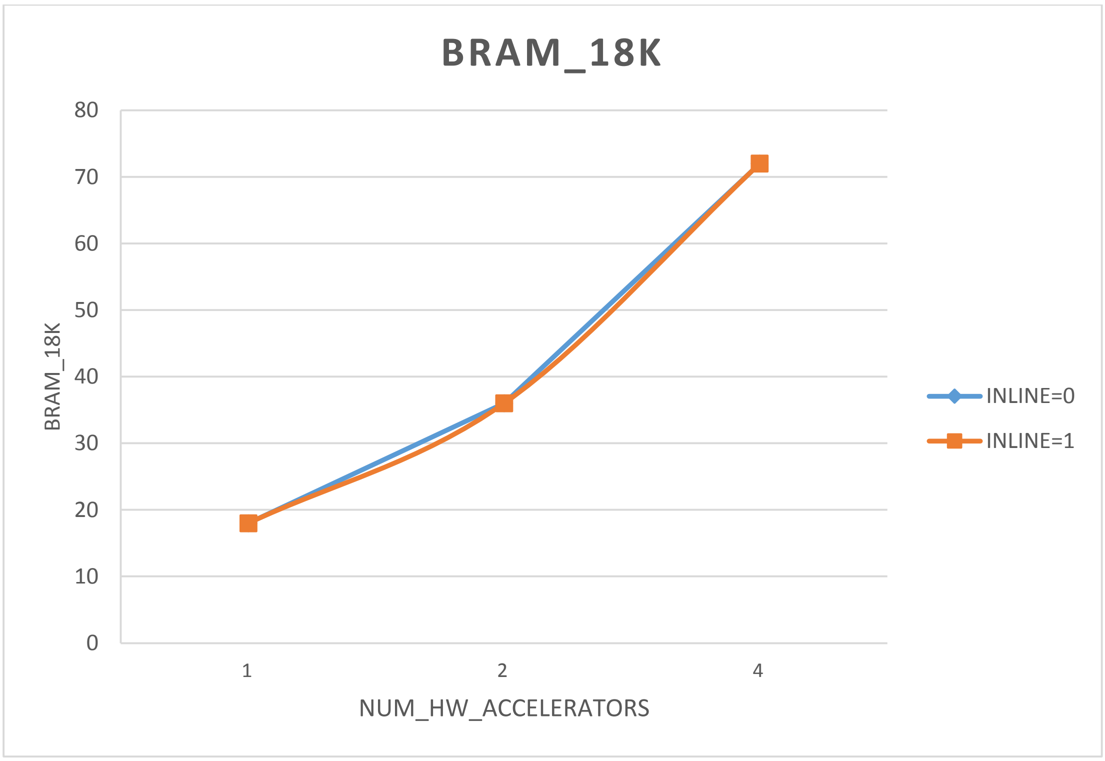
Σχήμα 5: Χρόνος εκτέλεσης συναρτήσει των heaps (INLINE)



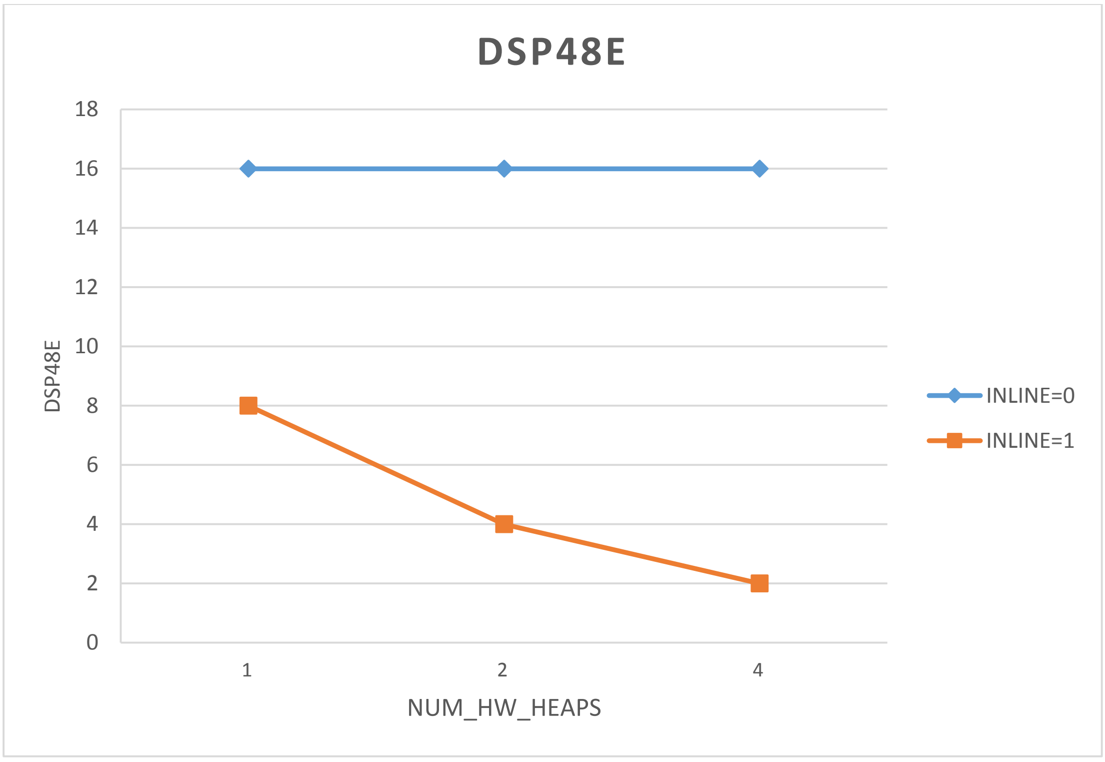
Σχήμα 6: Χρόνος εκτέλεσης συναρτήσει των πολλαπλών αλγορίθμων (INLINE)



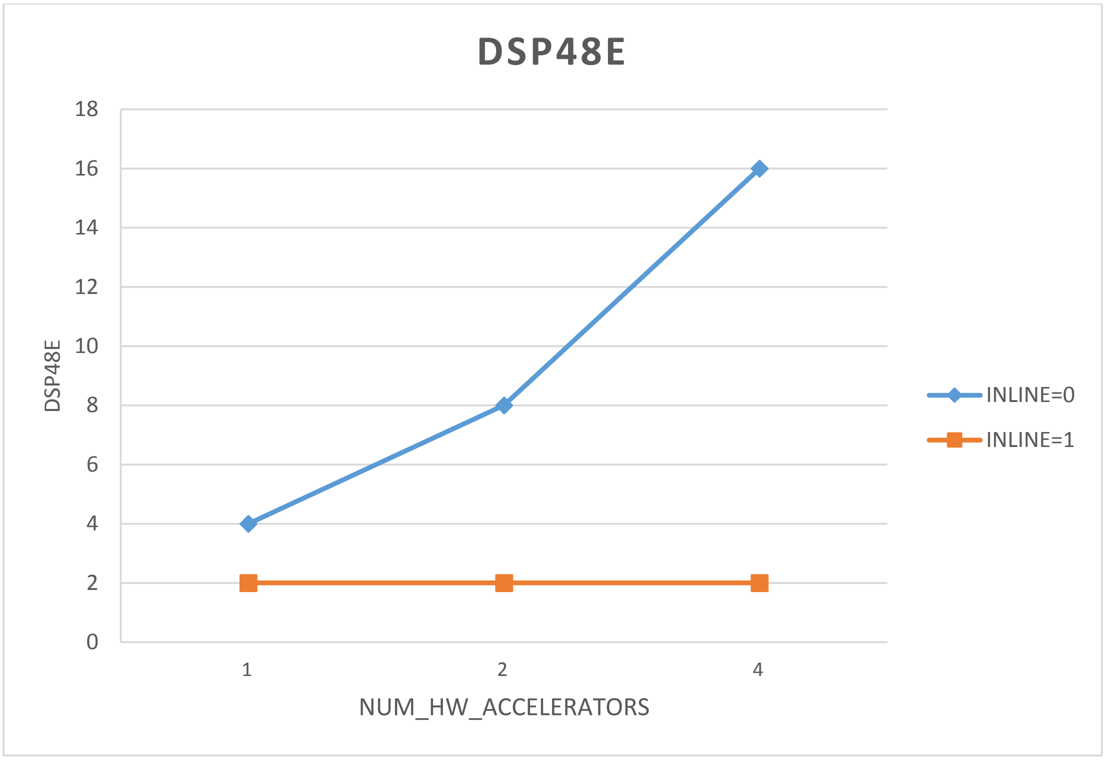
Σχήμα 7: Χρήση BRAM συναρτήσει των heaps (INLINE)



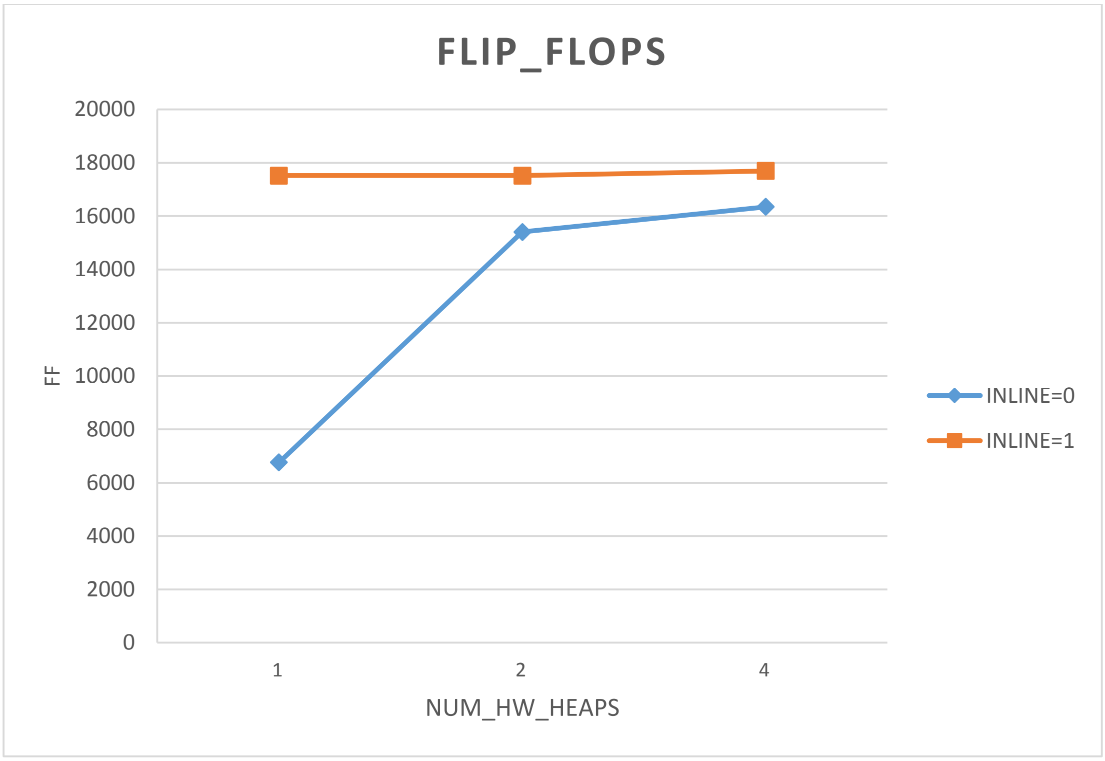
Σχήμα 8: Χρήση BRAM συναρτήσει των πολλαπλών αλγορίθμων (INLINE)



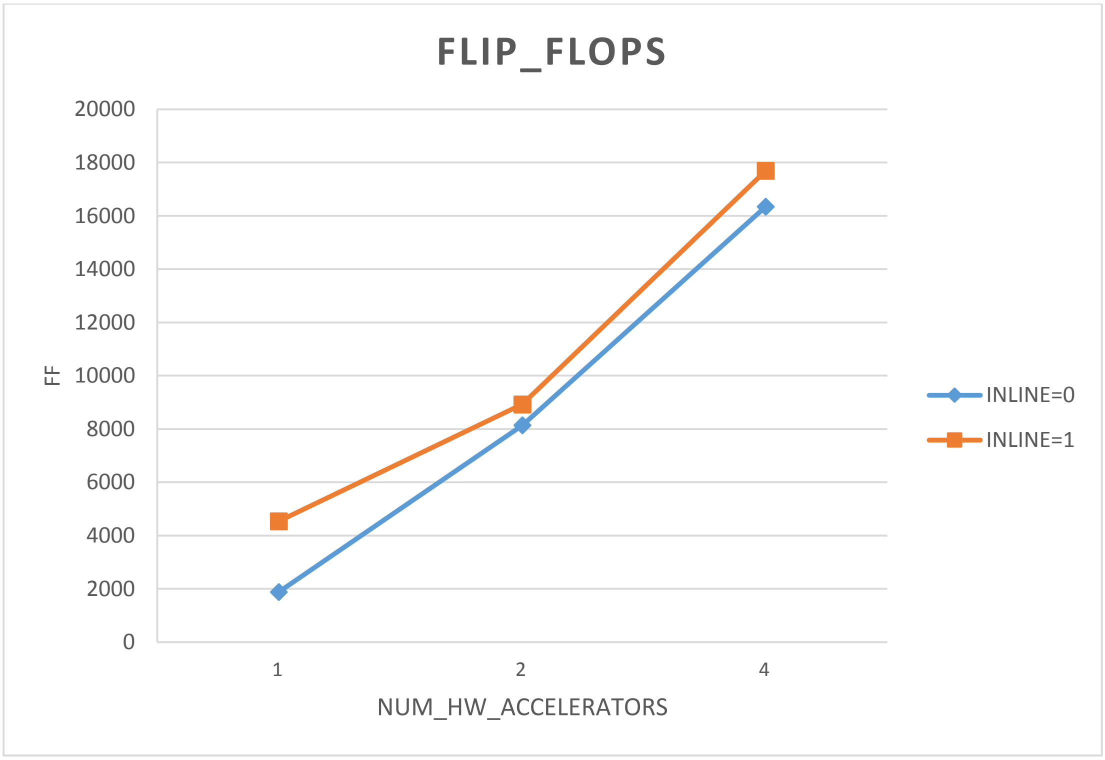
Σχήμα 9: Χρήση DSP συναρτήσει των heaps (INLINE)



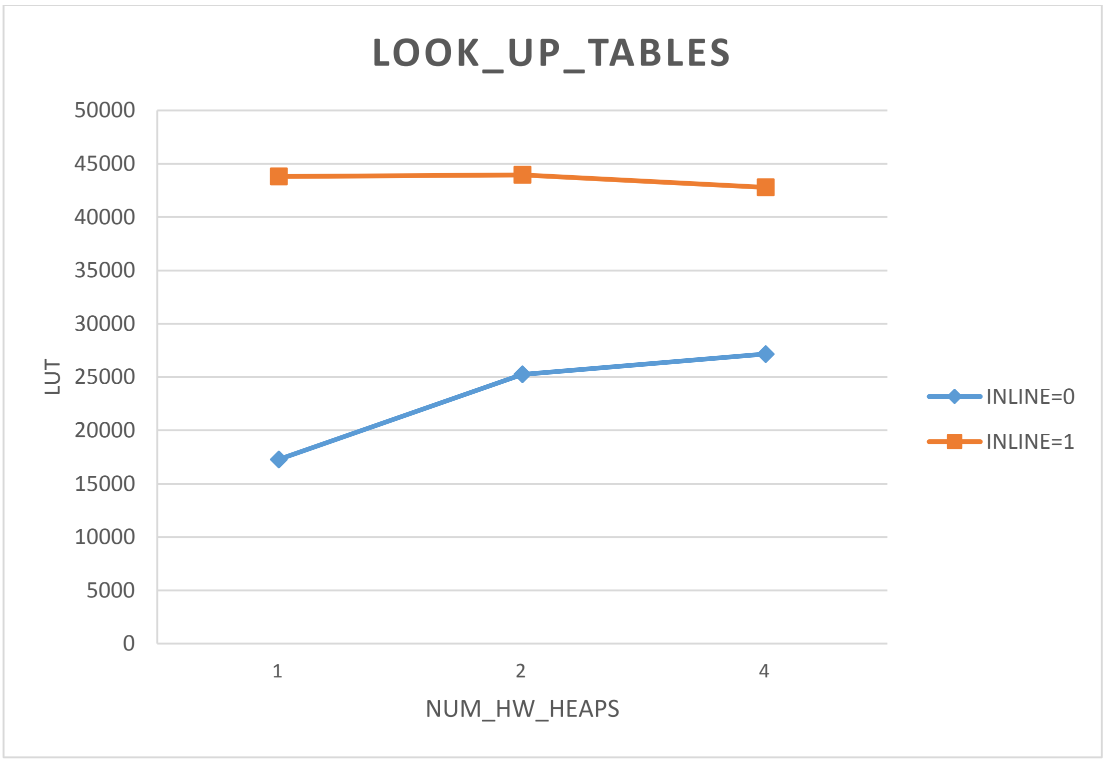
Σχήμα 10: Χρήση DSP συναρτήσει των πολλαπλών αλγορίθμων (INLINE)



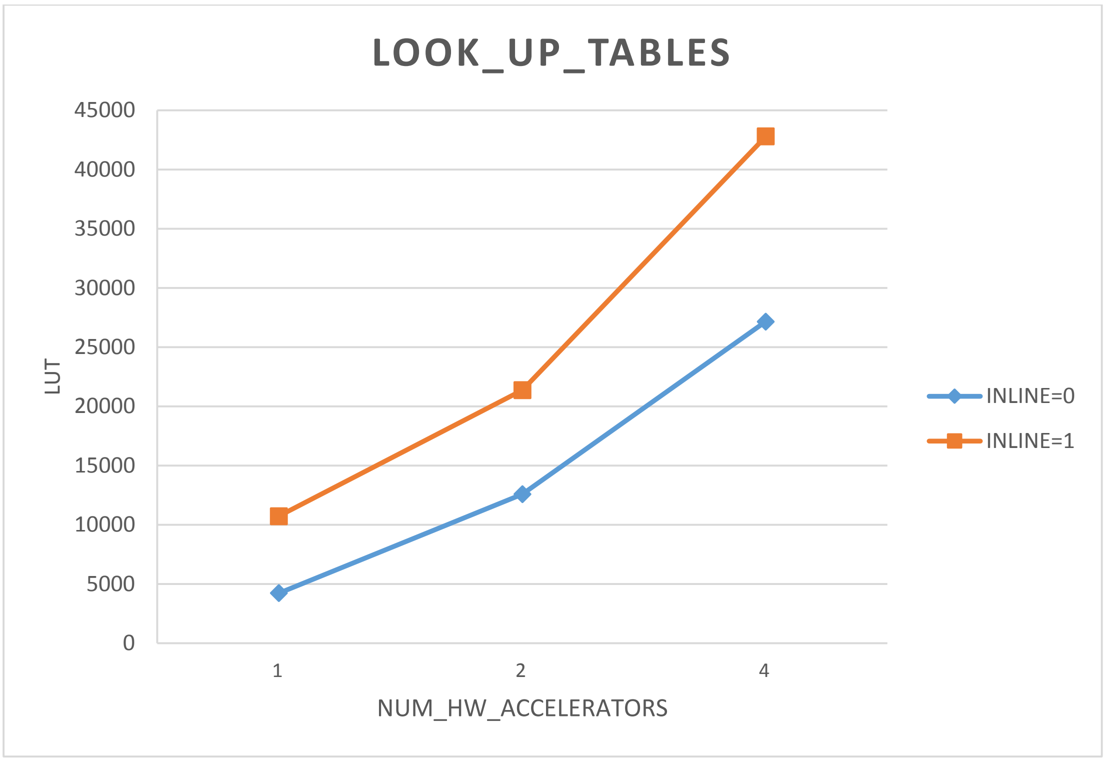
Σχήμα 11: Χρήση FF συναρτήσει των heaps (INLINE)



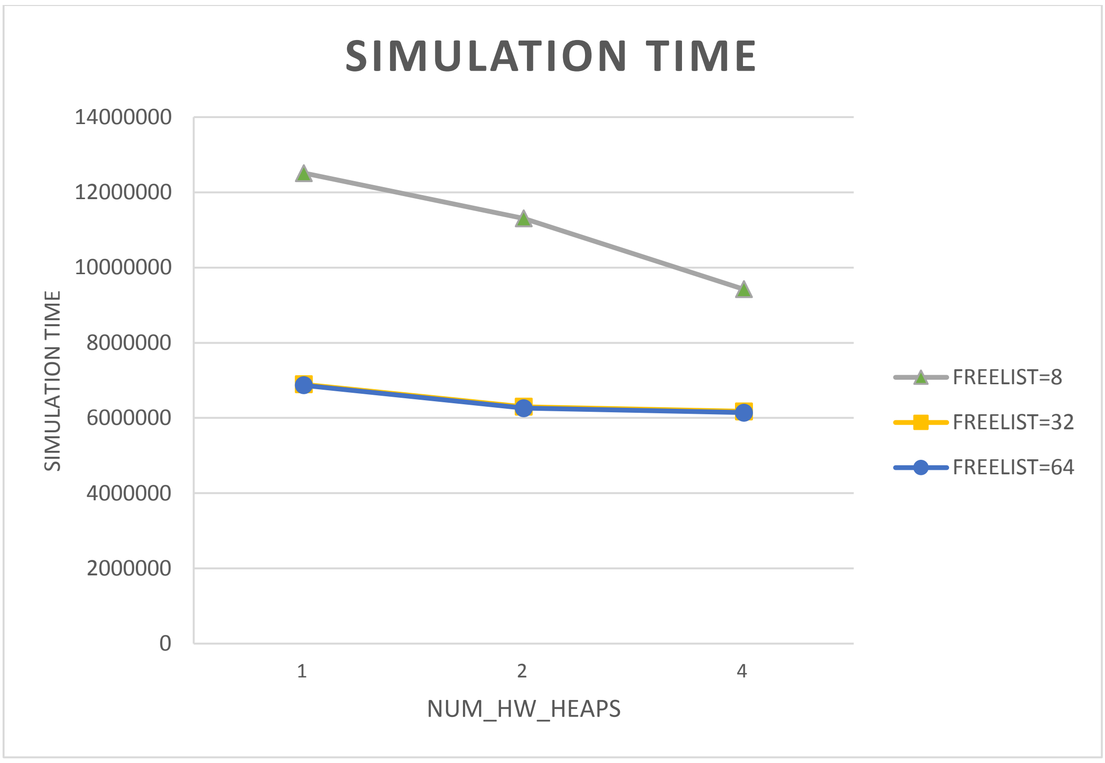
Σχήμα 12: Χρήση FF συναρτήσει των πολλαπλών αλγορίθμων (INLINE)



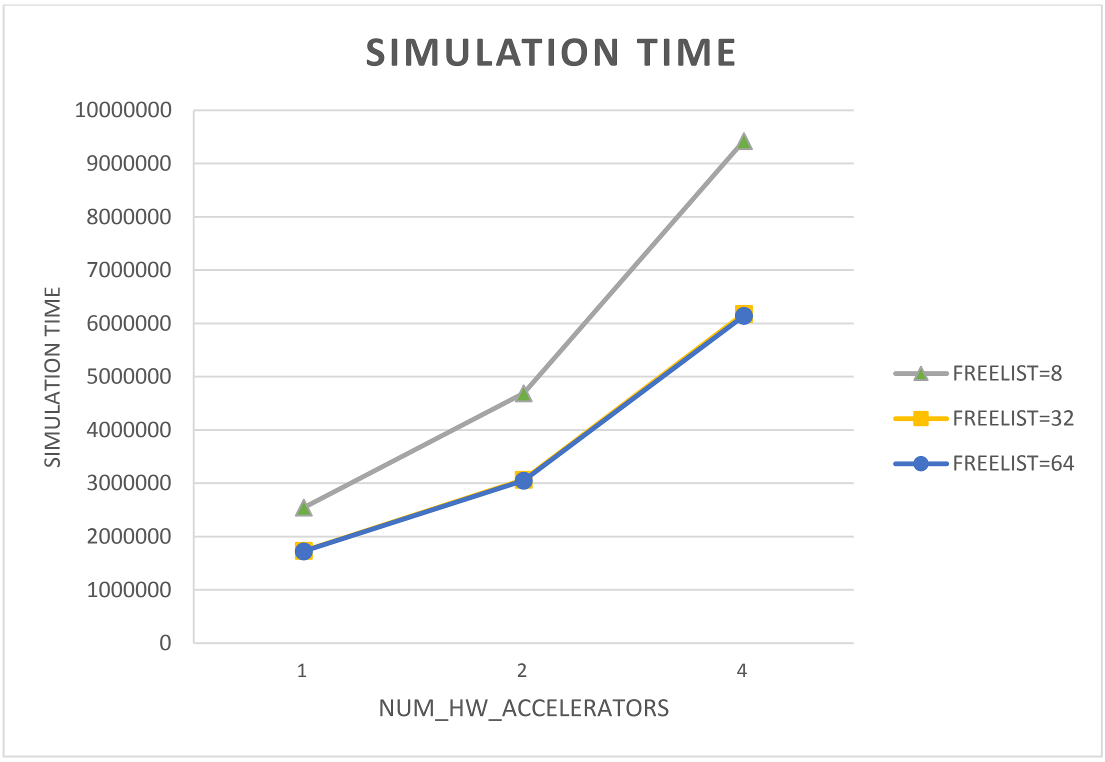
Σχήμα 13: Χρήση LUT συναρτήσει των heaps (INLINE)



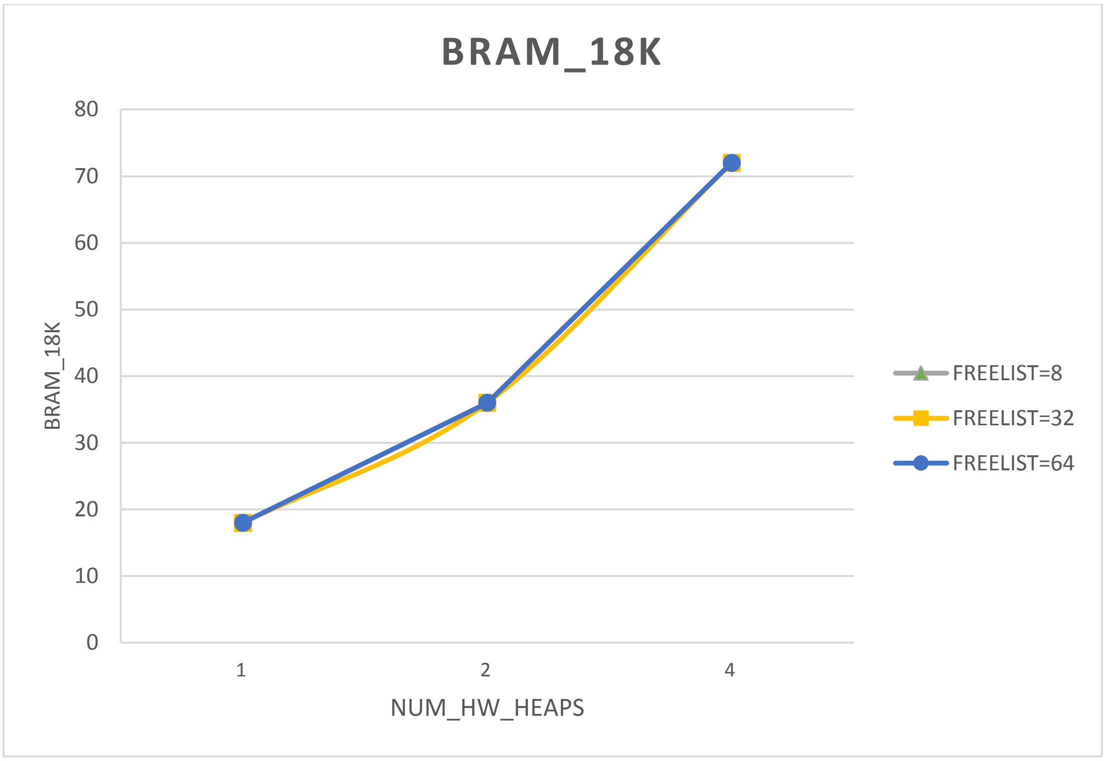
Σχήμα 14: Χρήση LUT συναρτήσει των πολλαπλών αλγορίθμων (INLINE)



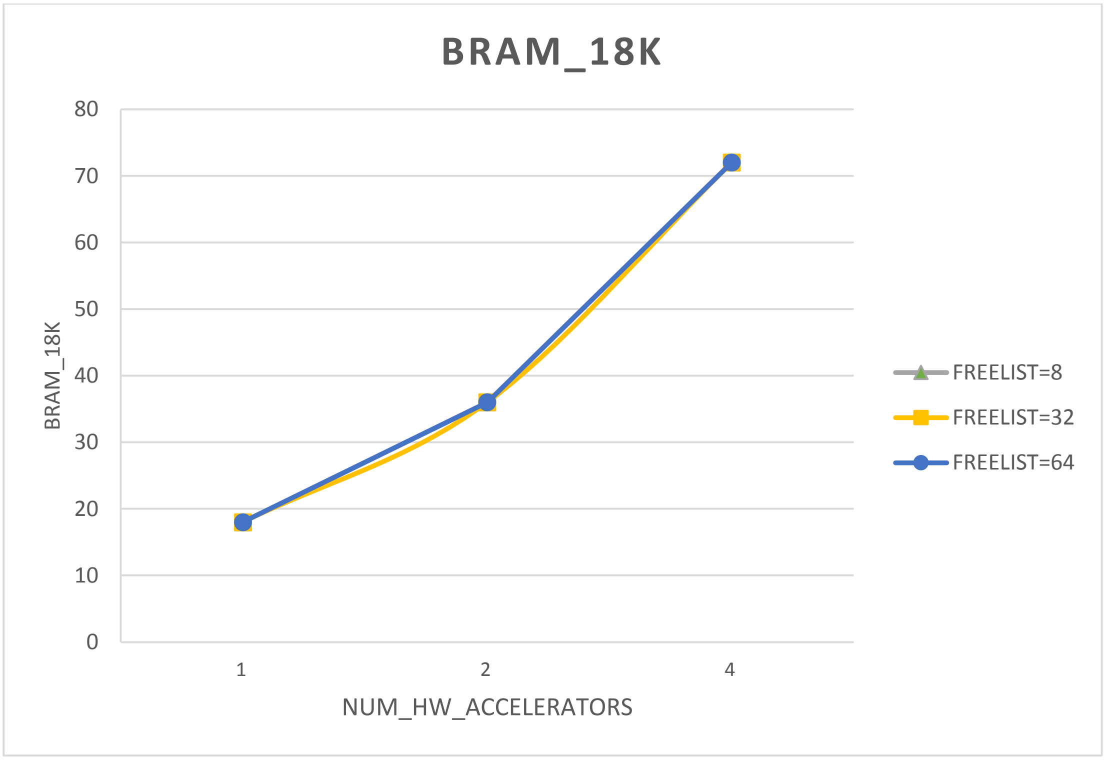
Σχήμα 15: Χρόνος εκτέλεσης συναρτήσει των heaps (Freelist width)



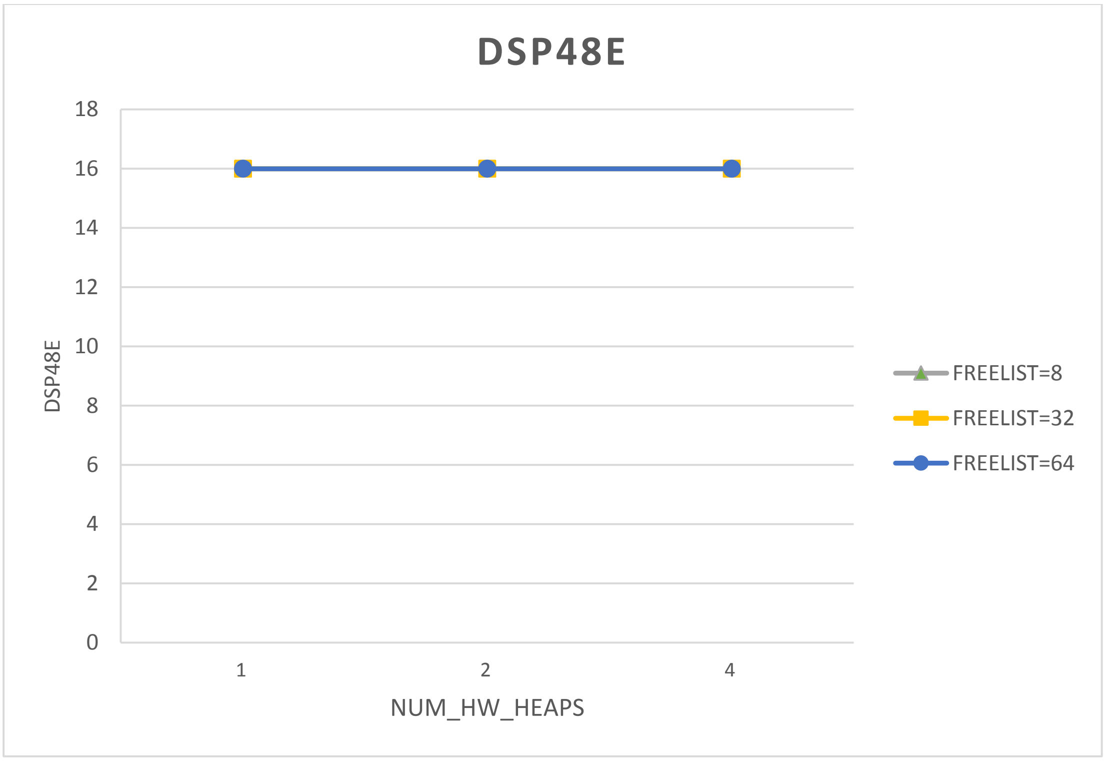
Σχήμα 16: Χρόνος εκτέλεσης συναρτήσει των πολλαπλών αλγορίθμων (Freelist width)



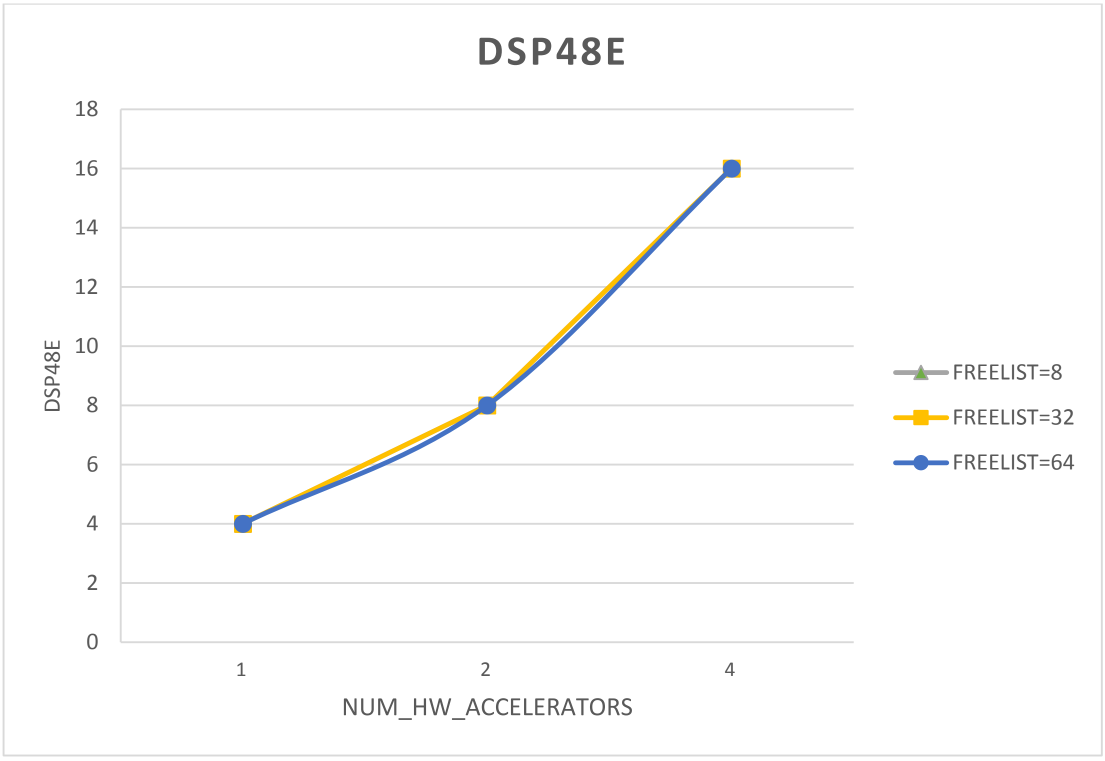
Σχήμα 17: Χρήση BRAM συναρτήσει των heaps (Freelist width)



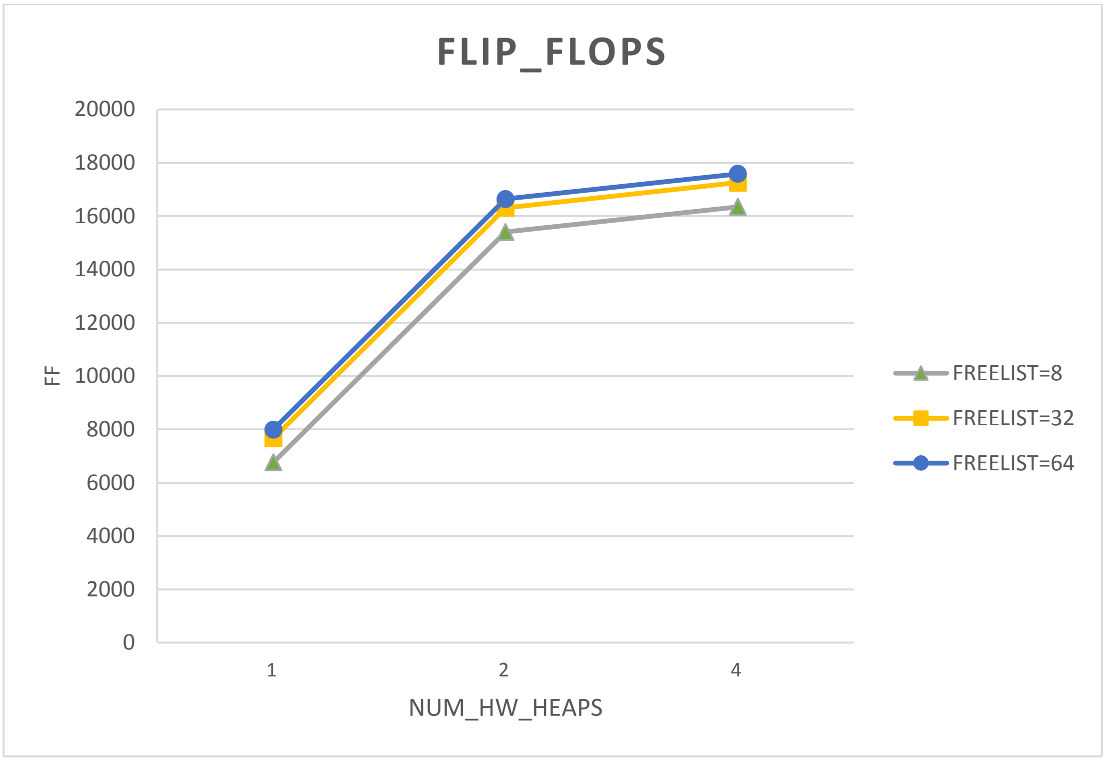
Σχήμα 18: Χρήση BRAM συναρτήσει των πολλαπλών αλγορίθμων (Freelist width)



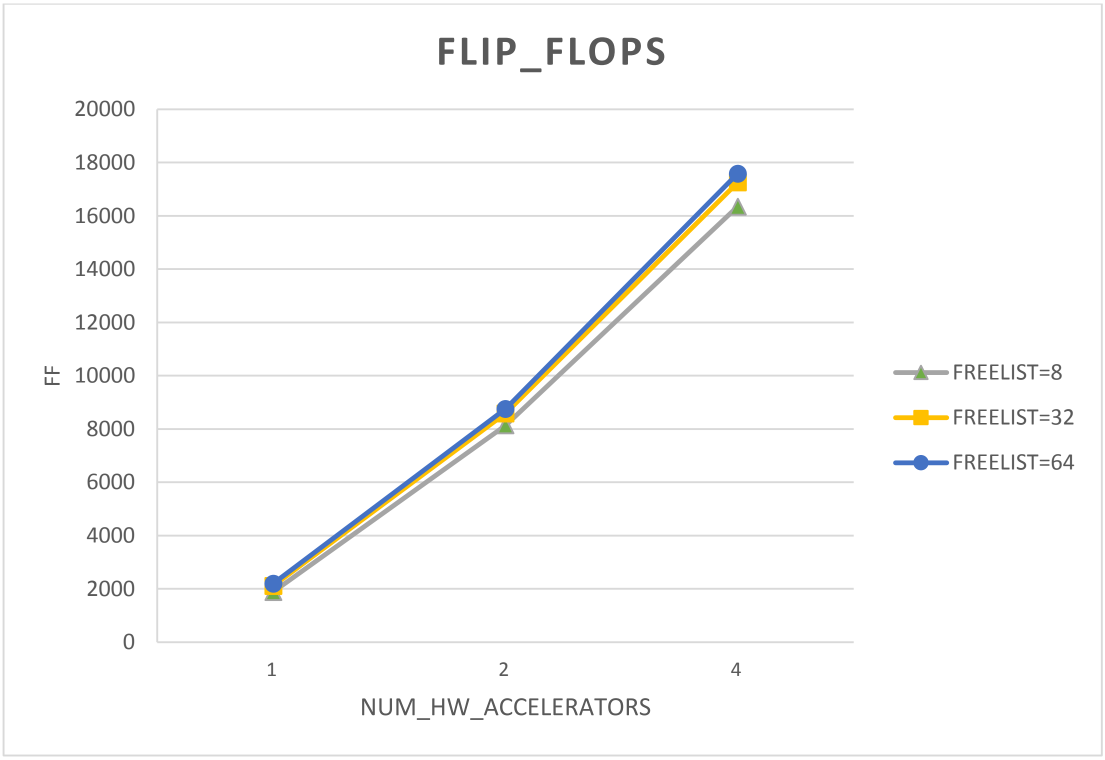
Σχήμα 19: Χρήση DSP συναρτήσει των heaps (Freelist width)



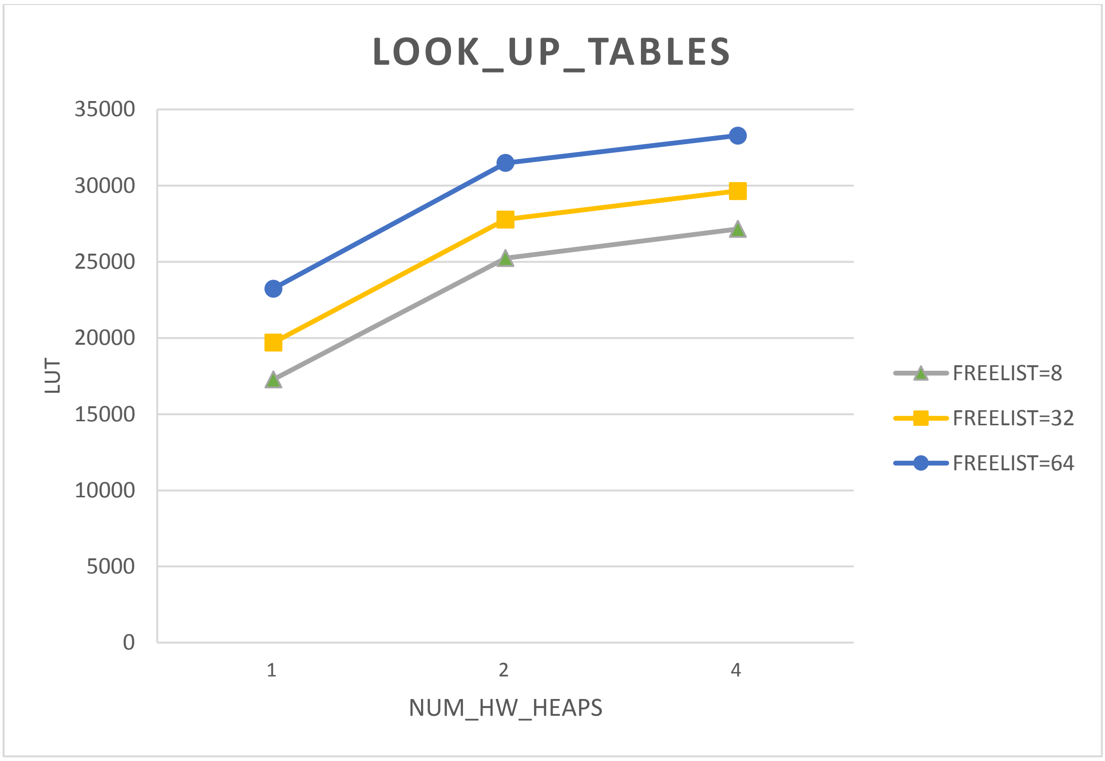
Σχήμα 20: Χρήση DSP συναρτήσει των πολλαπλών αλγορίθμων (Freelist width)



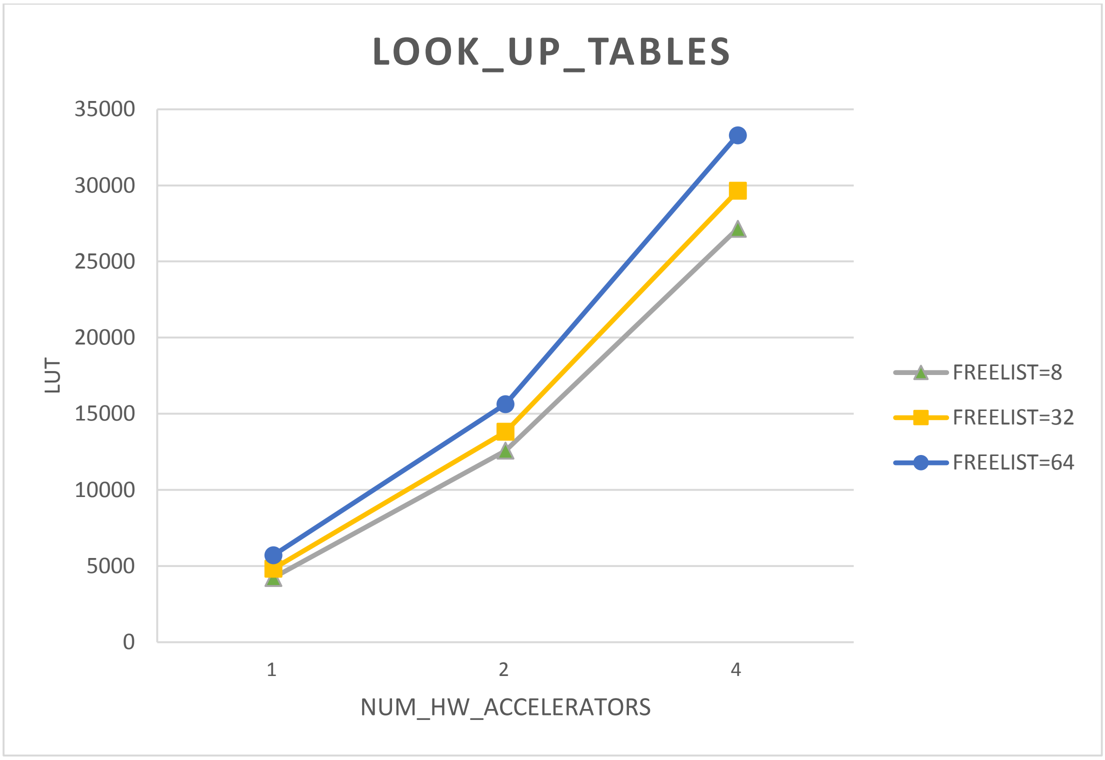
Σχήμα 21: Χρήση FF συναρτήσει των heaps (Freelist width)



Σχήμα 22: Χρήση FF συναρτήσει των πολλαπλών αλγορίθμων (Freelist width)



Σχήμα 23: Χρήση LUT συναρτήσει των heaps (Freelist width)



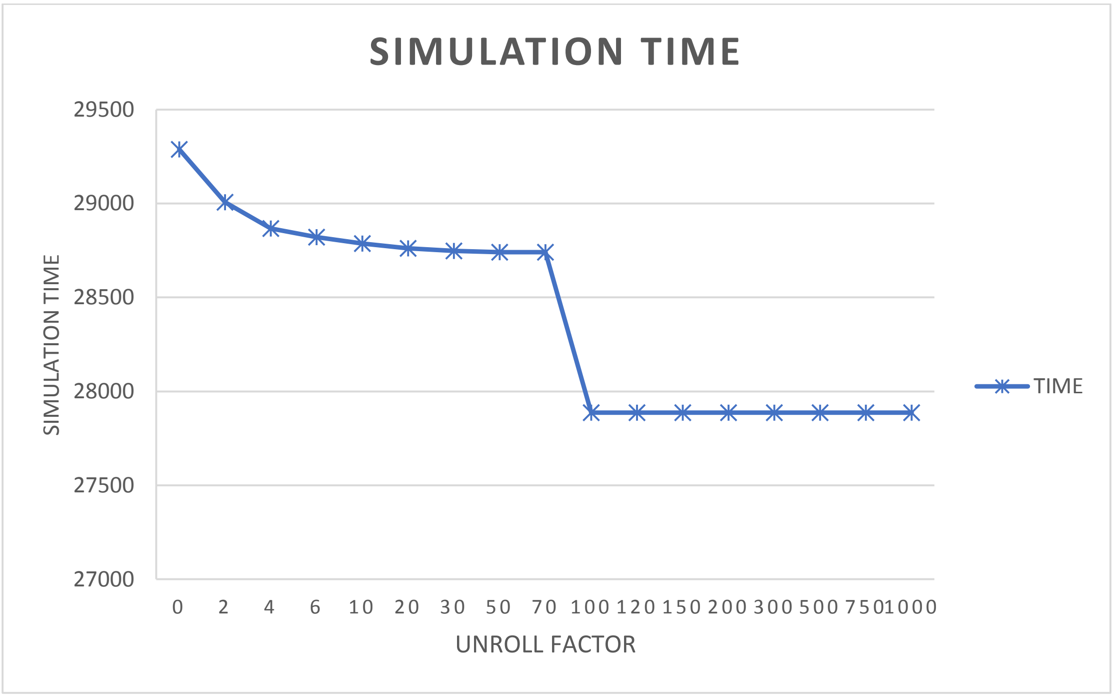
Σχήμα 24: Χρήση LUT συναρτήσει των πολλαπλών αλγορίθμων (Freelist width)

Στο πλαίσιο της εξερεύνησης του χώρου λύσεων για το inline, παρατηρούμε ότι ο χρόνος εκτέλεσης του προγράμματος μειώνεται με μεγαλύτερη κλίση συγκριτικά με την κατάσταση που το inline ήταν απενεργοποιημένο. Αυτό συμβαίνει καθώς το inline επιτυγχάνει ακόμη μεγαλύτερη παραλληλία στις προσβάσεις στους διάφορους heaps της μνήμης (πέρα από την παραλληλία που ήδη υπάρχει καθώς αυξάνουμε τους heaps). Καθώς το σύστημά μας κλιμακώνεται (με την προσθήκη επιπλέον αλγορίθμων και heaps), παρατηρούμε ότι το inline συγκρατεί το χρόνο εκτέλεσης στα ίδια επίπεδα, όσους αλγορίθμους και αν προσθέσουμε, με την προϋπόθεση ότι διατηρούμε την αναλογία 1-1 μεταξύ αλγορίθμων και heaps. Επιπλέον φαίνεται ότι το inline δεν επηρεάζει καθόλου τη χρήση των μονάδων BRAM. Σχετικά με τα DSP, φαίνεται ότι όταν ενεργοποιώ το inline, τα DSP σε χρήση μειώνονται. Αυτό συμβαίνει επειδή οι συναρτήσεις της δυναμικής βιβλιοθήκης συνθέτονται κανονικά σε κώδικα RTL, έχοντας inline ίσο με 0, ενώ για inline ίσο με 1 παραλληλοποιούνται και χρονοδρομολογούνται κατά τέτοιο τρόπο ώστε να επαναχρησιμοποιούν τα ήδη υπάρχοντα κυκλώματα. Γι' αυτό το λόγο παρατηρούμε αυτή τη μείωση στο DSP τόσο στην περίπτωση που αυξάνουμε τους heaps, όσο και στην περίπτωση που αυξάνουμε τους accelerators και τους heaps. Σχετικά με τα FF και τα LUT, παρατηρούμε ότι με την ενεργοποίηση του inline έχουμε αύξηση των μονάδων που χρησιμοποιούνται από αυτούς τους πόρους. Αυτό συμβαίνει επειδή τα στοιχεία αυτά αποτελούν δομικές μονάδες πολυπλεκτών, οι οποίοι ελέγχουν το διαμοιρασμό και την επαναχρησιμοποίηση των υπολγιστικών πόρων (τα οποία προκαλούνται από την ενεργοποίηση του inline).

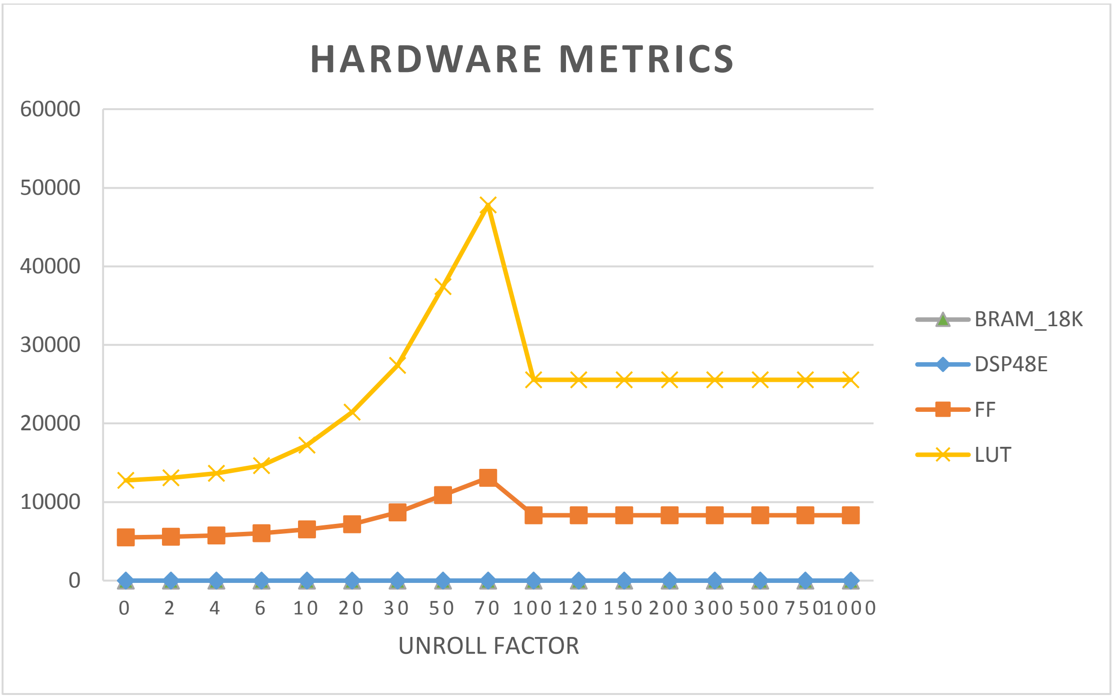
Στο πλαίσιο της εξερεύνησης του χώρου λύσεων για το freelist width, παρατηρούμε ότι ο χρόνος εκτέλεσης του προγράμματος μειώνεται καθώς αυξάνουμε το πλάτος του πίνακα. Αυτό συμβαίνει διότι όσο αυξάνουμε το πλάτος του πίνακα freelist τόσο περισσότερα στοιχεία μπορούμε να ελέγξουμε σε μία επανάληψη. Έτσι, είμαστε σε θέση να εντοπίσουμε γρηγορότερα ένα ελέυθερο τμήμα μνήμης προκειμένου να ικανοποίησουμε ένα αίτημα για δέσμευση μνήμης. Καθώς το σύστημά μας κλιμακώνεται (με την προσθήκη επιπλέον αλγορίθμων και heaps), παρατηρούμε την ίδια συμπεριφορά, δηλαδή για μεγαλύτερα πλάτη του πίνακα freelist παρατηρούμε μείωση στο χρόνο εκτέλεσης. Επιπλέον φαίνεται ότι το freelist width δεν επηρεάζει τη χρήση των μονάδων BRAM και DSP. Σχετικά με τα FF και τα LUT, παρατηρούμε ότι καθώς αυξάνεται το πλάτος του πίνακα freelist έχουμε αύξηση των μονάδων που χρησιμοποιούνται από αυτούς τους πόρους. Αυτό συμβαίνει επειδή τα στοιχεία αυτά χρησιμοποιούνται για την υλοποίηση λογικών μασκών, ολισθήσεων των περιεχομένων των μεταβλητών ή και στην αποθήκευση επιπλέον μεταβλητών, όταν αυτό κρίνεται απαραίτητο. Όλα όσα αναφέραμε εντάσσονται στο πλαίσιο της λειτουργίας του πίνακα freelist και οι ανάγκες για μεγαλύτερες μάσκες ή ολισθήσεις ή για περισσότερες επιπλέον μεταβλητές αυξάνονται όσο αυξάνεται το πλάτος του πίνακα.

Στους υπόλοιπους αλγορίθμους παρατηρούμε σε γενικές γραμμές την ίδια συμπεριφορά για το χρόνο εκτέλεσης και τους υπολογιστικούς πόρους με ορισμένες εξαιρέσεις που οφείλονται είτε σε μικρό σετ δεδομένων εισόδου είτε στα ιδιαίτερα χαρακτηριστικά του κάθε αλγορίθμου.

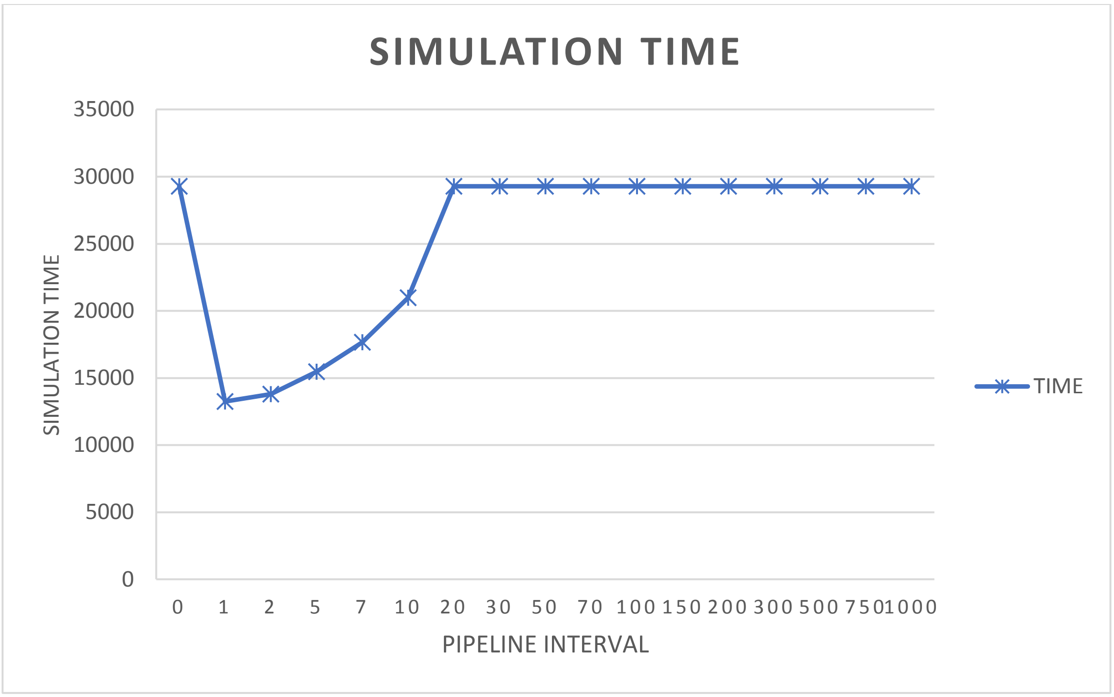
Ακολουθούν τα διαγράμματα που αφορούν την εξερεύνηση του χώρου λύσεων στο επίπεδο του κώδικα του αλγορίθμου Histogram.



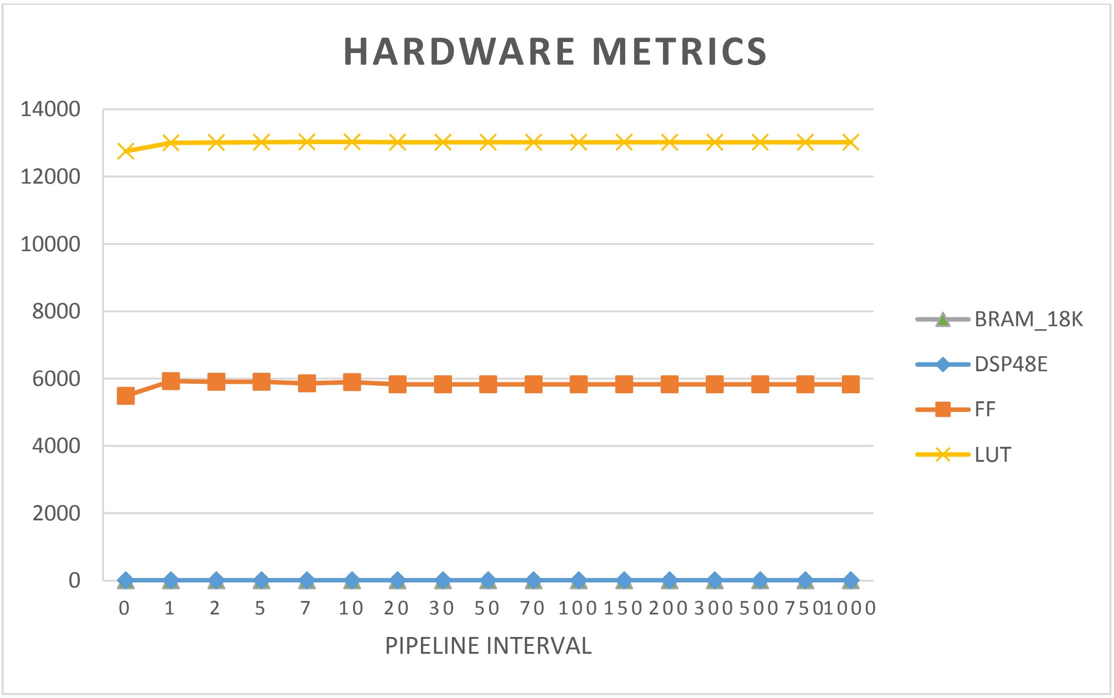
Σχήμα 25: Χρόνος εκτέλεσης εφαρμόζοντας loop unroll



Σχήμα 26: Χρήση υπολογιστικών πόρων εφαρμόζοντας loop unroll



Σχήμα 25: Χρόνος εκτέλεσης εφαρμόζοντας loop pipeline



Σχήμα 26: Χρήση υπολογιστικών πόρων εφαρμόζοντας loop pipeline

Ο αλγόριθμος Histogram φαίνεται να επωφελείται περισσότερο από το loop pipeline. Αυτό συμβαίνει λόγω των ιδιαίτερων χαρακτηριστικών του κώδικα με τον οποίο υλοποιήθηκε ο αλγόριθμος, καθώς και από τις εξαρτήσεις δεδομένων που περιλαμβάνει. Επιπρόσθετα, παρατηρούμε ότι το loop pipeline κοστίζει πολύ λιγότερο σε υπολογιστικούς πόρους σε σχέση με το loop unroll. Αυτό εξηγείται από το γεγονός ότι το loop unroll προκειμένου να βελτιστοποίησει το χρόνο εκτέλεσης κατασκευάζει πολλαπλά κυκλώματα όμοια με το κύκλωμα του βρόγχου που

ξεδιπλώνουμε ώστε να τρέχει ορισμένες επαναλήψεις του βρόγχου παράλληλα. Συνεπώς καταναλώνει πολλούς υπολογιστικούς πόρους συγκριτικά με το loop pipeline, το οποίο επεμβαίνει μόνο στο χρονοπρογραμματισμό των εντολών, διατηρώντας τις ίδιες υπολογιστικές μονάδες. Τέλος δοκιμάσαμε να εφαρμόσουμε και το loop merge, λόγω της μορφής του κώδικα (δύο σειριακοί βρόγχοι), αλλά το Vivado HLS απέτυχε να το εφαρμόσει καθώς δεν τηρούνταν ορισμένες προϋποθέσεις.

Λέξεις κλειδιά

Συστοιχία επιτόπια προγραμματιζόμενων πυλών, FPGA, Πληροφορική Υπερυψηλής Απόδοσης, HPC, Σύνθεση Υψηλού Επιπέδου, HLS, Εξερεύνηση του Χώρου Λύσεων, DSE, Δυναμική Διαχείριση Μνήμης, DMM, Ενσωματωμένα Συστήματα

Abstract

Design Space Exploration (DSE) is the process of discovering and evaluating different design alternatives during system development and before the final implementation [2]. DSE is a very useful and important concept in case of designing and implementing different algorithms in embedded systems, as these systems have limited hardware resources. Nowadays, as embedded systems, and more precisely FPGAs, are used to run HPC applications (High Performance Computing), there is a serious performance bottleneck, due to the starvation of on-chip memory. It is worth mentioning that we have to use a High Level Synthesis (HLS) tool to transform the high-level algorithmic description of these applications to hardware description language in order to be implemented on a FPGA board.

In this thesis we introduce the use of a Dynamic Memory Management (DMM) library and we apply the concept of DSE to memory structures that we allocate through this library. The DSE in case of memory is related with two features of the library, the inline and the width of the freelist array, and examines time and hardware metrics such as simulation time, block RAMs in use, etc. DSE of DMM library is accomplished using several algorithms of Phoenix MapReduce suite [7].

Furthermore, using the same set of algorithms, we apply DSE to the implementation of these algorithms using some loop optimizations, as loop unroll, loop pipeline etc., provided by Vivado HLS, the HLS tool that we use. We analyze the same time and hardware metrics, as before, to evaluate the results of the optimizations.

Key Words

Field-programmable gate array, FPGA, High Performance Computing, HPC, High Level Synthesis, HLS, Design Space Exploration, DSE, Dynamic Memory Management, DMM, Embedded Systems

Ευχαριστίες

Με την παρούσα διπλωματική εργασία κλείνει ο κύκλος των προπτυχιακών σπουδών μου στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών. Στη διάρκεια της φοίτησής μου στο Εθνικό Μετσόβιο Πολυτεχνείο ήρθα αντιμέτωπη με αρκετές προκλήσεις και δυσκολίες, οι οποίες με διαμόρφωσαν ως μηχανικό. Στο σημείο αυτό θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα αυτής της διπλωματικής, κ. Δημήτριο Σούντρη, ο οποίος μου έδωσε τη δυνατότητα να ασχοληθώ σε βάθος με τον προγραμματισμό των ενσωματωμένων συστημάτων. Επιπλέον, θα ήθελα να ευχαριστήσω τον Δρα. Διονύση Διαμαντόπουλο καθώς και τον Δρα. Σωτήρη Ξύδη για την συνεχή βοήθεια και καθοδήγηση κατά την εκπόνηση της παρούσας εργασίας. Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου και τους φίλους μου που στάθηκαν δίπλα μου και με στήριξαν όλα τα χρόνια της φοιτητικής μου ζωής.

Αγγελική Α. Δαβουρλή,
Αθήνα 9^{η} Μαρτίου 2016

Contents

List of figures

List of tables

Chapter 1

Introduction

Field-Programmable Gate Array, or FPGA, is a type of a general-purpose integrated circuit, which is able to be configured as many times as needed by the designer or the customer after manufacturing. This basic characteristic is the main reason for using FPGAs in several markets and applications as aerospace and defence, automotive, medical applications, video and image processing, wired and wireless communications, High Performance Computing (HPC) etc.. Although, given the fact that the FPGA configuration is specified using Hardware Description Languages (HDL), FPGAs become difficult to be programmed by designers that are not familiar with these languages. Therefore, alongside the evolution of FPGA devices, there is the evolution of High Level Synthesis tools (HLS tools). The idea of High Level Synthesis consists of “an automated design process that deals with the generation of behavioral hardware descriptions from high-level algorithmic specifications” [1]. Besides the concept of HLS, the wide variety of uses of FPGAs has led to the deployment of another stage of the system development process, the Design Space Exploration (DSE). Design Space Exploration is the process of discovering and evaluating different design alternatives during system development and before the final implementation [2]. DSE is very useful in case of designing and implementing different algorithms in an FPGA board, as the board has limited hardware resources.

In this diploma thesis, we study the concept of DSE, by applying it initially in the level of memory management and evaluating the result using several algorithms of Phoenix MapReduce suite [7]. In order to explore the different memory structures that may suit to our implementations, we propose the use of Dynamic Memory Management (DMM) concerning the FPGA’s on-chip memory. After that, we apply DSE into the implementation of those algorithms using some directives provided by the HLS tool that we are using.

Chapter 2 is an introduction to FPGA devices. We discuss the definition of a FPGA and its basic characteristics. The components and the basic building blocks of FPGA boards are studied, focusing on hardware resources that we measure during our DSE. Afterwards, we highlight the differences between programming an FPGA and a processor and finally we analyze some basic stages of FPGA programming as scheduling, pipelining and dataflow.

Chapter 3 refers extensively to the concept of Design Space Exploration. The idea of High Level Synthesis is analyzed in order to be combined with DSE. We emphasize the importance of HLS tools in embedded systems and we study some HLS directives for loop optimizations. In addition, Dynamic Memory Management is introduced as a concept in embedded systems and DMM-HLS library is proposed in order to overcome performance bottlenecks. We further analyze the two DMM features that we evaluate through DSE, inline and width of freelist array.

Chapter 4 includes the experimental results of the DSE of DMM library concerning inline and freelist width. We explore these DMM options in case of many-accelerator (MA) systems in order to maximize and analyze further their impact. We also study the role of scheduler and synthesizer of Vivado HLS in several simulations.

Chapter 5 consists of the experimental results of DSE using HLS directives. We analyze compiler and code optimizations concerning loops and we examine the special circumstances under which they may be applied by Vivado HLS. We finally go through a dependency analysis regarding the kernels that we want to apply optimizations.

Finally, this thesis concludes in Chapter 6 where general remarks are discussed and potential future work is proposed.

Chapter 2

Introduction to FPGA

Field-Programmable Gate Array, or FPGA, is a type of a general-purpose integrated circuit, which is designed to be configured by the designer or the customer after manufacturing. A hardware description language or HDL specifies the configuration of an FPGA. A big variety of algorithms can be implemented into an FPGA and this is because FPGAs contain a large amount of programmable logic blocks combined with reconfigurable interconnects. Those hardware elements can be connected in several ways to perform either a complex algorithmic concept or a simple logic function, as AND, OR etc. Furthermore, it is worth mentioning that in most FPGA devices the programmable logic blocks contain also some memory elements, such as flip-flops. One of the remarkable advantages of an FPGA, regarding other integrated circuits, is their ability to be reconfigurable dynamically. In particular, we can reconfigure an FPGA, as many times as we want and this process is similar to the process of loading a program in a processor.

2.1 History

The FPGA industry comes from the combination of the PROM (programmable read-only memory) and the PLD (programmable logic devices) technologies. PLDs are electronic devices which include an array of logic gates AND and logic gates OR, while PROMs are non-volatile memory circuits. Both PLDs and PROMs can be configured and programmed during manufacturing or after that (field programmable). Altera delivered the EP300 in 1984, the industry’s first reprogrammable device. The EP300 was able to be reprogrammable because of a quartz window in the device. Therefore, when the user wanted to erase the EPROM cells that held the device configuration, the only thing to be done was to shine an ultra-violet lamp on that window. One year later, Xilinx co-founders Ross Freeman and Bernard Vonderschmitt invented the XC2064, the first feasible field-programmable gate array. This device included 64 configurable logic blocks (CLBs) with 2 three-input lookup tables (LUTs). These two companies, Xilinx and Altera, continue to be the industry leaders in the FPGA market, representing together approximately 77% of the market. Regarding the use of the FPGAs, they initially used in telecommunications and networking but their use expanded quickly in the areas of automotive, consumer and industrial applications.

2.2 FPGA Architecture

A Field-Programmable Gate Array consists of a big amount of Configurable Logic Blocks (CLBs), which is the main functional unit of an FPGA. CLBs are organized in a two dimensional array as shown in the following figure.

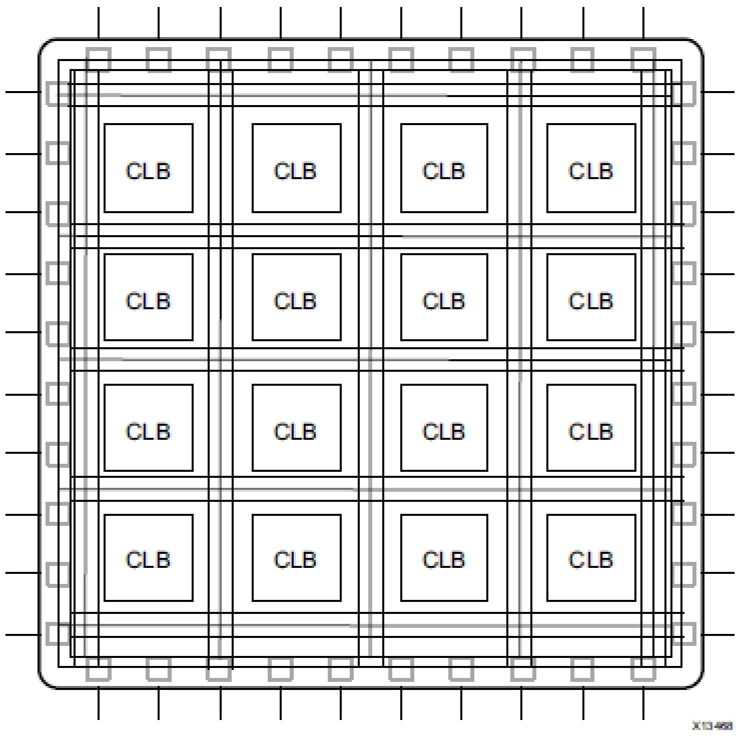


Figure 1: Array of CLBs [4]

Besides the CLBs, an FPGA board includes also some I/O blocks (IOB) in order to communicate with external devices and some programmable switch matrices (SM). A more detailed view can be shown in the following figure.

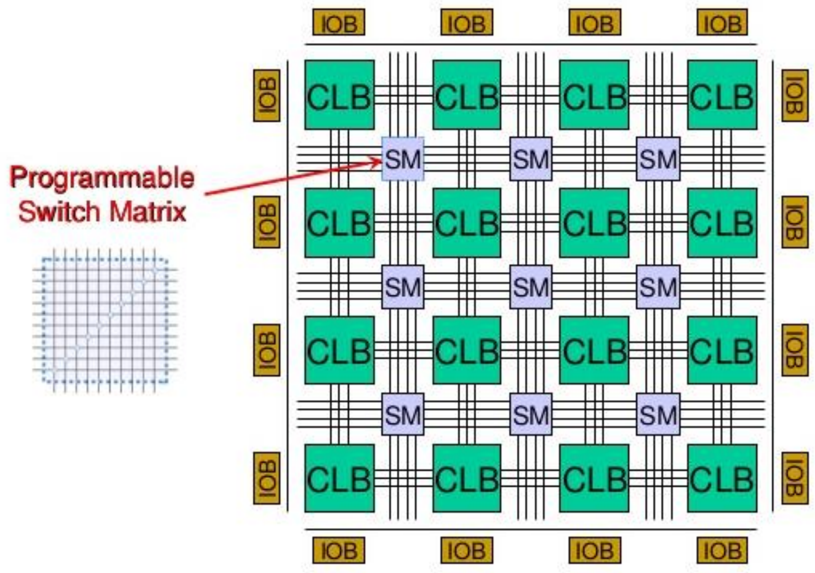


Figure 2: Structure of an FPGA [3]

The above structure is sufficient for the implementation of a variety of algorithms, but has some limitations regarding computational throughput, required resources and clock frequency [4].

2.2.1 FPGA Components

2.2.1.1 Configurable Logic Blocks (CLBs)

The CLBs are composed of the following hardware elements:

1. Look-up tables
2. Flip-Flops
3. Wires
4. Input/Output (I/O) pads

Some of these elements are been discussed in the following sections.

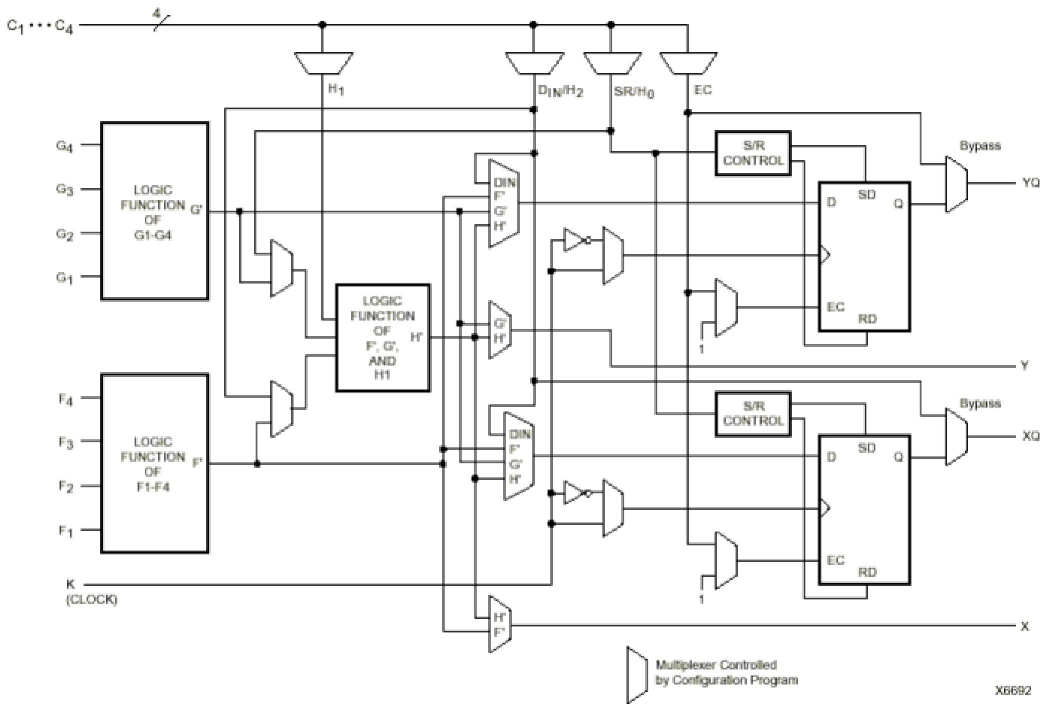


Figure 3: Simplified block diagram of XC4000 Series CLB [8]

As we can see, the look-up tables implement the logic functions into the CLB, the flip-flops store the results of the LUTs while the multiplexers set/clear input selection of the flip-flops and route the logic into the block and to and from external resources (using C1, C2, C3, C4, K signals) [4].

2.2.1.2 *Input Output Blocks (IOBs)*

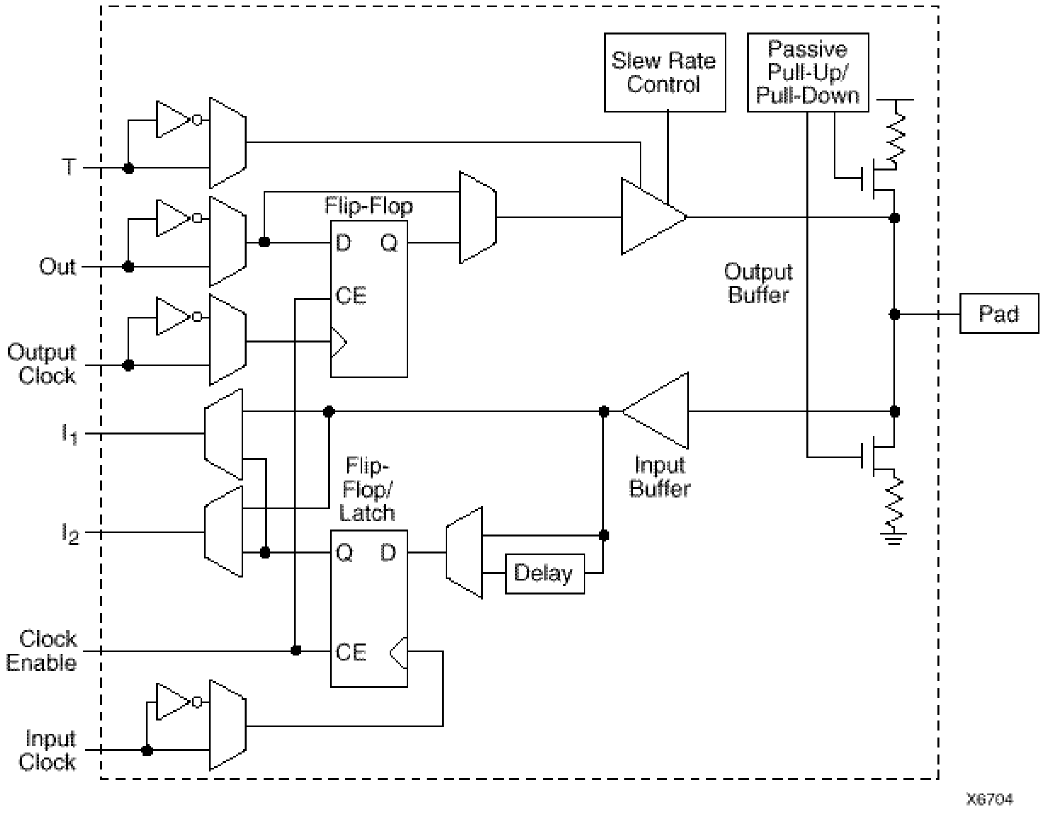


Figure 4: Simplified block diagram of Input Output Block of XC4000 Series [8]

The IOB contains an input buffer and an output buffer with three-state and open collector output controls. There are pull up and pull down resistors on the outputs in order to terminate signals and buses without requiring external resistors to do that. Furthermore there are flip-flops on inputs in order to minimize the circuit delay, while the flip-flops on the outputs not only reduce the delay but also synchronize the IOB with external devices [4].

2.2.1.3 *Programmable Switch Matrices*

Finally, regarding the programmable switch matrices, figure 5 shows us the interconnection between the CLBs. An FPGA board contains long and short lines that connect critical CLBs, which are physically far or close from each other respectively. These long and short lines are connected via switch matrices. There are also three-state buffers that connect many CLBs to a long line, creating a bus. Some special long lines, the global clock lines, are designed to carry signals with fast propagation times, thus these lines are connected to the clocked elements of each CLB in order to synchronize all the FPGA CLBs. However, in an FPGA, most of the delay in the chip comes from the interconnection that we described.

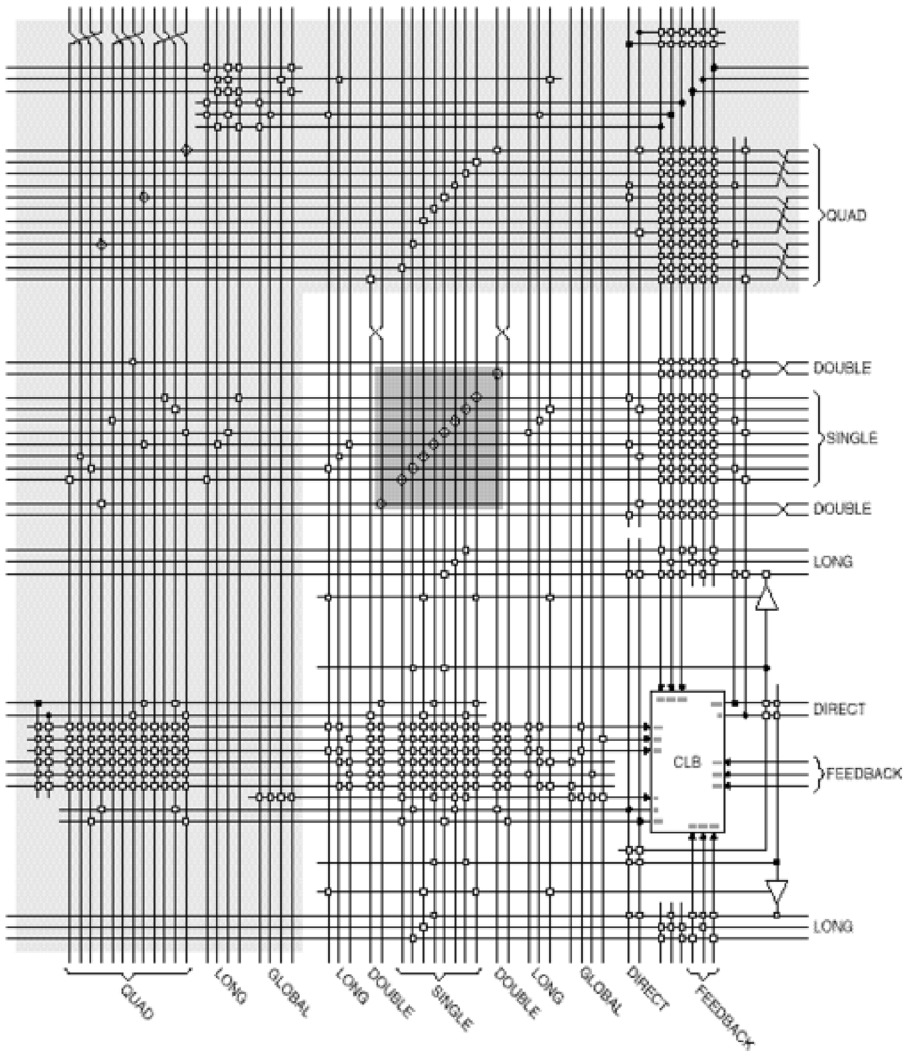


Figure 5: Programmable Interconnect Associated with XC4000 Series CLB [8]

Contemporary FPGA devices incorporate not only the basic elements, as they are described ab, but also some additional computational and data storage blocks. These additional elements are [4]:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

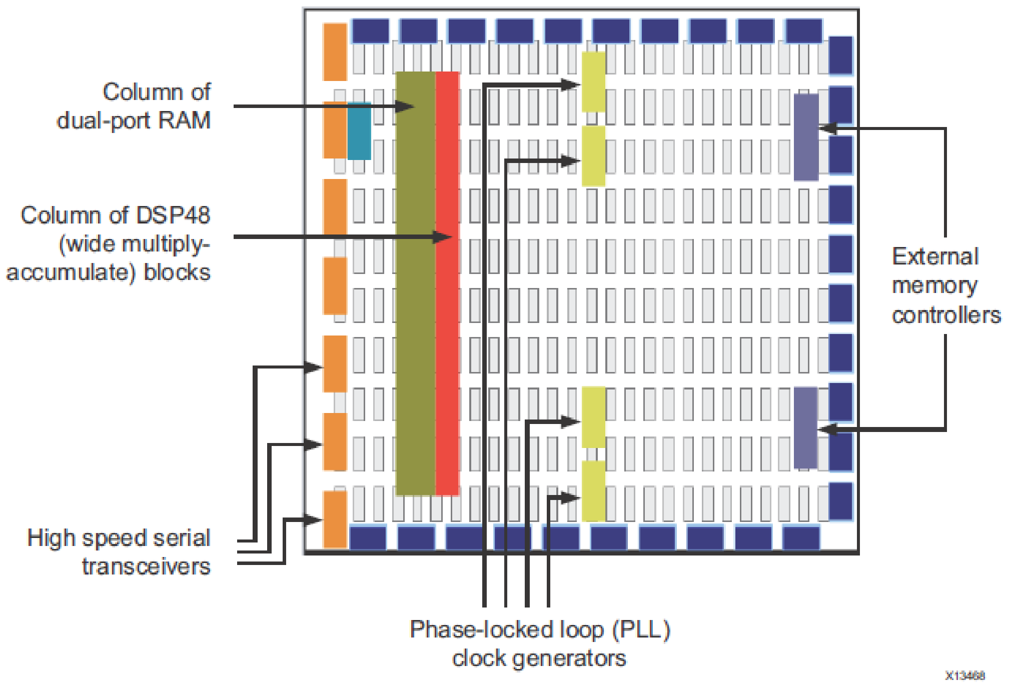

Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
Phase-locked loop (PLL) clock generators
External memory controllers
X13468

Figure 6: Contemporary FPGA Architecture [4]

2.2.2 Basic building blocks and units of FPGAs

2.2.2.1 Look-up Tables (LUTs)

As it is mentioned in 2.2.1.1, LUTs are used for the implementation of any logic function in the CLB, and therefore in the FPGA. This element is actually a truth table, in which different values as inputs to different logical functions generate different outputs. For a number of inputs equal to N, the size of the truth table is N and the number of memory locations that can be accessed by the table are 2^N, allowing the table to implement 2^{N^N} number of logical functions.

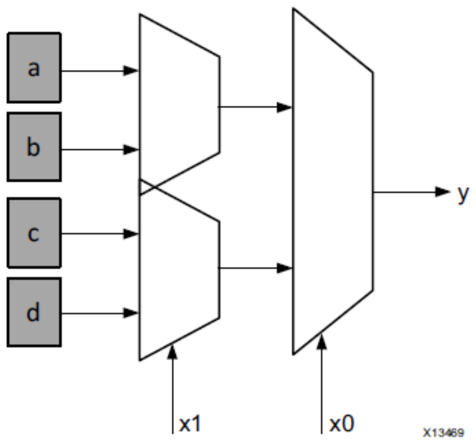


Figure 7: Functional Representation of a LUT as Collection of Memory Cells [4]

Regarding the hardware implementation of LUTs, we can describe this unit as a collection of memory cells connected to a structure of multiplexers. The a, b, c, d inputs of the LUT (Figure 7) may act as selection bits for the first level of multiplexers in order to select the output y. Because of this representation, it becomes clear that a LUT can be useful as both a function compute engine and a data storage element [4].

2.2.2.2 Flip-Flop

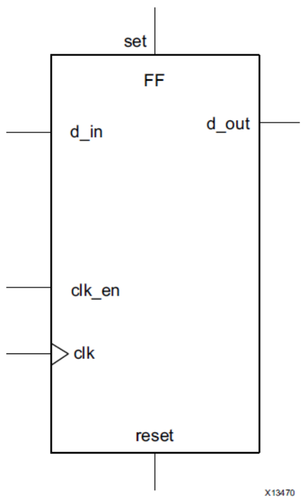


Figure 8: Structure of a Flip-Flop [4]

Flip-Flop circuit includes a data input, a clock input, a clock enable pin, a reset pin and a data output. The normal operation of a flip-flop is to pass the input value to the output on every pulse of the clock. Using the clock enable pin, the flip-flop has the ability to hold a specific value for more than one clock cycle before propagate it to the output. Therefore, the input values can be transmitted to the output when both clock and clock enable are equal to logical one. The flip-flop is the basic storage unit in an FPGA board and is always paired with a LUT for logic pipelining and data storage purposes [4].

2.2.2.3 DSP48 Block

The DSP48 block is the arithmetic logic unit (ALU) embedded in a Xilinx FPGA and it is maybe the most complex computational block. It is a computational block that was added in the FPGA later on, in order to fulfill the demands for extra computational load.

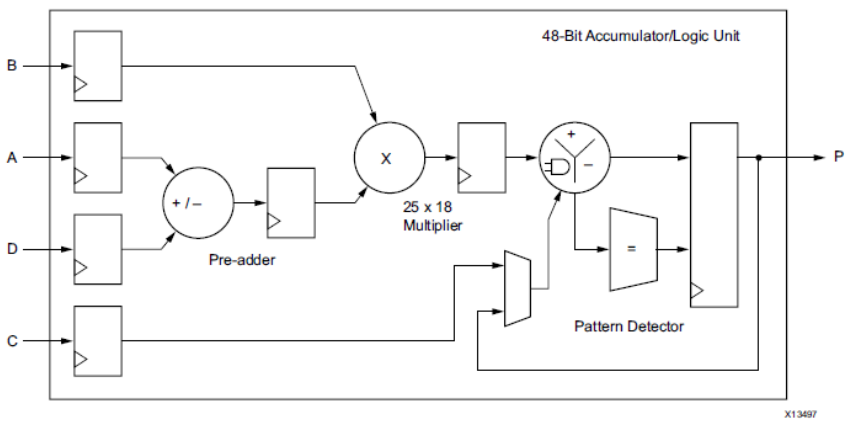


Figure 9: Structure of a DSP48 Block [4]

The DSP48 block is composed of a chain of 3 different blocks, an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine [4].

2.2.2.4 Memory structures

FPGA boards contain three types of embedded memory, random-access memory (RAM), read-only memory (ROM) and shift registers. These memory types are implemented using block RAMs (BRAMs), LUTs and shift registers [4].

2.2.2.4.1 BRAMs

Block RAM is a dual-port RAM embedded in the FPGA board for the storage of a large set of data on-chip. There are two types of BRAM memories on the FPGA, 18kbits and 36kbits, the number of each is determinate for a specific device. Block RAMs can implement either a RAM or a ROM [4].

2.2.2.4.2 LUTs

As it is mentioned in the 2.2.2.1 section, LUTs may be used as memory structures. In particular, they can be 64-bit memory units. These distributed memories, as LUTs are often called, are the fastest kind of memory on the FPGA because of the fact that this kind of memory can be instantiated in any part of the board to improve the performance of the implemented algorithm [4].

2.2.2.4.3 Shift registers

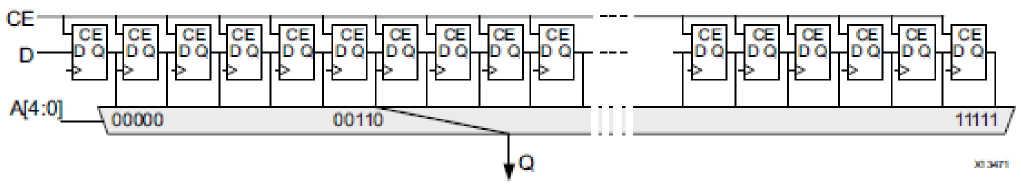


Figure 10: Structure of an Addressable Shift Register [4]

As we can see from Figure 10, a shift register is a chain of connected registers. The main purpose of this memory unit is the data reuse along a computational path [4].

2.2.3 Programming an FPGA

2.2.3.1 *Differences between an FPGA and a Processor*

The FPGA structures enable a high degree of parallelism regarding the algorithm that is implemented in the board, compared with processor architecture. More particular, when a program is executed on a processor, processor compiler transforms the algorithm's C/C++ code into assembly language. Therefore, one C command may be translated into multiple assembly commands, with different number of clock cycles to complete. Another issue, when we execute a program on a processor, is that the software engineers have to spend a lot of time restructuring their algorithms in order to increase spatial locality of data in memory and decrease the processor time spent per instruction. On the contrary, the same operation in an FPGA does not require this effort. The HLS compiler that is used for the implementation of an algorithm in the FPGA board, transforms the software description written in C/C++ into RTL (Register Transfer Language). This transformation has no limitations regarding cache and unified memory space. Instead of assembly commands, the C/C++ are transformed via HLS compiler into a several number of LUTs, depending on the size of the output of each command. Furthermore, the HLS compiler allocates multiple storage banks of memory as close as possible to the needs of every C/C++ command. This implementation leads to high parallelism during the execution and memory accesses of each command. Regarding execution, it is clear that using independent sets of LUTs instead of a central ALU (which exists in a processor) is a far more parallel process of command execution. Also, regarding memory accesses, using several storage banks of memory instead of one unified memory decreases the number of data dependencies and the stalling between commands. There are the processes of scheduling, pipelining and dataflow that are essential for the extraction of the best possible circuit-level implementation of a software application, taking advantage of all the capabilities of an FPGA board [4].

2.2.3.2 Scheduling

The process of identifying the data and control dependencies between several commands is called scheduling. This identification has to be done in order to determine when each command will be executed regarding the dependencies. We can also refer to scheduling as parallelizing the software algorithm for hardware implementation, which is a manual process necessary for the algorithm implementation in traditional FPGA boards.

Scheduling allows the HLS compiler to group all the operations that can be executed in the same clock cycle and to configure all the necessary hardware to allow the overlap of function calls, if any exists. The overlapping of function calls can raise any limit concerning the current function call to fully complete before the next function call. The process of executing 2 function calls in parallel is called pipelining [4].

2.2.3.3 Pipelining

In order to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation, the designers are using a digital designed technique named pipelining. More precisely, pipelining changes the source of data for each stage of the algorithm. Without pipelining, each stage of the algorithm computation starts when the previous stage has been completed and the data are available. Using pipelining, designers are able to separate a computation into several steps so as to have the data available for the next computation before the previous one is completed. Pipelining can be applied in hardware level by using registers between the circuits of each computation. These registers are flip-flop blocks implemented in the FPGA fabric. Using the registers, developers can isolate each electronic circuit (one circuit for each computation) into separate compute sections in time. This can lead to compute more than one values in parallel and allows the overlapping between the sequential computations [4].

2.2.3.4 Dataflow

Dataflow is another digital designed technique, which refers to the pipelining of functions into the same algorithm. More particular, this technique enables the parallel execution of functions in the same program. There are two different types of interaction between 2 functions. The first type of interaction is when the functions are using completely different data sets and do not communicate. In this case, the function parallelism is feasible and efficient. Regarding the second type of interaction, it refers to functions that may share data and provide results to other functions. Although this type of interaction is more complicated, the function parallelism is still feasible but not as efficient as in the previous case [4].

Chapter 3

Design Space Exploration

3.1 Introduction to Design Space Exploration in Embedded Systems

Design Space Exploration (DSE) is the process of discovering and evaluating different design alternatives during system development and before the final implementation [2]. DSE can be used for several purposes, like rapid prototyping, optimization, system integration, etc. In particular, DSE consists of the evaluation of a large set of design alternatives, which will finally implement the same specific system. Regarding embedded systems, these alternatives may concern different hardware component allocations or lower level design parameters (clock frequency etc.). They may also concern different mappings of software tasks to resources or different scheduling policies regarding shared resources. It becomes clear that, if we involve multiple different optimization goals in a DSE process, those design alternatives are a trade-off between these goals. The main challenge of DSE arises from the fact that there is a need of exploring the design space of a large system in a cost-effective manner (in case we have a large system with too many possible design alternatives) [2] [9].

DSE can be applied in various levels of abstraction during the designing process of the system.

In this diploma thesis, we implement DSE into

- Logic Design of each algorithm (using directives/pragmas of High Level Synthesis)
- Memory structure that supports each algorithm (using MEMLUV, a library of dynamic memory allocation)

3.2 DSE combined with High Level Synthesis

After the definition of DSE, it is necessary to determine and analyze the concept of High Level Synthesis (HLS) before use it for the needs of design space exploration.

“High-level synthesis (HLS) is an automated design process that deals with the generation of behavioral hardware descriptions from high-level algorithmic specifications” [1]. Although commercial HLS systems are known since 1990s, it is the rapid increase of complexity in system-on-a-chip (SoC) design that has encouraged engineers to seek for an automated synthesis of high-level descriptions to low-level cycle accurate RTL, for both FPGAs and application-specific integrated circuits (ASICs) [10] [1].

Some of the reasons why HLS tools play a central role in embedded technology are the following:

- *Embedded systems are used in a wide variety of devices and applications.* Considering that not all the designers are familiar with HDL (Hardware Description Languages), it is very convenient for them to be able to specify the functionality of an application by using high-level programming languages (like C/C++) for both embedded software and hardware logic of the SoC [10].

- *Hardware design written in high-level programming languages can manageable and maintainable*. Regarding the behavioral description of a design, it is known that HDL languages need many more code lines in order to describe a design than a high-level language like C. So, it is feasible to reduce the number of code lines up to 10 times by using high-level programming languages, and this can result in a manageable and maintainable code [10].

- *Reusability of behavioral intellectual properties (IPs).* In contrast to RTL IPs, which have specific and invariable interface protocols and architecture, behavioral IPs can be easily reused in many different system designs. This characteristic of behavioral IPs increases design productivity, because for every new requirement of the system, there are some IPs already designed and ready to be placed correctly [10].

- *Usage of high-level specification for system verification needs*. SystemC TLMs (Transaction-Level Modeling) and other C/C++ based extensions are used to describe software or hardware platforms regarding software development, exploration and modeling of system architecture and verification of its functionality. The connection of these SystemC models with HLS solutions can lead to generate automatically RTL code, instead of writing this code manually [10].

- *Modern computer systems use multiple accelerators with custom architecture and heterogeneous SoC.* More precisely, the implementation of multiple accelerators with custom architecture in an FPGA board can result in the reduction of power consumption simultaneously with high-performance achievement. The architecture of heterogeneous systems can be efficiently extracted by using HLS tools [10].

HLS is strongly correlated with the FPGA technology and DSE regarding FPGA boards because of the typical characteristics of FPGA boards.

Some of the FPGA characteristics that lead to the growing use of HLS tools are the following:

- *The concept of platform-based synthesis*. Many components of modern FPGA boards are predesigned and fabricated IPs (like arithmetic functional units, memories, system buses etc.) in order to reduce the area that they cover on the board. This can result in implementing as many IPs as possible in the same board. So, an HLS tool can help the designer to synthesize the algorithm, written in a high-level programming language, concerning the specific and predefined hardware that he/she has at his/her disposal [10].

- *Time-to-market criticality for FPGA platforms*. FPGA boards are often selected for systems that have to be delivered quickly to the market. By using HLS tools, designers may achieve a satisfactory reduction in design time and a sufficient code quality compared to hand-written RTL.

- *High-performance computing need for reconfigurable system architecture*. High-performance computing (HPC) applications may be accelerated by utilizing reconfigurable computing platforms, like FPGAs. In the case, designers have to use HLS tools in order to take advantage of the available hardware and adapt the algorithms without using HDL programming [10].

In this diploma thesis, we apply the concept of High Level Synthesis by using Vivado HLS tool from Xilinx Corporation. In fact, we use Vivado HLS for building and synthesizing algorithms for Design Space Exploration regarding either Dynamic Memory Management or HLS directives.

Among a wide variety of available HLS pragmas/directives, we decided to focus on pragmas regarding loop transformations because of the type of algorithms that we study. We further analyze these algorithms in Chapter 4. In Chapter 5 we apply the following HLS directives in order to explore the design space of those algorithms.

3.2.1 High Level Synthesis Directives

High Level Synthesis pragmas/directives contribute to the DSE inside the body of the algorithm. More precisely, we want to find the optimized design of a specific algorithm in a given hardware platform, the minimum latency simultaneously with the minimum number of hardware components in use.

In this section we examine the general concepts of loop unroll, loop pipeline, loop merge and loop flatten and the implementation of these code optimizations by the HLS tool that we use (Vivado HLS).

3.2.1.1 Loop Unrolling

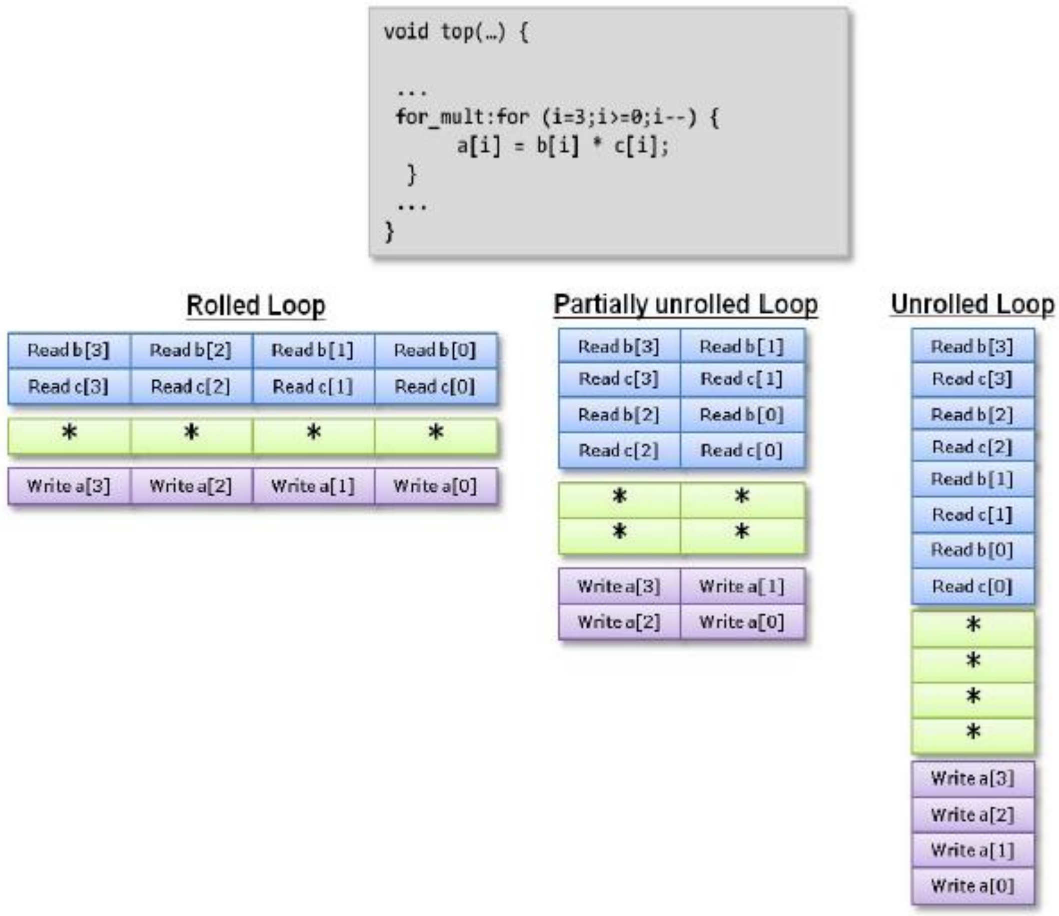
Loop unrolling is a compiler optimization, which is applicable to some kinds of loops and may reduce the loop maintenance instructions and the frequency of branches [11]. The main limitation of this optimization is that the number of iterations must be known before the execution of the loop optimization. By using loop unrolling, we may replicate the code inside the body of a loop a number of times. The loop unrolling factor is the number that indicates the replicates of the code. This software technique, loop unrolling, improves the execution time when the code has not any data dependencies and the execution of the multiple copies of the same code may be executed in parallel [11].

Vivado HLS keeps all the loops rolled by default. When a loop is rolled, all the operations into the body of the loop are executed using the same hardware resources for all the iterations. Vivado HLS is able to unroll, partially or completely, all the for-loops by using the UNROLL directive. Therefore, a loop may be rolled, partially unrolled or completely unrolled.

A rolled loop, as we stated before, is a loop where all the operations into the body of the loop are executed using the same hardware resources for all the iterations. So, each iteration of the loop starts when the previous is complete.

A completely unrolled loop is a loop where all its N iterations may be executed in parallel, using N identical groups of hardware resources, if there are no data dependencies.

A partially unrolled loop by a factor of x, is a loop where from all its N iterations, N/x iterations may be executed in parallel, using N/x identical groups of hardware resources, if there are no data dependencies [12].



```
void top(…) {

 ...
 for_mult:for (i=3;i>=0;i--) {
      a[i] = b[i] * c[i];
  }
 ...
}
```

Figure 11: *Execution of a rolled, a partially unrolled and a completely unrolled loop [12]*

From the above figure, taken from the Xilinx manual regarding Vivado HLS, we are able to analyze the application of the UNROLL directive on a for-loop. We may assume that the arrays a[i], b[i] and c[i] are mapped to block RAMs (BRAMs). We may also assume that each iteration of the rolled loop lasts n clock cycles.

Regarding the rolled loop, we notice that every iteration of the loop executes sequentially, so the for-loop completes after $4xn$ clock cycles. Considering the required hardware resources, we need 1 multiplier and 3 single-port BRAMs, as we perform only one read or write in each array.

Concerning the partially unrolled loop, we see that the for-loop unrolled by a factor of 2. So, we notice that every time 2 out of 4 iterations of the loop execute in parallel. In this way the for-loop completes after $2xn$ clock cycles. Considering the required hardware resources, now we need 2 multipliers and 3 dual-port BRAMs, as we perform 2 reads or writes in each array.

Regarding the completely unrolled loop, we notice that all the iterations are executed in parallel, so the for-loop completes after n clock cycles. Although the reduction of the execution time, we now have to implement 4 reads or writes to each array. In this case it is necessary to partition the arrays in order to access them by using dual-port BRAMs. Therefore, we are going to use 4 multipliers and 3 dual-port BRAMs.

In conclusion, we see that loop unrolling is able to reduce the execution time if there are no data dependencies. We also notice that the trade-off of this reduction is the increase of the required hardware resources. Because loop unroll results in the creation of more objects to schedule, sometimes it may lead to the increase of execution time [12].

3.2.1.2 Loop Pipelining

Loop pipelining is a compiler optimization, which may lead to the reduction of execution time of a loop without requiring extra hardware resources. More precisely, loop pipelining consists of the time scheduling of instructions during several loop iterations. Thus, this technique reduces stalls when the hardware resources are available. It may also exploits the instruction level parallelism on the superscalar and VLIW machines. As loop pipelining schedules the instructions inside the loop, it is clear that there will be an overlapping between the instructions and every loop iteration may starts before the previous has been completed [13].

In the following figure, we can see the three operations of a function. Without pipelining, the first function call completes after the sequential execution of all three operations. The second function call starts only when the first call is finished. With function pipeline, we notice that the operations of the two function calls are executed in parallel, when there are hardware resources available [12].

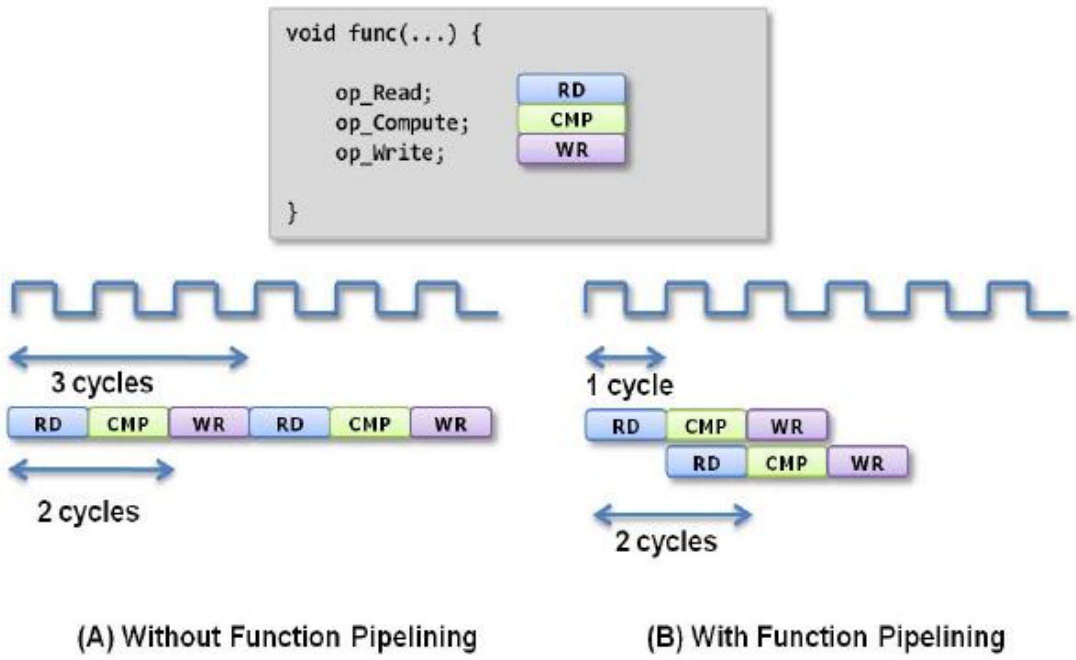


Figure 12: *Software Pipelining [12]*

Vivado HLS is able to pipeline both loops and functions. The loop pipelining is done using the PIPELINE directive, while the function pipelining is done using the DATAFLOW directive. In this diploma thesis, we studied the PIPELINE directive so as to pipeline only loops. We have to define the initiation interval, as the minimum number of clock cycles after which new inputs may be applied. Also we can define latency, as the number of cycles required in order to produce the requested output.
The pipeline technique has a great impact on the execution time of a program which includes one or more loops, if these loops can be pipelined.

We can examine an arithmetic example in order to notice the impact on execution time. Assuming that we have a loop which contains only a function which reads an integer. This loop is repeated for 50 times and the latency of the read function is 5 cycles.

Without loop pipelining, this loop will be complete after $latency\ x\ iterations = 5x50 = 250$ cycles, as the next iteration will start only when the previous is finished (part A, Figure 12).

If we pipeline this loop with an initiation interval $II = 1$, then the next iteration may be executed in parallel with the previous one (part B, Figure 12). The pipelined loop will be complete after $latency + (iterations - 1) * II = 5 + (50 - 1) * 1 = 54$ cycles.

We may also realize that we need to keep the initiation interval small (as close to 1 as we can) in order to reduce the execution time.

In case we have nested loops and we want to apply the PIPELINE directive to the top loop using Vivado HLS, all the sub-loops are unrolled recursively before pipelining. So, it worth noting that the implementation of PIPELINE directive (regarding Vivado HLS framework) contains also the UNROLL directive for all the sub-loops, if any [12]. Also, we have to mention that after the unrolling and the pipelining of the whole structure of the loop, Vivado HLS applies automatically the LOOP_FLATTEN directive in order to further reduce the latency. We analyze the LOOP_FLATTEN directive in Chapter 3.2.1.4.

3.2.1.3 Loop Merge

Loop merge is a code optimization which is able to merge consecutive loops, in order to "reduce overall latency, increase sharing and improve logic optimization" [12]. This software technique is able to reduce the execution time, as it is known that the cost of entering or leaving a loop is typically 1 clock cycle. So, by merging the loops, we decrease the number of loops, thus the number of transactions between loops and the corresponding clock cycles. Another benefit from loop merge is that, it enable us to implement the merged loops in parallel, if it is possible.

Although Vivado HLS provides us with the option of loop merge, with the LOOP_MERGE directive, there are some limitations concerning the use of loop merge.

There are two types of loop bounds, the variable and the constant bounds. If we want to merge two loops, they must have the same type of bounds, both either variable or constants. If they have variable bounds, they must have exactly the same number of iterations in order to be merged. If they have constant bounds and we want to merge them, the maximum constant bound will be the bound of the merged loop.

Another important thing, regarding merging, is the type of instructions contained into the loops. Merging may change the order of the instructions. So we can merge loops which contain instructions of type $a = b$, but we cannot merge loops which contain either FIFO reads or instructions of type $a = a + 1$, because these kinds of operations must be in sequence in order to provide the correct output.

It is worth noting that when we apply the LOOP_MERGE to a loop, the directive is applied on all the sub-loops, if any, but not to the loop itself. So, in this case we are going to merge all the sub-loops and not the initial loop with the other loops of the same hierarchy [12].

3.2.1.4 Loop Flatten

Loop flatten is a code optimization, which “allows nested loops to be collapsed into a single loop with improved latency” [12]. As we stated before regarding loop merge, loop flatten reduces the execution time by the decrease of the number of loops, since the cost of entering or leaving a loop is typically 1 clock cycle.

Vivado HLS provides us the directive LOOP_FLATTEN in order to flatten the suitable loops, if any. Regarding LOOP_FLATTEN directive, there are 3 types of loop structures, the perfect loop nest, the semi-perfect loop nest and the imperfect loop nest. All these types of nests contain a top-level loop and at least one loop, nested to the initial loop. There might be more sub-loops, nested to one another. Vivado HLS allows only perfect and semi-perfect loop nests to be flattened using this specific directive.

Perfect loop nests and semi-perfect loop nests are loop structures where only the innermost loop has body content and “there is no logic specified between the loop statements” [12]. The main difference between perfect loop nests and semi-perfect loop nests is that the loop bounds in a perfect loop nest structure are all constants, while in semi-perfect loop nest the outermost loop bound may be a variable.
Imperfect loop nests are loop structure which either the innermost loop has variable bounds or the loop content is not only into the innermost loop. These loop structures cannot be flattened by the LOOP_FLATTEN directive and we have to restructure the code in order to create a perfect or a semi-perfect loop nest.

It is worth noting that the LOOP_FLATTEN directive must be applied to the inner-most loop of a perfect or semi-perfect loop nest in order to flatten the whole structure to a single loop [12].

3.3 DSE of Dynamic Memory Management

In this chapter we discuss the significance of Dynamic Memory Management (DMM) regarding FPGA systems. We aim to apply the concept of DSE in the memory structures, created by the DMM mechanism.

As we stated in chapter 3.2, modern computer systems use multiple accelerators with custom architecture and heterogeneous SoC. Computer scientists state that this many-accelerator heterogeneous architecture may overcome the utilization/power wall, regarding the so-called "Breaking of the exascale barrier" challenge [14]. Heterogeneous FPGAs constitute a suitable platform for these MA (many-accelerator) architectures that we mentioned above. So, it is clear that HLS tools play a central role in the design of MA computing platforms, as these tools are highly correlated with FPGA boards.

It is worth noting that the memory organization is an important performance bottleneck because of the number and the diversity of the accelerators that form a MA architecture. Thus, there is a great need for a careful design of a memory subsystem, in order to achieve high utilization of the accelerator datapaths. It is known by experiment [15] that in modern FPGA boards, the starvation of the available on-chip memory cause limitations regarding the scalability of the number of accelerators. The memory starvation leads to the under-utilization of FPGA's resources. The problem regarding the on-chip memory is related with the fact that the FPGA design tools permit only the static memory allocation. Static memory allocation imposes the reservation of the maximum memory required of an application and keeps this memory reserved during the entire execution window. Although this type of memory allocation is compatible with systems, which include a limited number of accelerators, it cannot scale for MA systems.

We analyzed some experimental data [15] and we understood that the size of the on-chip memory of an FPGA board is important regarding the maximum number of accelerators that can be allocated. More precisely, BRAMs are the resource type of memory that starves faster than the others (DSPs, FFs, LUTs) and cause the bottleneck considering the number of accelerators.

In this diploma thesis, we use a DMM-HLS API for MA FPGA systems, based on Xilinx Vivado-HLS tool. By using this, we aim to eliminate the memory allocation based on the worst case scenario (static allocation). In fact we propose the dynamic memory allocation in order to "enable each accelerator to dynamically adapt its allocated memory according to the runtime memory requirements" [15]. So, the DMM-HLS framework that we propose (i) extends the HLS flow with DMM functionality (Figure 13) and (ii) provides a specific API, which includes function calls similar to glibc-dmm (malloc/free calls) in order to transform the statically allocated memory to dynamically allocated. More precisely, DMM-HLS API contains two main function calls regarding the malloc and free mechanisms:

- void* HlsMalloc(size_t size, uint heap_id)
- void HlsFree(void *ptr, uint heap_id)

respectively. Concerning the functions' arguments, size is the requested memory size to be allocated (in bytes), heap_id is the identification number of the memory heap that the allocation will take place and *ptr is the pointer that will be freed up [5] [15] [6].

The DMM-HLS framework is suitable for Single-Chip Many-Accelerator architectures and uses the back-end of Vivado HLS tool in order to be synthesized into RTL implementation. Before using the proposed DMM extension of Vivado HLS, we have to transform the original code from statically to dynamically allocated code regarding the data structures of the algorithm, which have global scope. We can also keep the statically allocation concerning accelerator's internal memory structures. Then we have to add DMM-HLS function calls into the transformed code and finally to synthesize the code into RTL code using Vivado HLS.

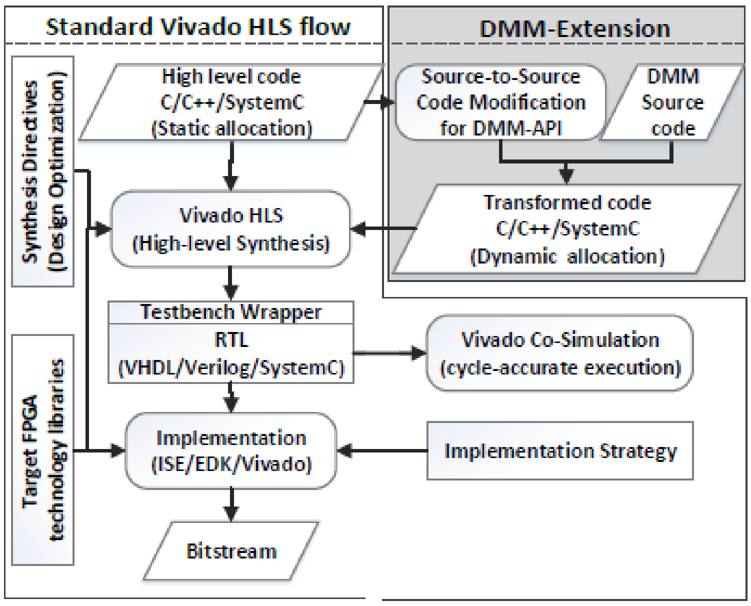

Standard Vivado HLS flow
DMM-Extension
Synthesis Directives (Design Optimization)
High level code C/C++/SystemC (Static allocation)
Source-to-Source Code Modification for DMM-API
DMM Source code
Transformed code C/C++/SystemC (Dynamic allocation)
Vivado HLS (High-level Synthesis)
Testbench Wrapper
RTL (VHDL/Verilog/SystemC)
Vivado Co-Simulation (cycle-accurate execution)
Target FPGA technology libraries
Implementation (ISE/EDK/Vivado)
Implementation Strategy
Bitstream

Figure 13: *DMM-HLS framework as DMM-Extension on the Standard Vivado HLS flow [5]*

The factor that increases the performance of the system regarding the DMM-HLS API is that we are able to "feed accelerators with data so that no processing stalling occurs due to memory read/write latency" [15]. All the transformations that are mentioned before may lead to the creation of one, unique memory module. This module will include all the BRAMs and as a result, all the allocation/deallocation requests will be related with this module. The consequence of using only one unique memory module is to have a bottleneck concerning the great amount of allocation/deallocation requests, which will execute in series, in case of systems containing hundreds of accelerators. Thus, we are going to implement parallelism regarding memory by grouping BRAM units into heaps, which are unique memory banks. Every heap will have a separate DM allocator. An HLS DM allocator is able to allocate any simple data type (integer, double, float etc.) on the same heap [5] [15] [6].

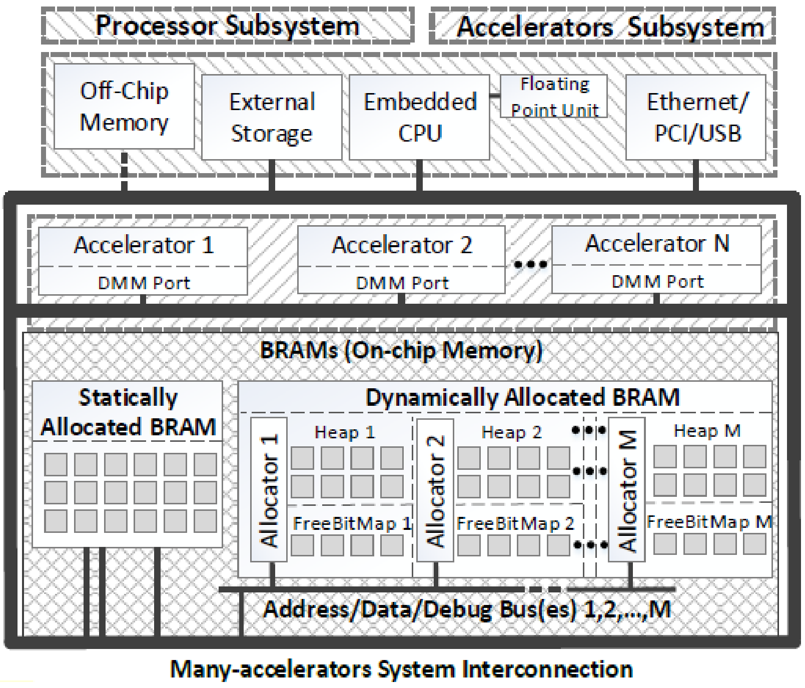


Figure 14: *Proposed memory structure using DMM-HLS API [15]*

Each DM allocator includes two components i) the freelist memory structure and ii) the fit allocation algorithm. The freelist memory structure tracks the reserved and the freed memory blocks and the fit allocation algorithm searches over the freelist structure and allocates the first memory block that fits to the requested memory size (first fit algorithm).

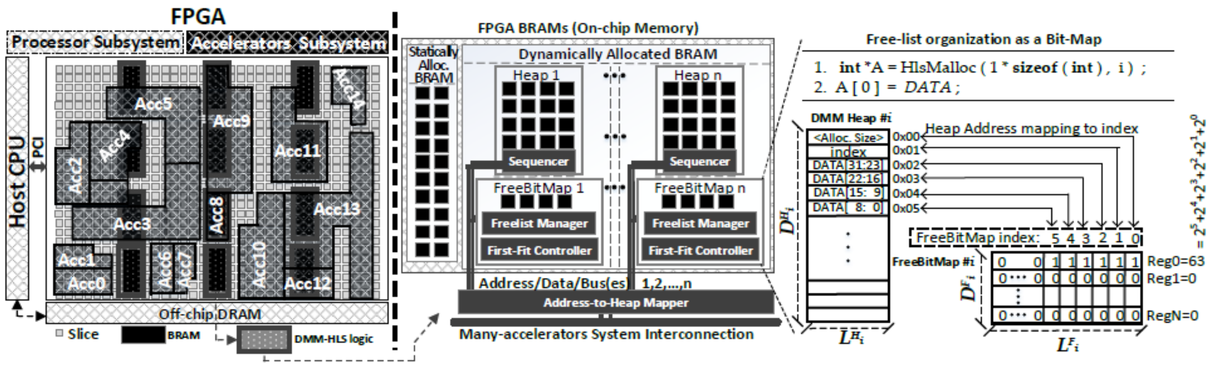

```
1. int *A = HlsMalloc ( 1 * sizeof ( int ) , i ) ;
2. A [ 0 ] = DATA ;
```

Figure 15: *Architectural design of DMM-HLS framework showing Freelist memory structure [6]*

As we mentioned before, BRAM modules are grouped into memory heaps. Into this DMM memory structure that we described, each memory heap can be bound for allocating data to more than 1 accelerators. The maximum number of heaps is correlated with the level of parallelism concerning the memory, because “less accelerators are sharing the same heap" [5]. As we increase the number of heaps, we see a reduction of the average latency of the system (Figures 16-17-18). The throughput gain that we see is due to accelerator parallelism in combination with the overlapped execution because of DMM heaps [5] [15] [6].

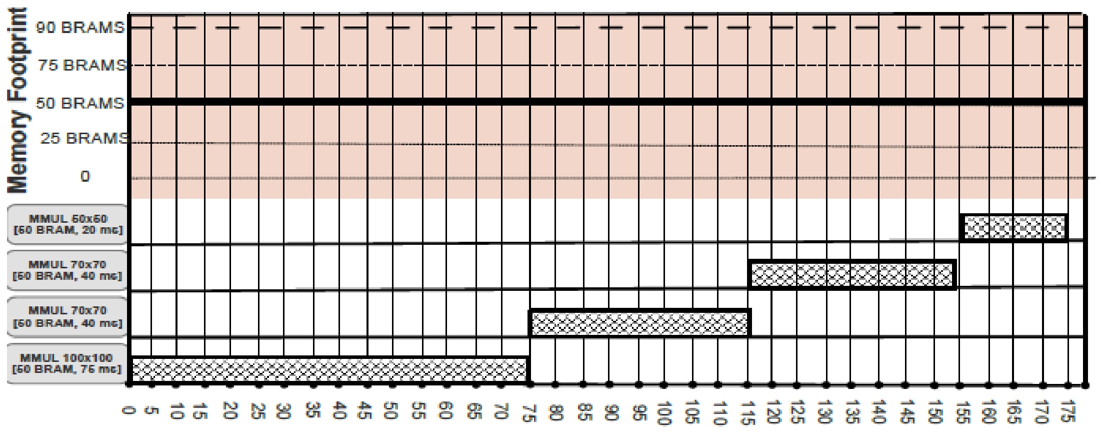


Figure 16: Memory footprint and scheduling of 4 MMUL accelerators using HLS with static allocation [6]

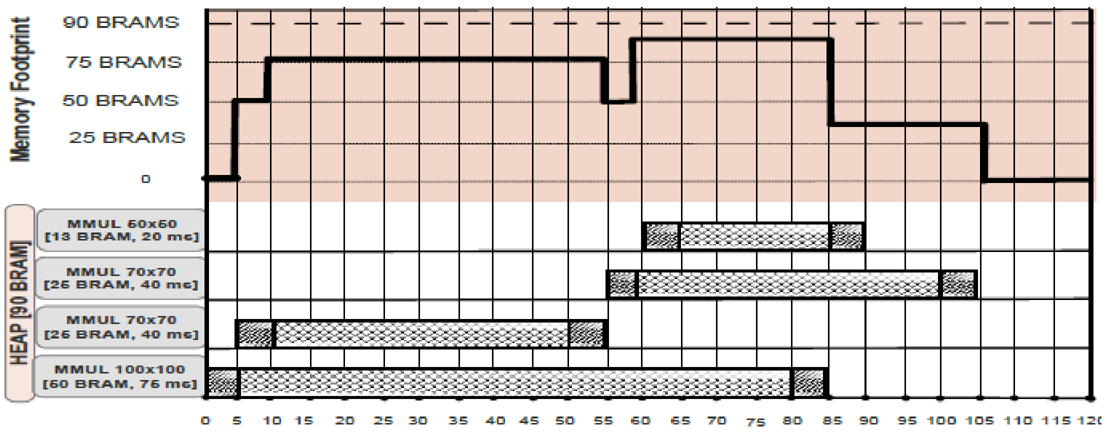


Figure 17: Memory footprint and scheduling of 4 MMUL accelerators using DMM-HLS with 2 memory heaps [6]

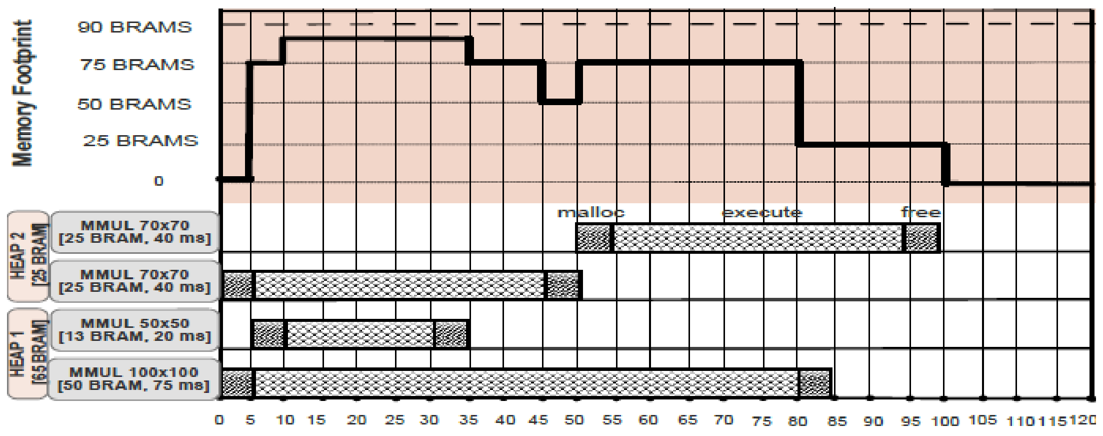


Figure 18: Memory footprint and scheduling of 4 MMUL accelerators using DMM-HLS with 2 memory heaps [6]

The tradeoff of this gain is the fact that the configuration and implementation of more memory heaps need extra hardware resources. Thus, we have to decrease the number of accelerators that we implement onto the FPGA. It is worth noting that even if every accelerator has its own heap, we will still face an overhead because of DMM internal operations (first-fit algorithm, freelist check).

Furthermore, there are three major runtime issues concerning the DMM in MA systems. At first, we have to analyze the memory fragmentation, divided into alignment and request fragmentation. As alignment fragmentation, we refer to the extra bytes needed in order to keep every allocation padded to the heap word length L_i^H. Request fragmentation is the fragmentation which occurs when we skip several freed memory blocks so as to find a continuous block equivalent to the size of the memory request. It is clear that the level of the request fragmentation is dependent on every accelerator's memory allocation pattern. It is worth noting that in homogeneous MA systems, request fragmentation is zero. Also regarding memory coherency, DMM-HLS has eliminated these types of memory problems as every accelerator has its own memory space, inaccessible to other accelerators. Finally it is possible to observe some memory access conflicts when a large set of accelerators is using the same heap [5] [15] [6].

For the DMM-HLS evaluation, we use some algorithms from the fields of artificial intelligence, scientific computing, enterprise computing etc. These algorithms may be found in Phoenix MapReduce framework for shared-memory systems [7] (Table 1).

Application Domain	**Kernel**	**Description**
Image Processing	Histogram	Determine frequency of RGB channels in image
Scientific Computing	Matrix Multiplication	Dense integer matrix multiplication.
Enterprise Computing	String Match	Search file with keys for an encrypted word
Artificial Intelligence	PCA	Principal components analysis on a matrix
Artificial Intelligence	K_{means}	Iterative clustering algorithm to classify n-D data points into groups

Table 1: Evaluation algorithms for DMM-HLS functionality [6]

3.3.1 Functionality of Function Inlining

We faced some limitations regarding the execution parallelism, when multiple accelerators request to allocate/free memory using HlsMalloc/HlsFree simultaneously, even if the memory blocks are located in different heaps. We introduced the function inlining feature in order to overcome this issue. The function inlining is applicable to Hls Malloc and Hls Free and is able to allow the simultaneous access on different heaps by many accelerators as well as the increase of the resource occupation. We evaluate this feature of DMM-HLS framework in Chapter 4 [5].

3.3.2 Functionality of Freelist Array

As we mentioned before, freelist memory structure or freelist array tracks the reserved and the freed memory blocks and fit allocation algorithm searches over the freelist structure allocating the first memory block that fits to the requested memory size (Figure 15). The speed of accessing a memory heap of a specific size depends on the freelist width. Increasing the width of freelist array, we may check a bigger number of data addresses in one iteration than having a smaller width. Therefore, the increase of freelist width results in the reduction of latency and simulation time of every application. In Chapter 4 we evaluate the efficient freelist array size for every application (8, 32, 64 bits).

Chapter 4

DSE of DMM in many-accelerator FPGAs

In this Chapter we evaluate two features of DMM HLS API that we mentioned before, the function inlining and the width of freelist array. The analysis of each feature includes research regarding metrics as Simulation time, BRAMs in use, DSPs in use, FFs in use and LUTs in use. The study of every metric consists of two parts. In the first part we keep the same number of accelerators (4 accelerators) and we increase the number of memory heaps in order to notice the impact of multiple heaps in the specified metric. In the second part of our analysis, we have one memory heap per accelerator, thus for 1 accelerators we use 1 heap, for 2 accelerators we use 2 heaps etc., as we saw that deploying 1 heap per accelerator results in the maximum parallelism simultaneously with the minimum overhead. In order to plot the curves for INLINE we assumed that the width of freelist array is 8 bits and for plotting freelist curves we assumed the INLINE factor equal to zero. It is worth noting that the simulation times are measured in nanoseconds (ns).

4.1 Inline

4.1.1 Simulation time

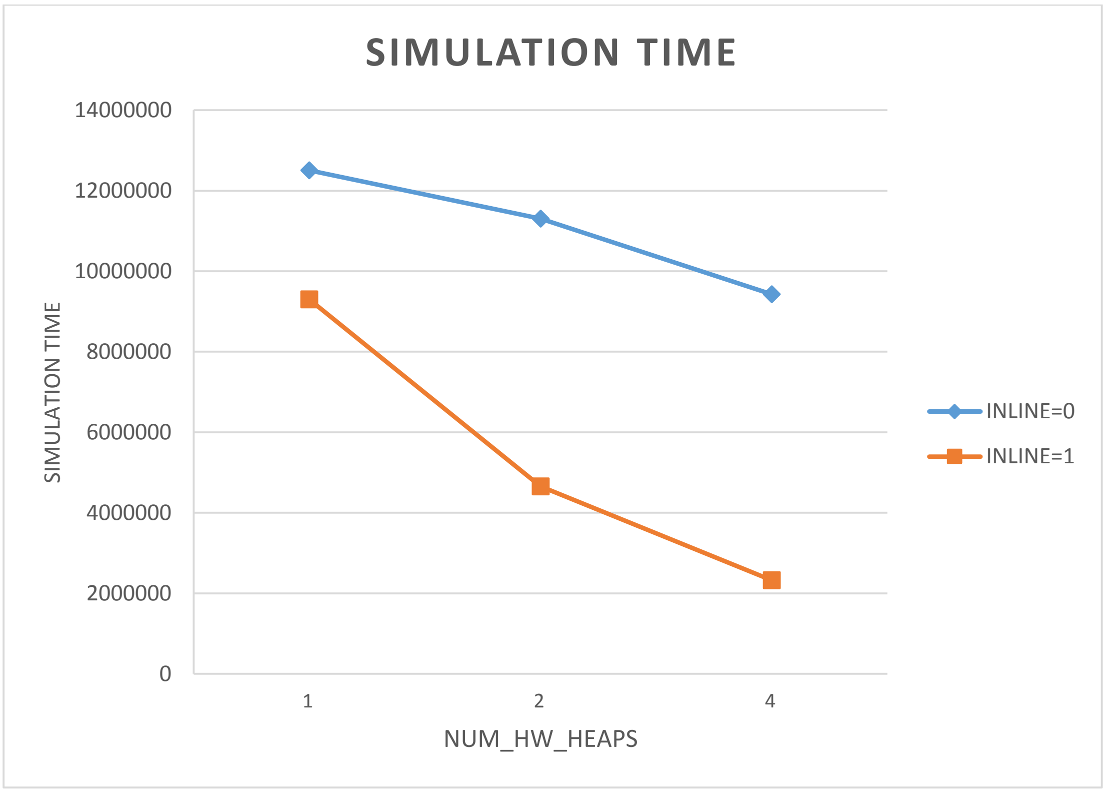


Figure 19: Simulation time regarding memory heaps for Histogram kernel

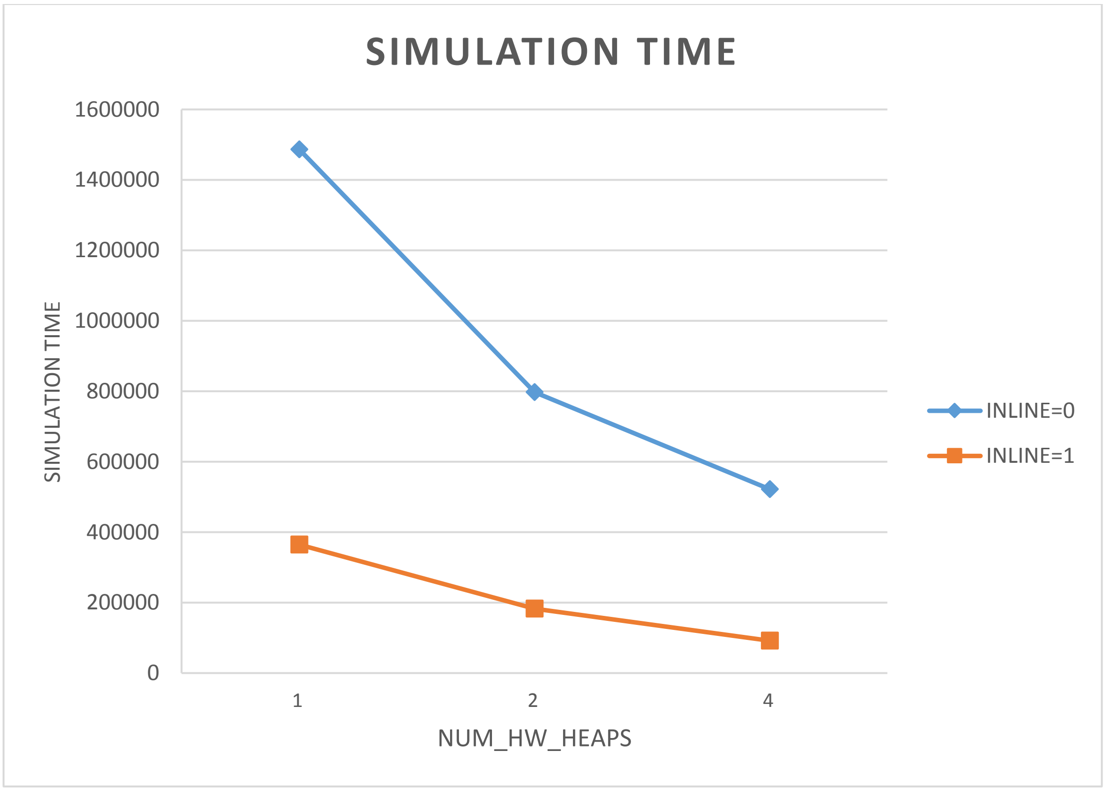


Figure 20: Simulation time regarding memory heaps for MMUL kernel

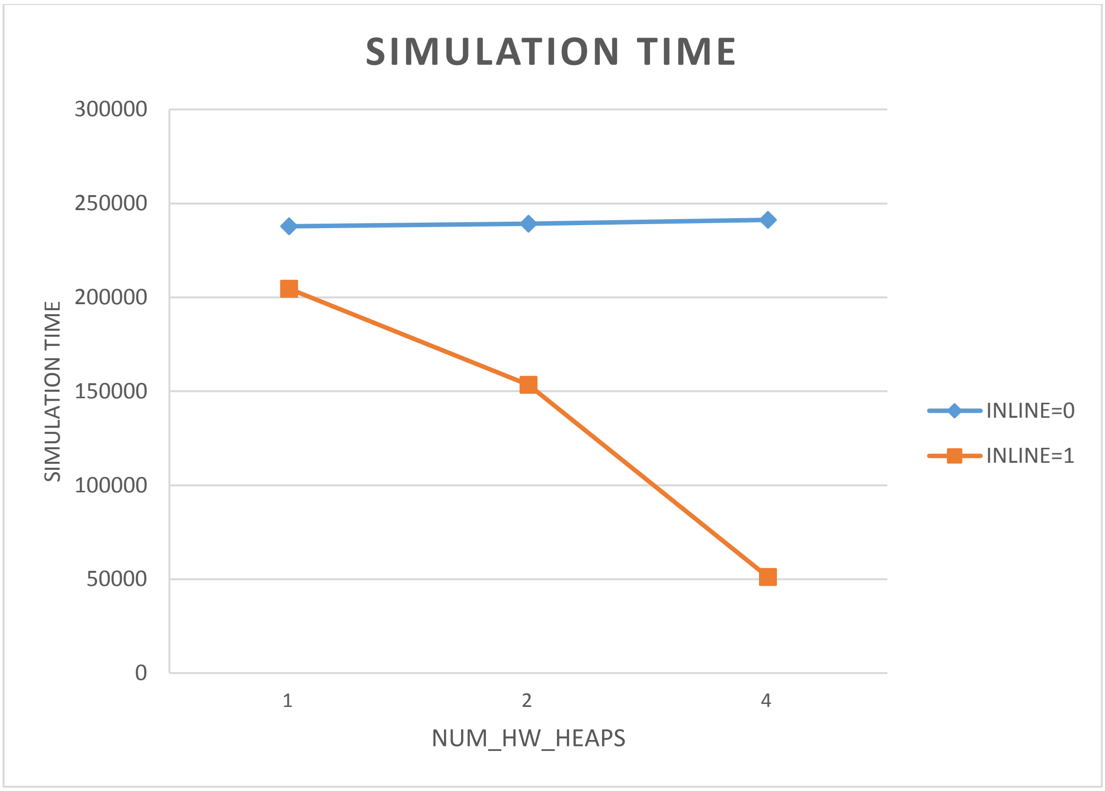


Figure 21: Simulation time regarding memory heaps for PCA kernel

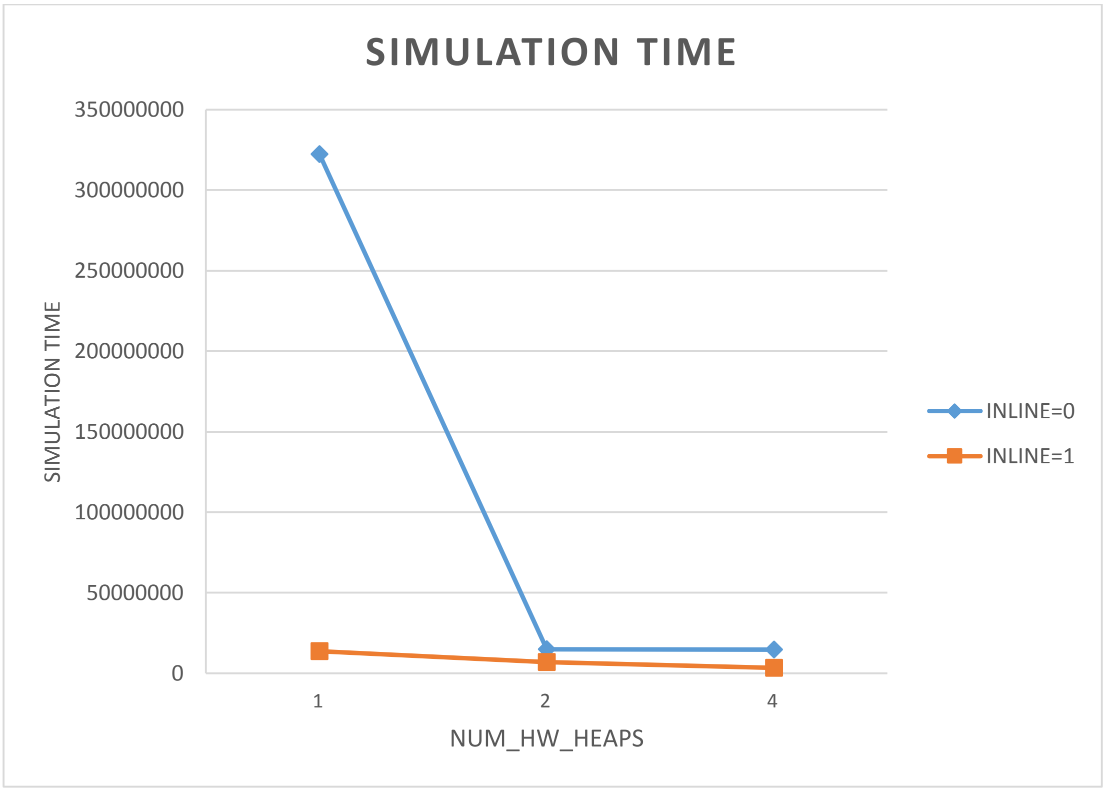


Figure 22: Simulation time regarding memory heaps for Kmeans kernel

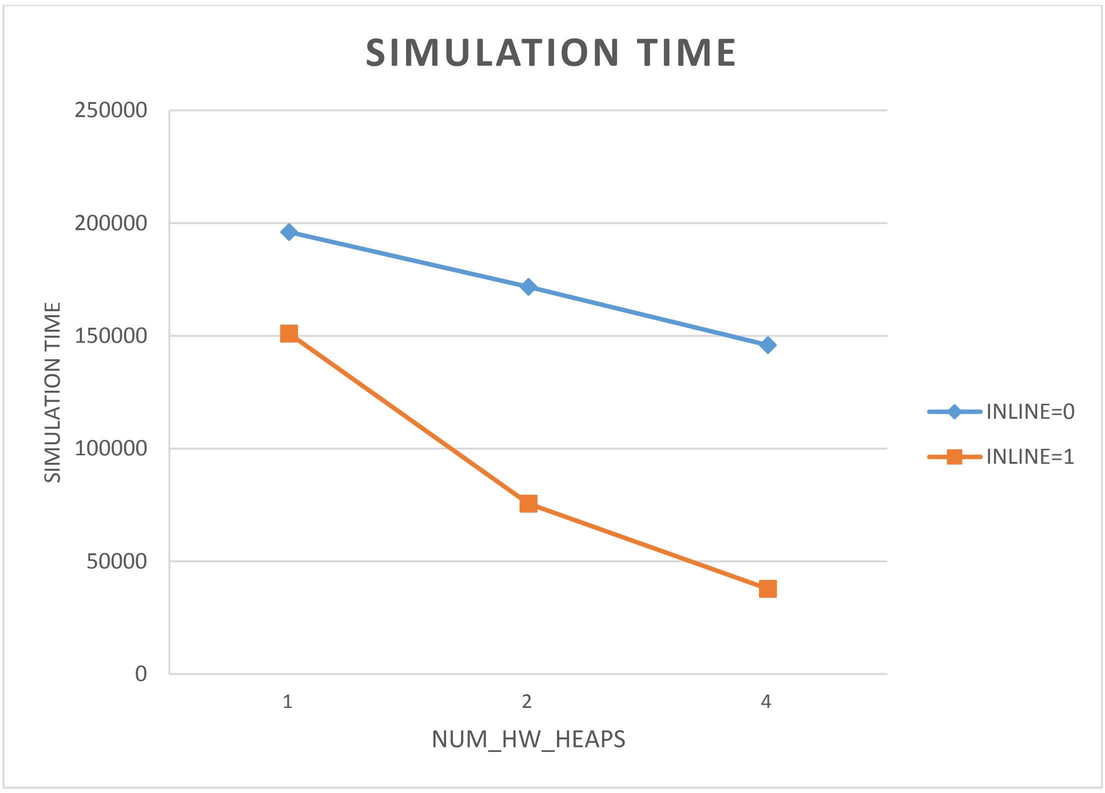


Figure 23: Simulation time regarding memory heaps for Strmatch kernel

We observe that for inline equal to zero, as we increase the number of memory heaps the simulation time decreases almost for every kernel, as expected. We also see a similar behavior for inline equal to one. The only difference is that the gradient of the curve regarding inline equal to one is bigger, as function inlining “unlocks” the parallelism of malloc and free mechanisms.

Regarding PCA kernel, we notice a completely different behavior when inline equals to zero. Although as we increase the width of freelist array, the simulation time decreases, we observe an increase of simulation time as we increase the number of heaps. This is happening because we used a small data set as input during execution of PCA kernel. As we add memory heaps to the system, there is sharing and reuse of hardware resources between the heap managers. The efficient exploitation of shared resources depends on the application (the data accesses over time and the memory footprint of the application) and on the scheduler of Vivado HLS. We have noticed in several studies that the scheduler of Vivado HLS fails to accomplish parallel execution of several computations and efficient exploitation regarding shared resources, especially for small data sets. Thus, for small data sets as inputs, the sharing of hardware resources in combination with the inability of efficient scheduling results in the small increase of simulation time.

Regarding Kmeans kernel, we observe a great simulation time when our system consists of only one memory heap.

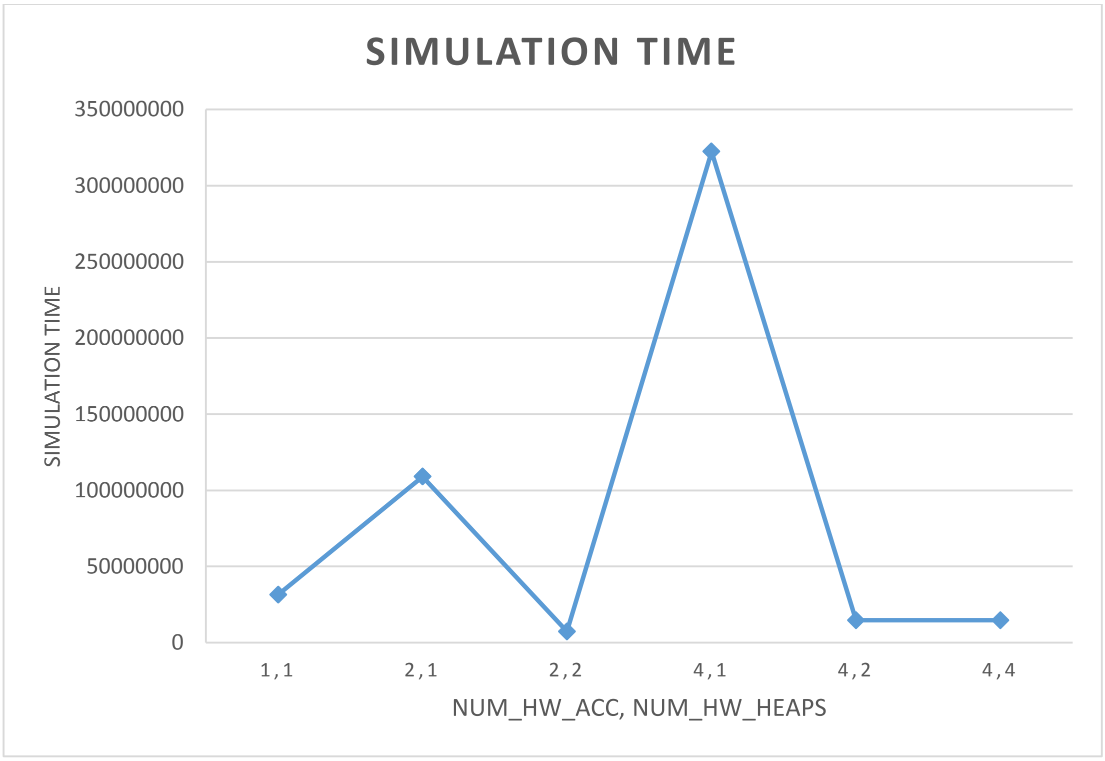


Figure 24: Simulation time of Kmeans kernel without inlining for all possible combinations of accelerators and memory heaps.

In figure 24, we notice some peaks of simulation time when our system consists of one heap (regardless the number of accelerators). We have concluded that it is the memory footprint of the accelerator's implementation and the data accesses over time that cause this behavior.

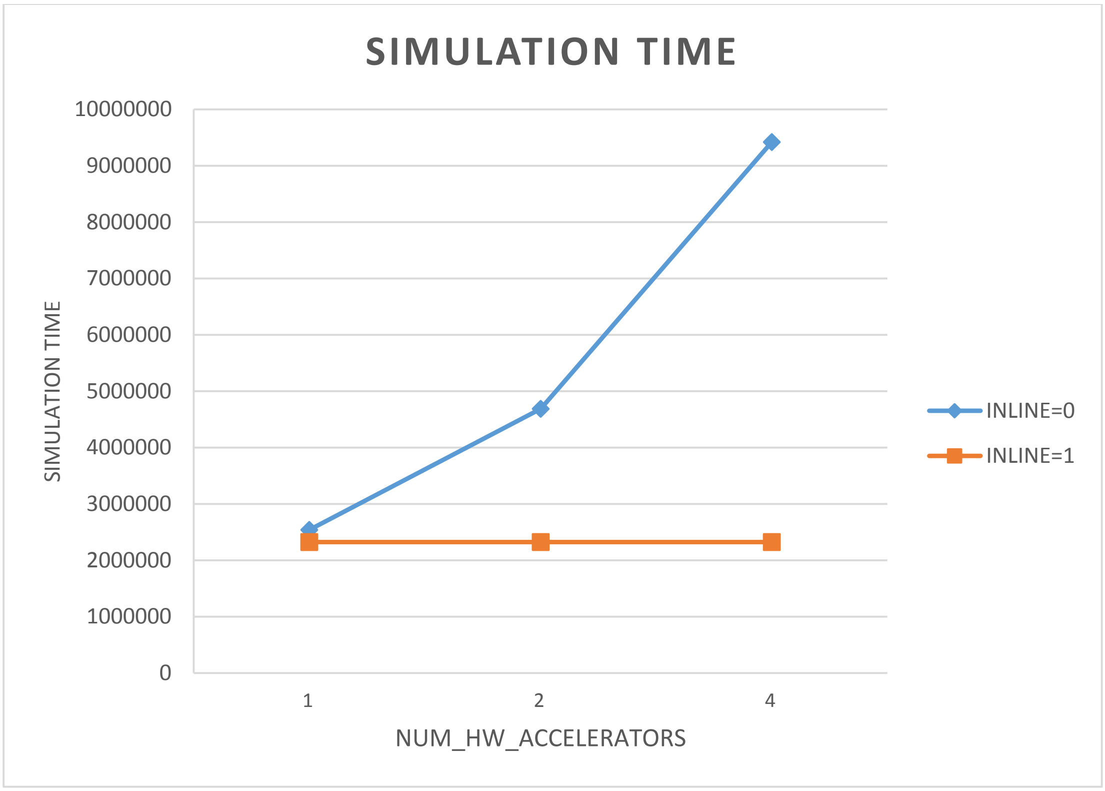


Figure 25: Simulation time regarding accelerators for Histogram kernel

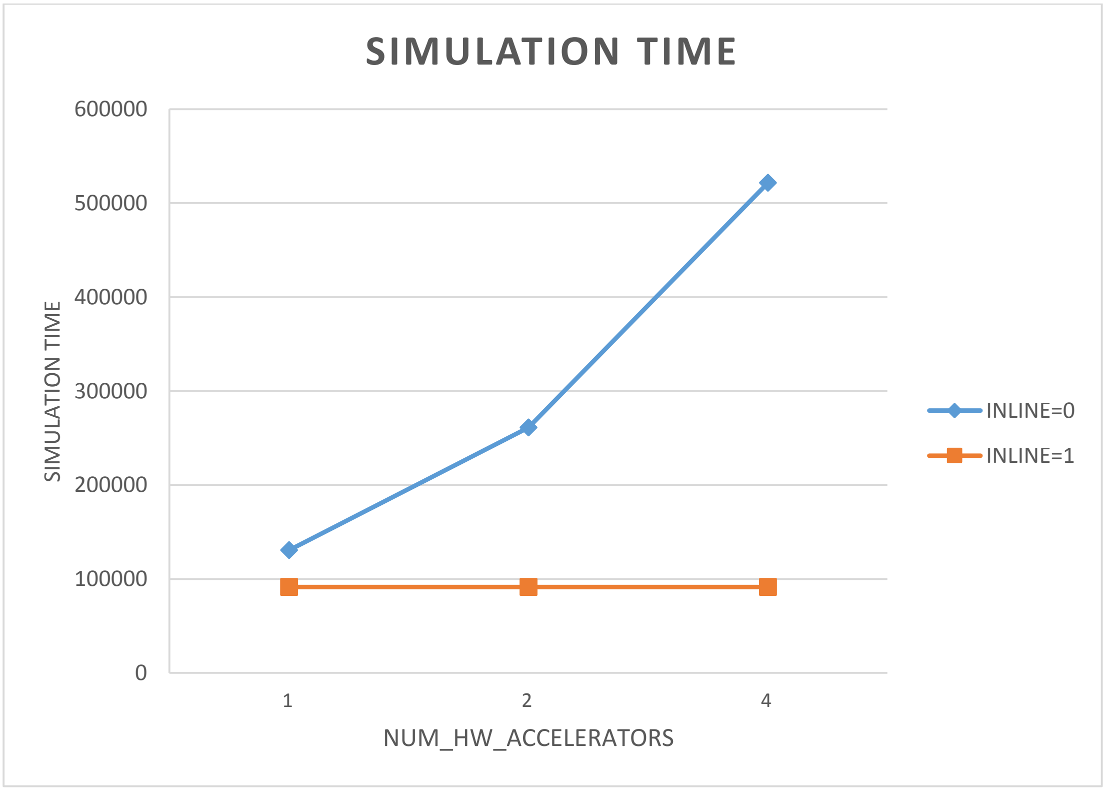


Figure 26: Simulation time regarding accelerators for MMUL kernel

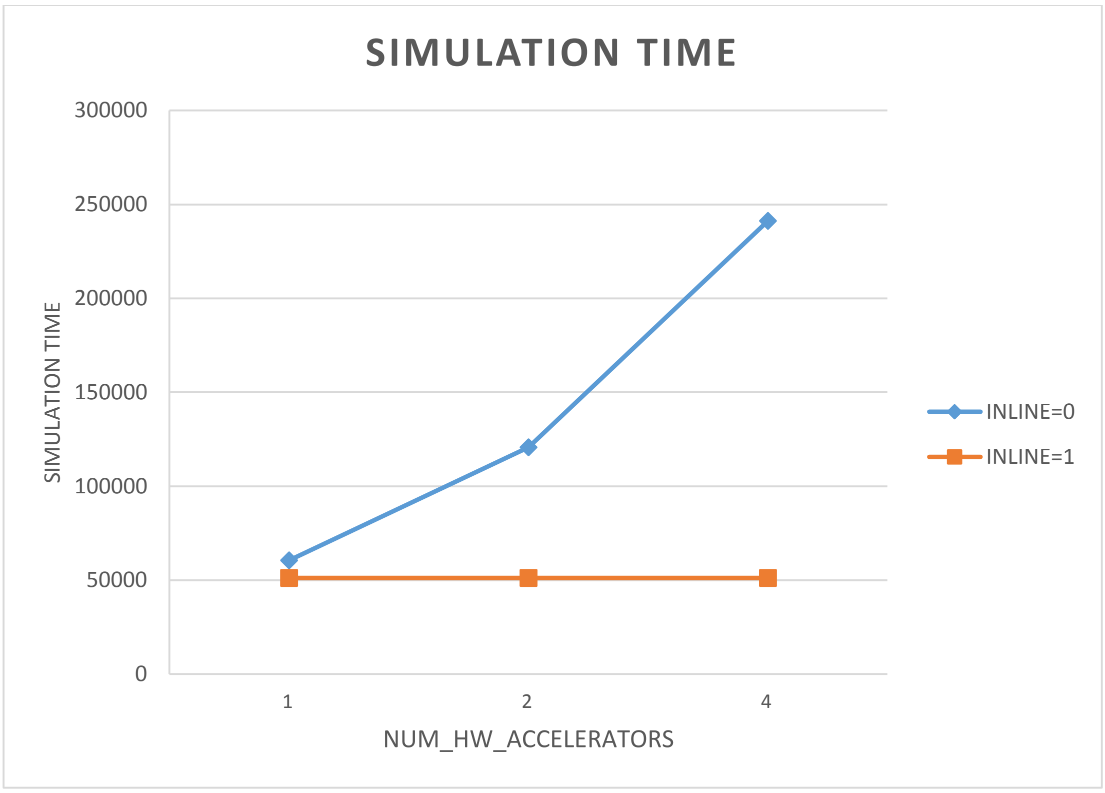


Figure 27: Simulation time regarding accelerators for PCA kernel

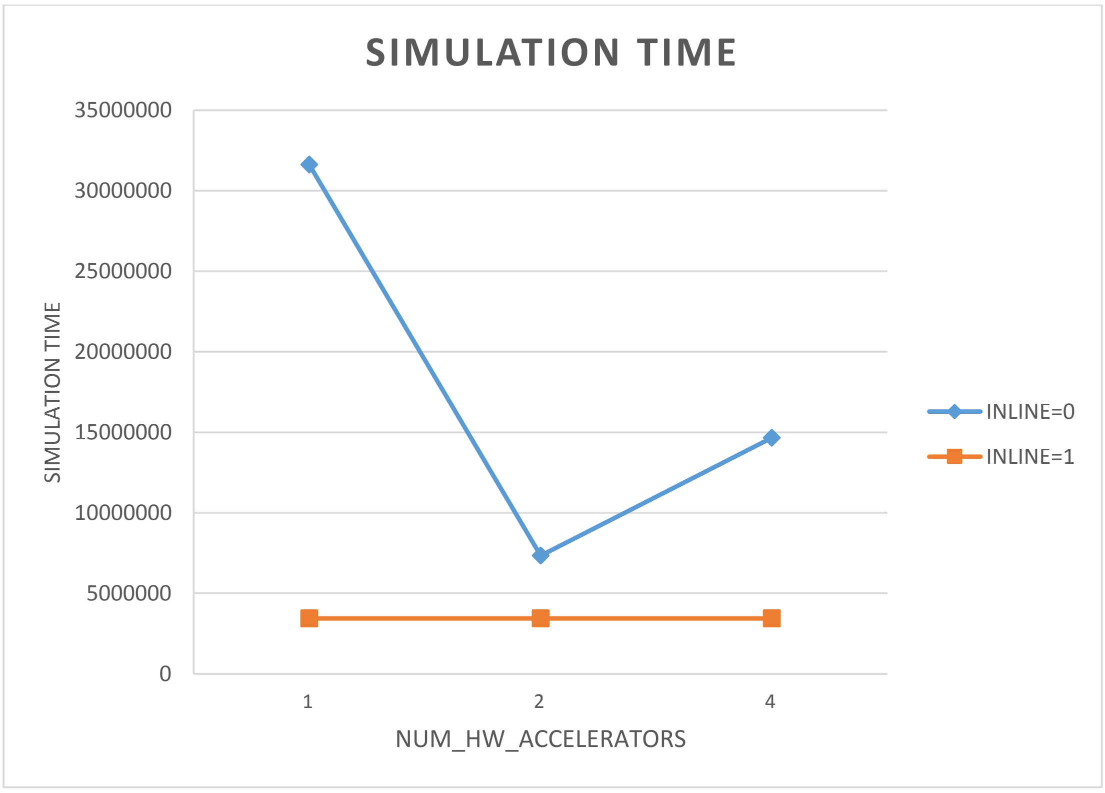


Figure 28: Simulation time regarding accelerators for Kmeans kernel

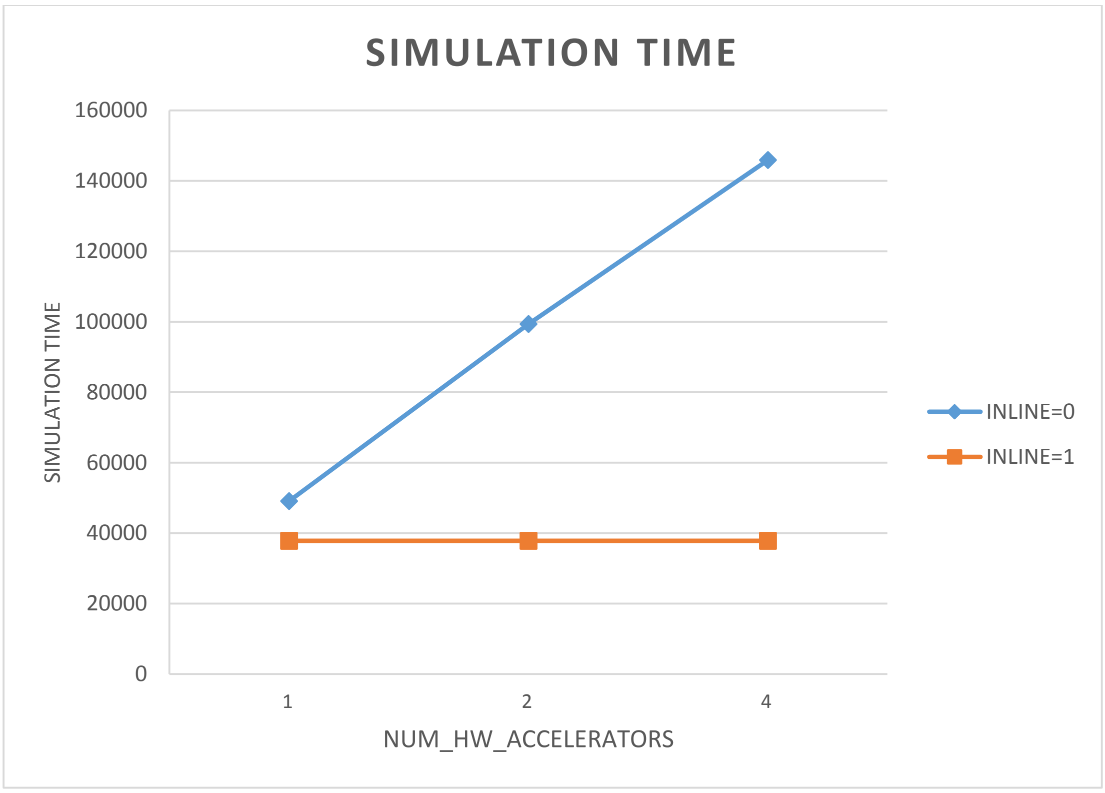


Figure 29: Simulation time regarding accelerators for Strmatch kernel

We observe that, for inline equal to zero, as we escalate our system by adding accelerators and memory heaps, simulation time raises. For inline equal to one, the simulation time remains the same as we keep the ratio between accelerators and memory heaps to 1-1 (one memory heap per accelerator).

Regarding Kmeans kernel, the line for inline equal to zero is different, compared with the other kernels. This is because Kmeans' implementation supports parallelism (even without function inlining), which is not feasible with only one memory heap.

4.1.2 BRAM

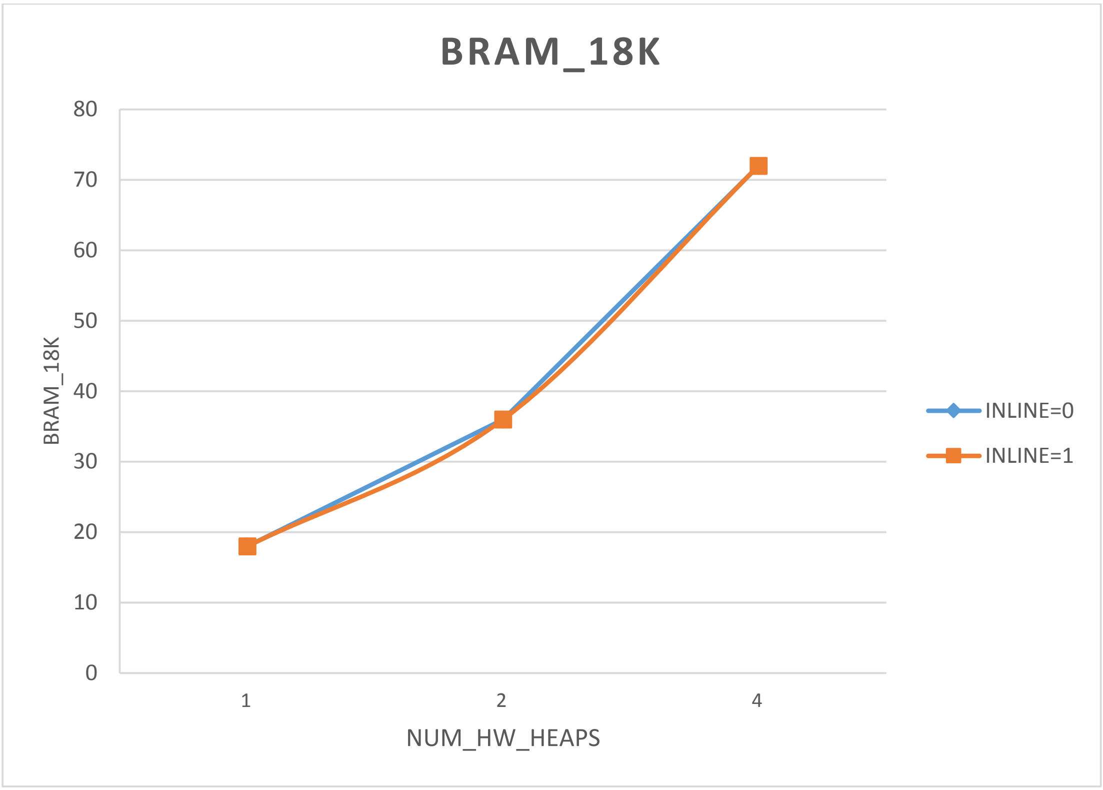


Figure 30: BRAM regarding memory heaps for Histogram kernel

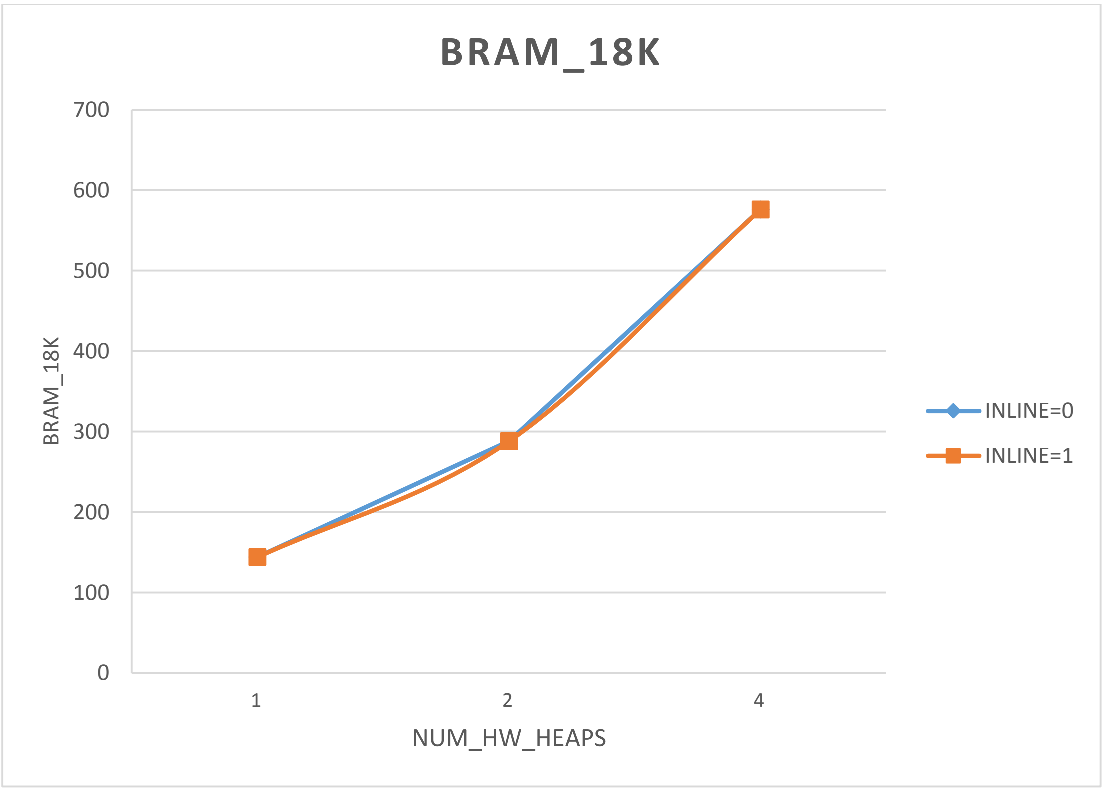


Figure 31: BRAM regarding memory heaps for MMUL kernel

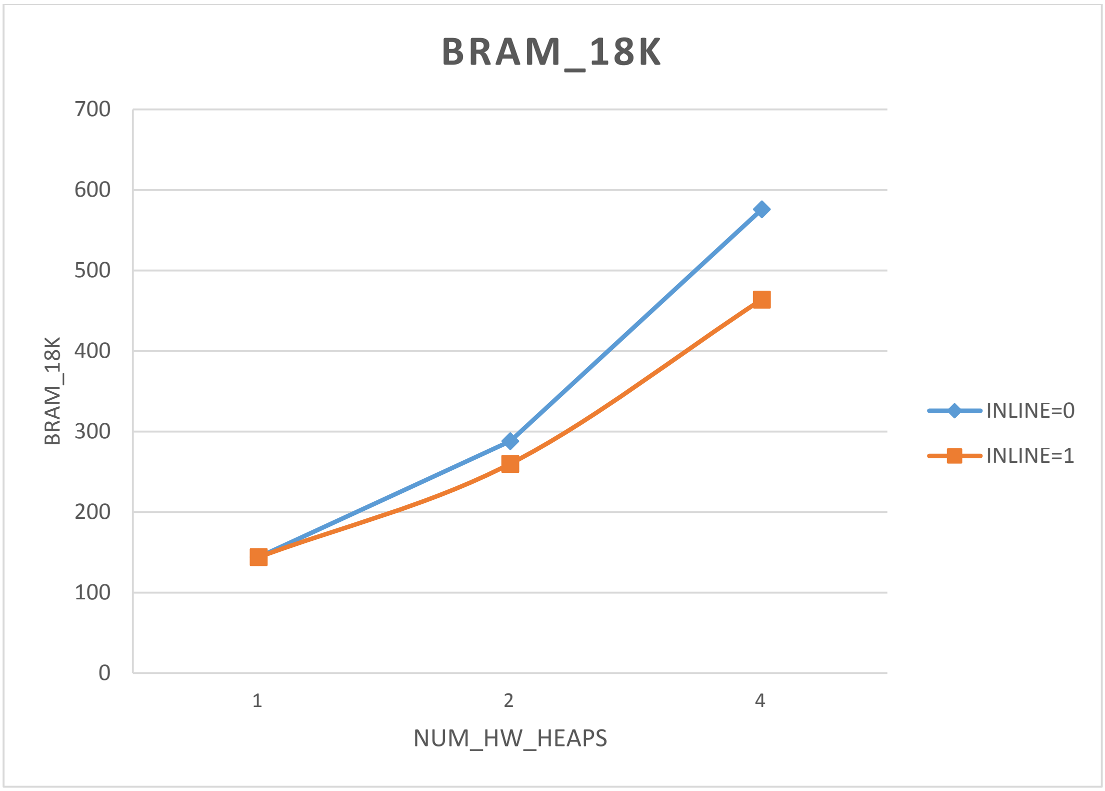


Figure 32: BRAM regarding memory heaps for PCA kernel

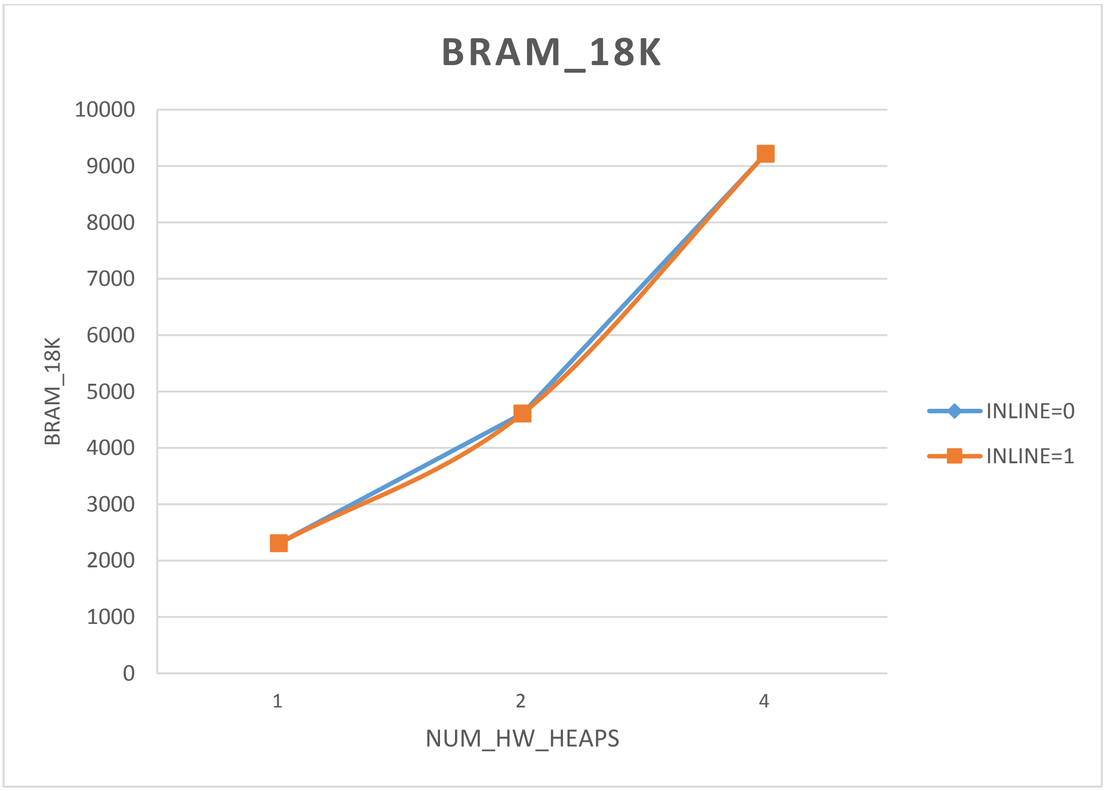


Figure 33: BRAM regarding memory heaps for Kmeans kernel

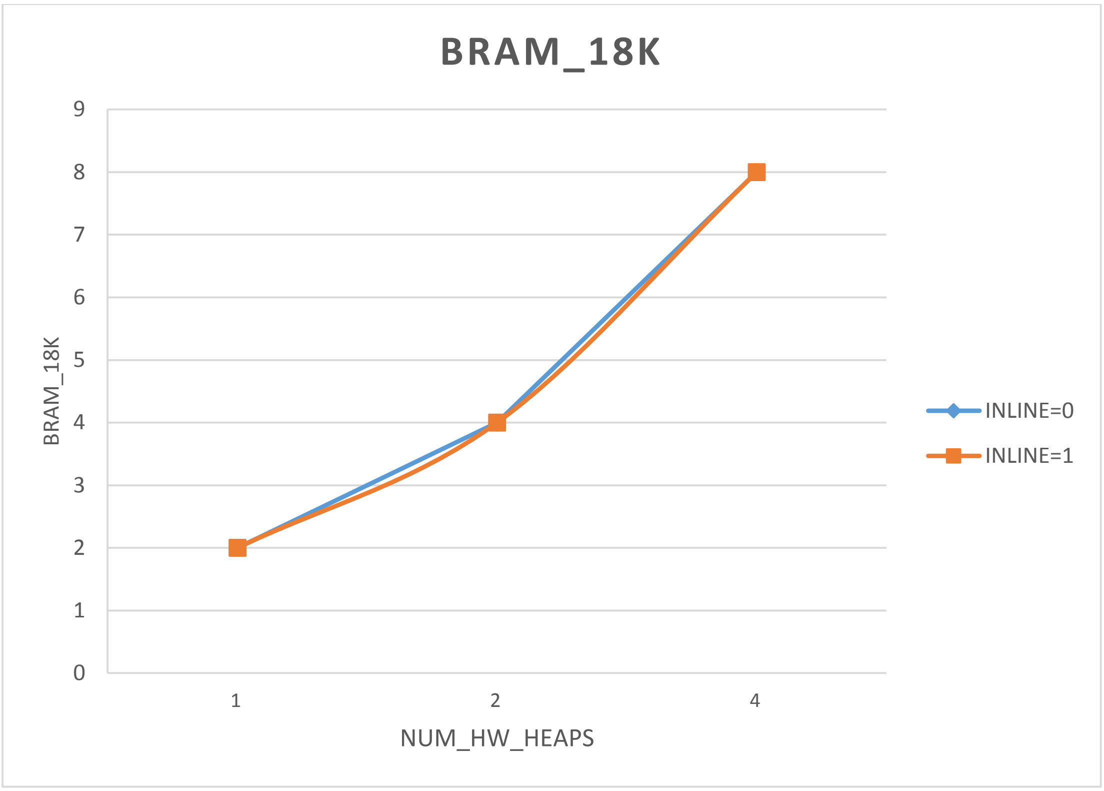


Figure 34: BRAM regarding memory heaps for Strmatch kernel

We see that as we increase the memory heaps, the BRAMs in use increase. This is expected as BRAMs are the building blocks of the memory heaps. We also notice that function inlining does not affect the number of BRAMs in use. That is because the inlining has impact only to hardware resources that perform calculations.

Regarding PCA, we notice a different ratio between function inlining and BRAMs in use, when we have four accelerators. This issue is strongly correlated with Vivado HLS tool. More precisely, when we want only one memory heap to be part of our system, Vivado HLS creates an RTL implementation only for this purpose. However, when we add two memory heaps to our system, Vivado HLS creates only one RTL implementation for memory allocation/deallocation purposes (heap implementation) and prefers to allocate the second heap in an RTL implementation of an accelerator (for optimization purposes). Finally, when we want to have four memory heaps, Vivado HLS prefers to allocate all the heaps into the RTL implementations of the four accelerators. All these optimized implementations occur for both inline equal to zero and inline equal to one. Nevertheless, when we enable the inline feature (inline equal to one), the memory heaps that have been implemented into the RTL circuits of the accelerators are affected. The result of inlining in those heaps is the reduction of the size of each memory address. Thus, even if we configured to have 32-bit addresses into the memory heap, we finally have 27-bit addresses ($32 - \log_2 32$). From this reduction of address size, we observe the decrease of BRAMs in use.

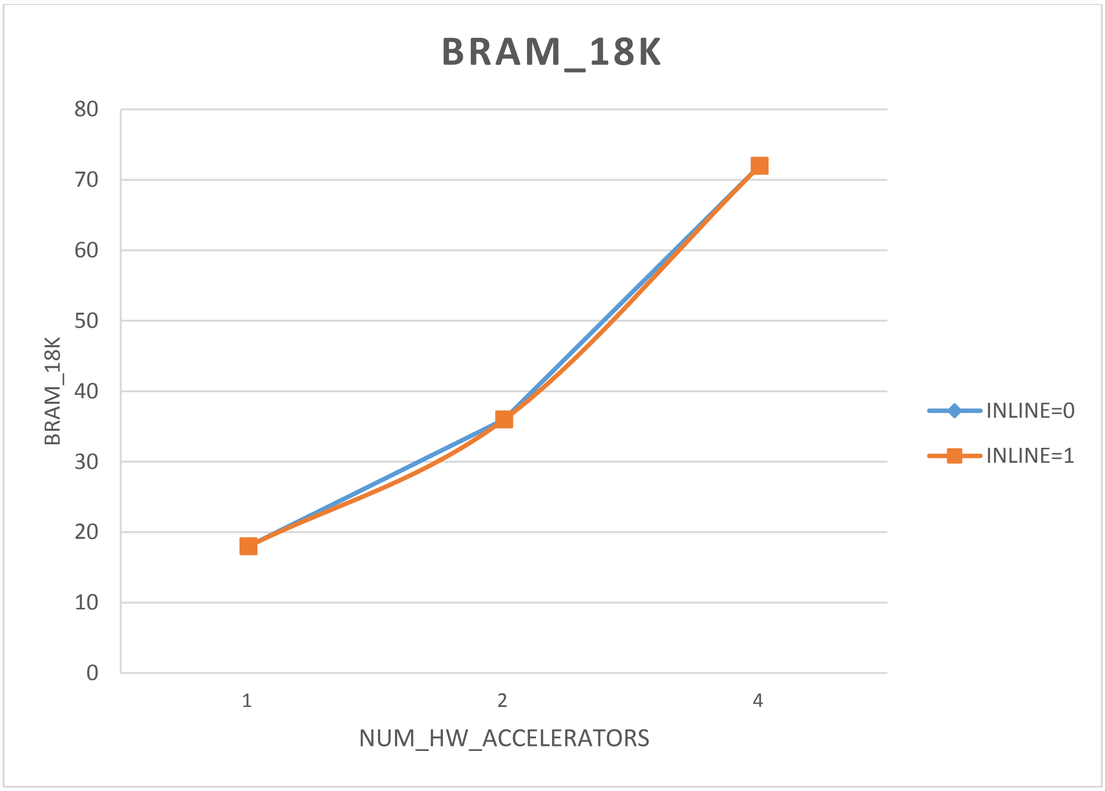


Figure 35: BRAM regarding accelerators for Histogram kernel

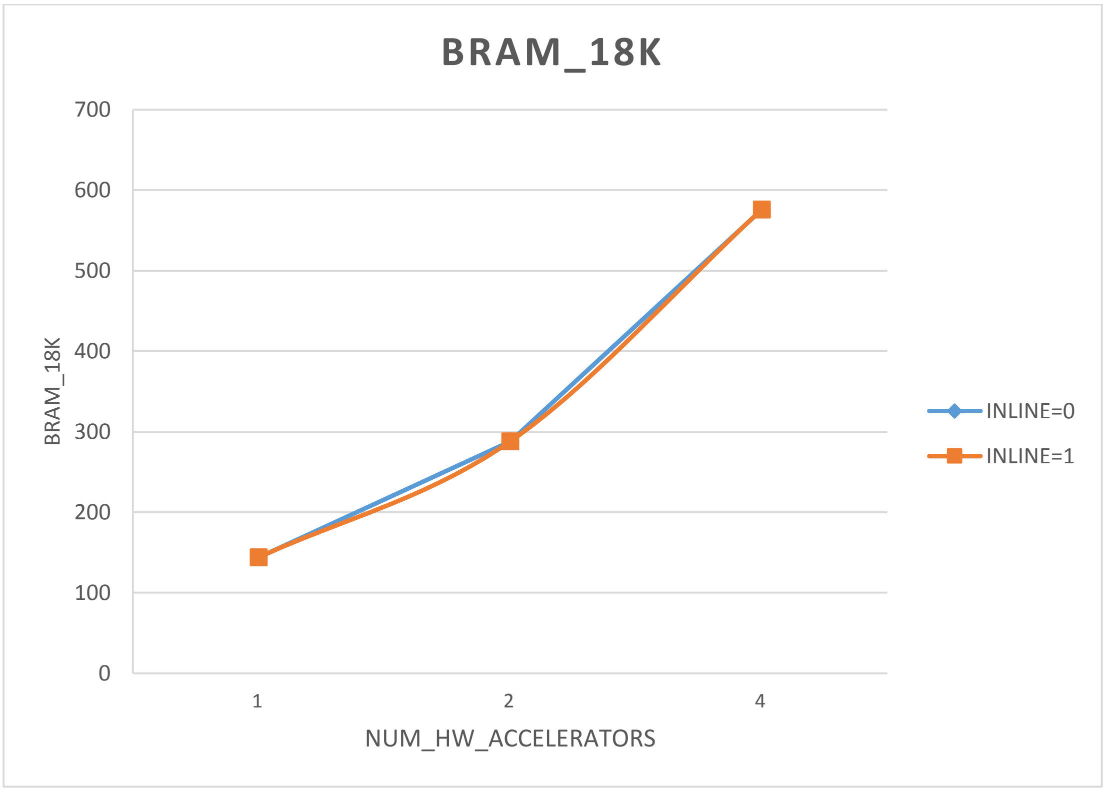


Figure 36: BRAM regarding accelerators for MMUL kernel

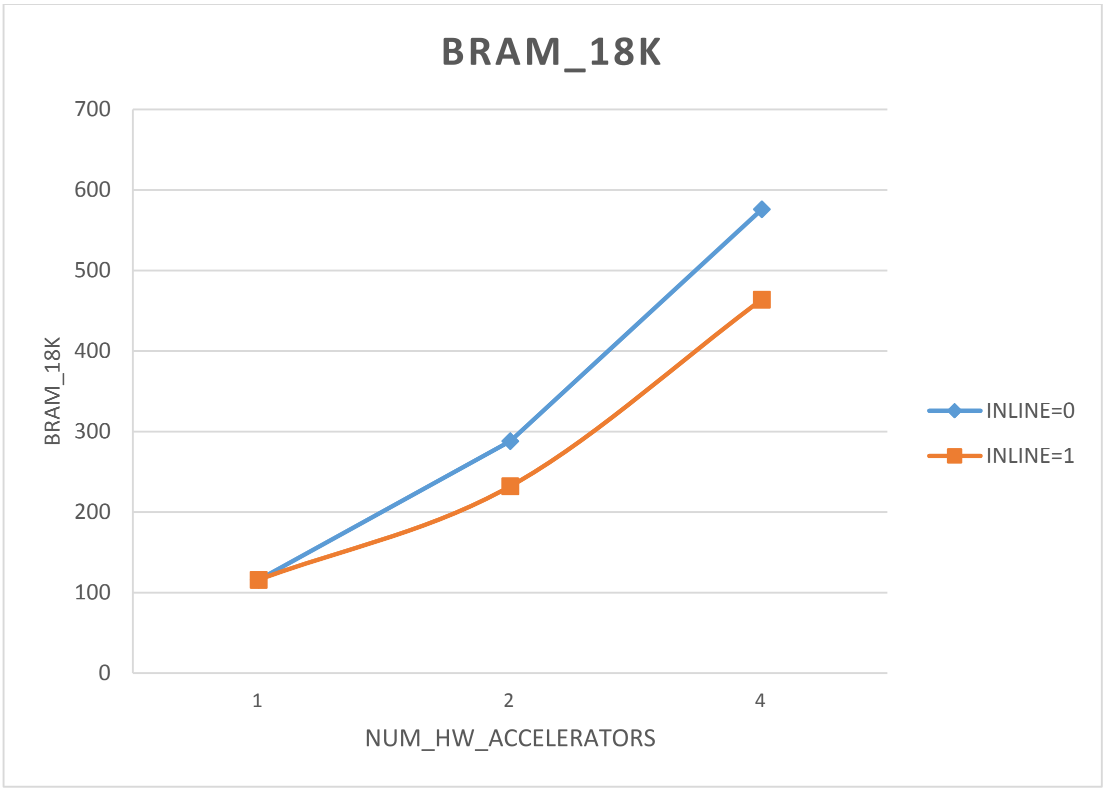


Figure 37: BRAM regarding accelerators for PCA kernel

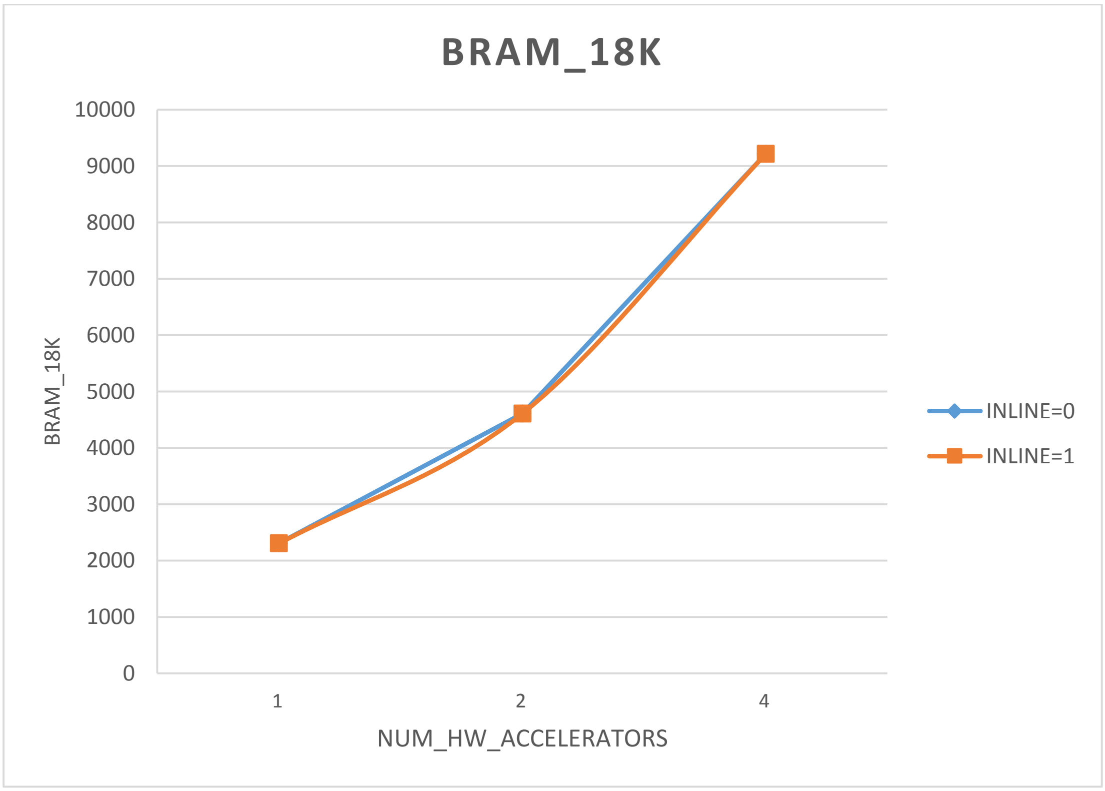


Figure 38: BRAM regarding accelerators for Kmeans kernel

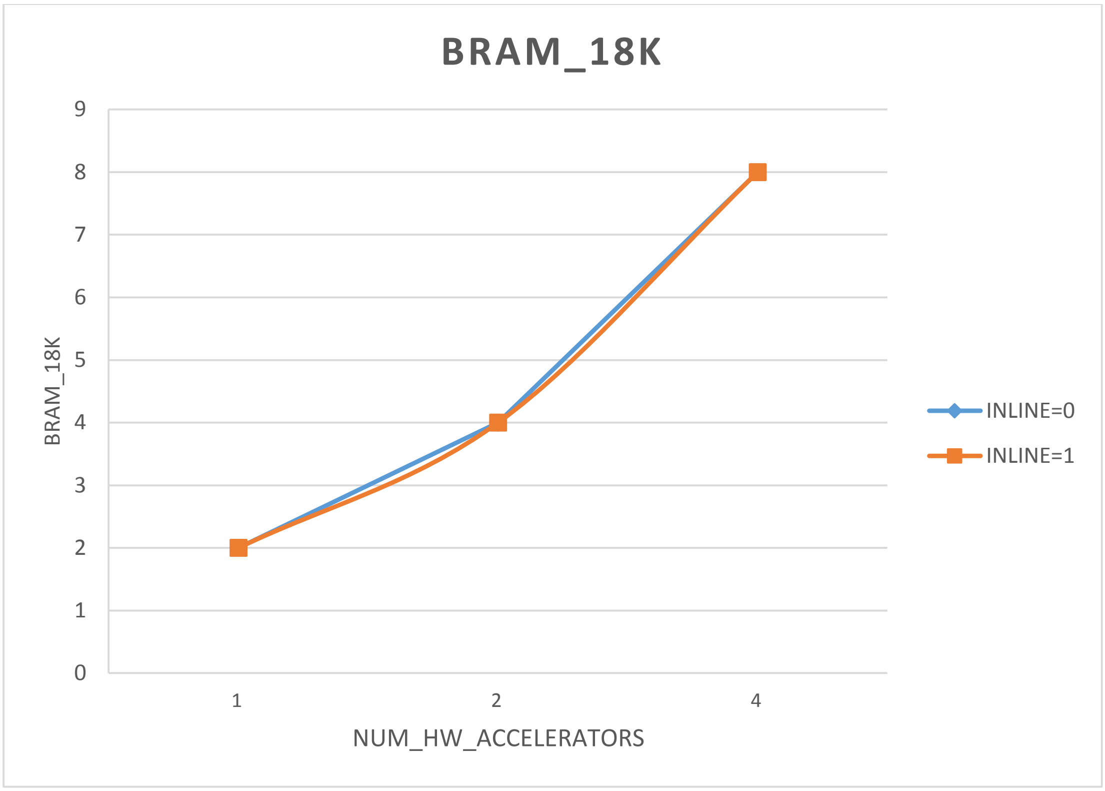


Figure 39: BRAM regarding accelerators for Strmatch kernel

As we escalate our system by adding accelerators and memory heaps, the number of BRAMs in use raises. We don't notice any difference when the inline factor is equal to one, as we also mentioned before.

Regarding PCA, we face the same issue as before (allocation of memory heaps and accelerators into the same RTL implementations). Thus, the application of function inlining results in the decrease of BRAMs in use.

4.1.3 DSP

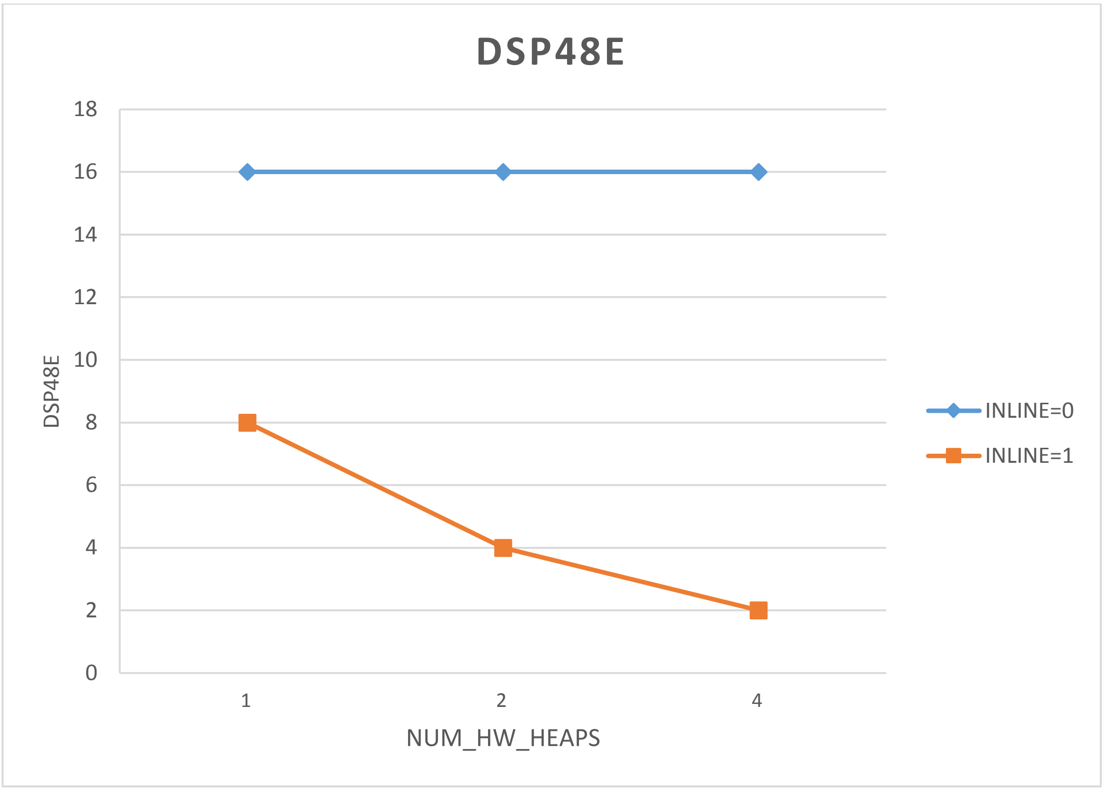


Figure 40: DSP regarding memory heaps for Histogram kernel

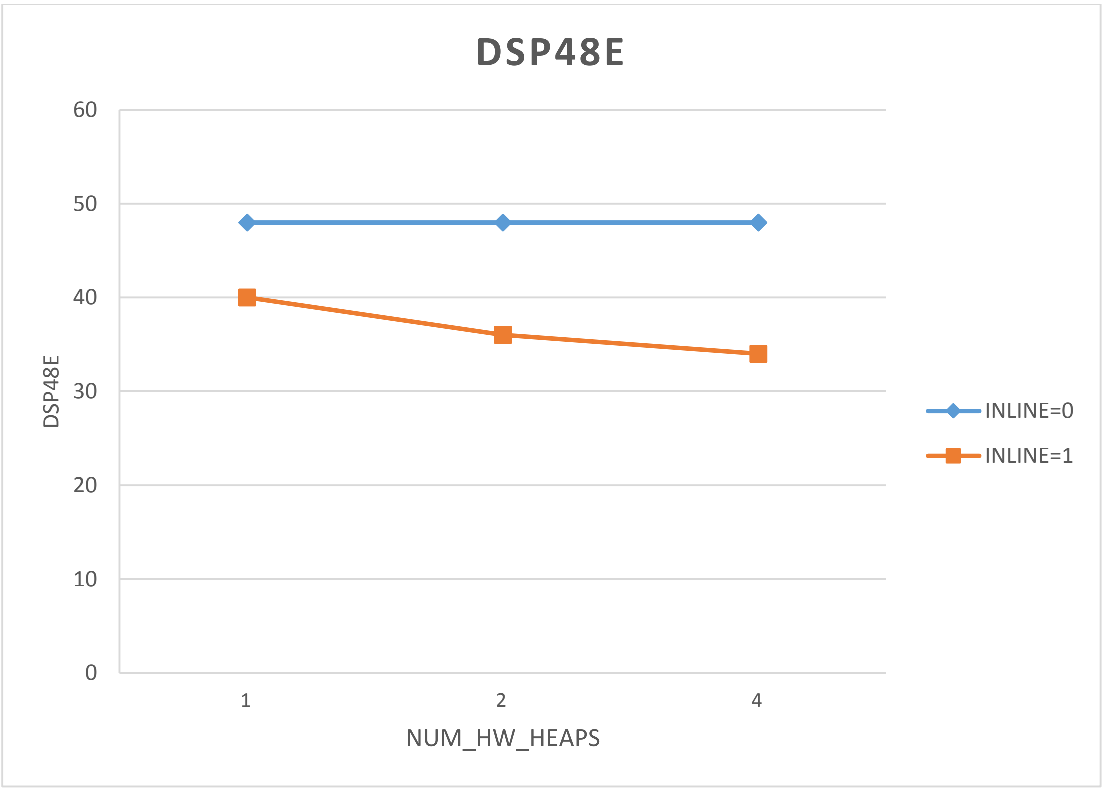


Figure 41: DSP regarding memory heaps for MMUL kernel

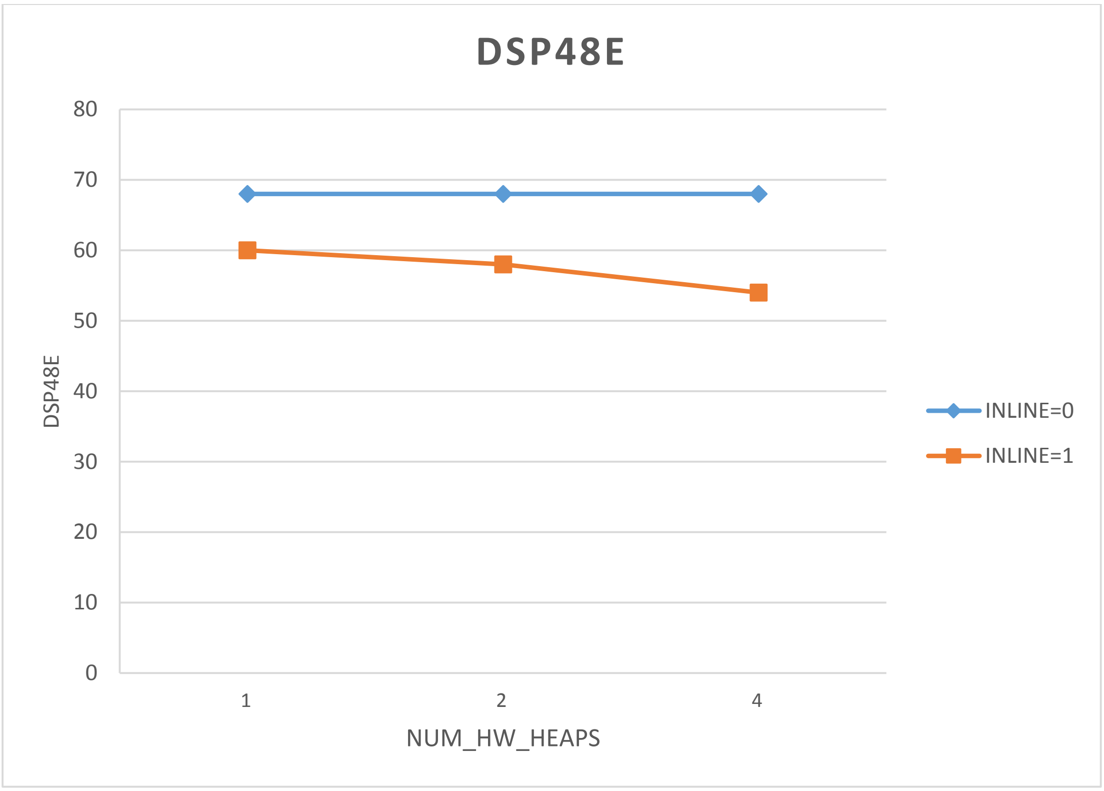


Figure 42: DSP regarding memory heaps for PCA kernel

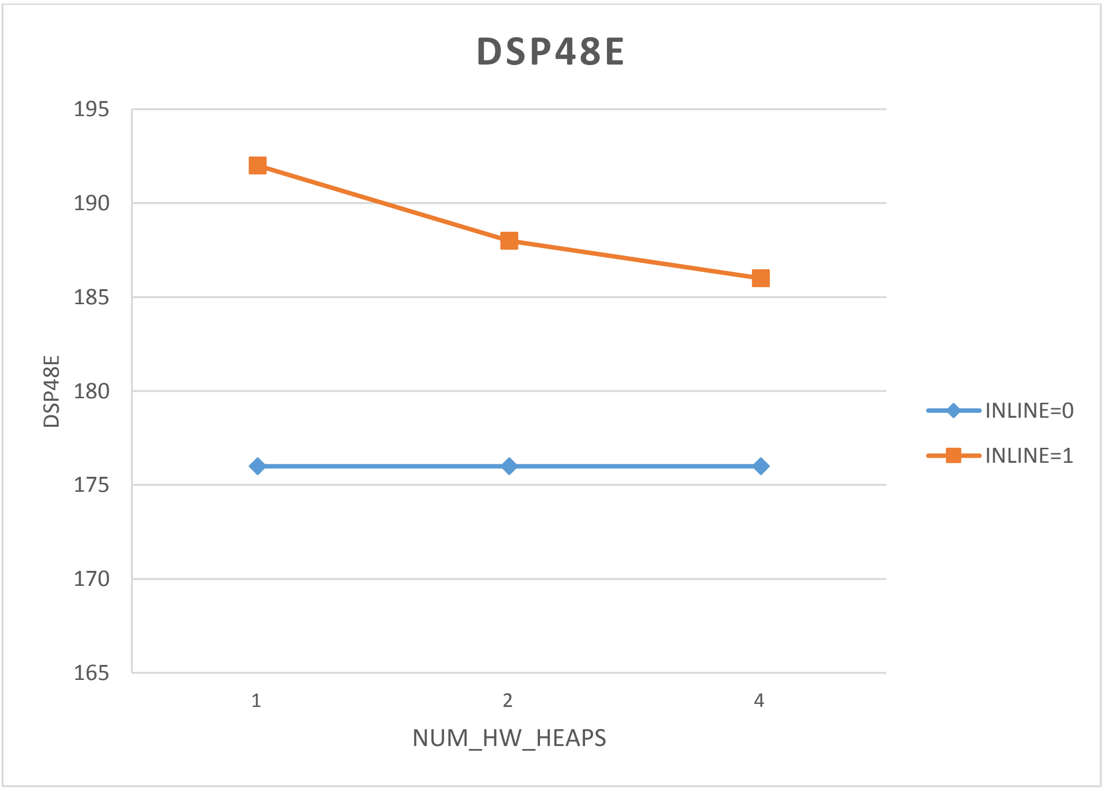


Figure 43: DSP regarding memory heaps for Kmeans kernel

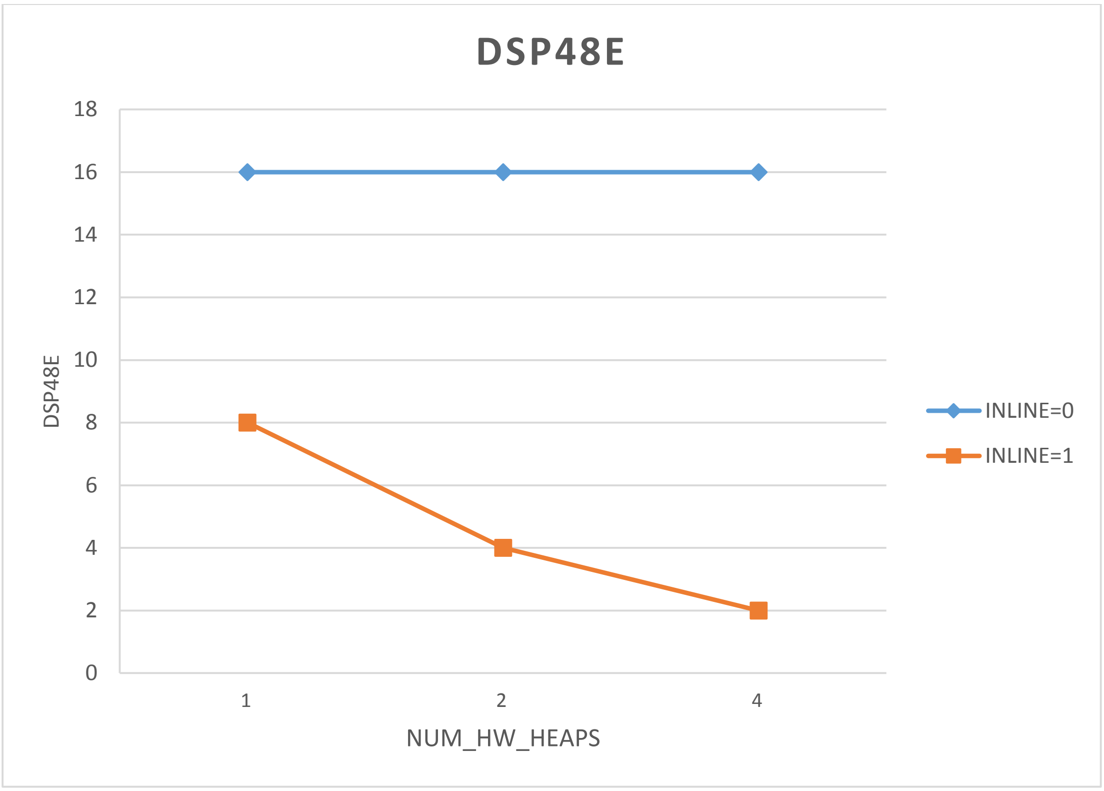


Figure 44: DSP regarding memory heaps for Strmatch kernel

We observe that when we enable function inlining, the number of DSPs in use reduces. This is happening because, when inline is equal to zero, Vivado HLS implements all the DMM functions into RTL circuits. The RTL implementations of DMM library use many DSPs in order to execute multiple computations regarding memory allocations/deallocations. Besides DMM functions, the accelerators of each kernel also utilize DSP units. When we inline all DMM functions, DSP units are used only for the implementation of the accelerators. Thus, we notice the reduction of DSPs in use. Furthermore, as we add more memory heaps to our system, the DSPs in use decrease more. This is reasonable, as the parallelism of memory requests result in the sharing and reuse of RTL circuits and so, we need to synthesize less circuits (for the functionality of accelerators).

Regarding Kmeans kernel, we see that the activation of inline results in the raise of DSPs in use. This is a different behavior than the rest of the kernels and it is related to Vivado HLS tool. More precisely, without inlining, Vivado HLS synthesizes the DMM functionality to RTL circuits. Vivado also schedules all the tasks regarding Kmeans kernel and manages to reuse the DMM RTL implementation for the accelerators' functionality. When inline is enabled, there is no reduction in DSPs, as the RTL implementations used by DMM library are still synthesized due to the accelerators' needs. The increase of DSPs in use is because Vivado HLS finally creates some extra circuits in order to exploit the advantages of parallelism and achieve a smaller simulation time.

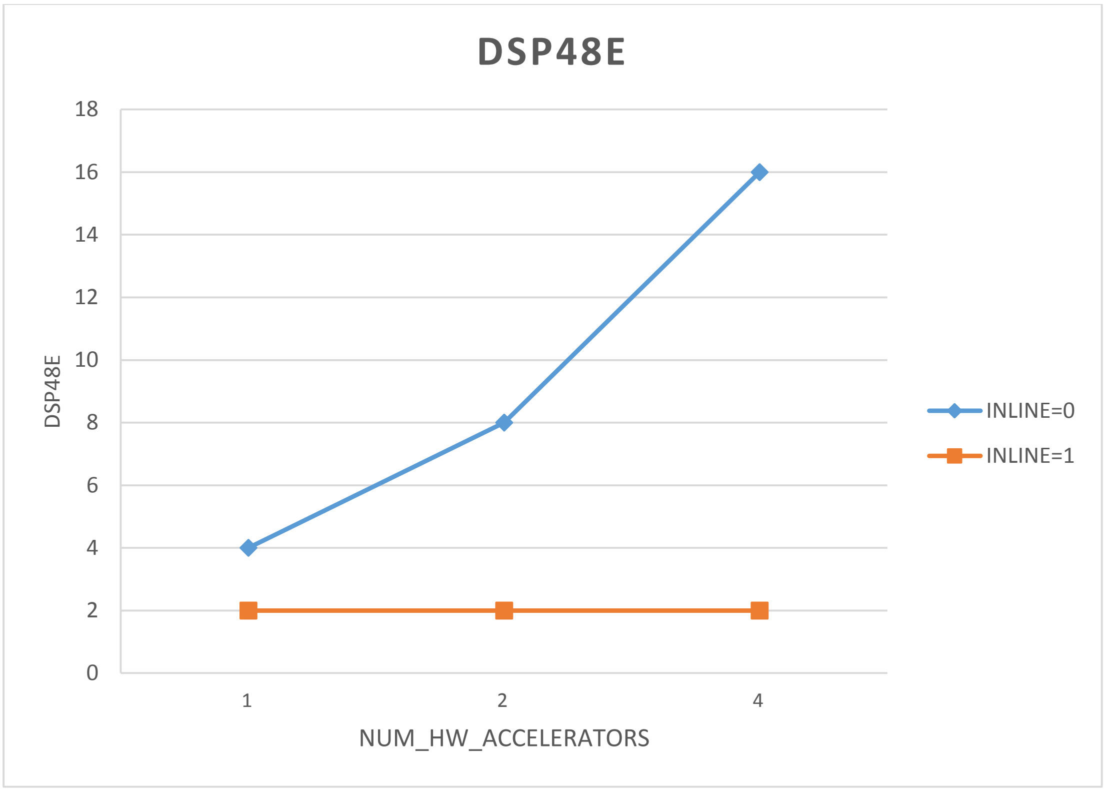


Figure 45: DSP regarding accelerators for Histogram kernel

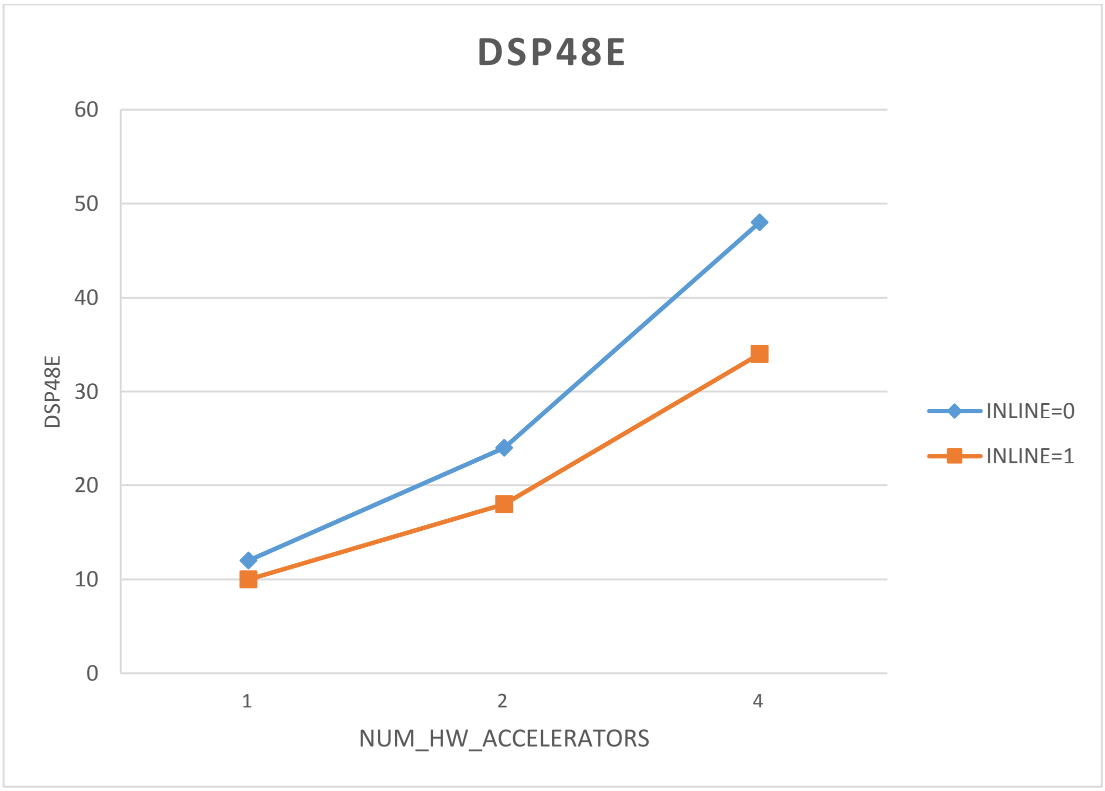


Figure 46: DSP regarding accelerators for MMUL kernel

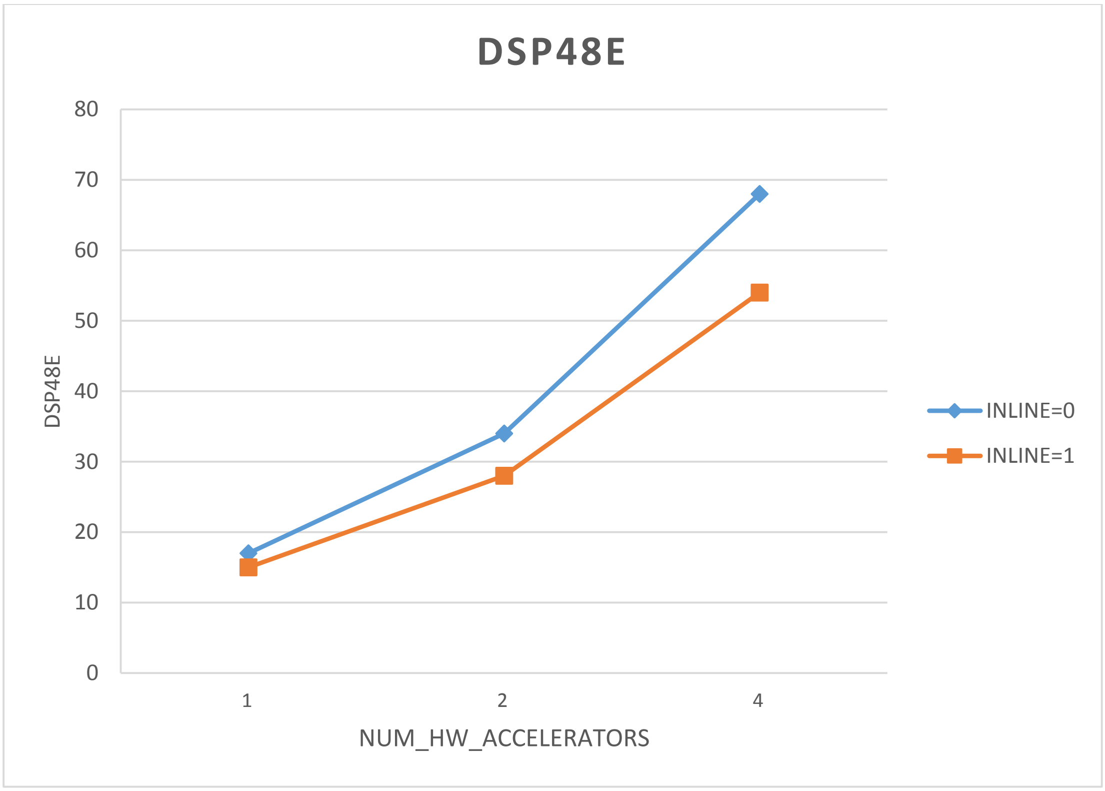


Figure 47: DSP regarding accelerators for PCA kernel

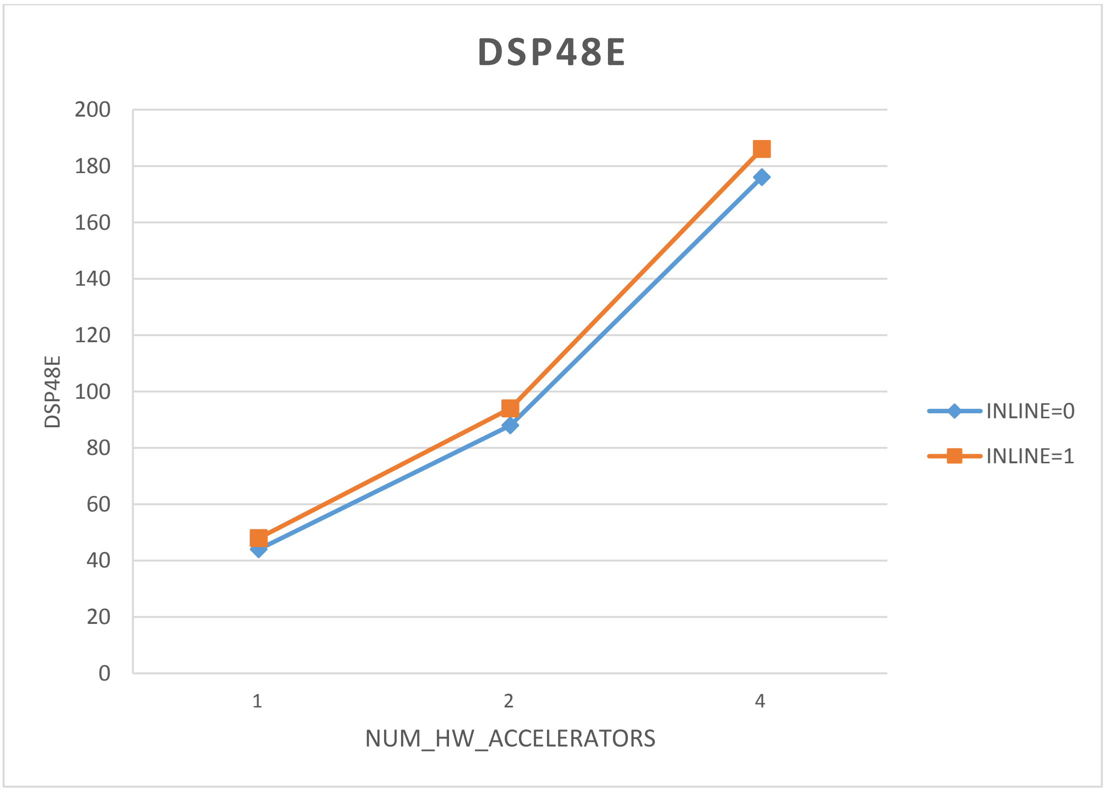


Figure 48: DSP regarding accelerators for Kmeans kernel

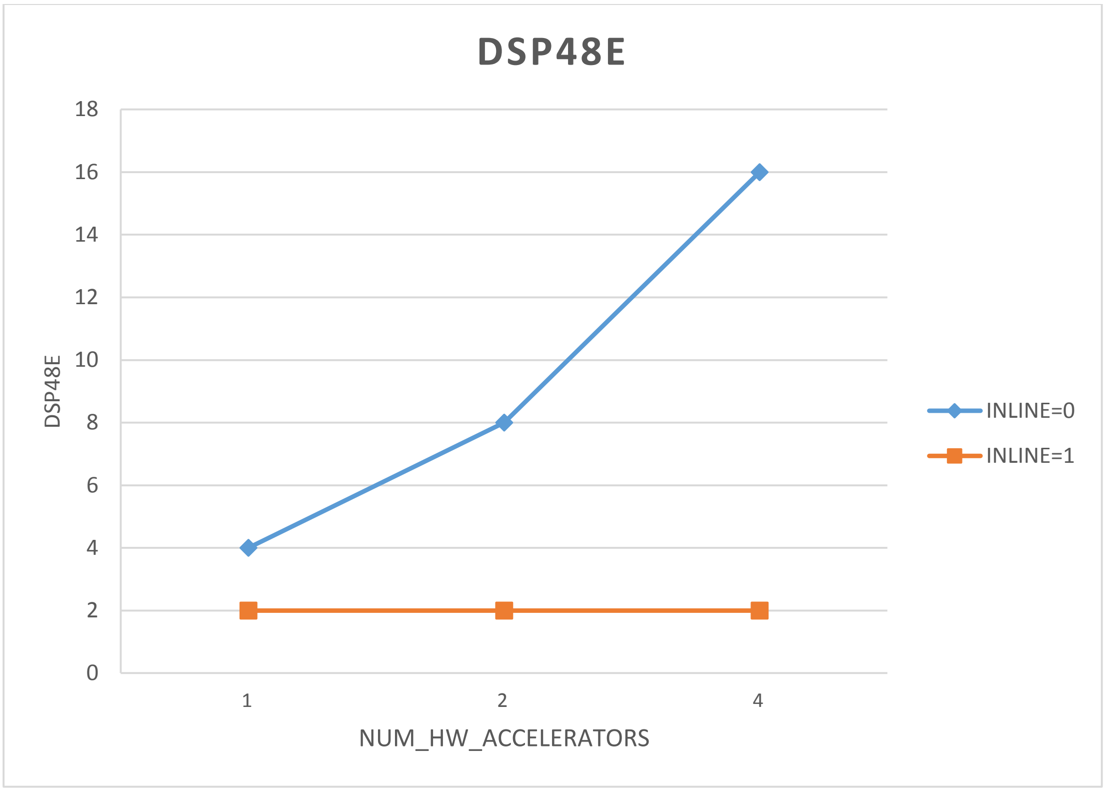


Figure 49: DSP regarding accelerators for Strmatch kernel

Generally, for inline equal to zero, as we add accelerators and memory heaps, the number of DSPs in use (regarding the whole system) increases, which is meaningful. For inline equal to one, we notice two different behaviors. Considering the kernels that do not use DSPs (Histogram, Strmatch), as we add accelerators and heaps and having the function inlining enabled, the number of DSPs remains the same (as these DSPs are used from the whole system and are independent of the number of accelerators or heaps). Regarding the kernels that use DSPs (MMUL, PCA, Kmeans) we observe that when we add accelerators and memory heaps and having the function inlining enabled, the number of DSPs in use raises, as expected, but remains smaller than the number of DSPs without inlining.

Regarding Kmeans kernel, we notice that as we escalate the system (adding more accelerators and heaps) we always need more DSPs when inline is activated. This is happening, as we mentioned before, due to the scheduling of Vivado HLS for inline equal to zero and the Vivado target for the minimum simulation time (when inline is activated).

4.1.4 FF

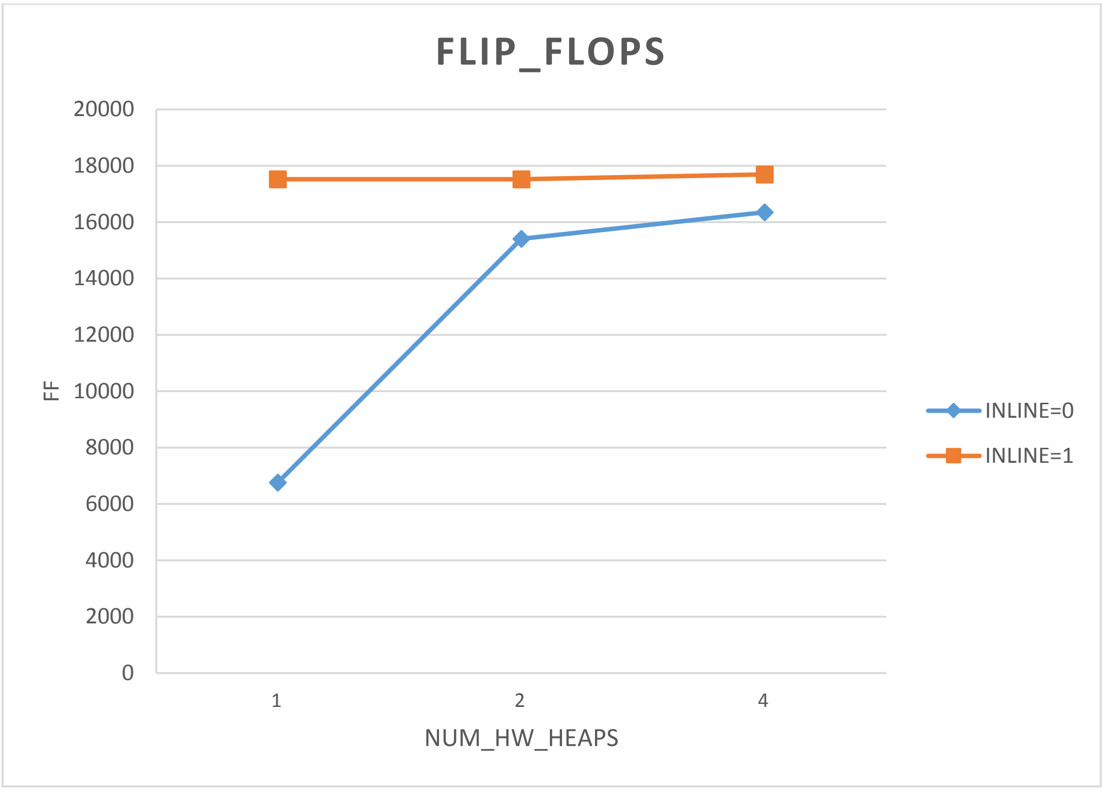


Figure 50: FF regarding memory heaps for Histogram kernel

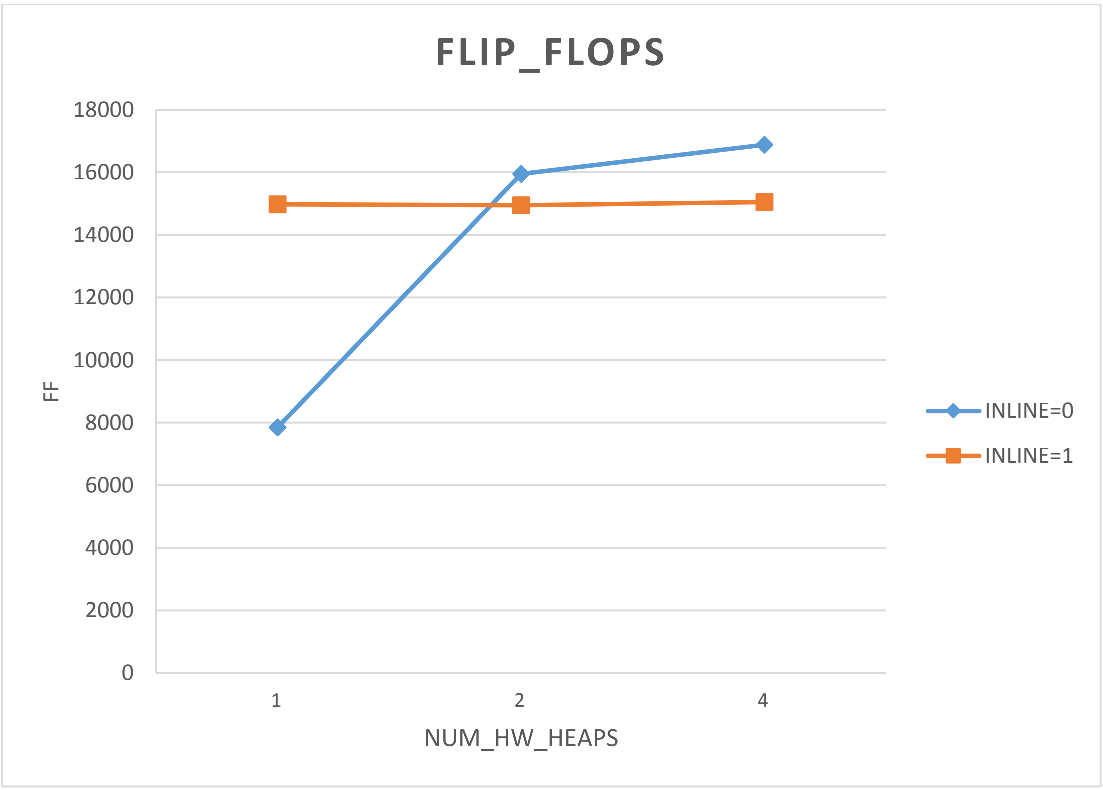


Figure 51: FF regarding memory heaps for MMUL kernel

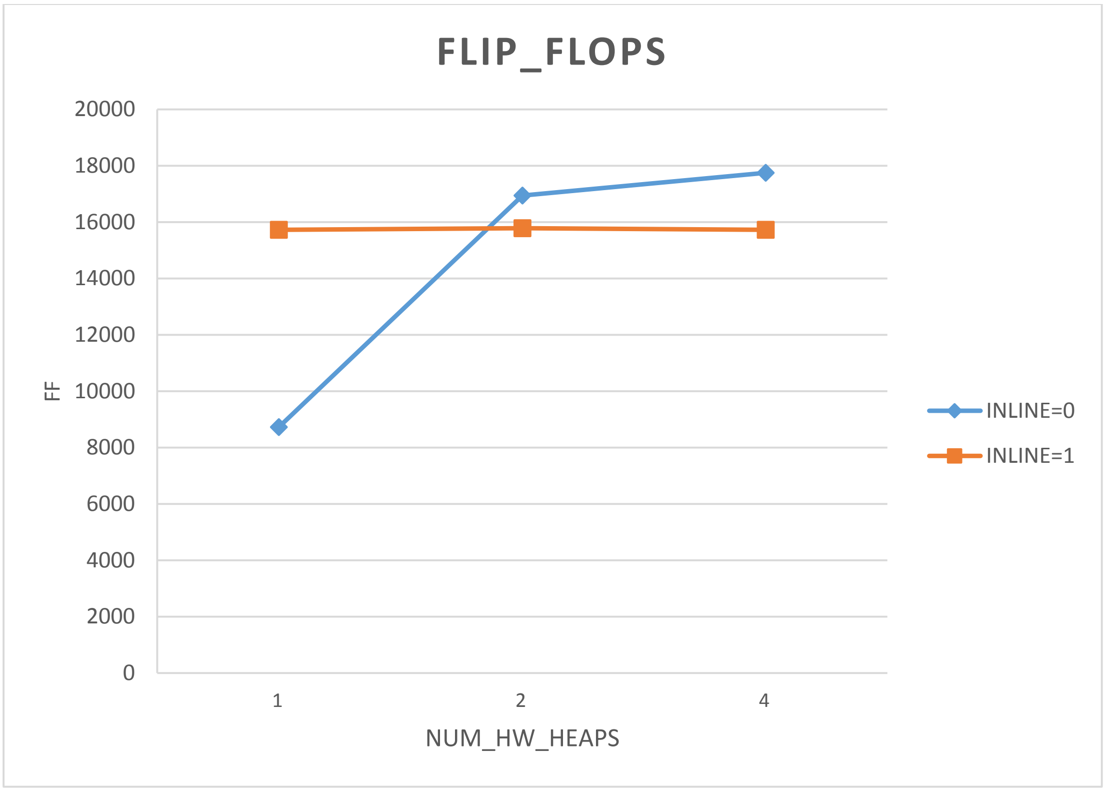


Figure 52: FF regarding memory heaps for PCA kernel

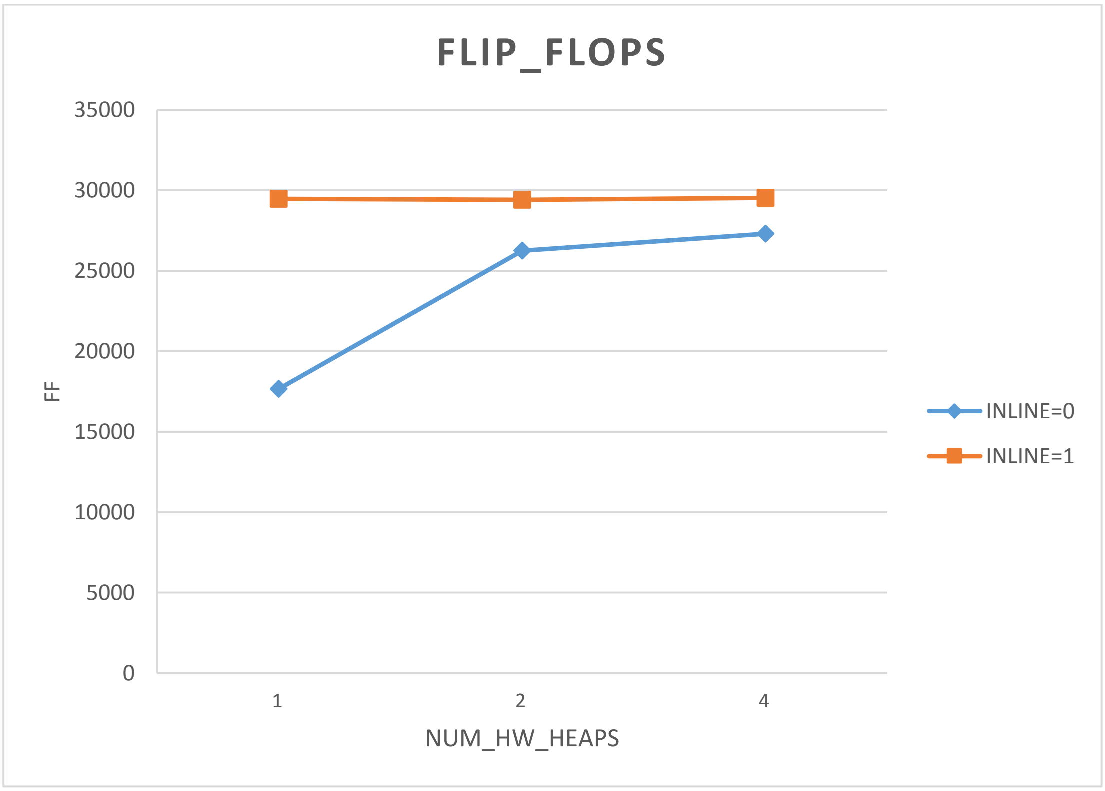


Figure 53: FF regarding memory heaps for Kmeans kernel

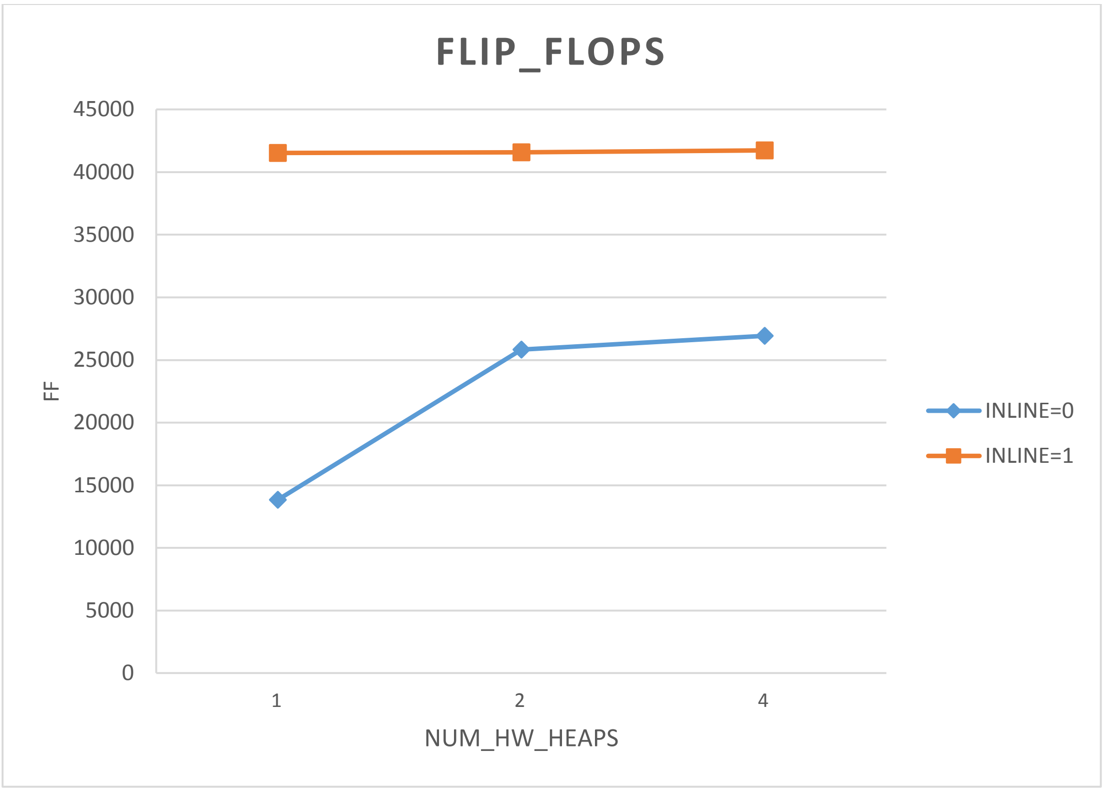


Figure 54: FF regarding memory heaps for Strmatch kernel

From figures 50-54, we observe that function inlining results in the raise of the number of FFs. This is happening because there is a trade off in FFs due to the utilization of many multiplexers that control the sharing and reuse of several hardware components, as inline push the system to a more optimal and parallel execution of tasks.

Regarding PCA and MMUL kernels, we observe that, for number of heaps greater than one, the number of FFs in use for inline equal to one is smaller than those when inline is equal to zero. This is happening because, when inline is zero, Vivado HLS creates RTL implementations for the DMM library that cost mainly in FFs. When we enable the function inlining for DMM functions, Vivado HLS does not create RTL circuits for DMM library any more. Thus, there is a great reduction of FFs in use. Although function inlining needs FFs due to the great number of multiplexers, the amount of needed FFs can not exceed the previous large number of FFs.

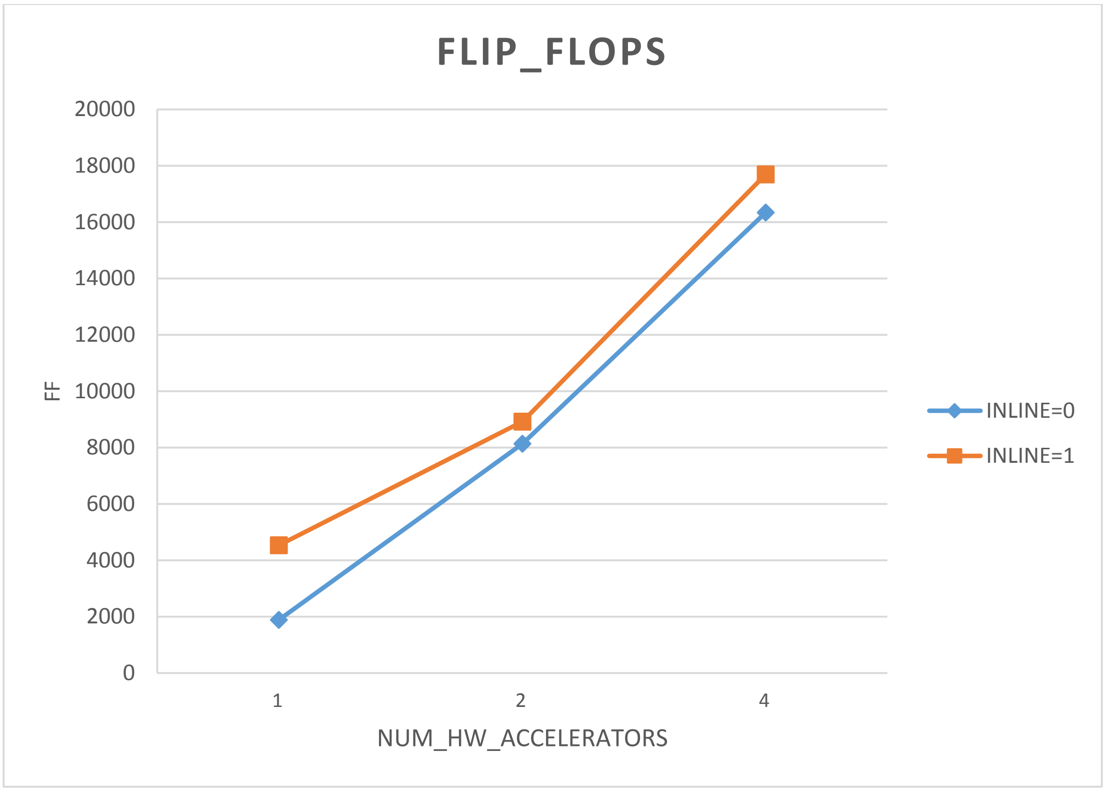


Figure 55: FF regarding accelerators for Histogram kernel

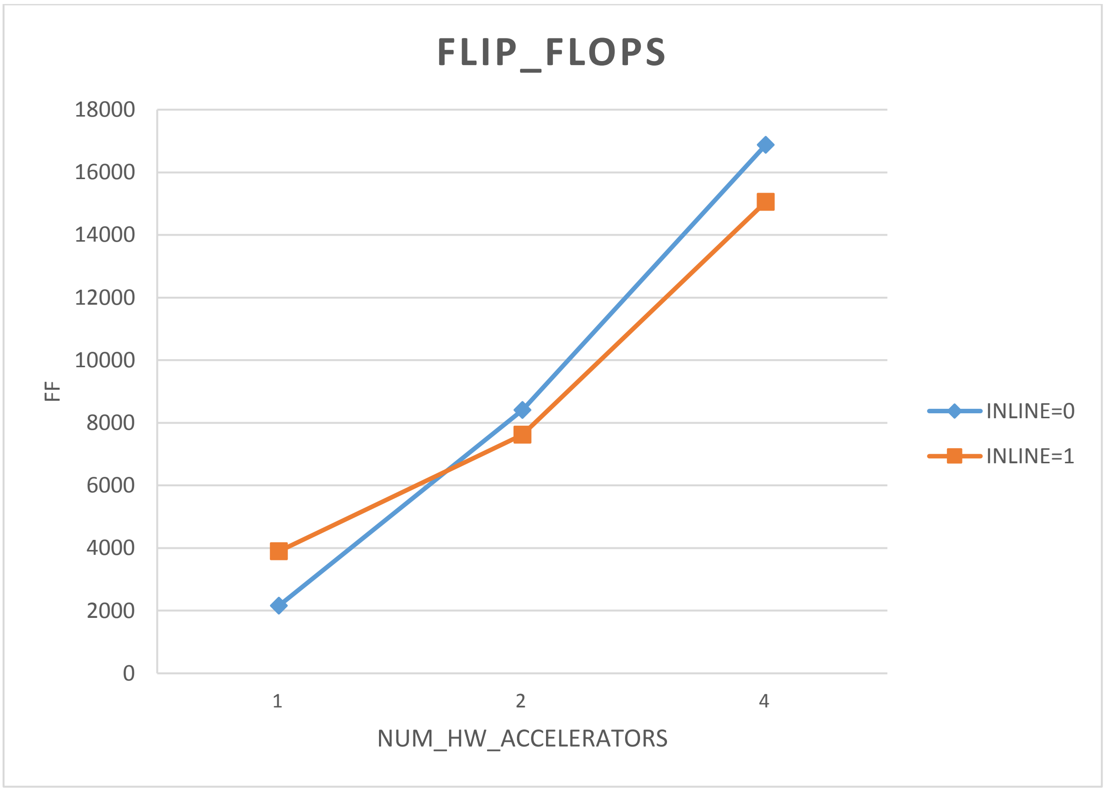


Figure 56: FF regarding accelerators for MMUL kernel

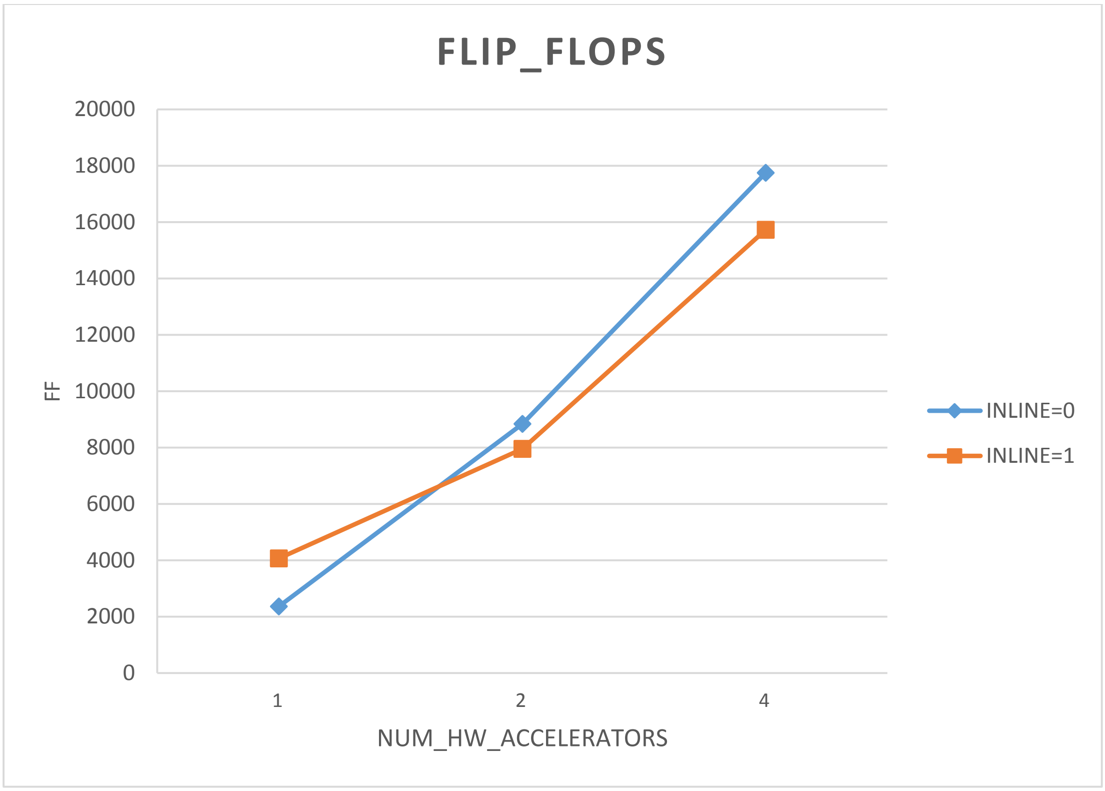


Figure 57: FF regarding accelerators for PCA kernel

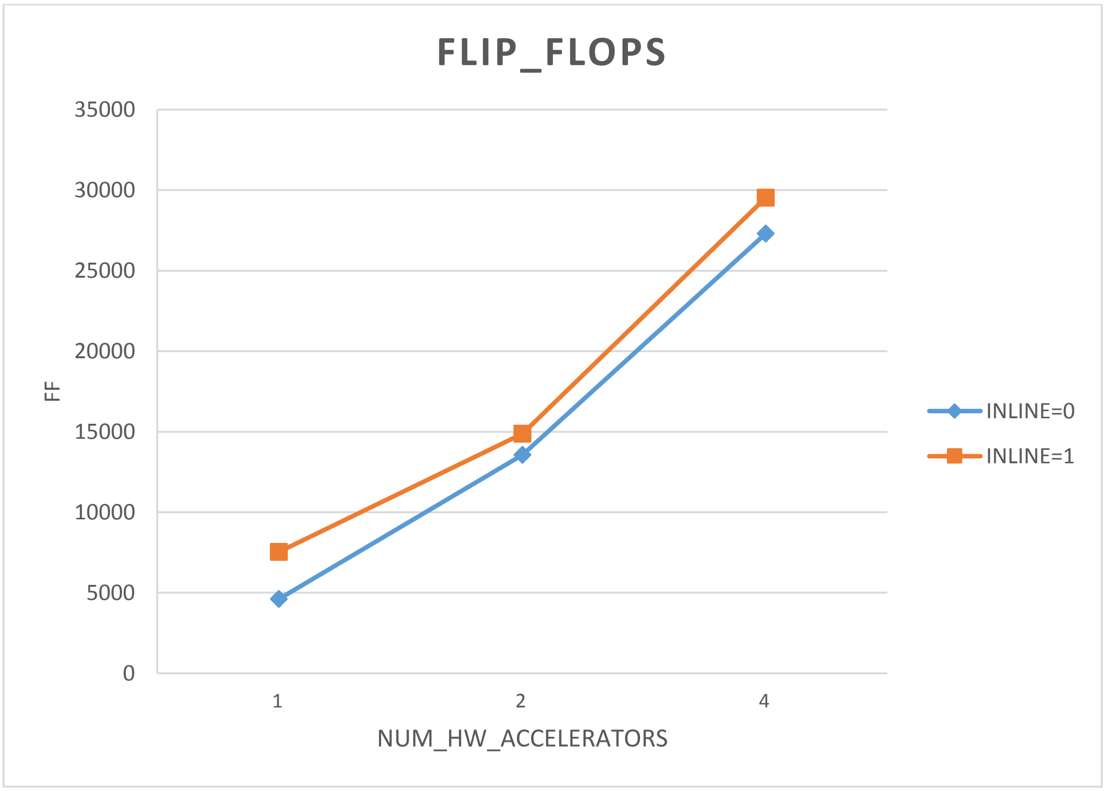


Figure 58: FF regarding accelerators for Kmeans kernel

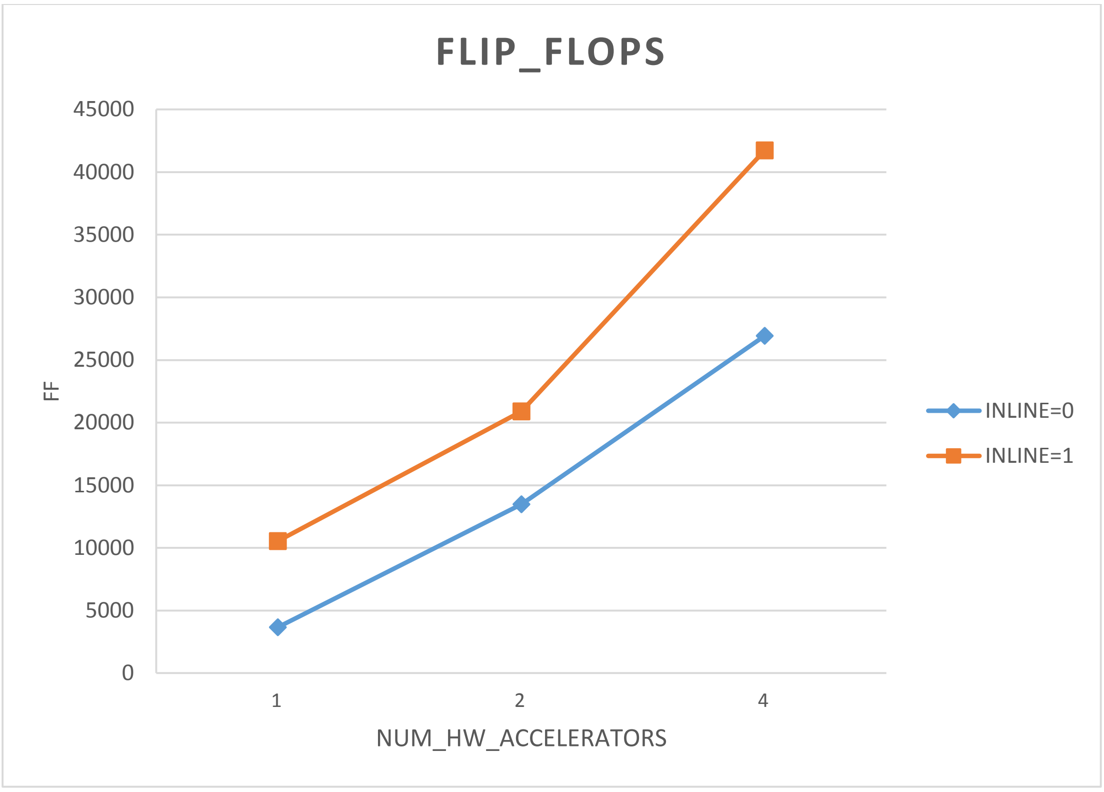


Figure 59: FF regarding accelerators for Strmatch kernel

It is reasonable that as we escalate our system (by adding more accelerators and memory heaps), the number of FFs in use raises for both inline 1 and inline 0. In most kernels, the number of FFs without inlining is smaller than the amount of FFs with inline enabled. The two exceptions concerning MMUL and PCA kernels can be explained, as before. More precisely, although function inlining needs FFs due to the great number of multiplexers, their number can not exceed the previous large amount of FFs (because of the DMM library synthesis).

4.1.5 LUT

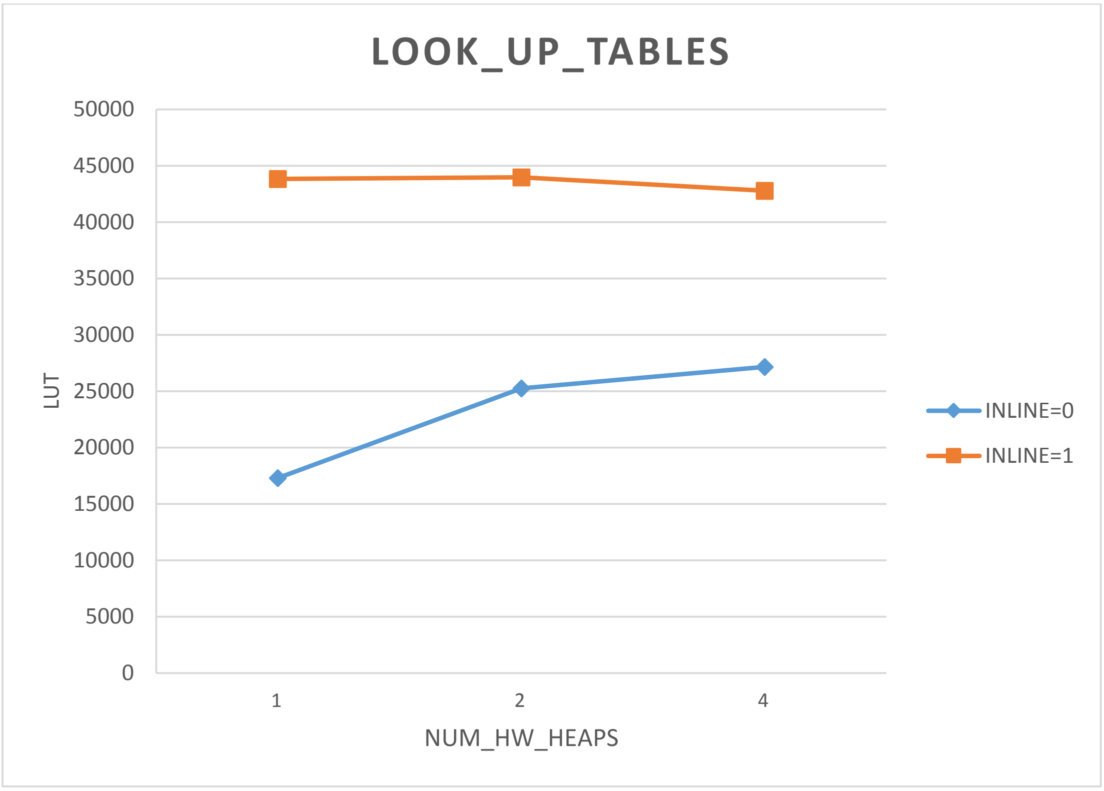


Figure 60: LUT regarding memory heaps for Histogram kernel

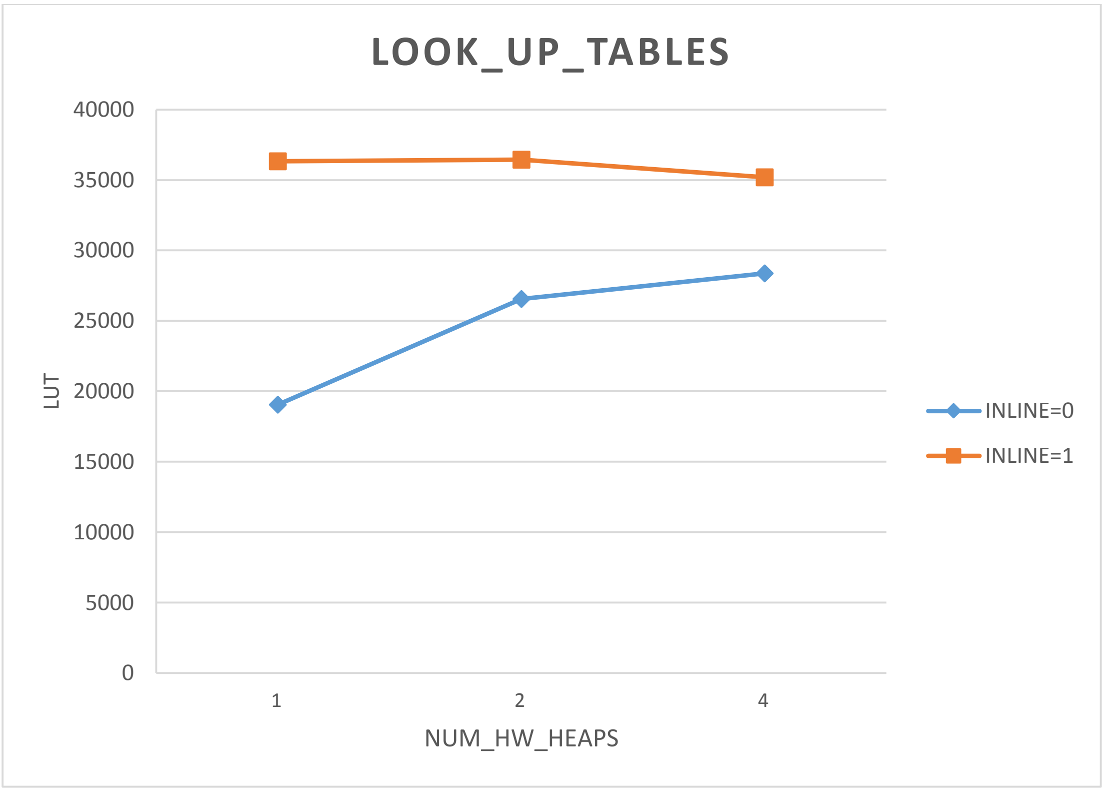


Figure 61: LUT regarding memory heaps for MMUL kernel

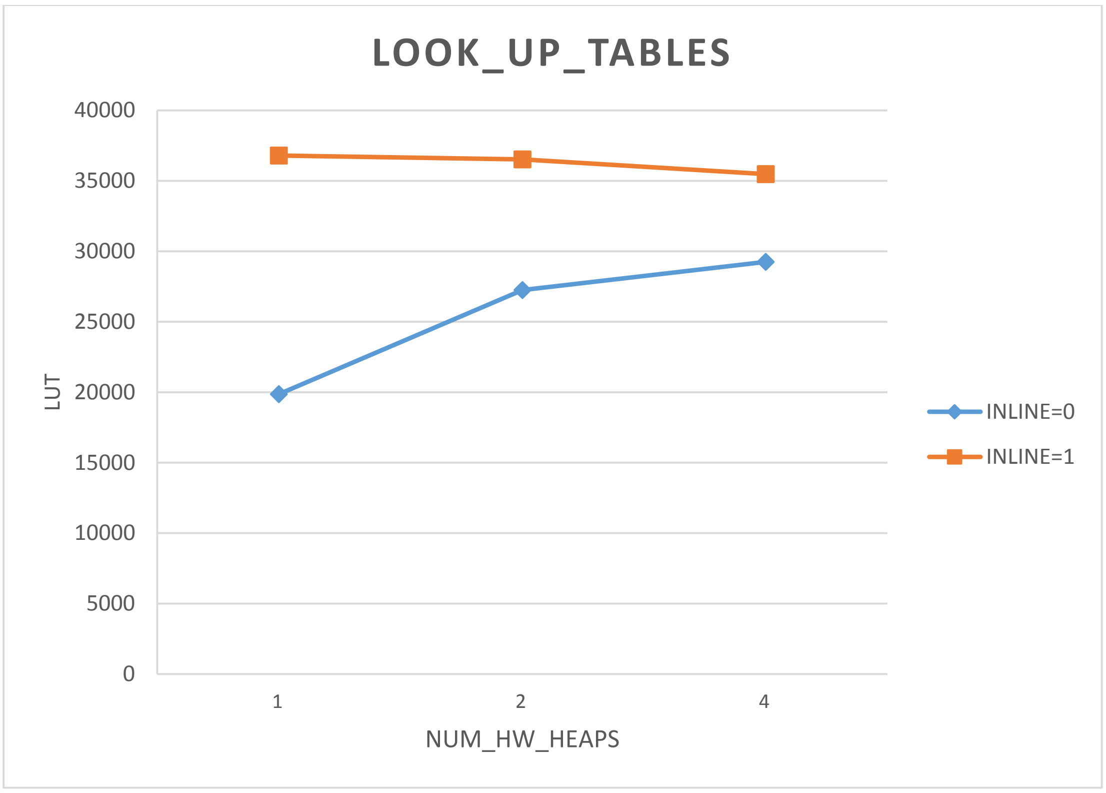


Figure 62: LUT regarding memory heaps for PCA kernel

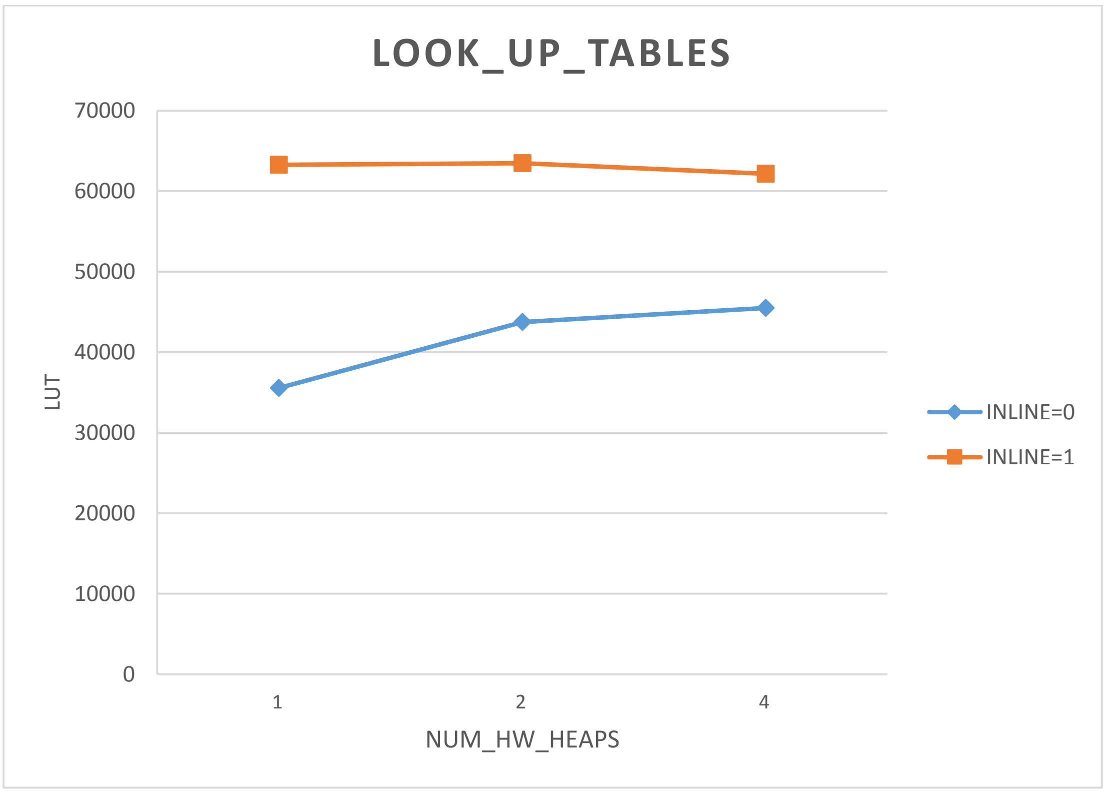


Figure 63: LUT regarding memory heaps for Kmeans kernel

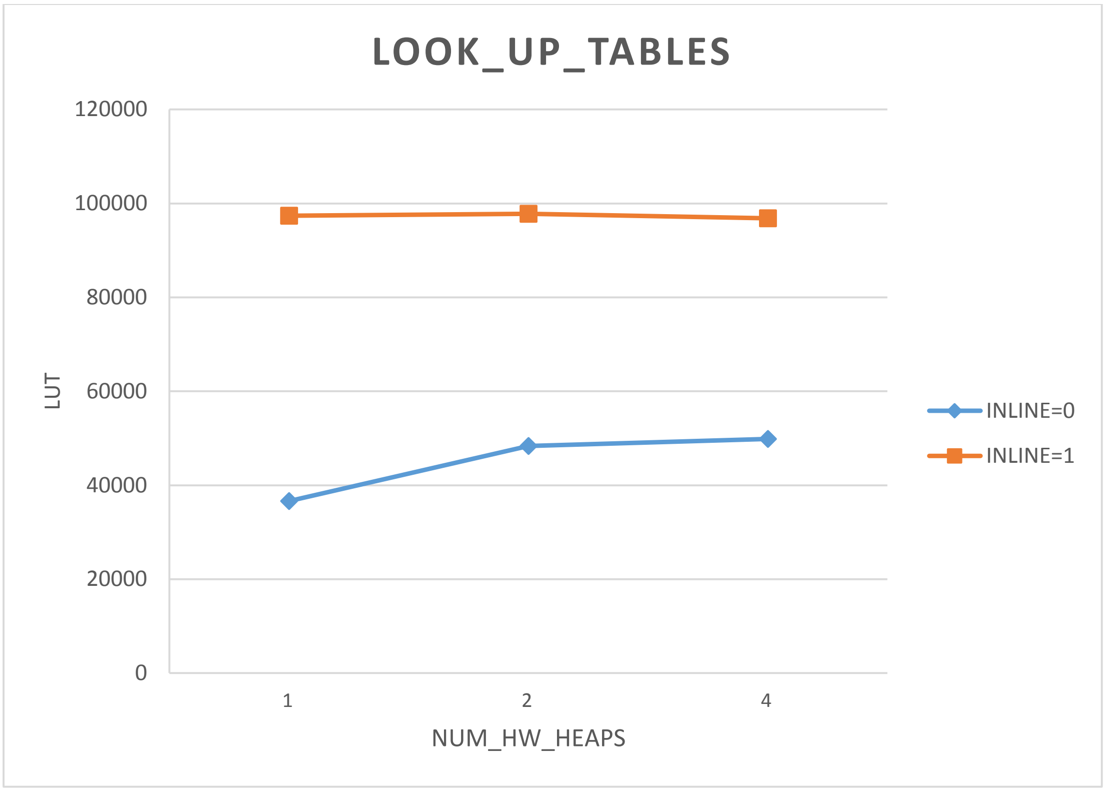


Figure 64: LUT regarding memory heaps for Strmatch kernel

From figures 60-64, we observe that function inlining results in the raise of the number of LUTs in use compared to the number of LUTs in use without inlining. This is happening because, as inline push the system to a more optimal and parallel execution of tasks, there is a trade off in LUTs due to the utilization of many multiplexers that control the sharing and reuse of several hardware components.

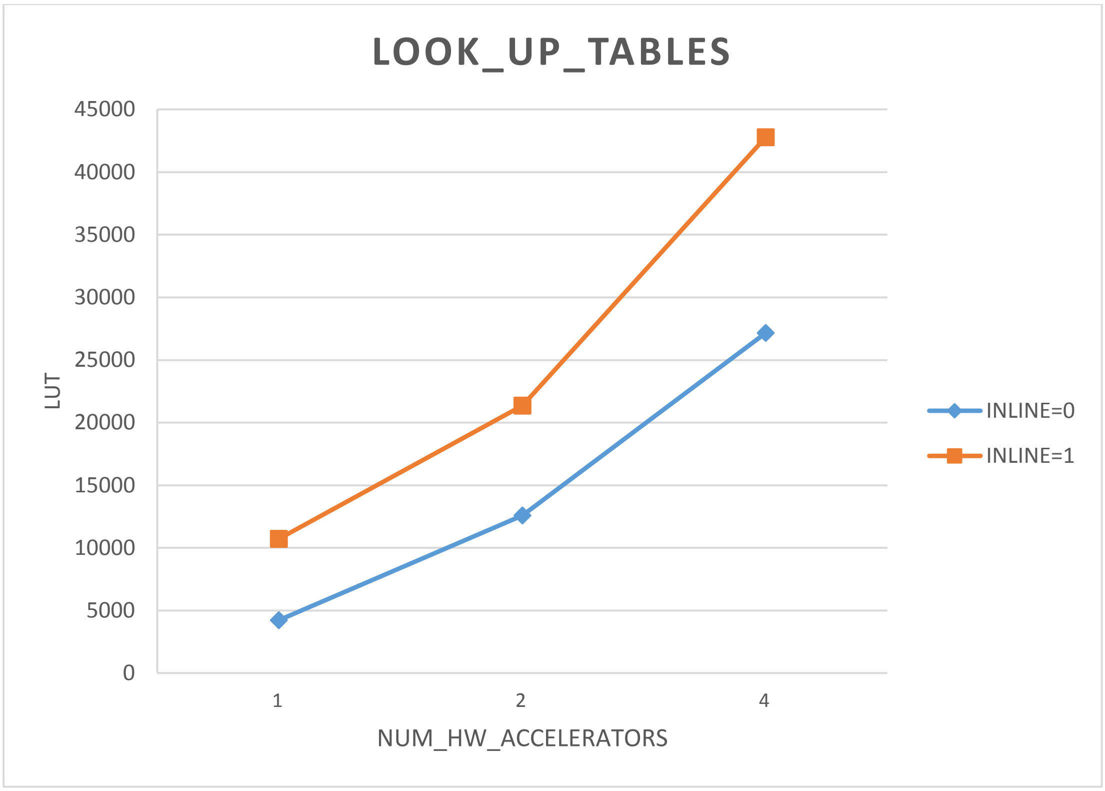


Figure 65: LUT regarding accelerators for Histogram kernel

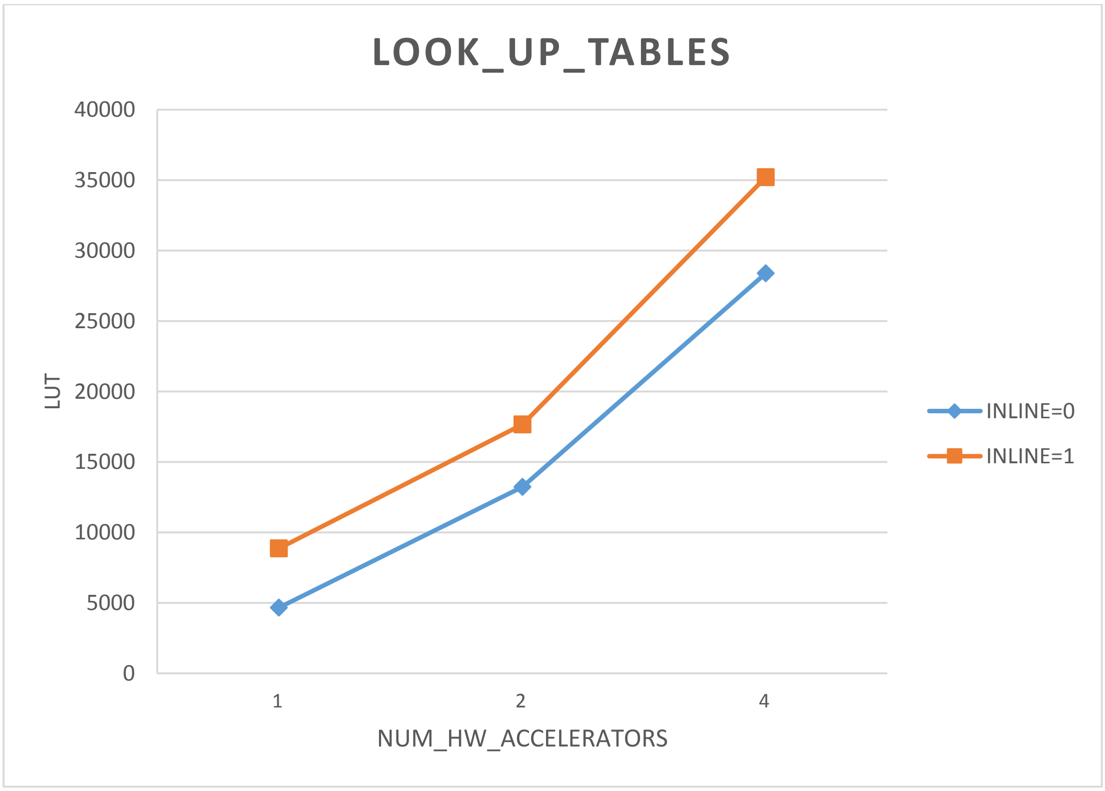


Figure 66: LUT regarding accelerators for MMUL kernel

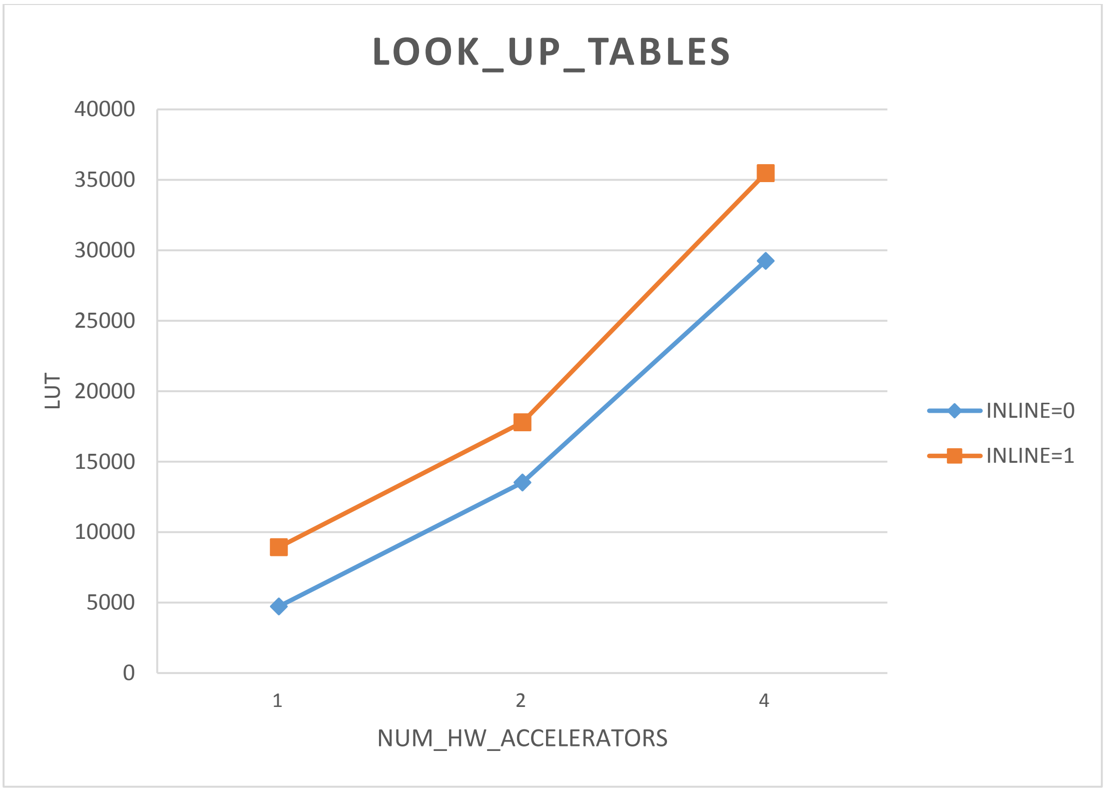


Figure 67: LUT regarding accelerators for PCA kernel

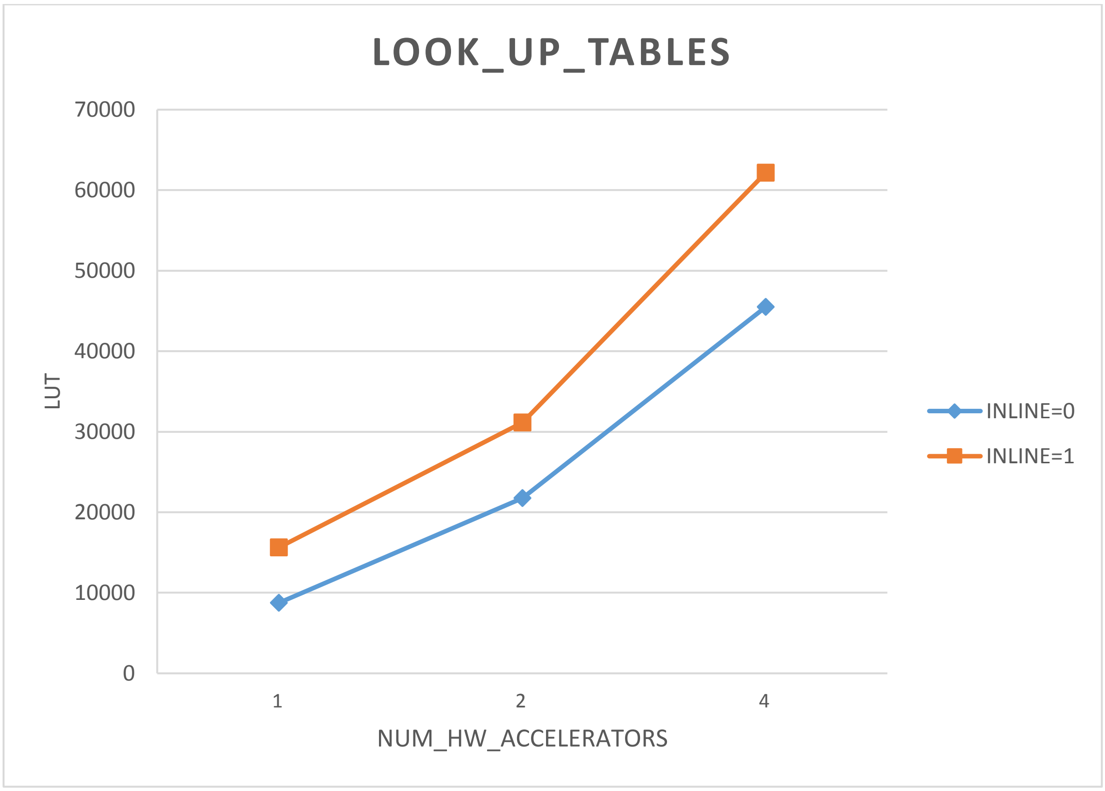


Figure 68: LUT regarding accelerators for Kmeans kernel

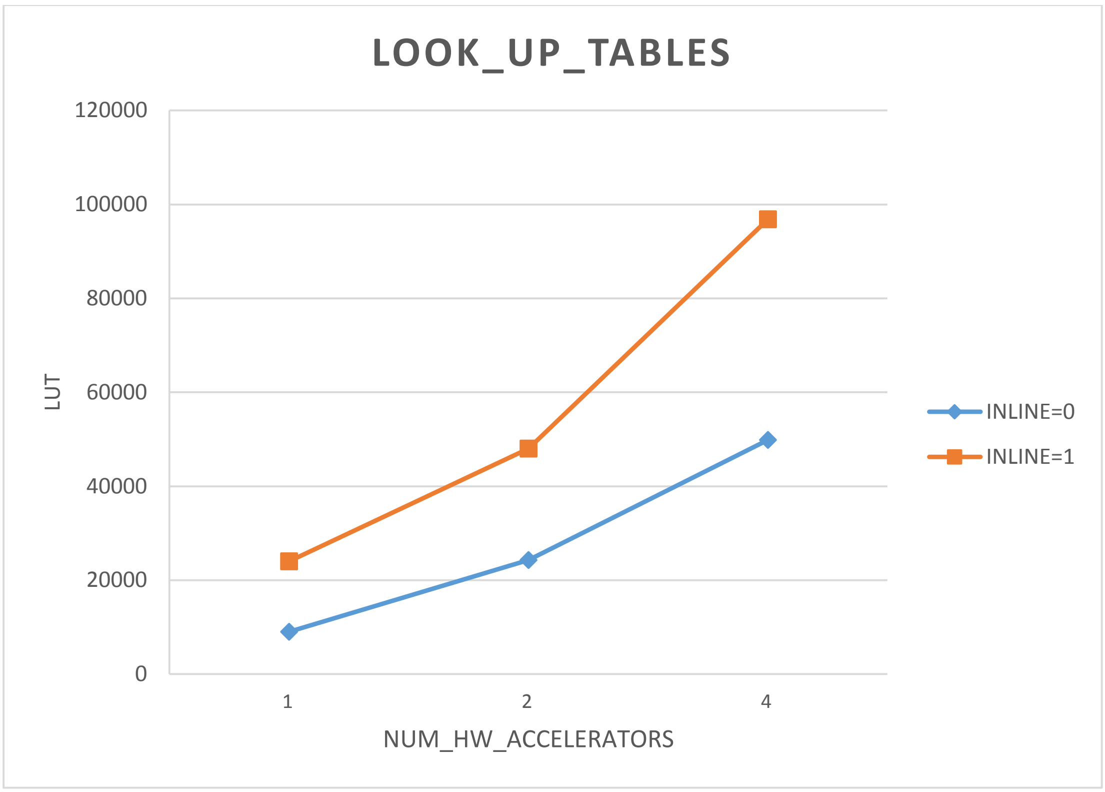


Figure 69: LUT regarding accelerators for Strmatch kernel

From figures 65-69, as we escalate the system (by adding more accelerators and memory heaps), the number of LUTs in use increases for both inline equal to zero and inline equal to one. The reason why the inline curve (inline equal to one) is always above the curve without inlining is the same as before, it happening due to many multiplexers that control the sharing and the reuse of several hardware components.

4.2 Freelist Array

4.2.1 Simulation time

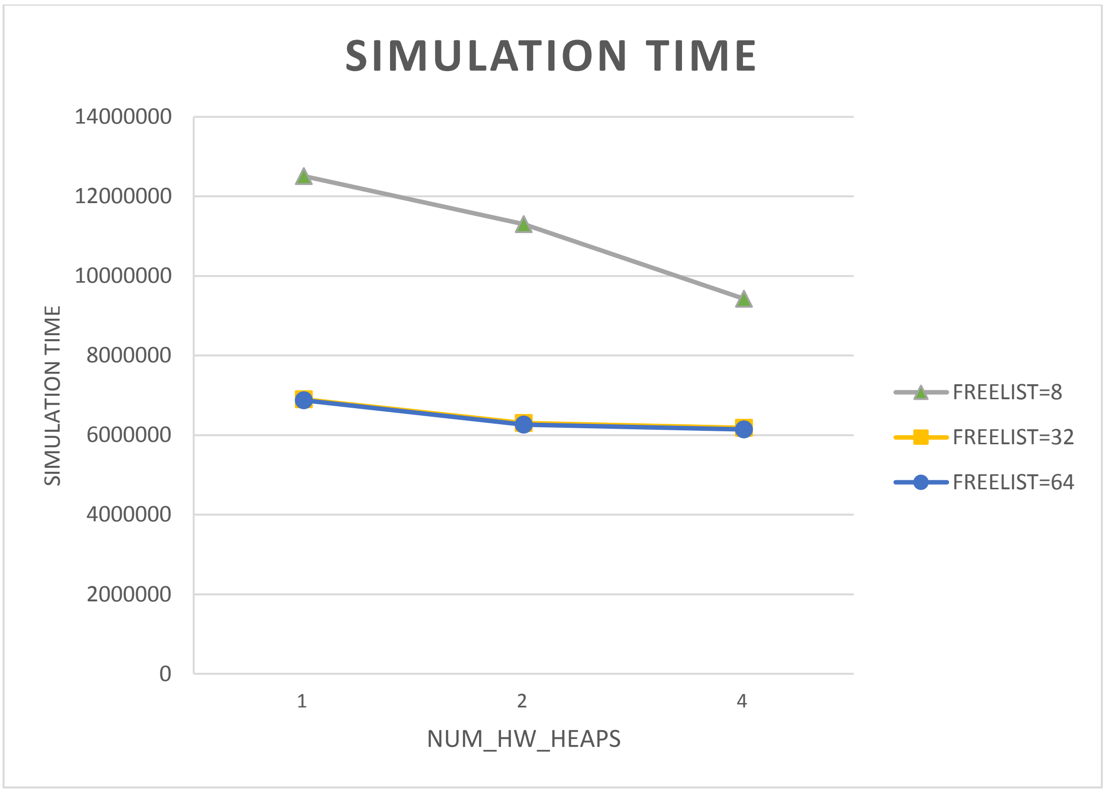


Figure 70: Simulation time regarding memory heaps for Histogram kernel

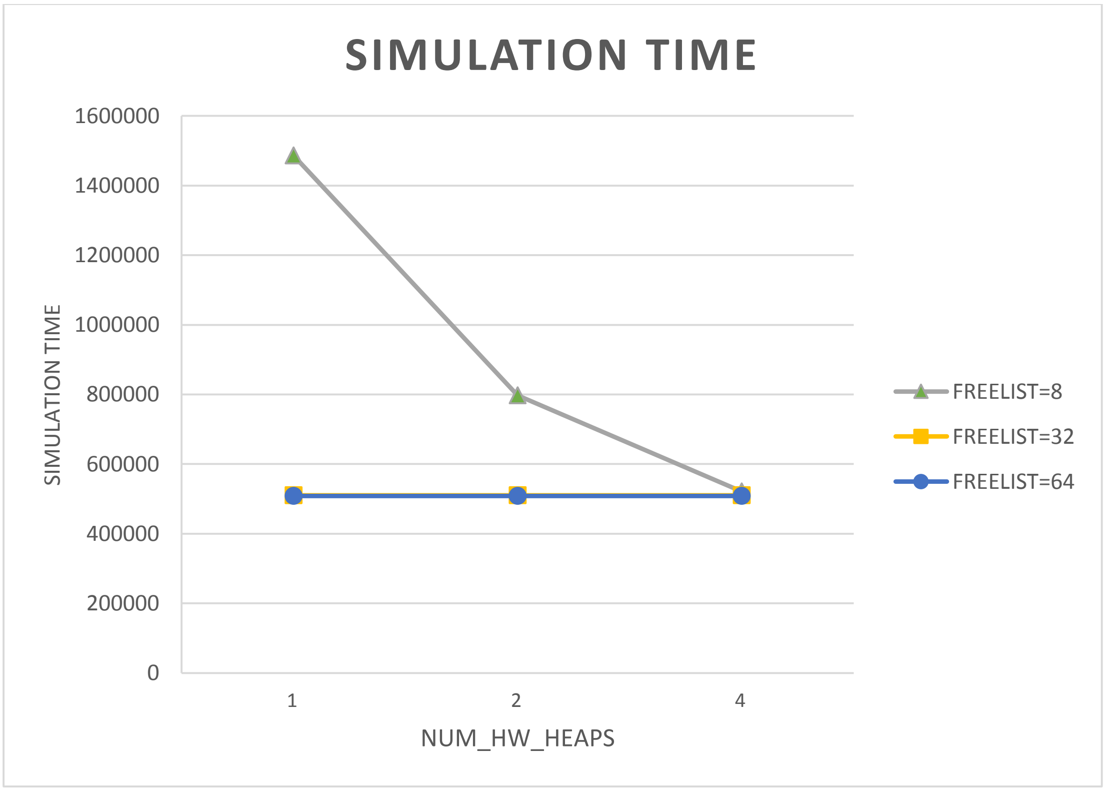


Figure 71: Simulation time regarding memory heaps for MMUL kernel

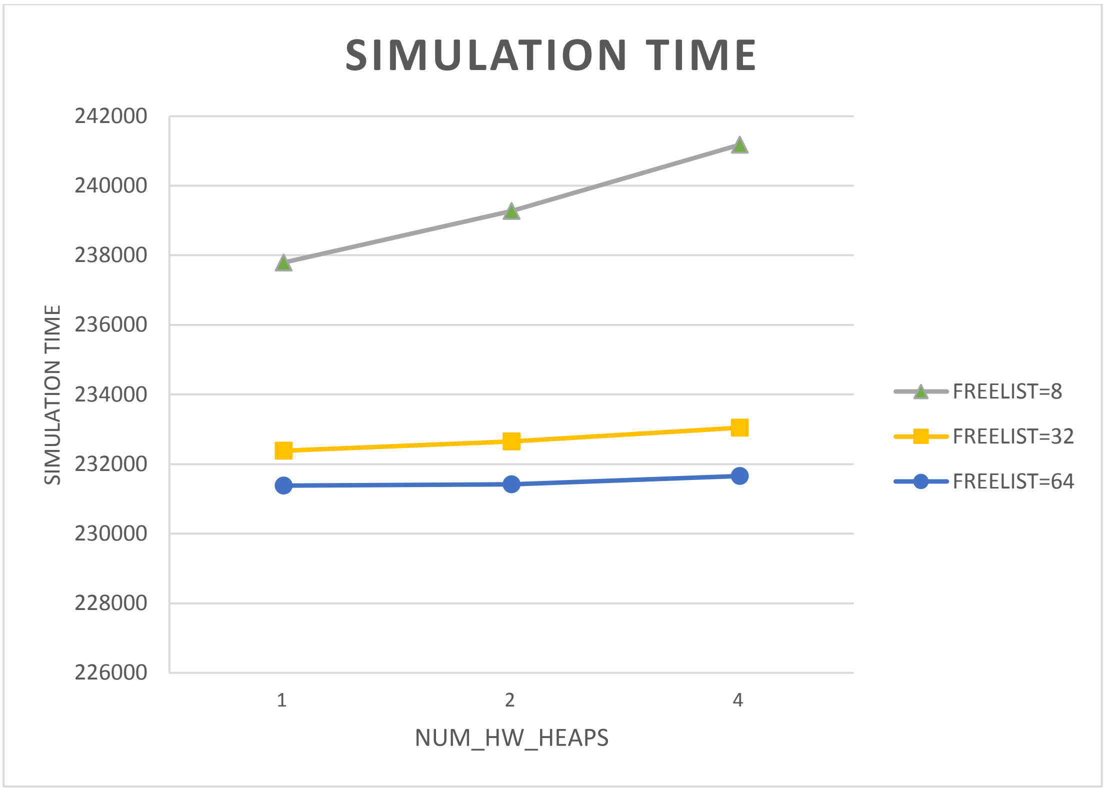


Figure 72: Simulation time regarding memory heaps for PCA kernel

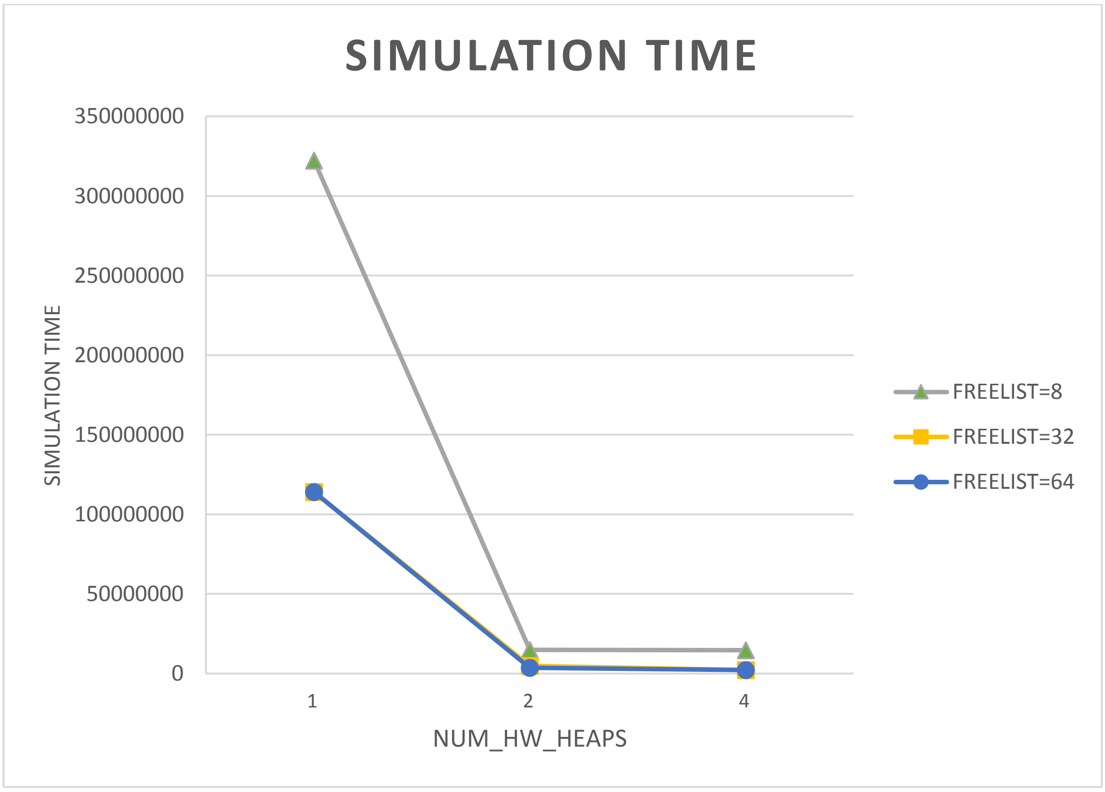


Figure 73: Simulation time regarding memory heaps for Kmeans kernel

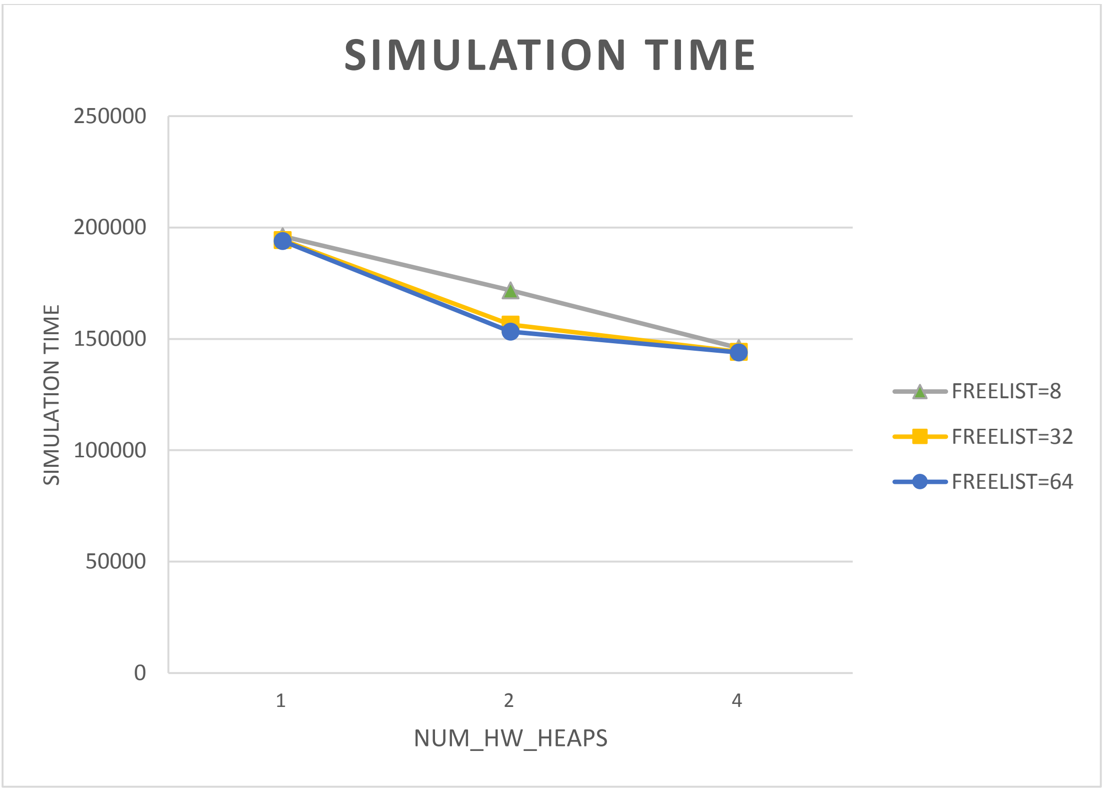


Figure 74: Simulation time regarding memory heaps for Strmatch kernel

We observe that as we increase the width of freelist array, the simulation time decreases. This is reasonable because as we increase the width, we may access bigger memory segments into the heap in one iteration. Thus, we can find a free memory block equal to the requested bytes faster. We also notice that for width equal to eight, adding more heaps has a great impact on simulation time. This is happening because this case (width equal to eight) is the least optimal case and we can see clearly the impact of adding more heaps (which is the decrease of simulation time). Finally, we do not see a notable difference in simulation time between width 32 and width 64 for almost all the applications. Both widths result in the optimization of simulation time, and the optimization factor of each (width 32 and width 64) depends on the memory pattern of every kernel and the type of variables that the application needs to allocate (int, double, char etc.).

Regarding MMUL kernel, we observe that only freelist width 8 is affected from the allocation of extra memory heaps. There are two main reasons for this behavior. On the one hand, the small data set that we used as input to this kernel in order to complete the process of cosimulation. We noticed that, as we increased the size of input data sets, the allocation of extra memory heaps affected also the width 32 and width 64 (as expected). On the other hand, this behavior is related to the size of the memory footprint and the data accesses over time regarding MMUL kernel.

Regarding PCA kernel, we notice a different behavior. This behavior can be explained by the fact that we use a small data set as input (as we mentioned before for the inline curves). More precisely, Vivado HLS scheduler fails to accomplish parallel execution of several computations and efficient exploitation regarding shared resources, especially for small data sets. Thus, the sharing of hardware resources in combination with the inability of efficient scheduling results in the increase of simulation time.

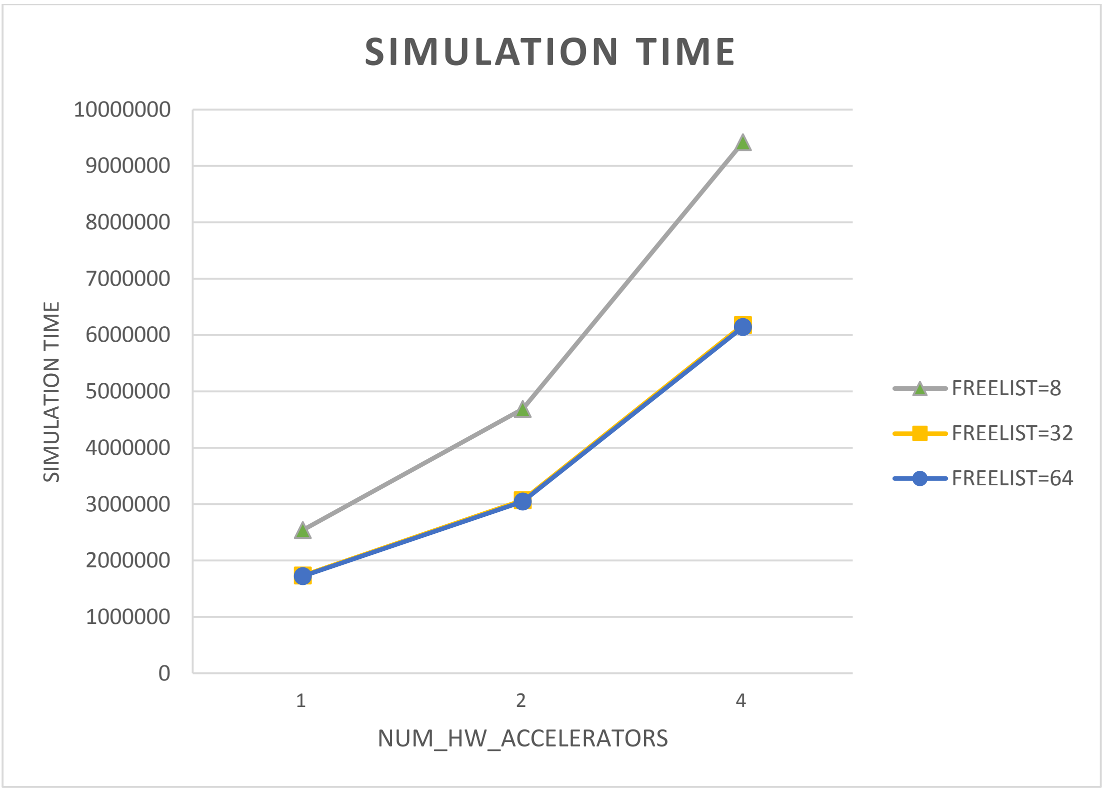


Figure 75: Simulation time regarding accelerators for Histogram kernel

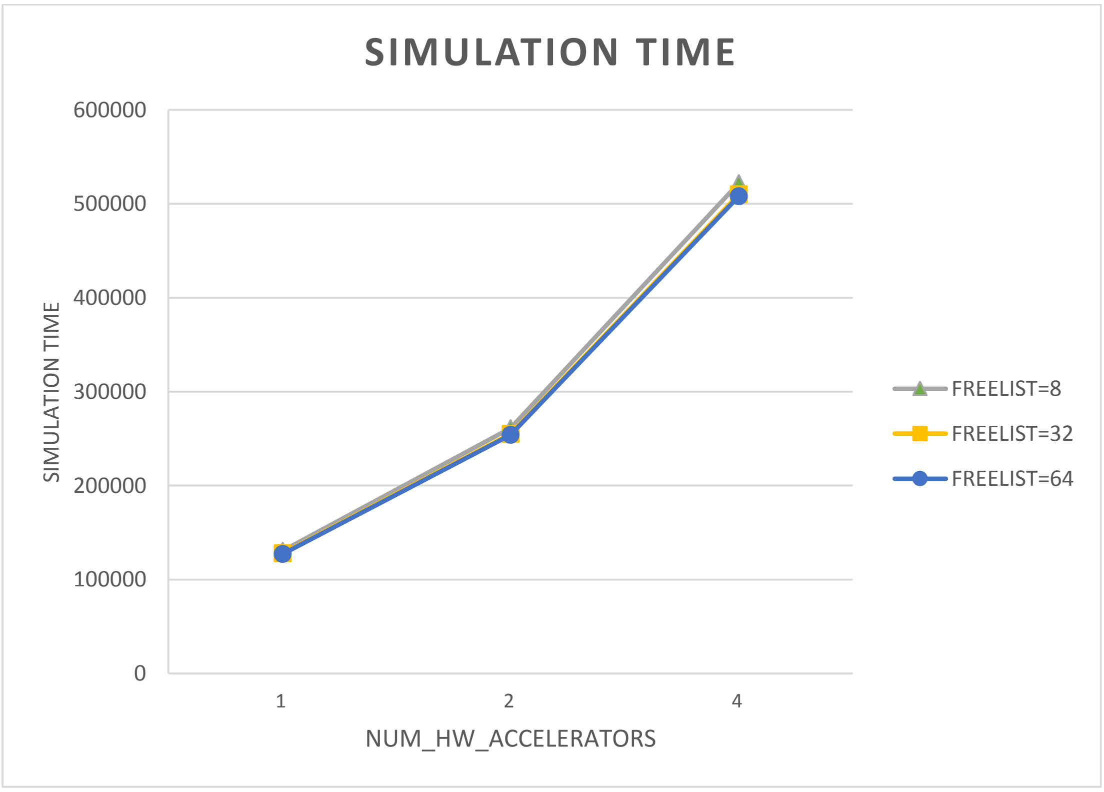


Figure 76: Simulation time regarding accelerators for MMUL kernel

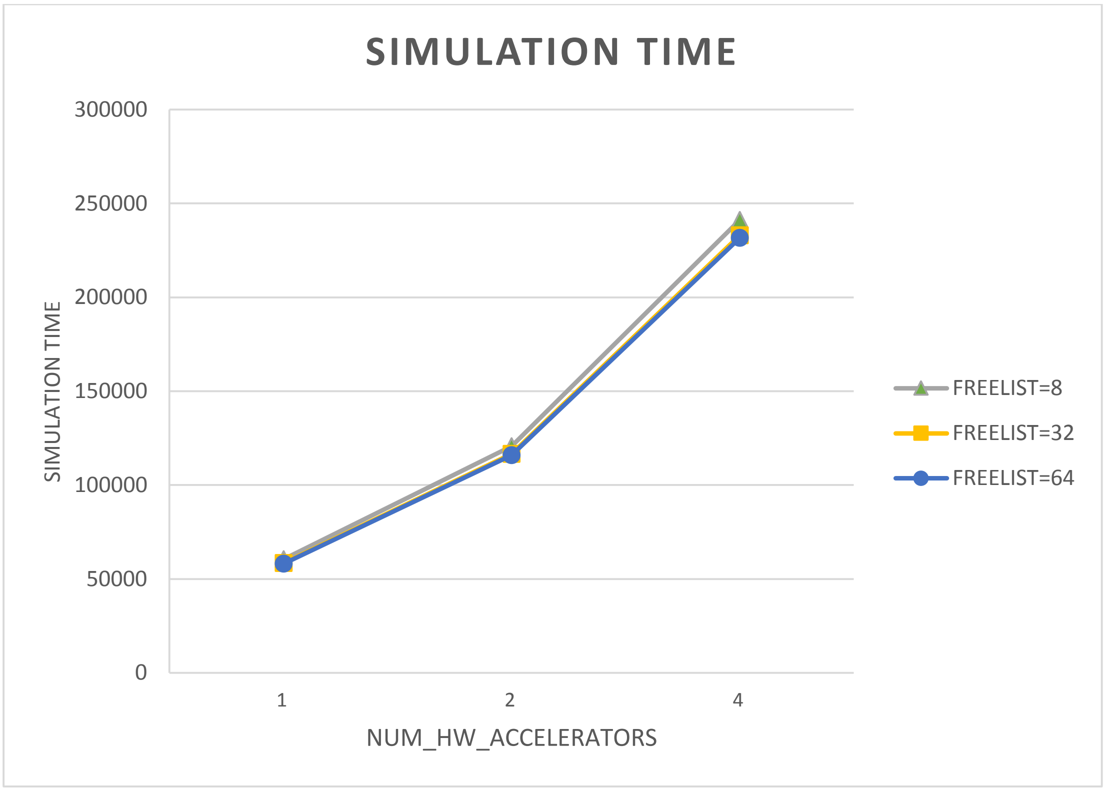


Figure 77: Simulation time regarding accelerators for PCA kernel

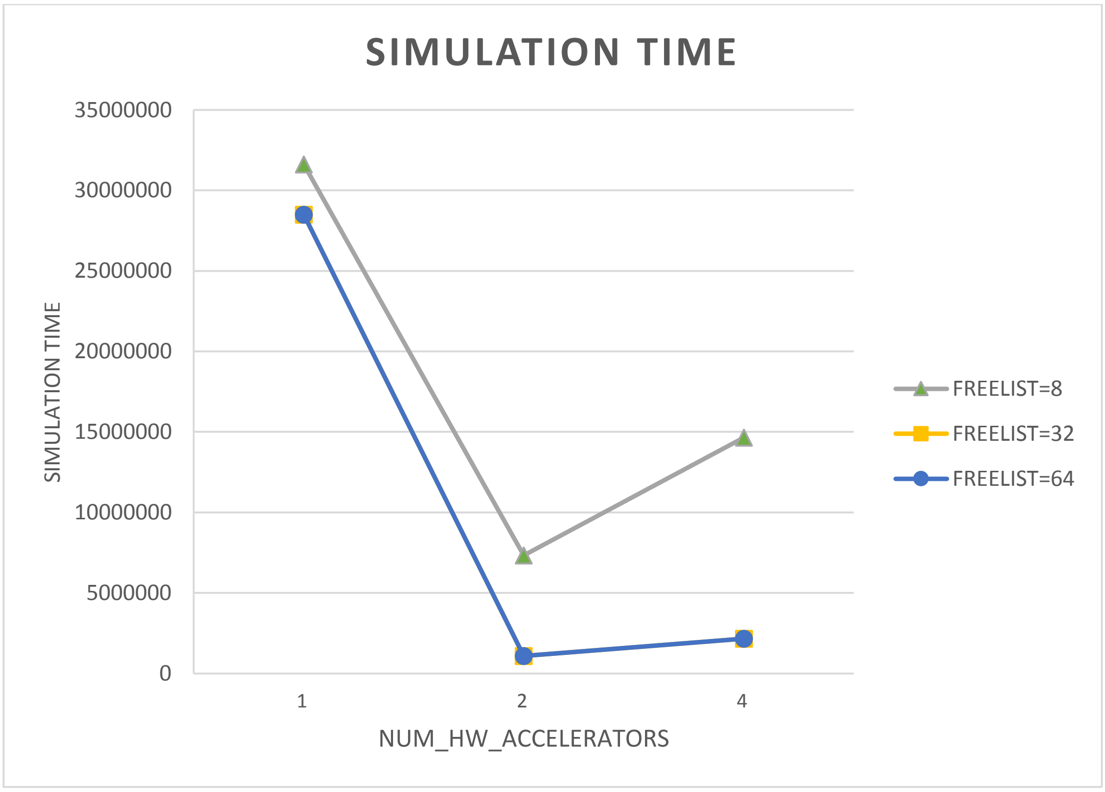


Figure 78: Simulation time regarding accelerators for Kmeans kernel

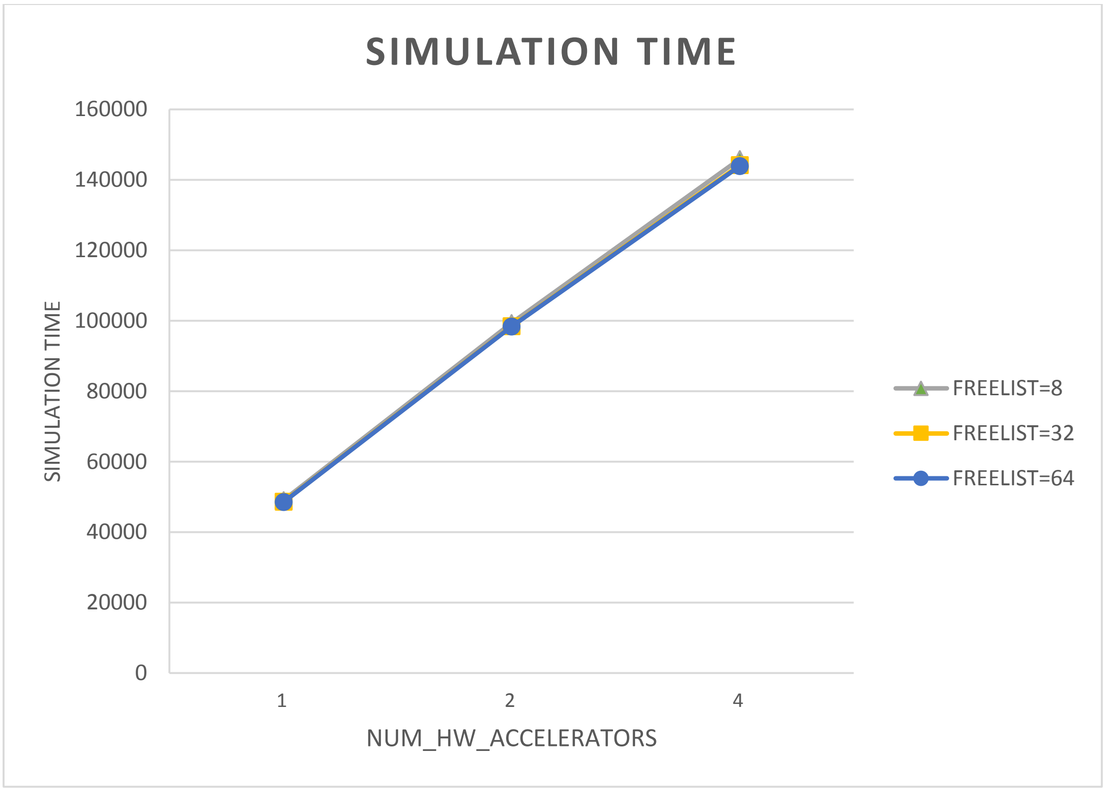


Figure 79: Simulation time regarding accelerators for Strmatch kernel

It is reasonable that as we escalate the system with the addition of more accelerators and heaps, the simulation time raises. The differences between width 8, 32, 64 depend on the memory pattern and the size of allocations/deallocations (as we said before).

4.2.2 BRAM

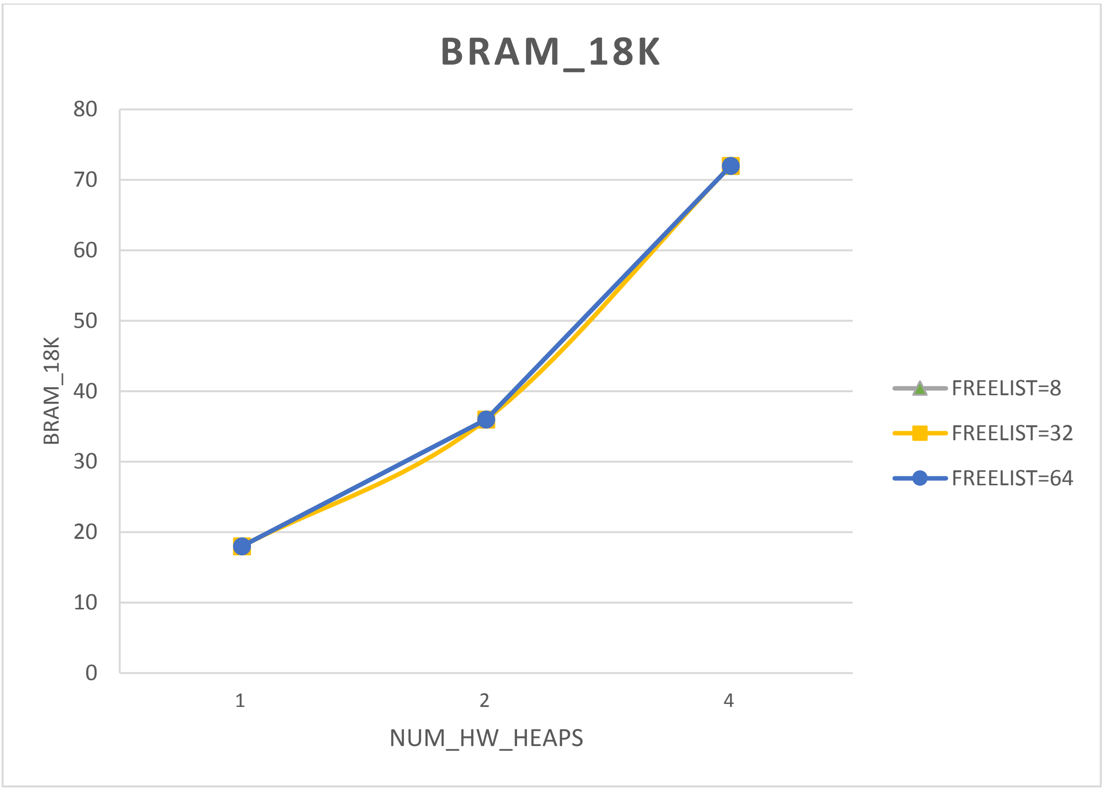


Figure 80: BRAM regarding memory heaps for Histogram kernel

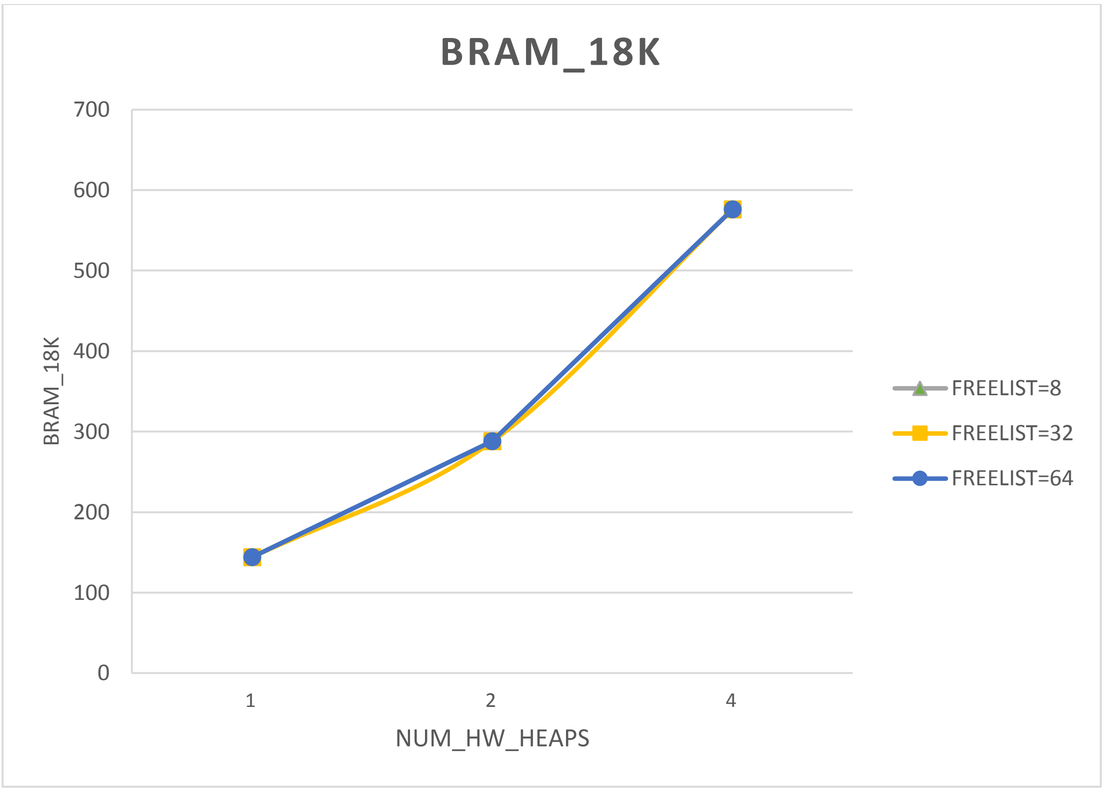


Figure 81: BRAM regarding memory heaps for MMUL kernel

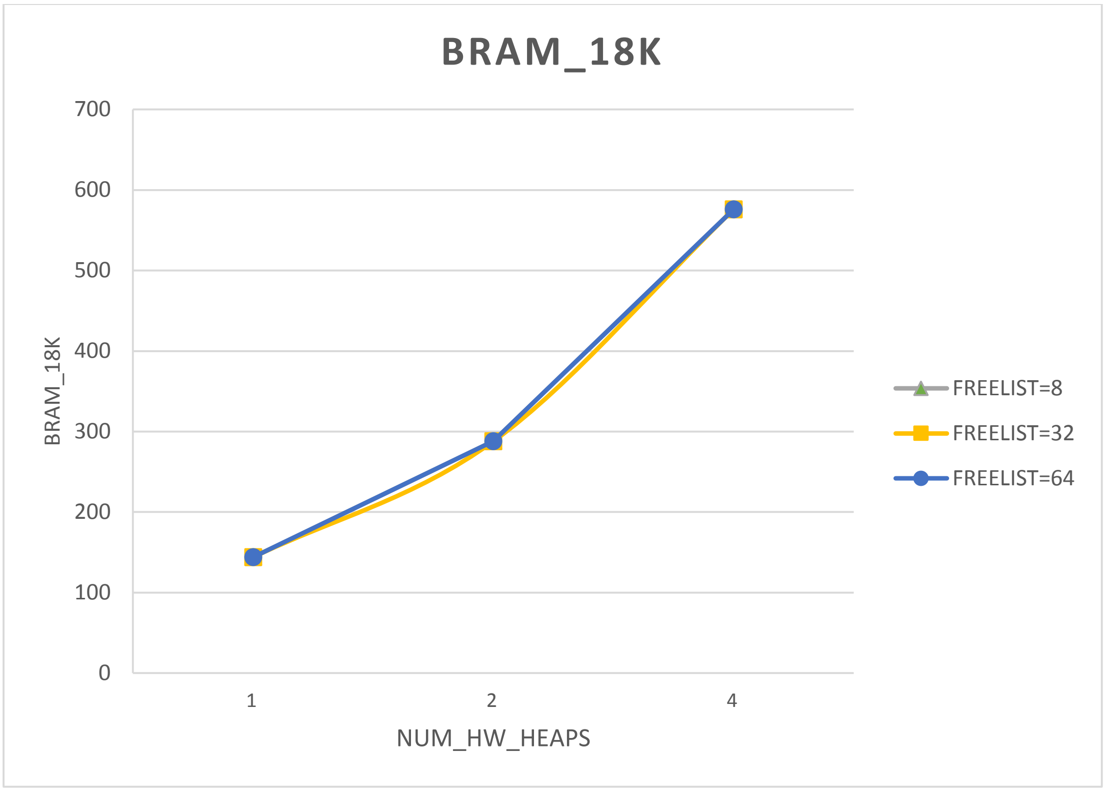


Figure 82: BRAM regarding memory heaps for PCA kernel

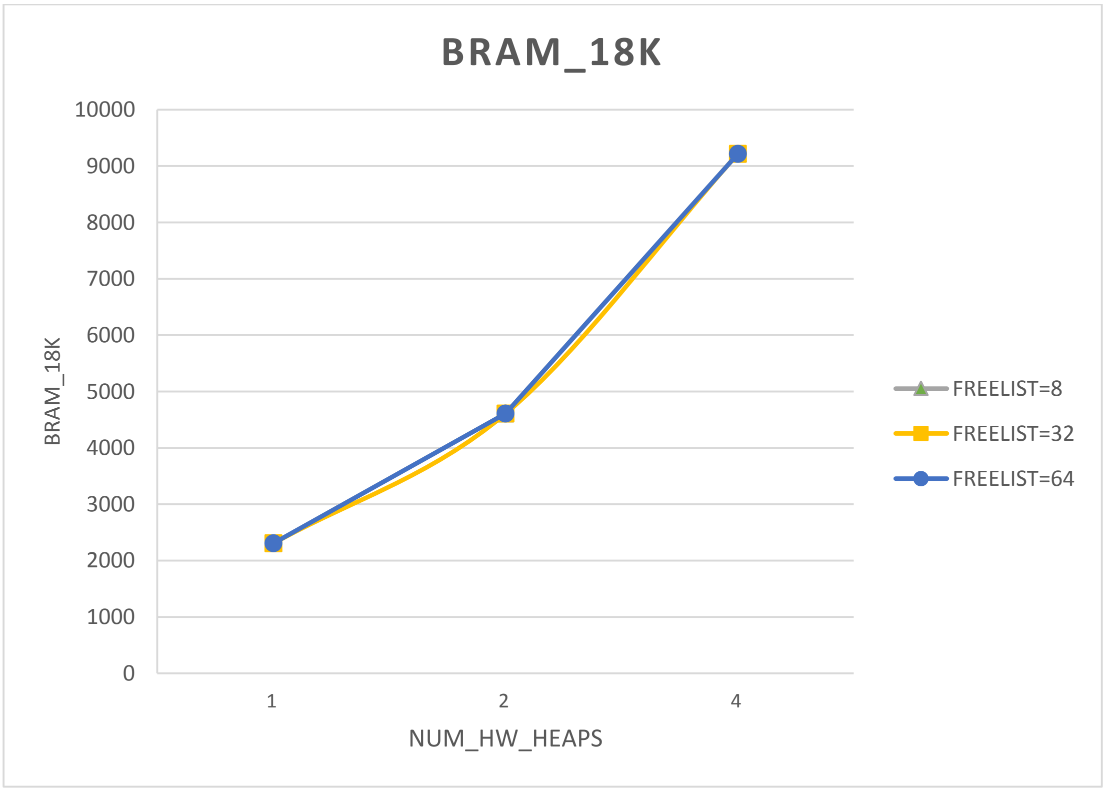


Figure 83: BRAM regarding memory heaps for Kmeans kernel

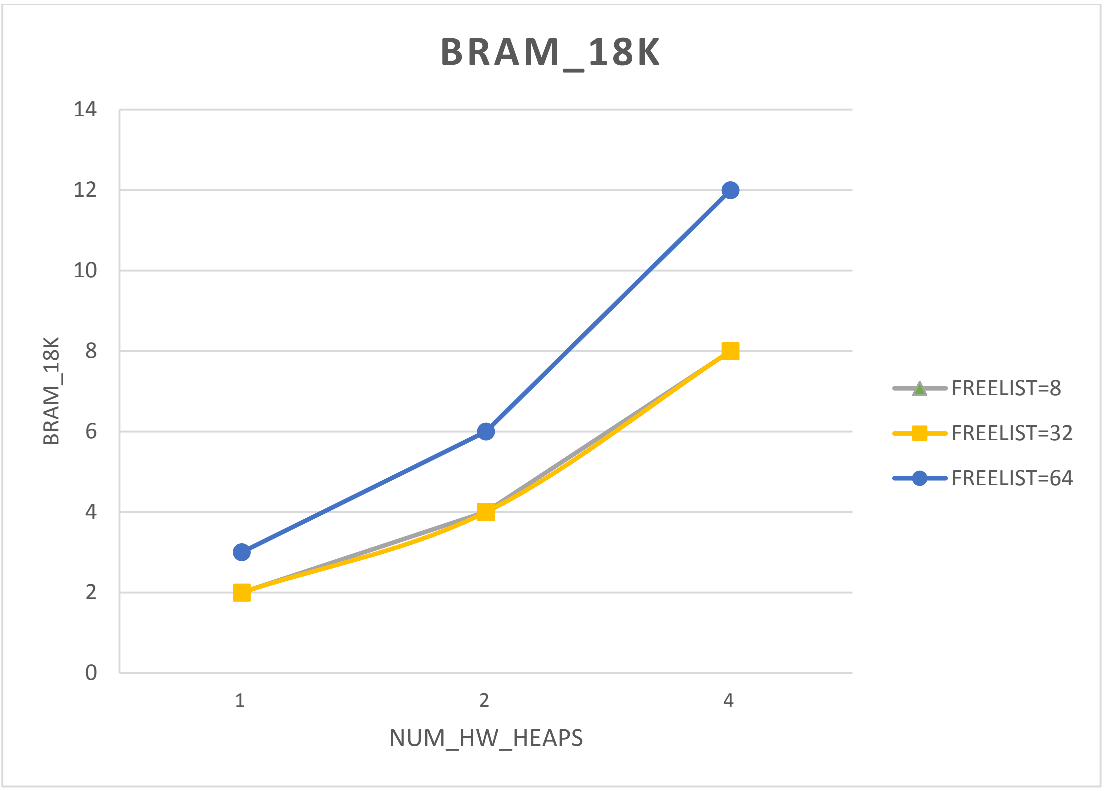


Figure 84: BRAM regarding memory heaps for Strmatch kernel

We observe that as we increase the number of memory heaps, the number of BRAMs in use increases, without notable differences between the different widths (8, 32, 64). This is reasonable, as BRAMs are the building blocks of memory heaps. Thus, the number of BRAMs in use depends only on the number of memory heaps.

Regarding Strmatch kernel, it seems that when width size is equal to 64, we need extra BRAM units. The reason for this observation concerns the mapping between virtual and physical memory. More precisely, when we introduced the DMM library, we actually proposed a structure of virtual memory where we are able to dynamically create or delete memory heaps. Thus, we initially define the number of unique addresses of the virtual memory (using the memluv depth variable) and we allocate this memory segment on the available BRAMs of the FPGA. Then we can manage this memory block as we want, regarding the accelerators' needs. We may add or delete memory heaps, or change the width of freelist array. Regarding figure 83, the need for extra BRAM appears when synthesis tool allocates the virtual memory that we selected to use, with a specific size, to the physical memory of the board. During this process, we may face some memory mismatches or incompatibilities that may cause the raise of BRAMs in use.

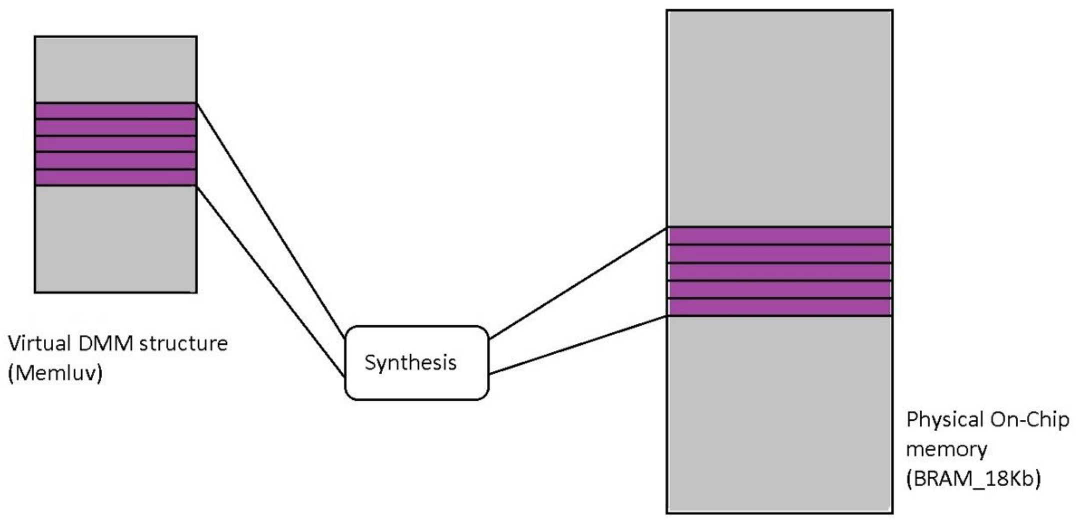


Figure 85: Mapping of virtual to physical memory

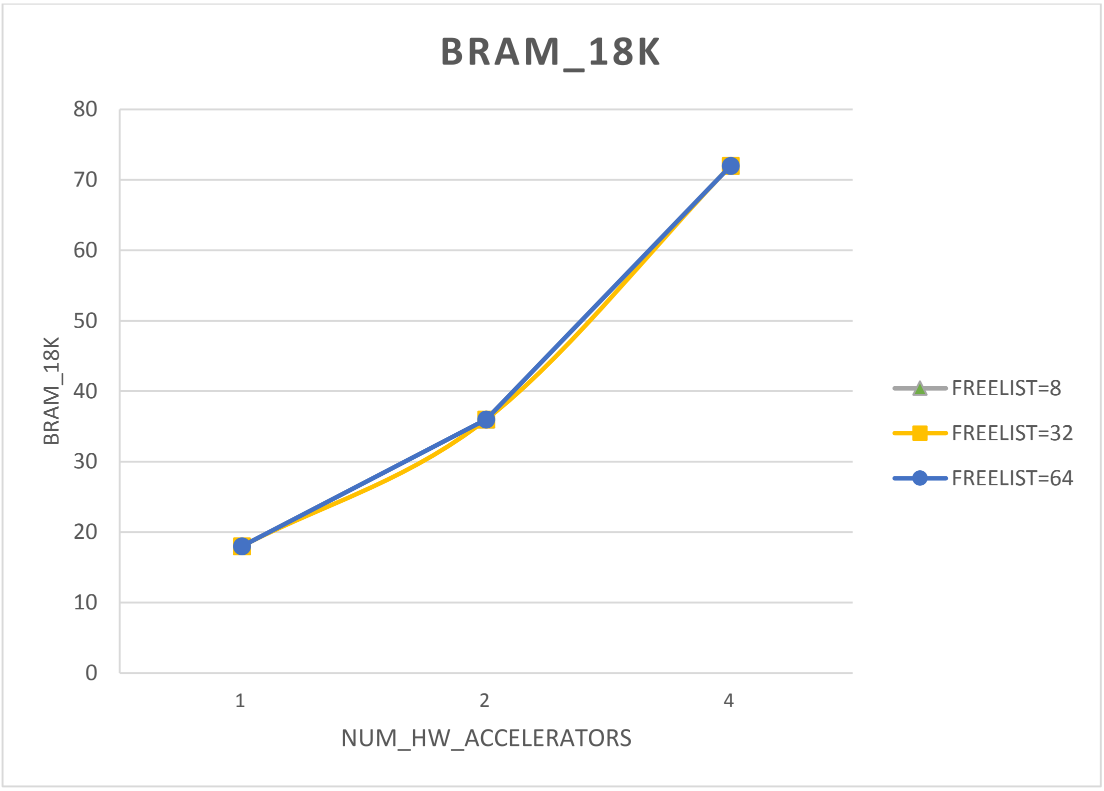


Figure 86: BRAM regarding accelerators for Histogram kernel

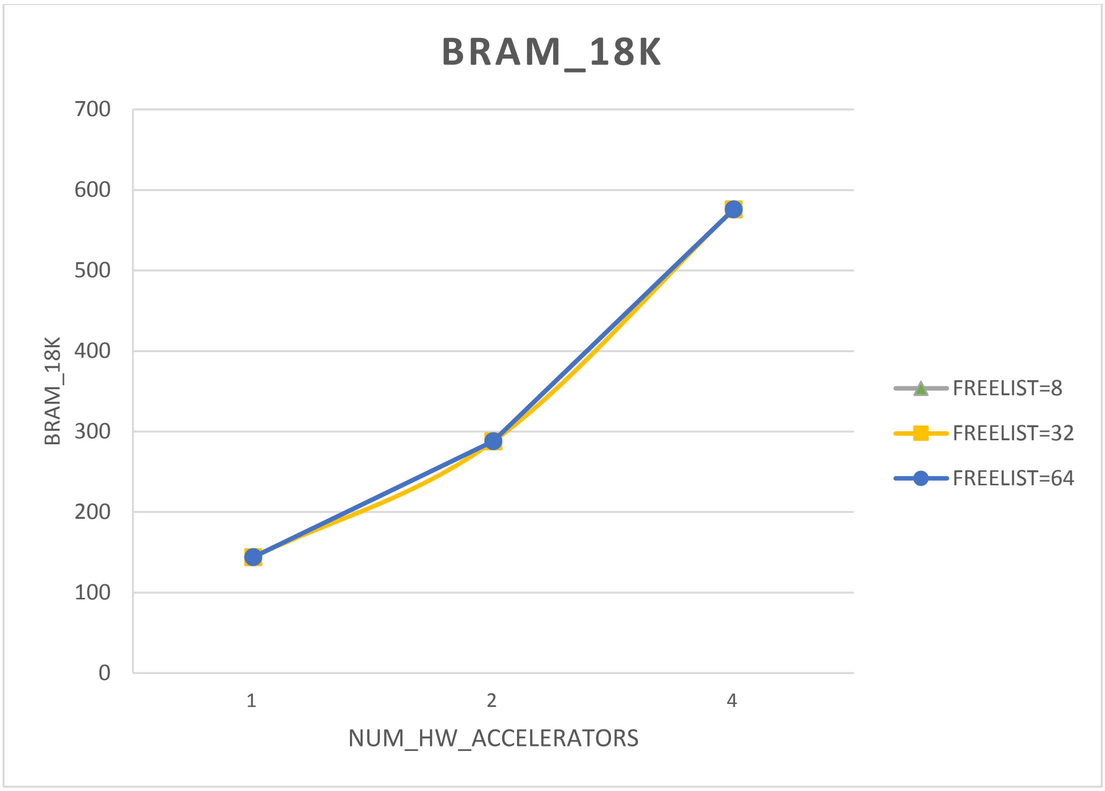


Figure 87: BRAM regarding accelerators for MMUL kernel

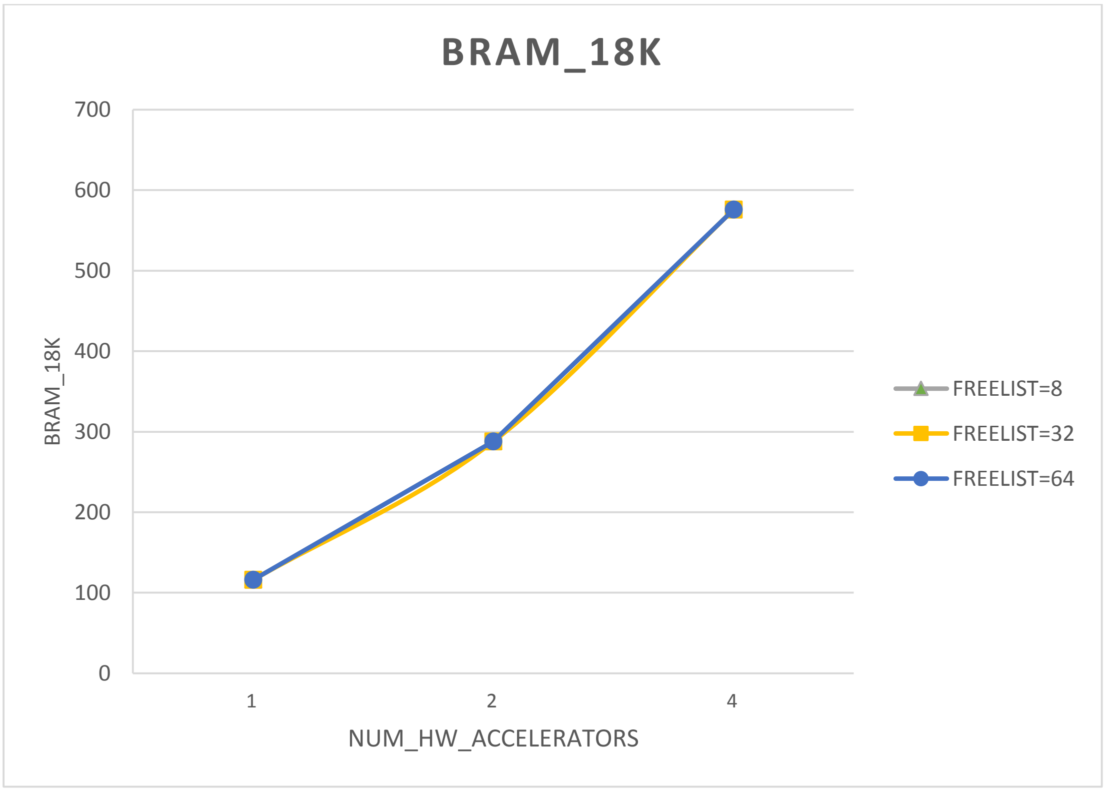


Figure 88: BRAM regarding accelerators for PCA kernel

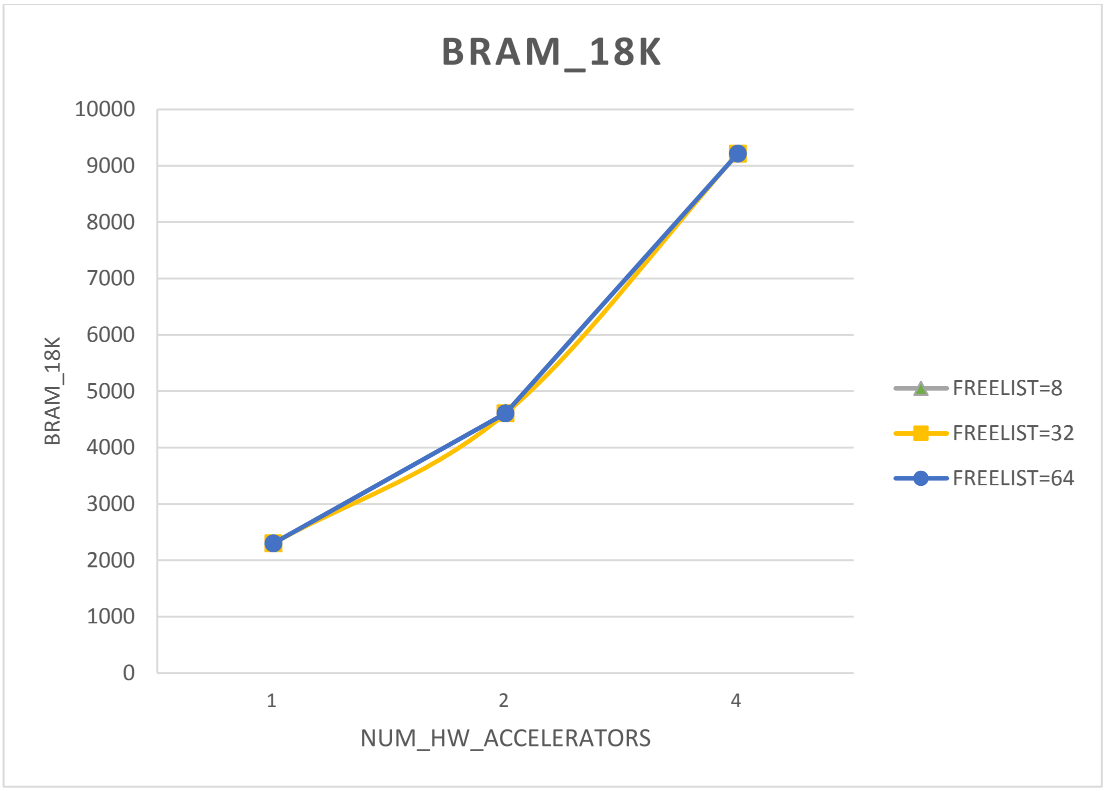


Figure 89: BRAM regarding accelerators for Kmeans kernel

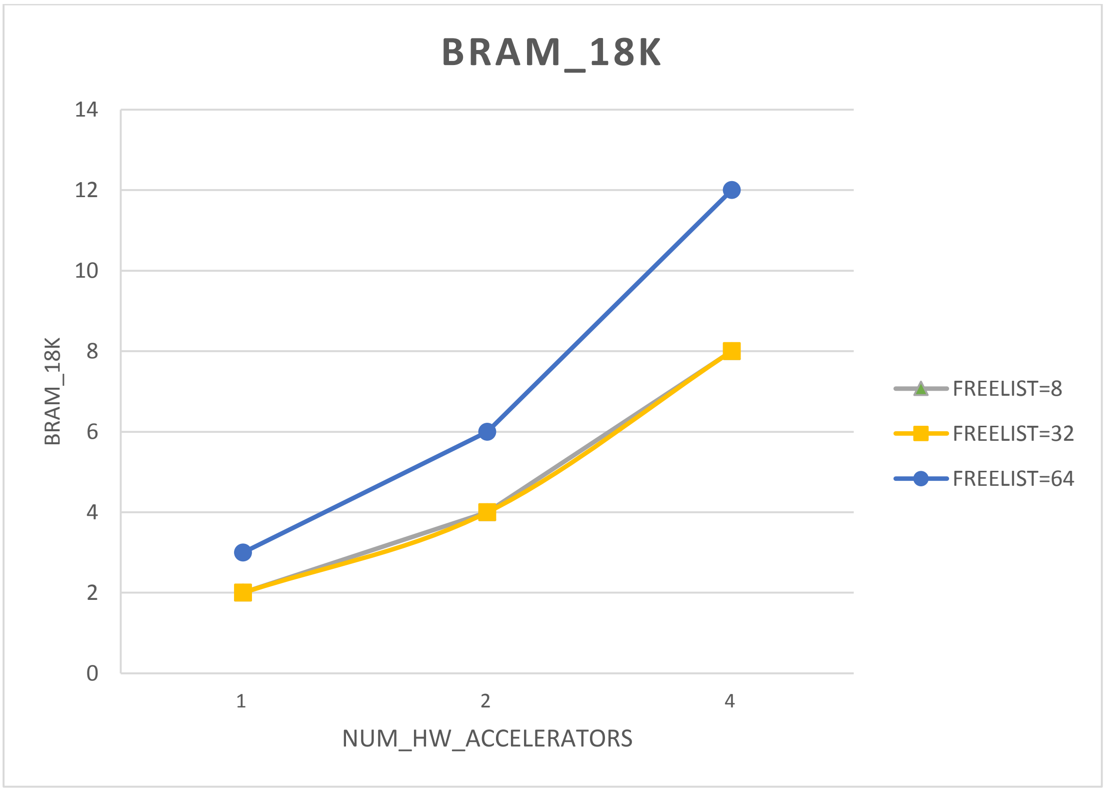


Figure 90: BRAM regarding accelerators for Strmatch kernel

As we mentioned before, as we increase the number of accelerators and heaps, the number of BRAMs in use increases. The difference regarding Strmatch is due to the same thing as before (mapping of virtual to physical memory).

4.2.3 DSP

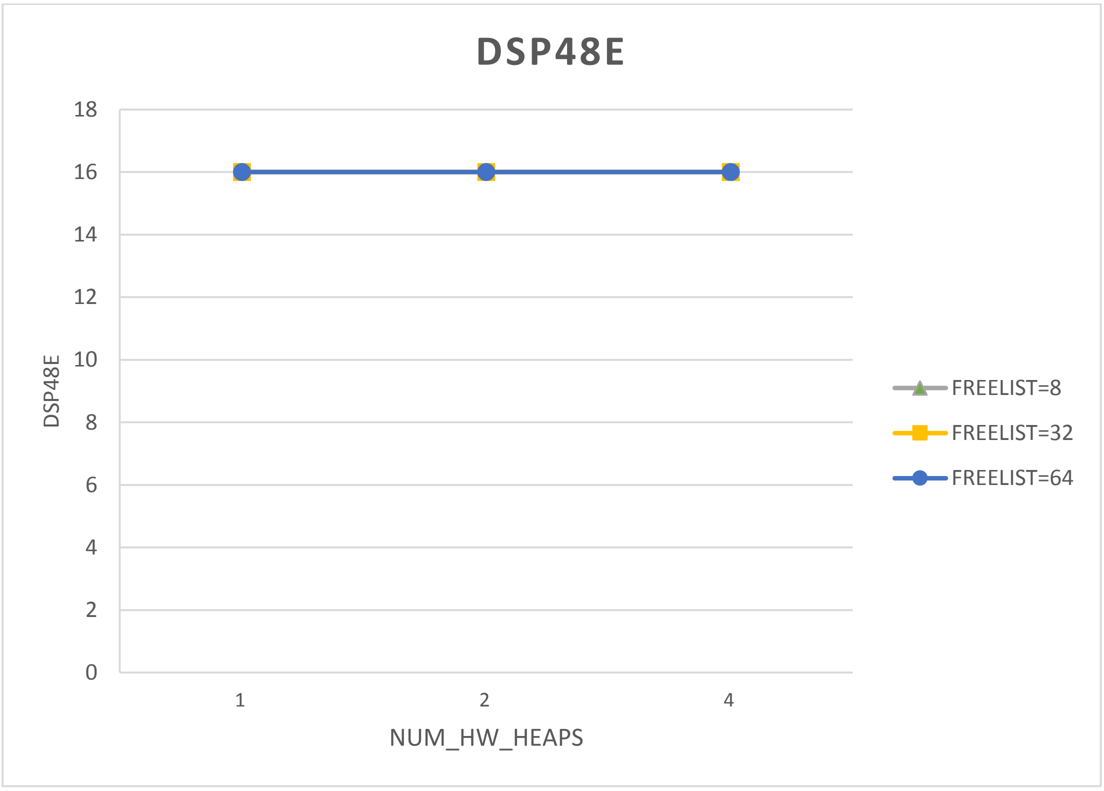


Figure 91: DSP regarding memory heaps for Histogram kernel

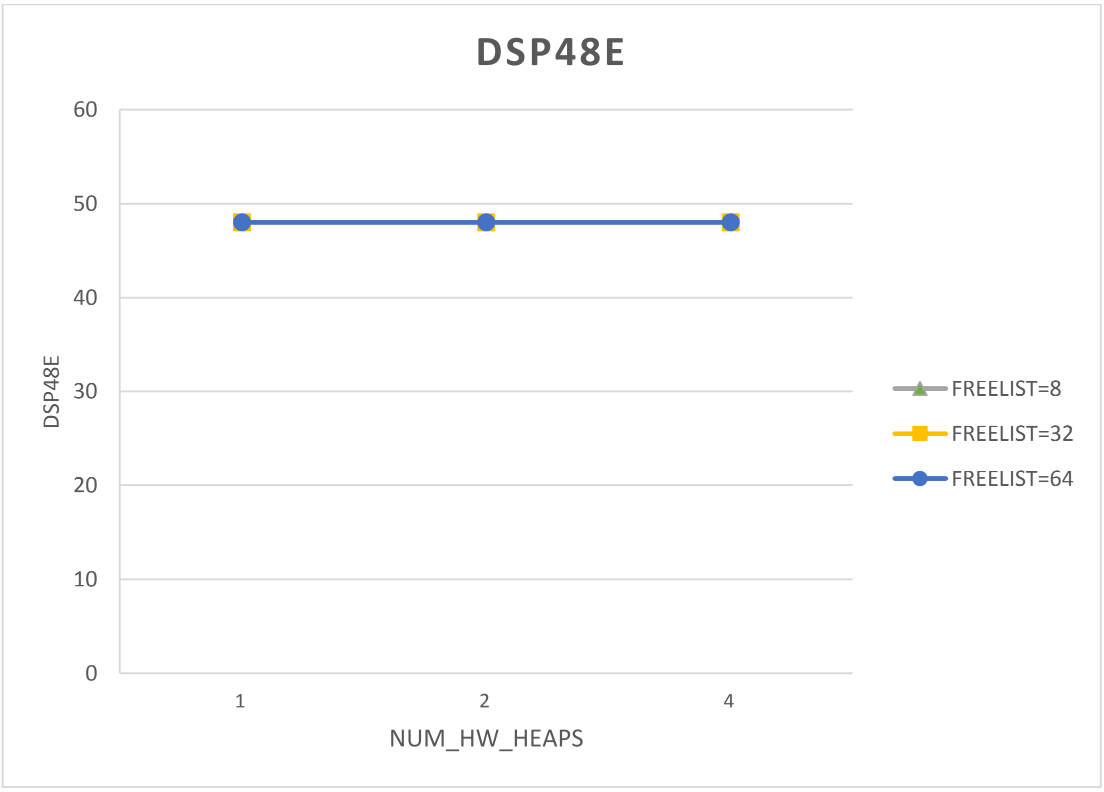


Figure 92: DSP regarding memory heaps for MMUL kernel

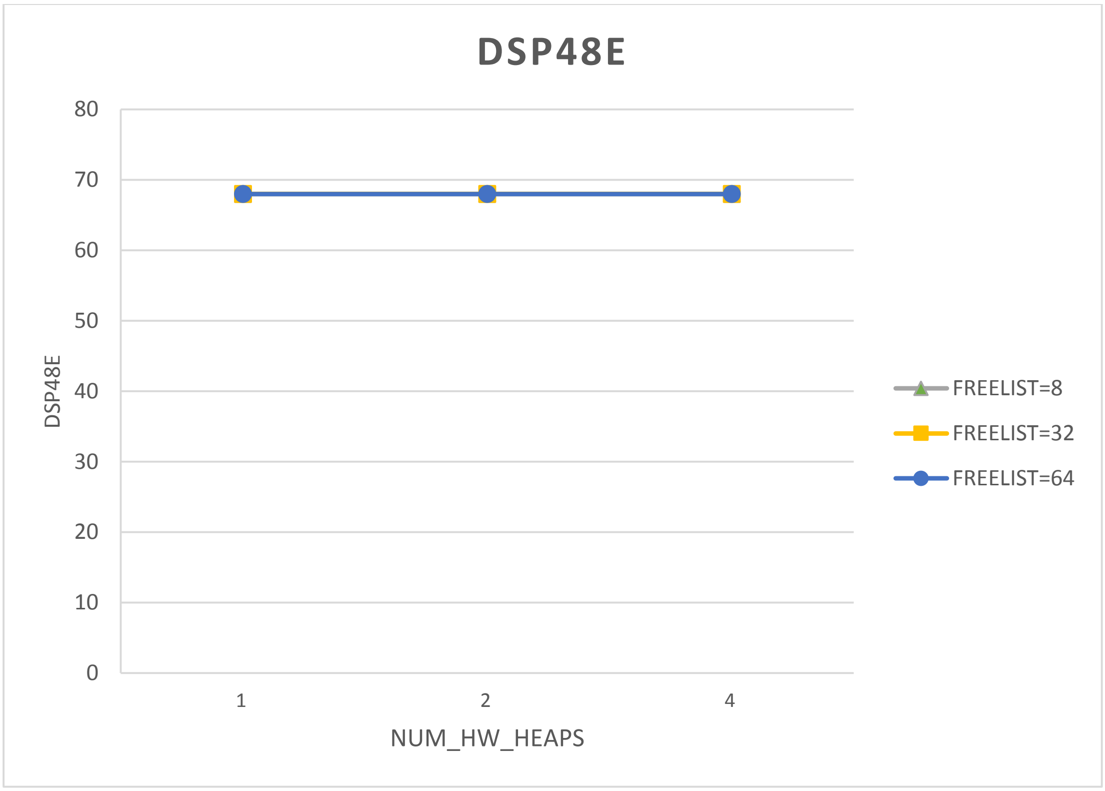


Figure 93: DSP regarding memory heaps for PCA kernel

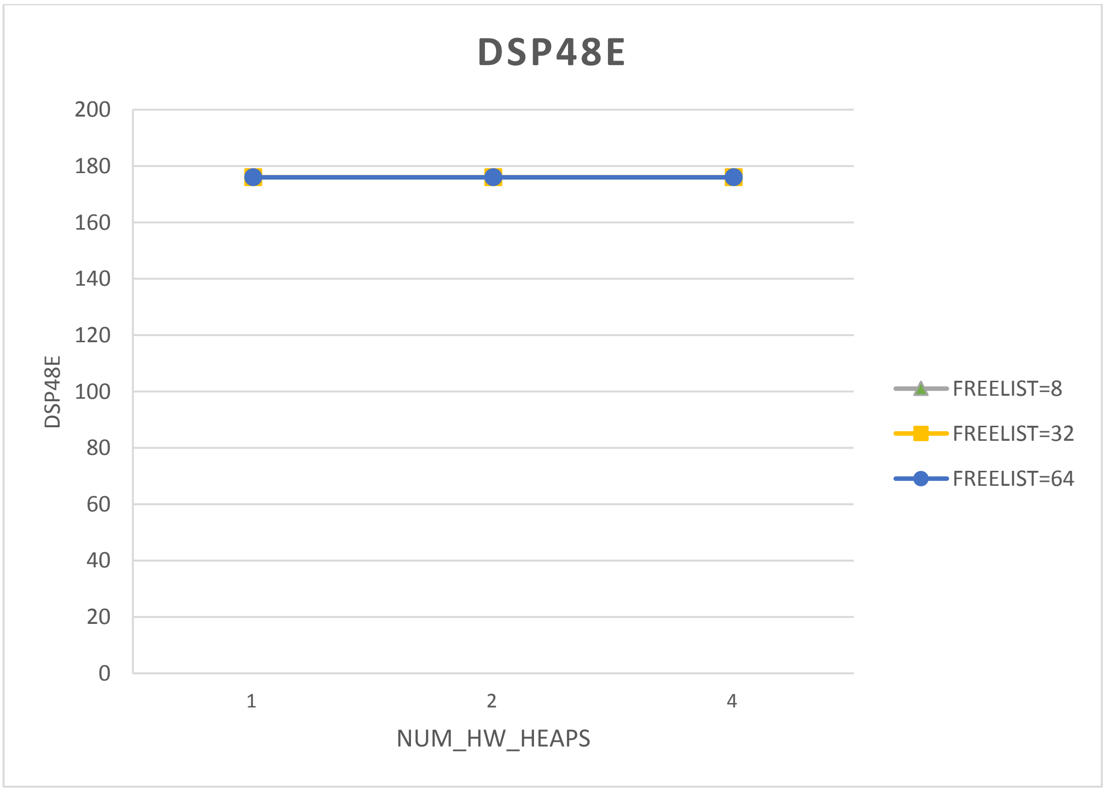


Figure 94: DSP regarding memory heaps for Kmeans kernel

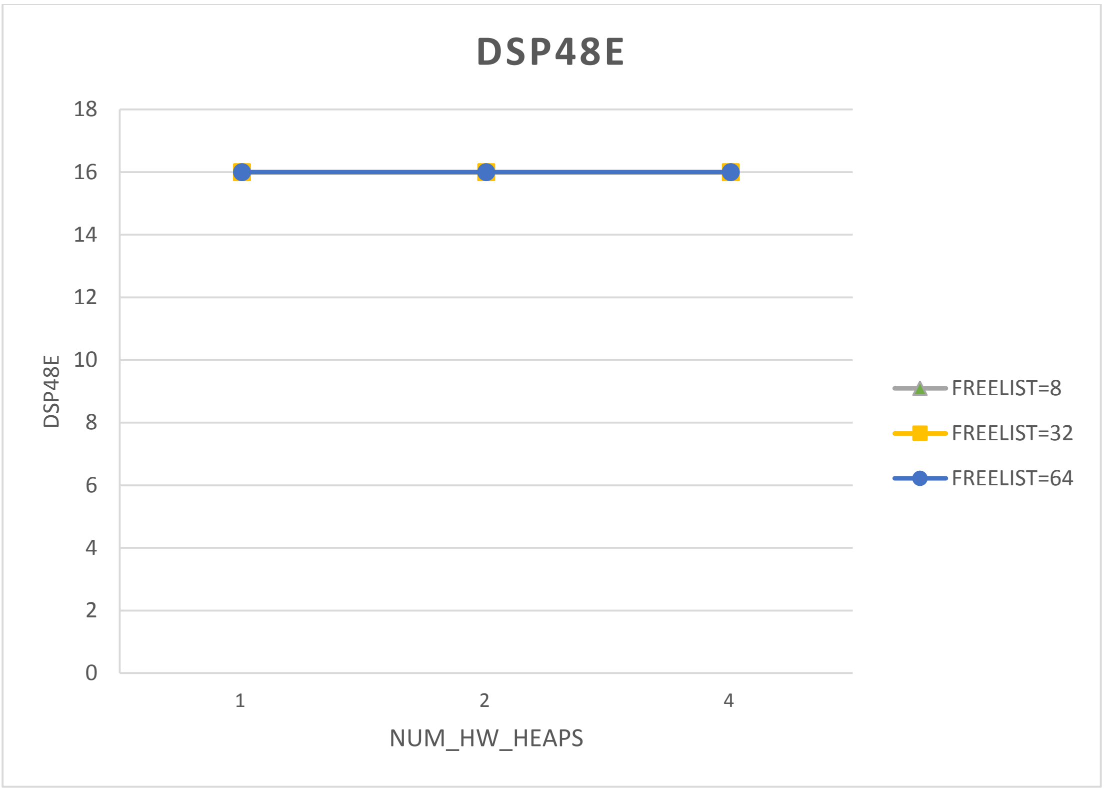


Figure 95: DSP regarding memory heaps for Strmatch kernel

We observe that changing the width of freelist array has no effect on the use of DSPs. This is happening because DSP units are irrelevant with the functionality of freelist array.

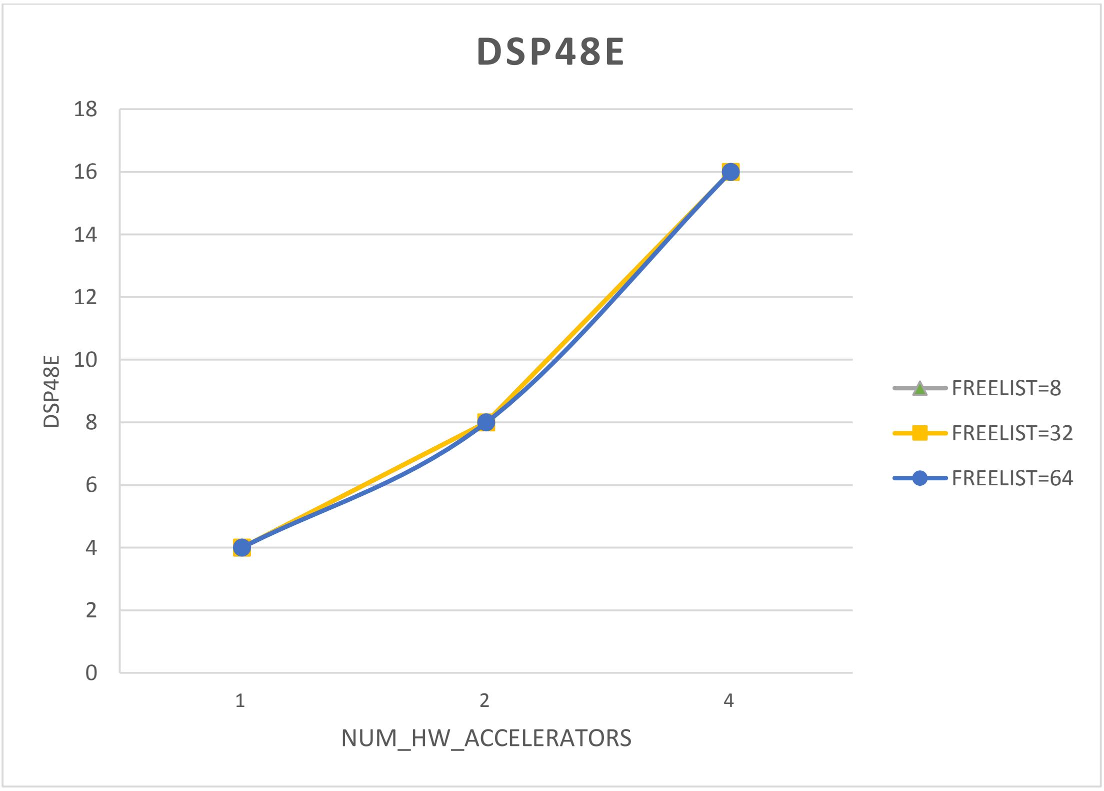


Figure 96: DSP regarding accelerators for Histogram kernel

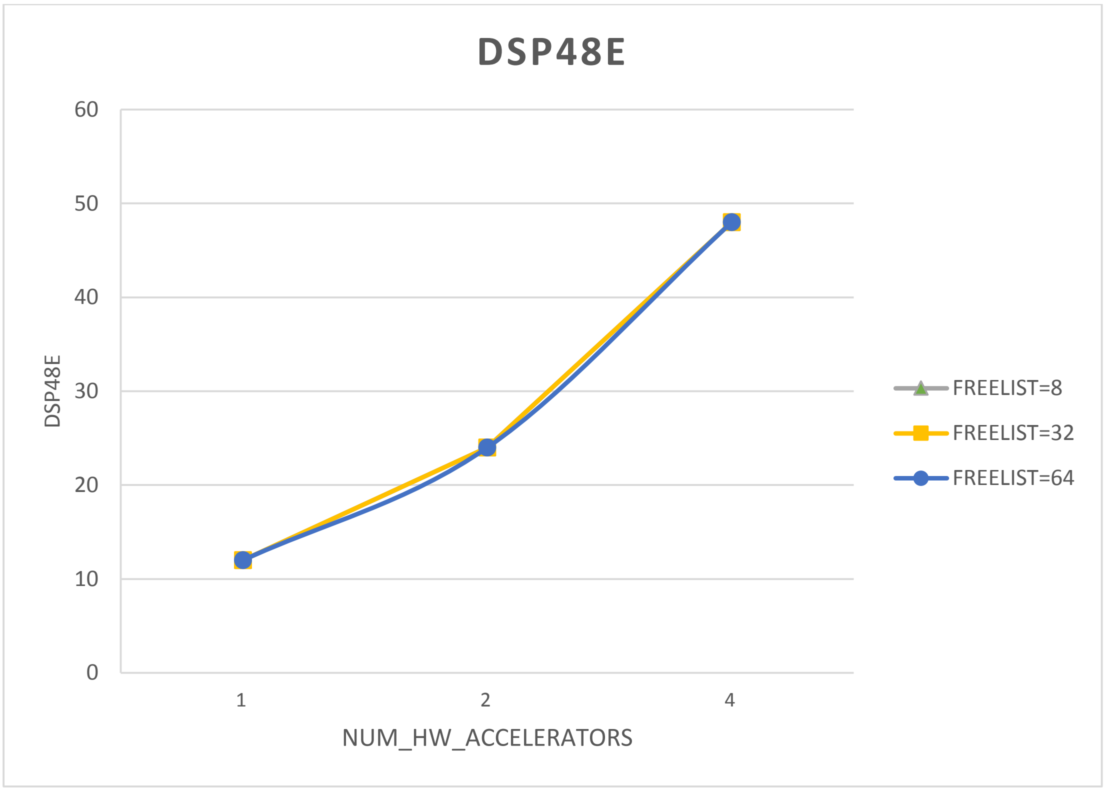


Figure 97: DSP regarding accelerators for MMUL kernel

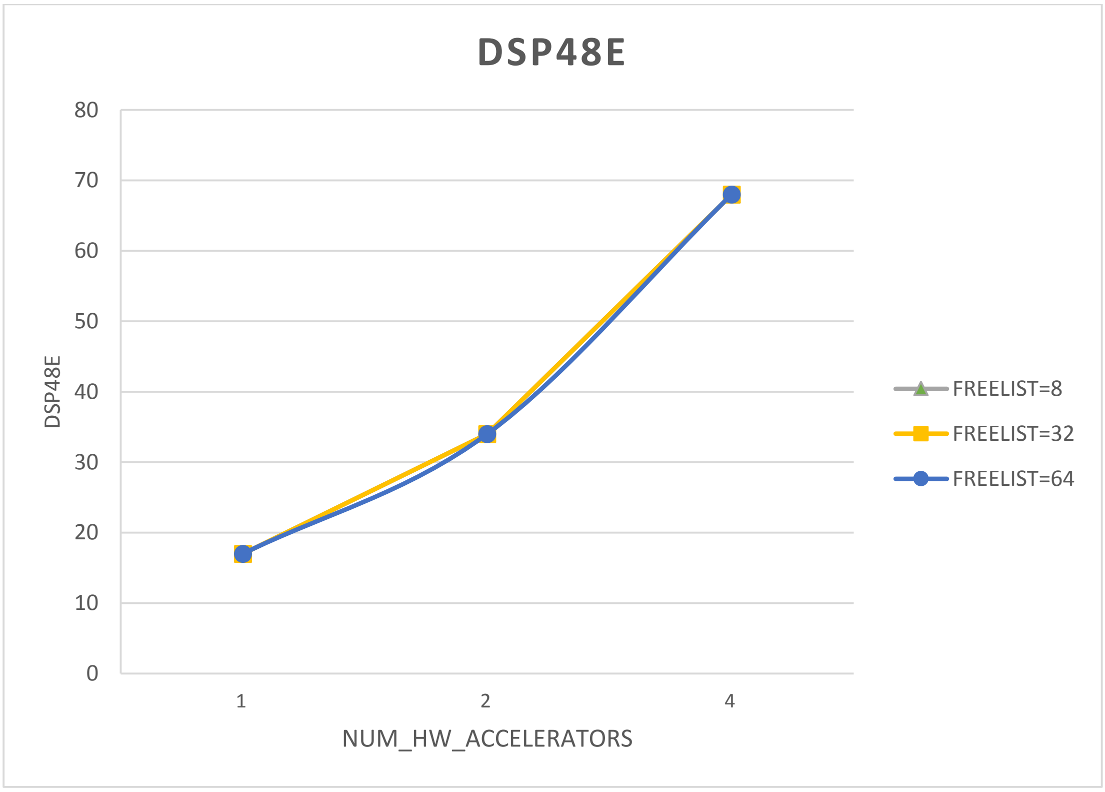


Figure 98: DSP regarding accelerators for PCA kernel

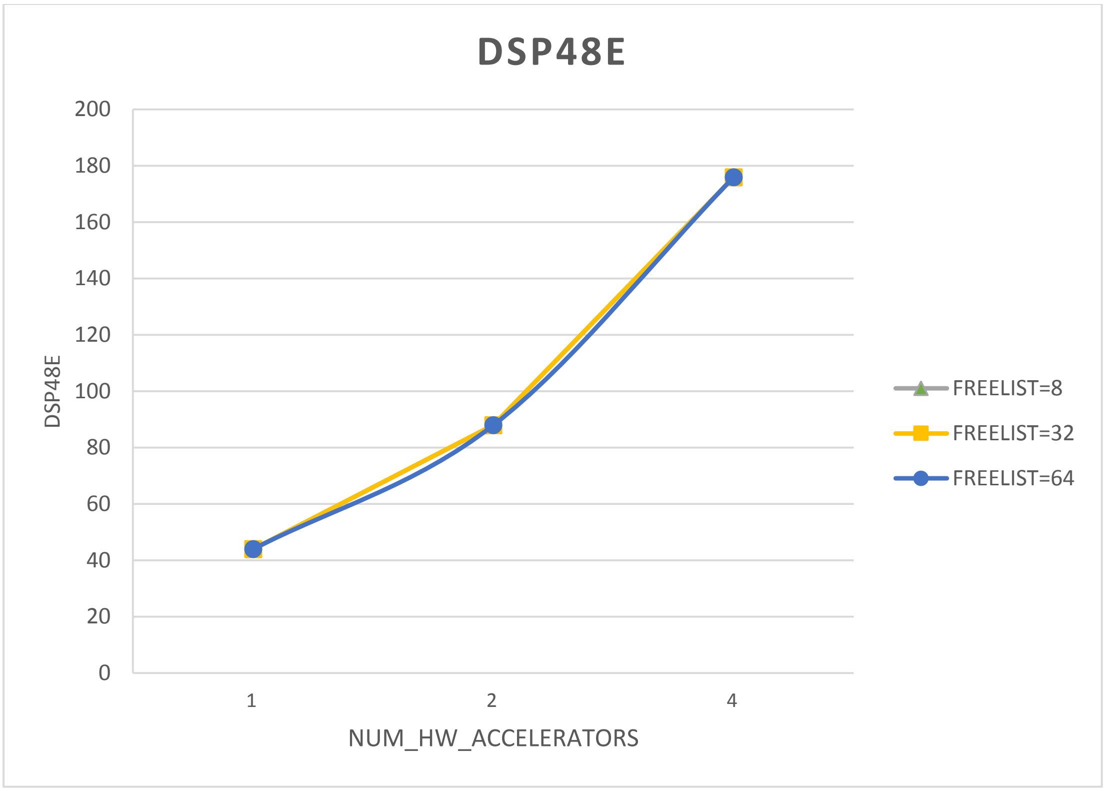


Figure 99: DSP regarding accelerators for Kmeans kernel

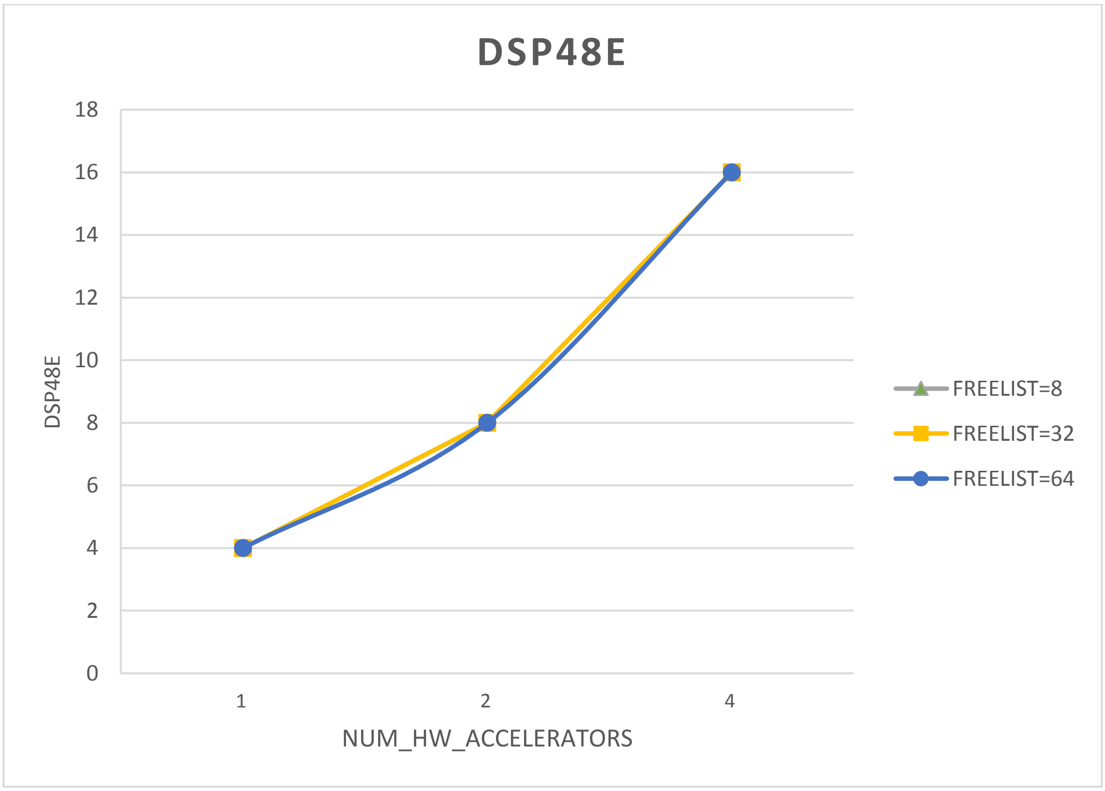


Figure 100: DSP regarding accelerators for Strmatch kernel

As we mentioned before, DSP units increase because of the escalation of the system (by adding more accelerators and heaps) and are independent of the width of the freelist array.

4.2.4 FF

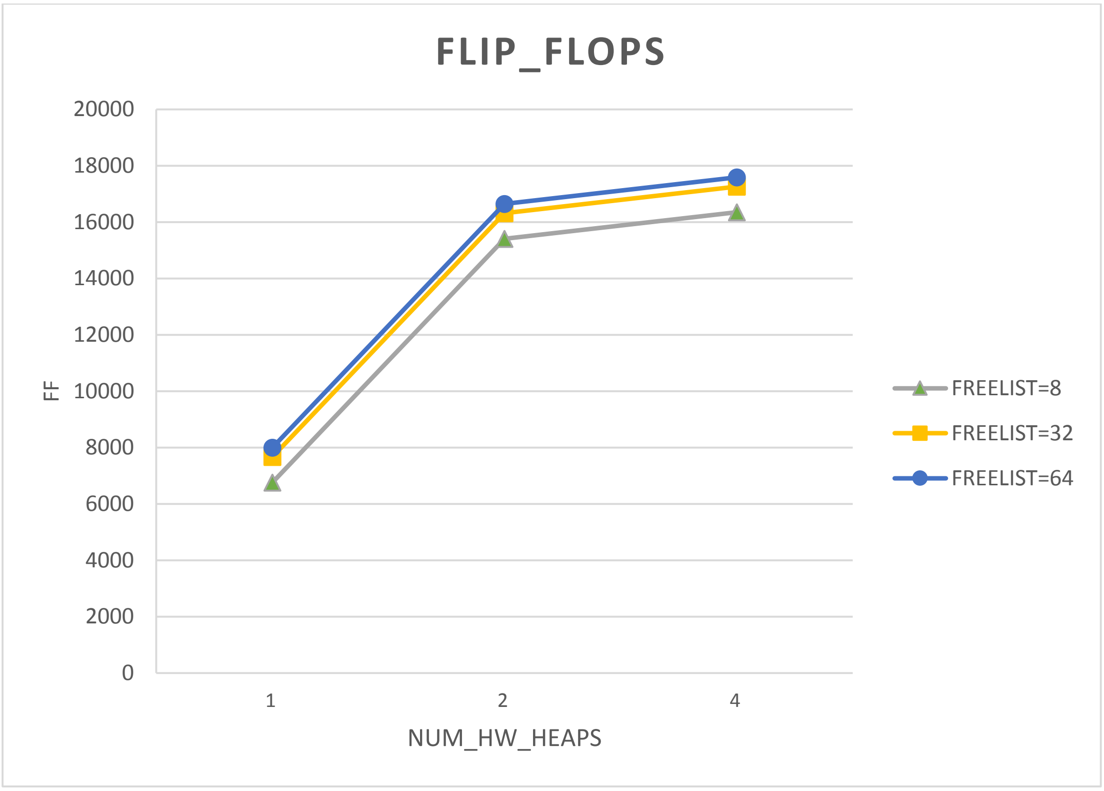


Figure 101: FF regarding memory heaps for Histogram kernel

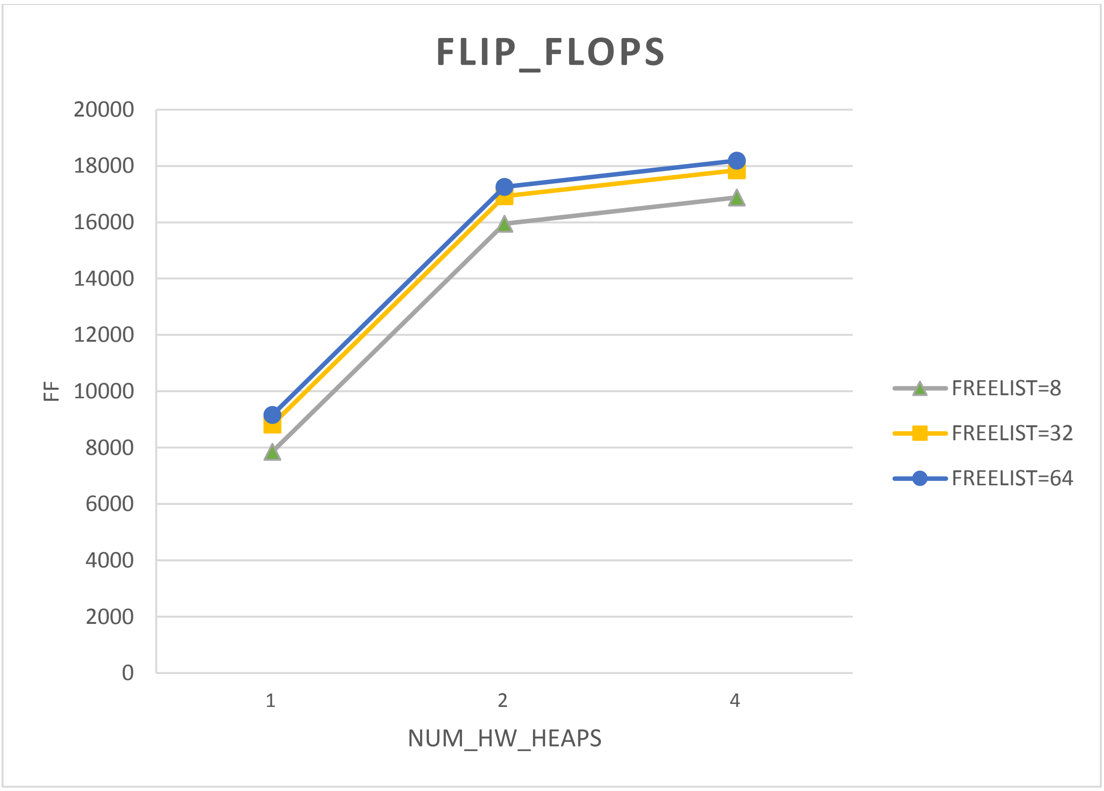


Figure 102: FF regarding memory heaps for MMUL kernel

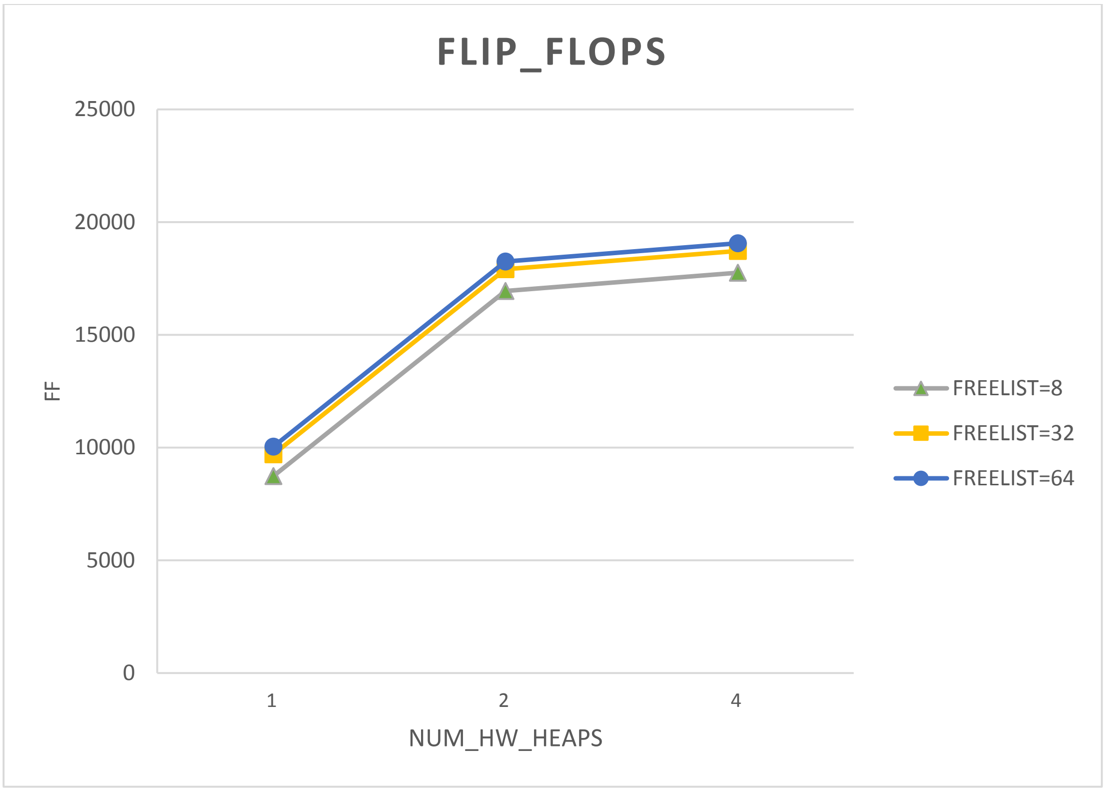


Figure 103: FF regarding memory heaps for PCA kernel

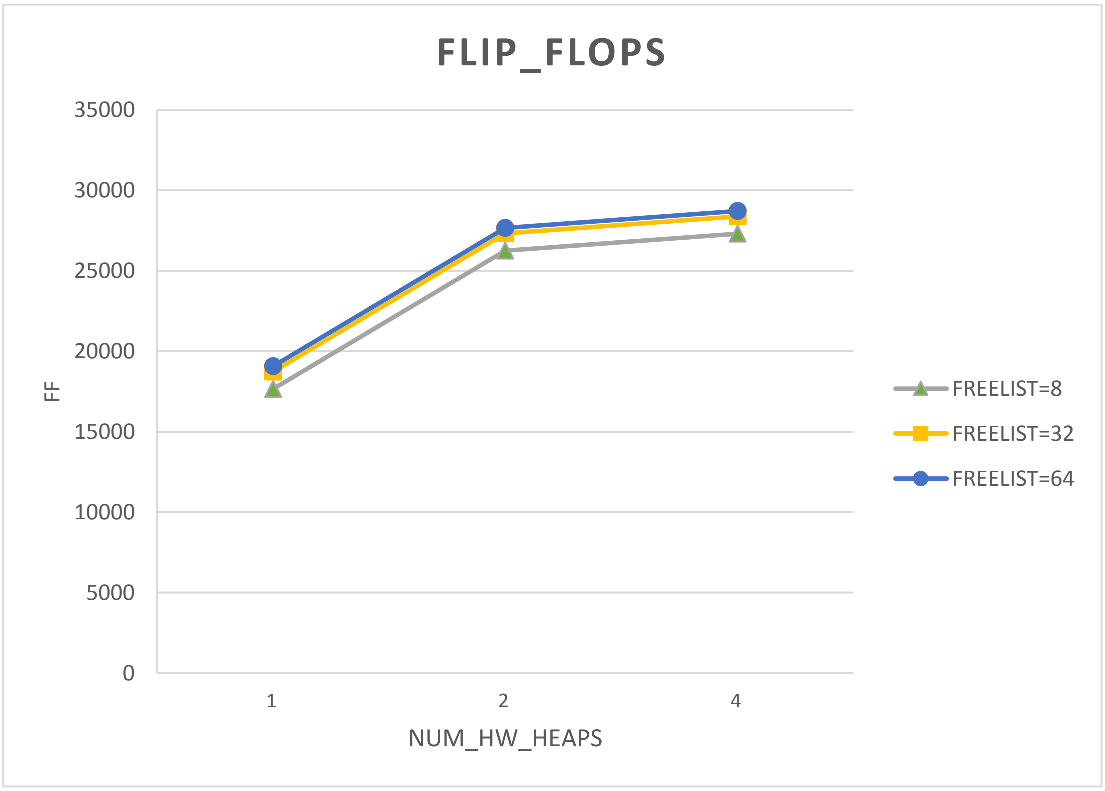


Figure 104: FF regarding memory heaps for Kmeans kernel

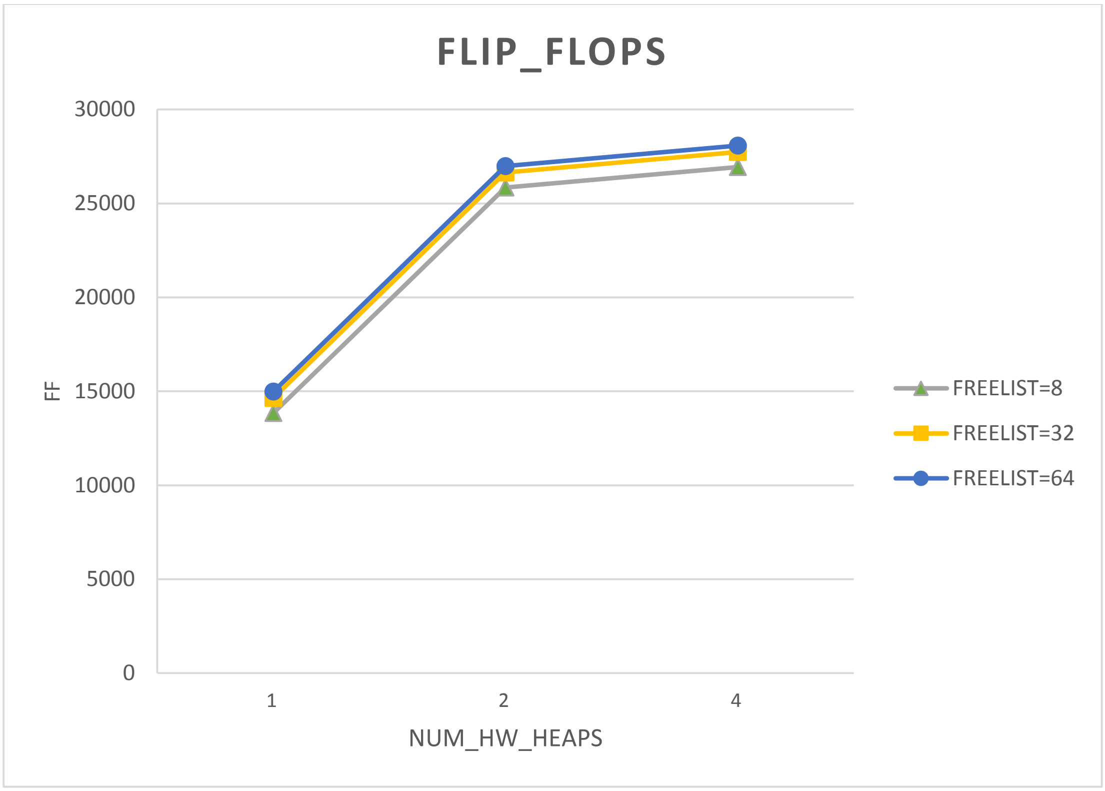


Figure 105: FF regarding memory heaps for Strmatch kernel

We notice that the width 64 of freelist array has an impact on the number of FFs in use. This is reasonable as FFs are used for logical masks, for the allocation of extra variables or shifts into registers, needed for the functionality of freelist array. As we increase the width of the array, we need bigger masks, maybe more extra variables for computations and more shifts. This is why we observe that the number of FFs in use depends on the width of freelist array.

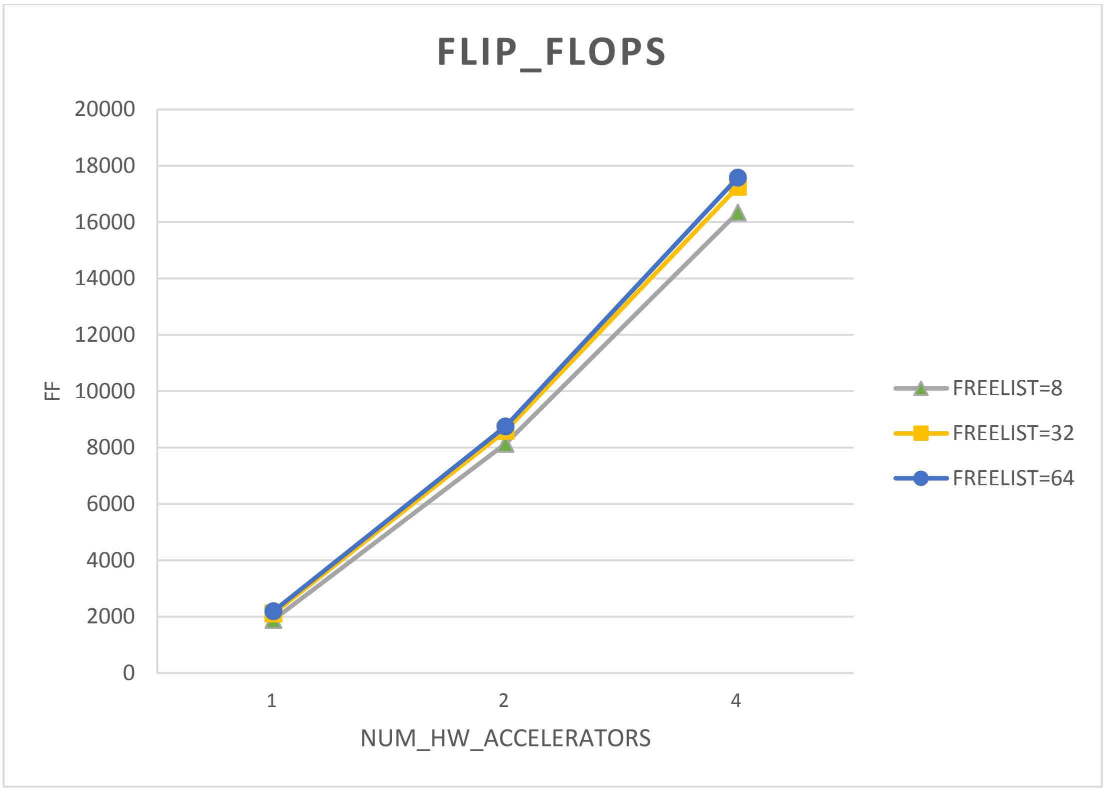


Figure 106: FF regarding accelerators for Histogram kernel

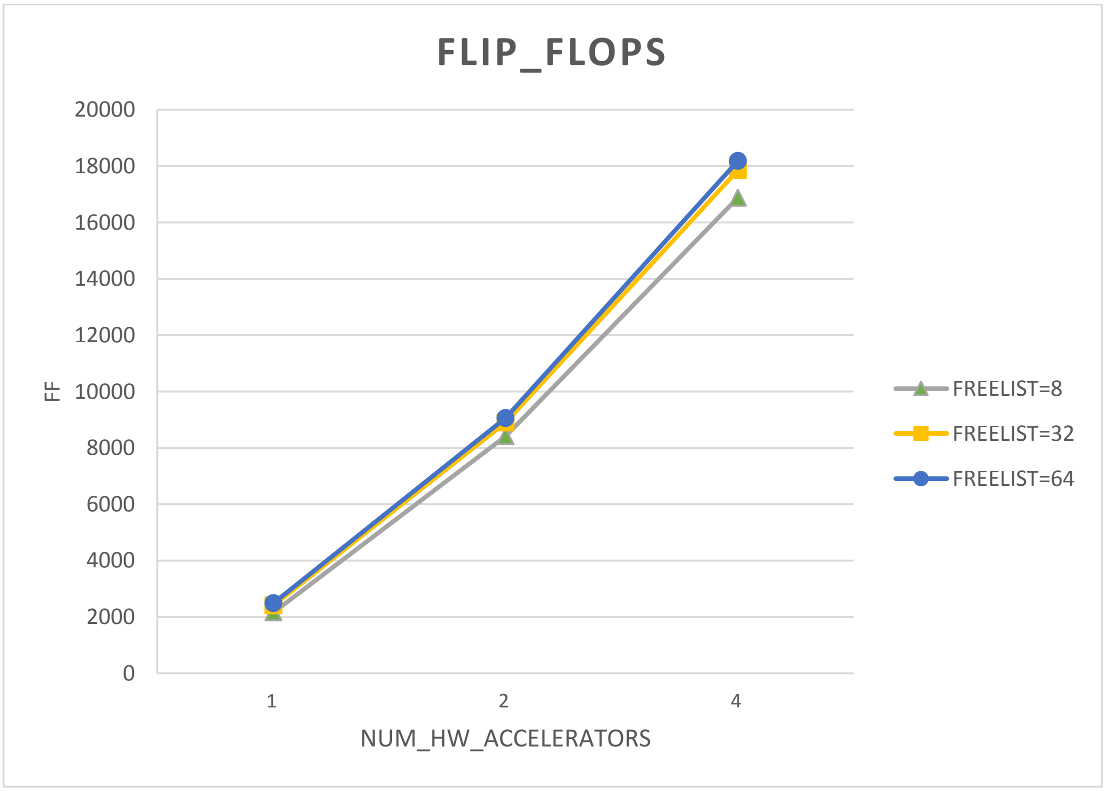


Figure 107: FF regarding accelerators for MMUL kernel

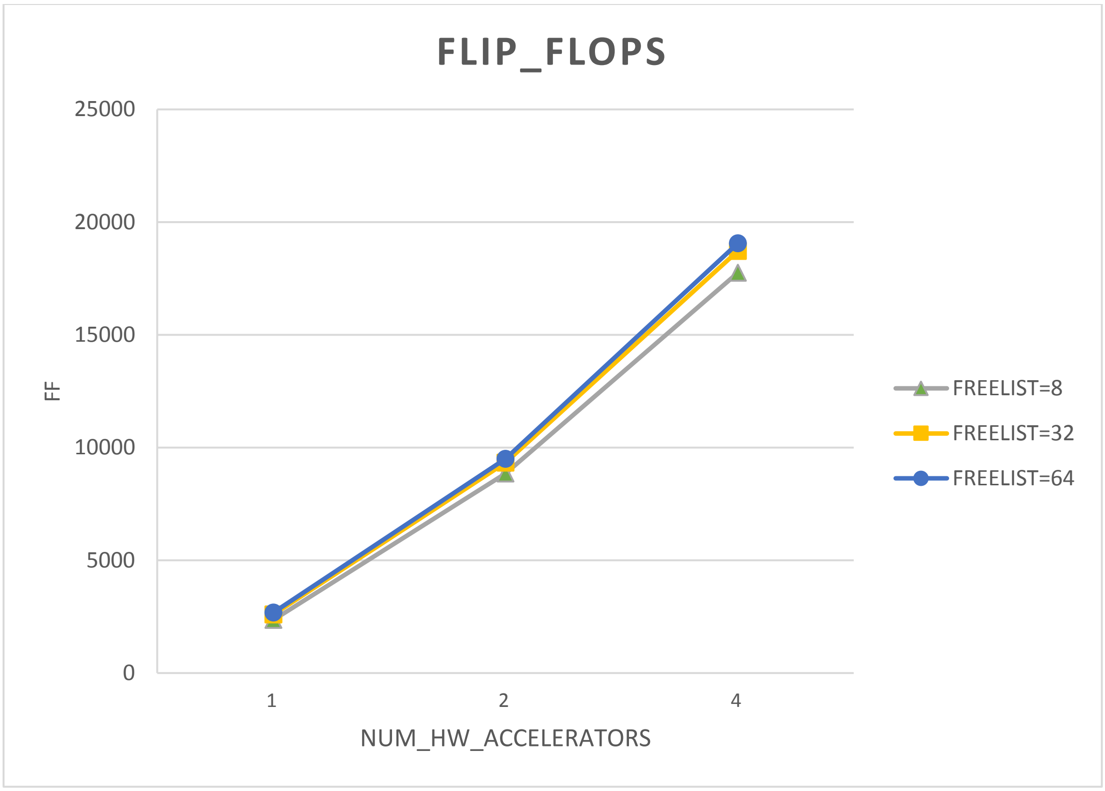


Figure 108: FF regarding accelerators for PCA kernel

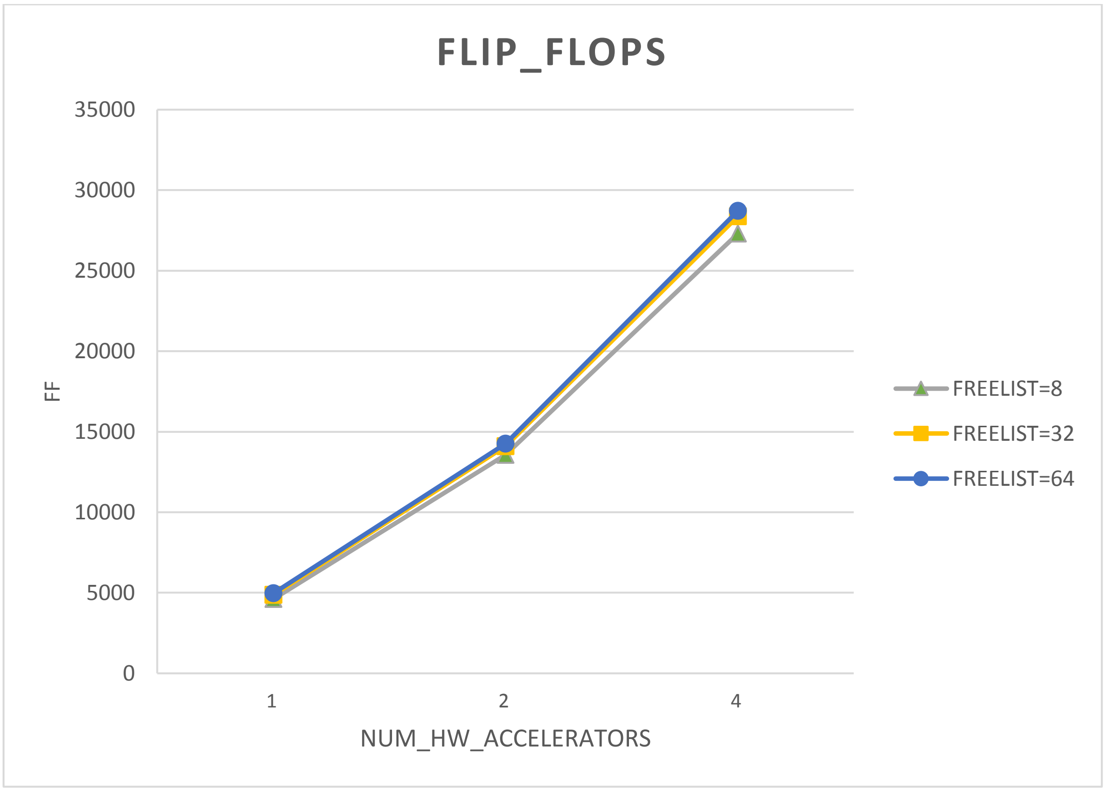


Figure 109: FF regarding accelerators for Kmeans kernel

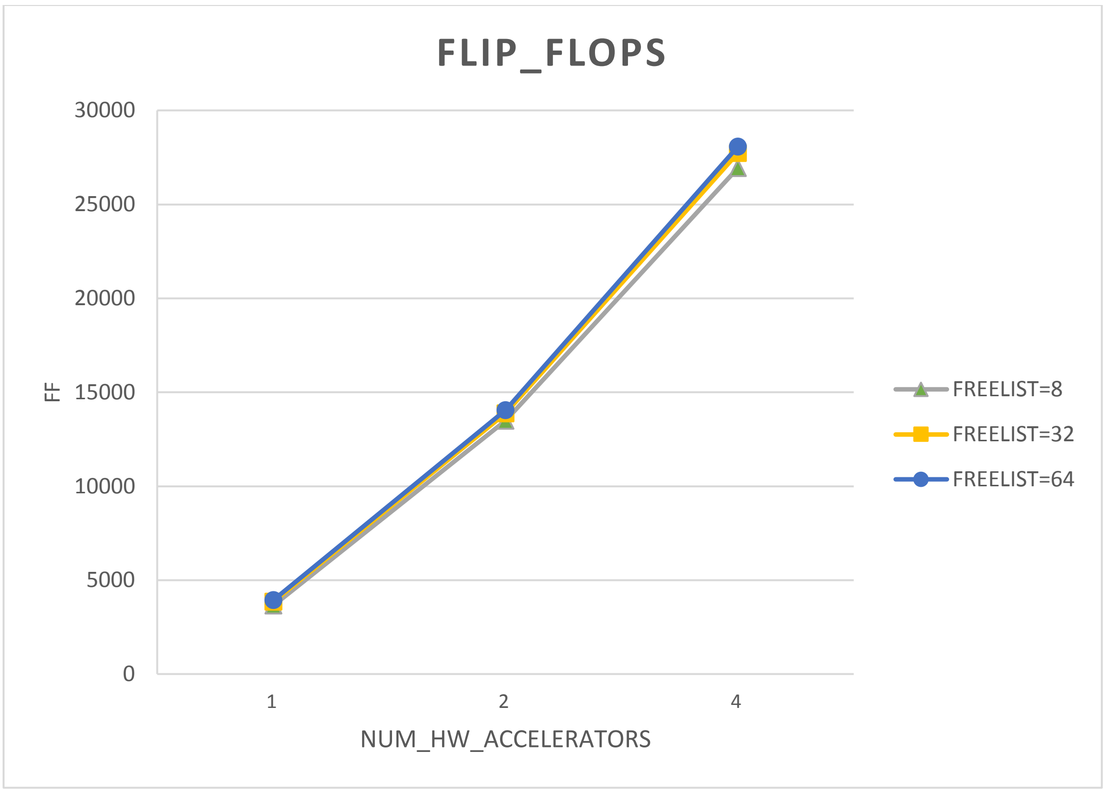


Figure 110: FF regarding accelerators for Strmatch kernel

As we add more accelerators and heaps, the use of FFs increases. Also, from the above figures, it is clear that (as we mentioned before) the FFs in use depend on the width of freelist array.

4.2.5 LUT

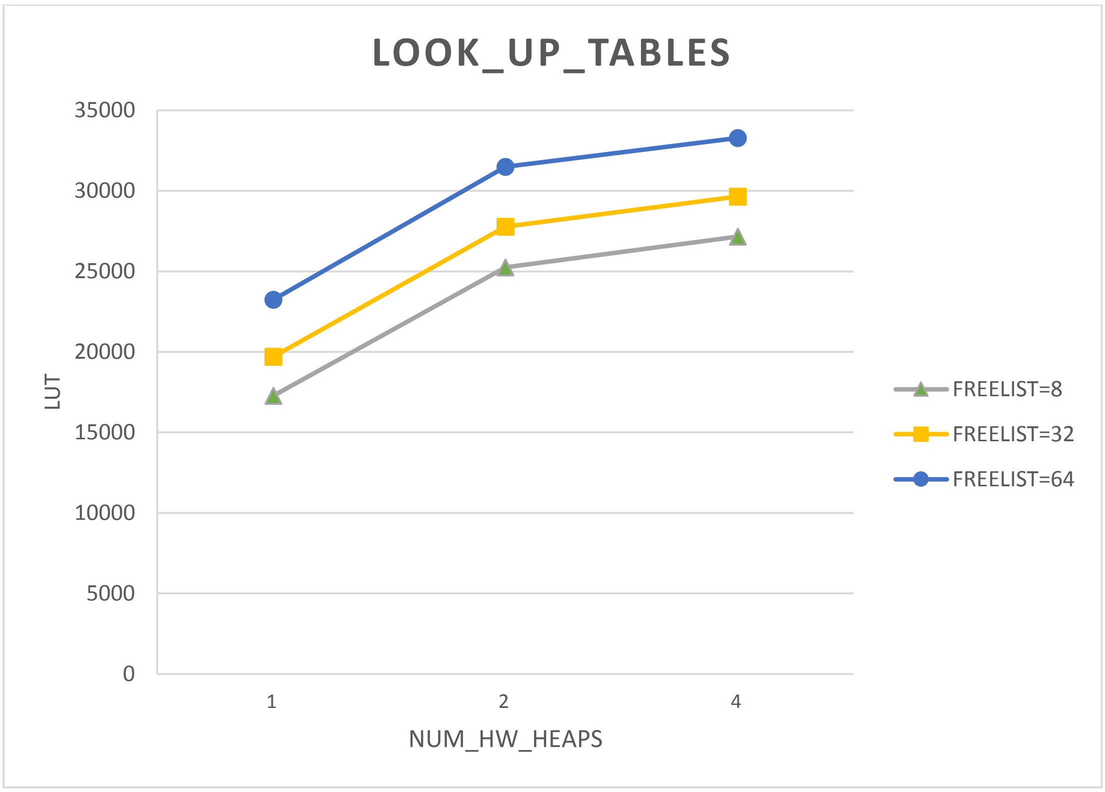


Figure 111: LUT regarding memory heaps for Histogram kernel

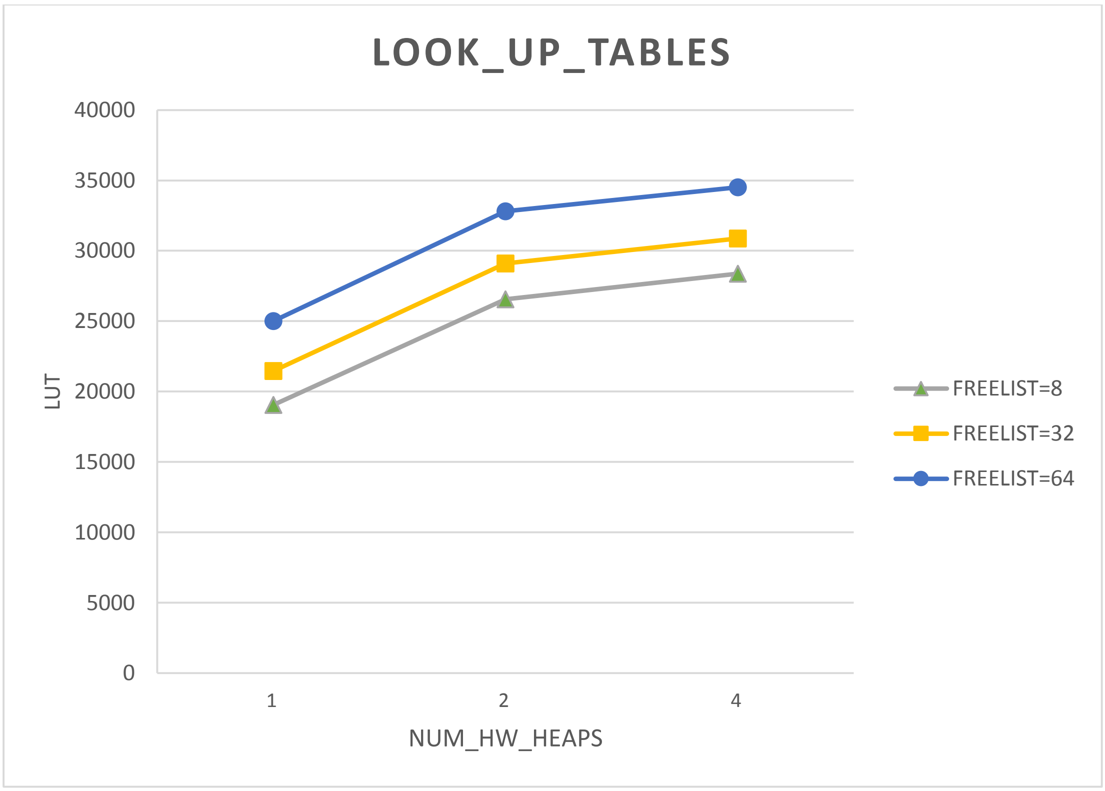


Figure 112: LUT regarding memory heaps for MMUL kernel

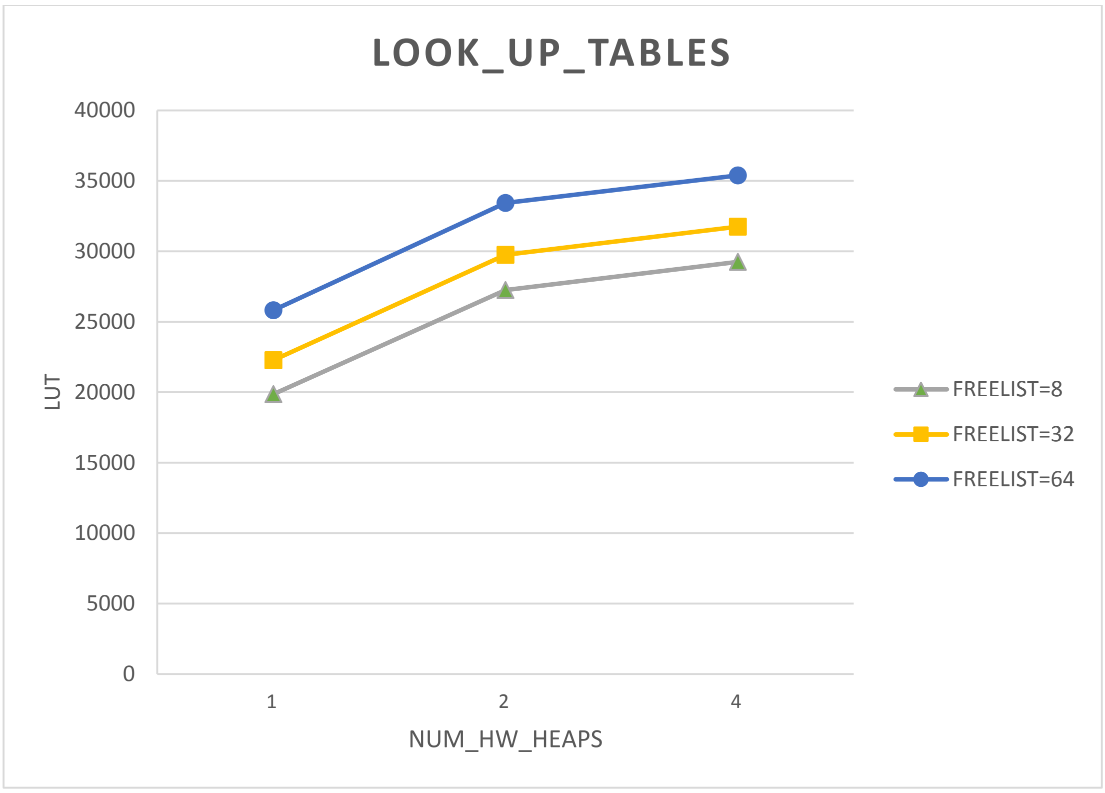


Figure 113: LUT regarding memory heaps for PCA kernel

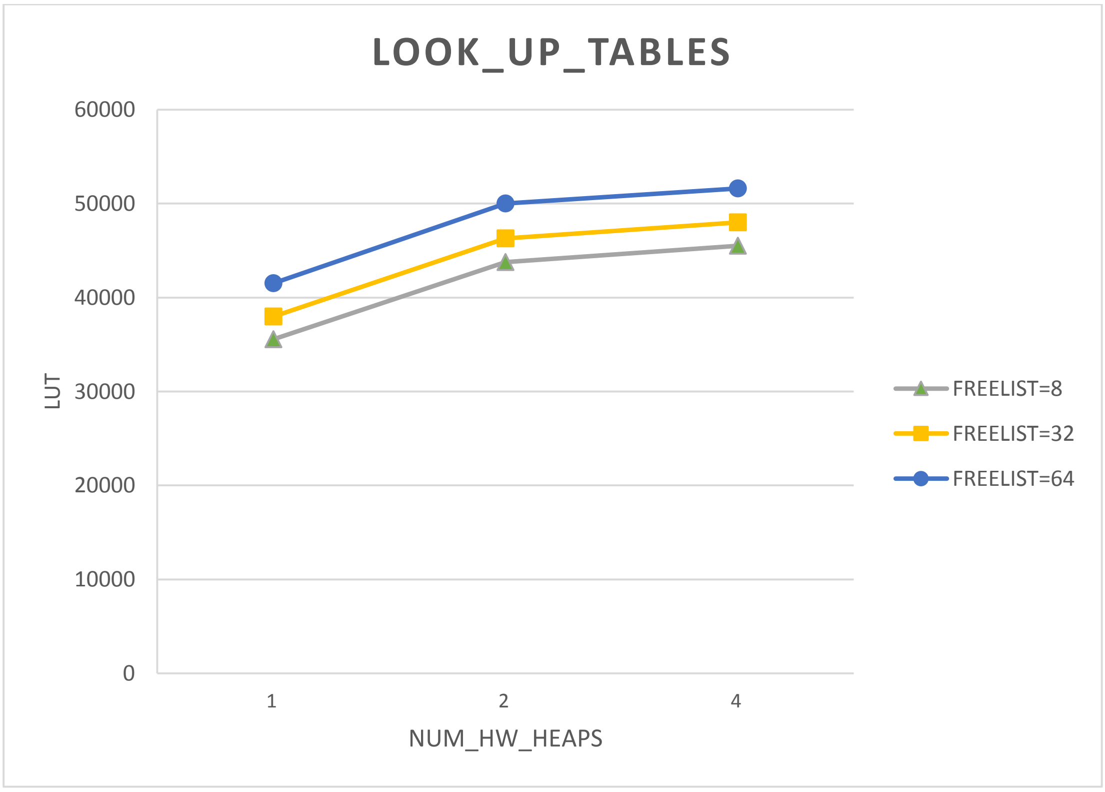


Figure 114: LUT regarding memory heaps for Kmeans kernel

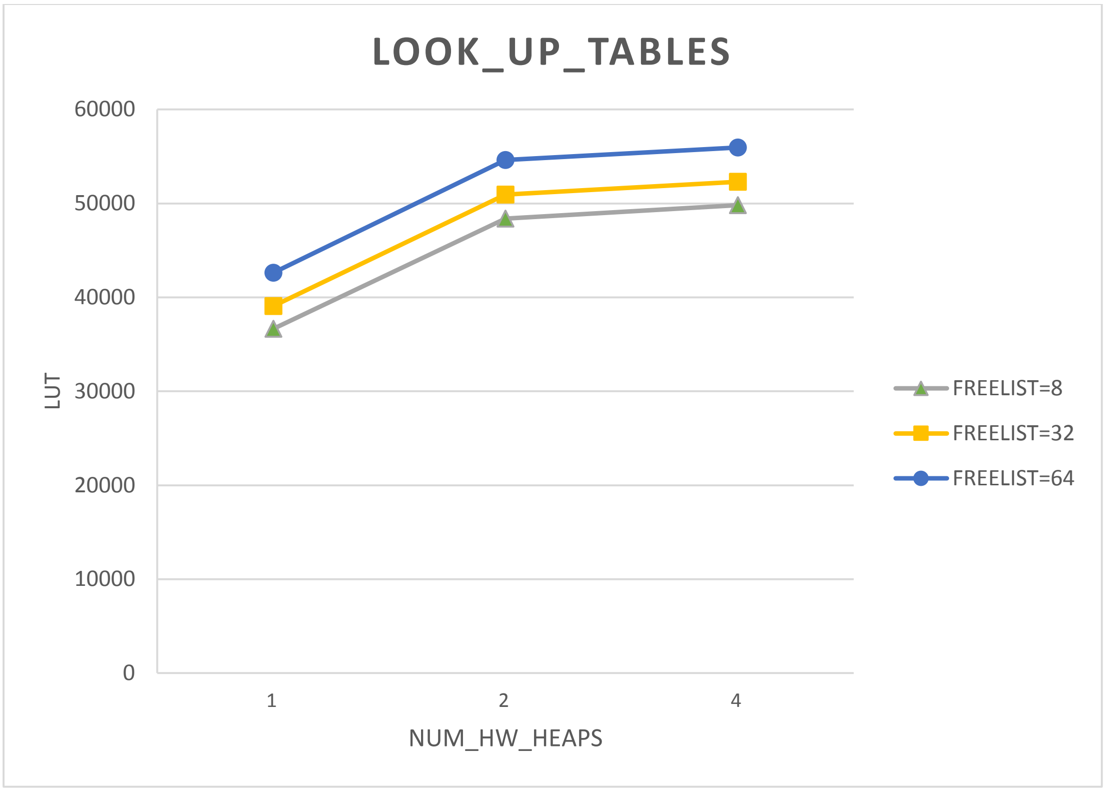


Figure 115: LUT regarding memory heaps for Strmatch kernel

We notice that the width 64 of freelist array has an impact on the number of LUTs in use. This is reasonable as LUTs are used for the same operations as FFs (logical masks, allocation of extra variables, shifts into registers) and they are needed for the functionality of freelist array. As we increase the width of the array, the number of LUTs in use raises. Thus, the number of LUTs in use depends on the width of freelist array.

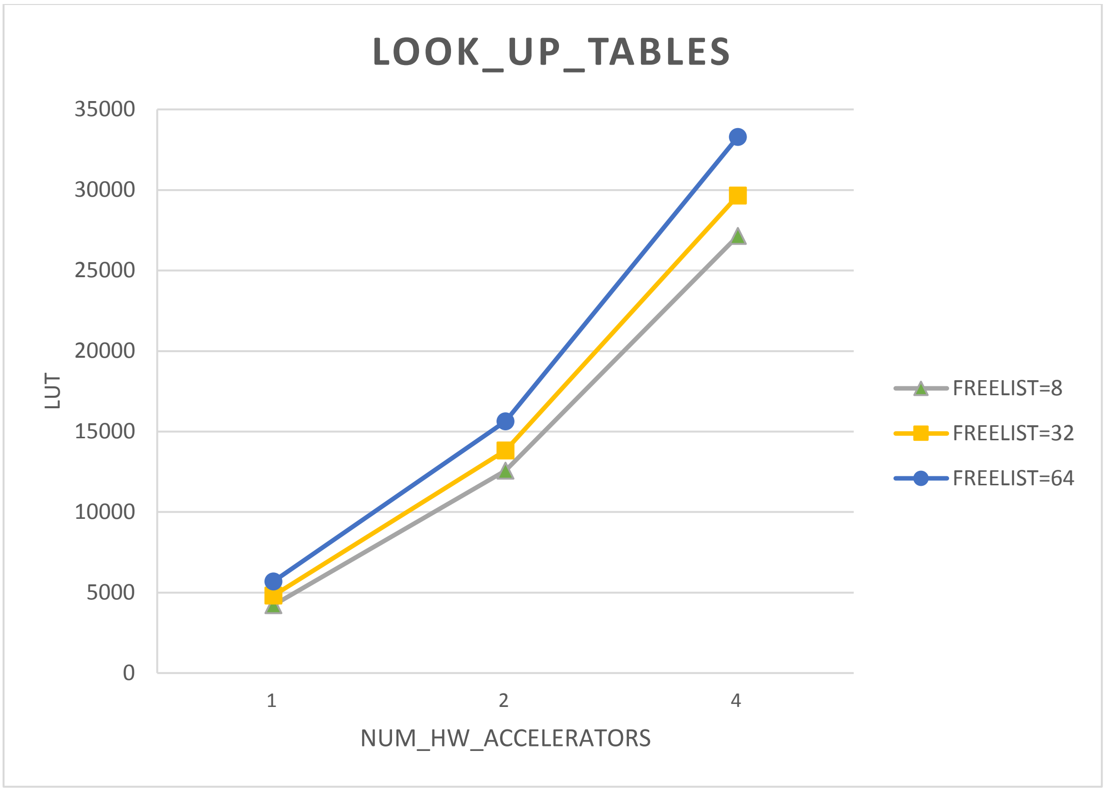


Figure 116: LUT regarding accelerators for Histogram kernel

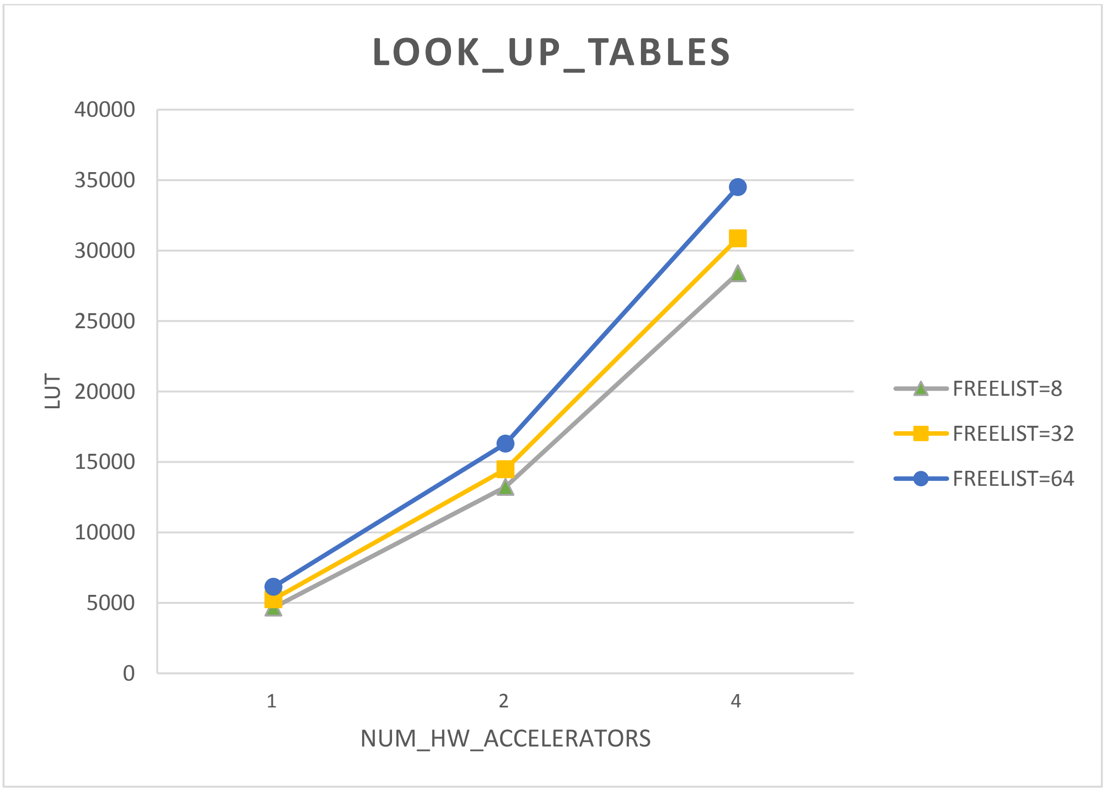


Figure 117: LUT regarding accelerators for MMUL kernel

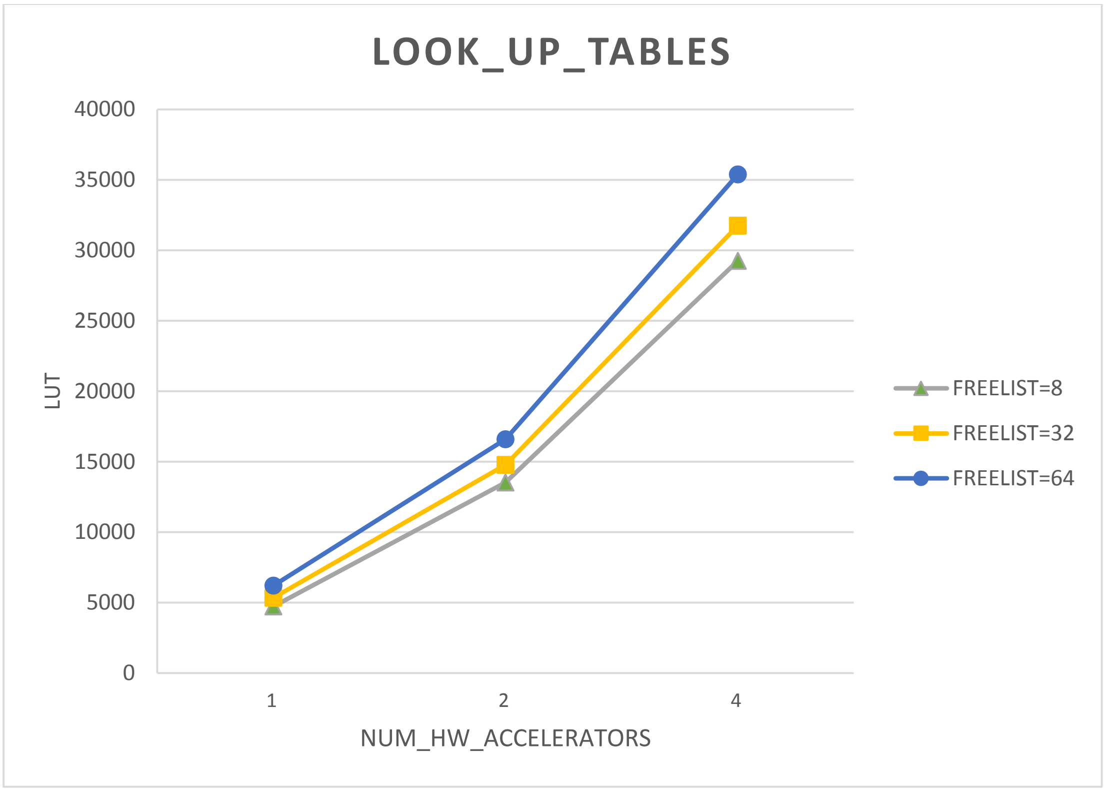


Figure 118: LUT regarding accelerators for PCA kernel

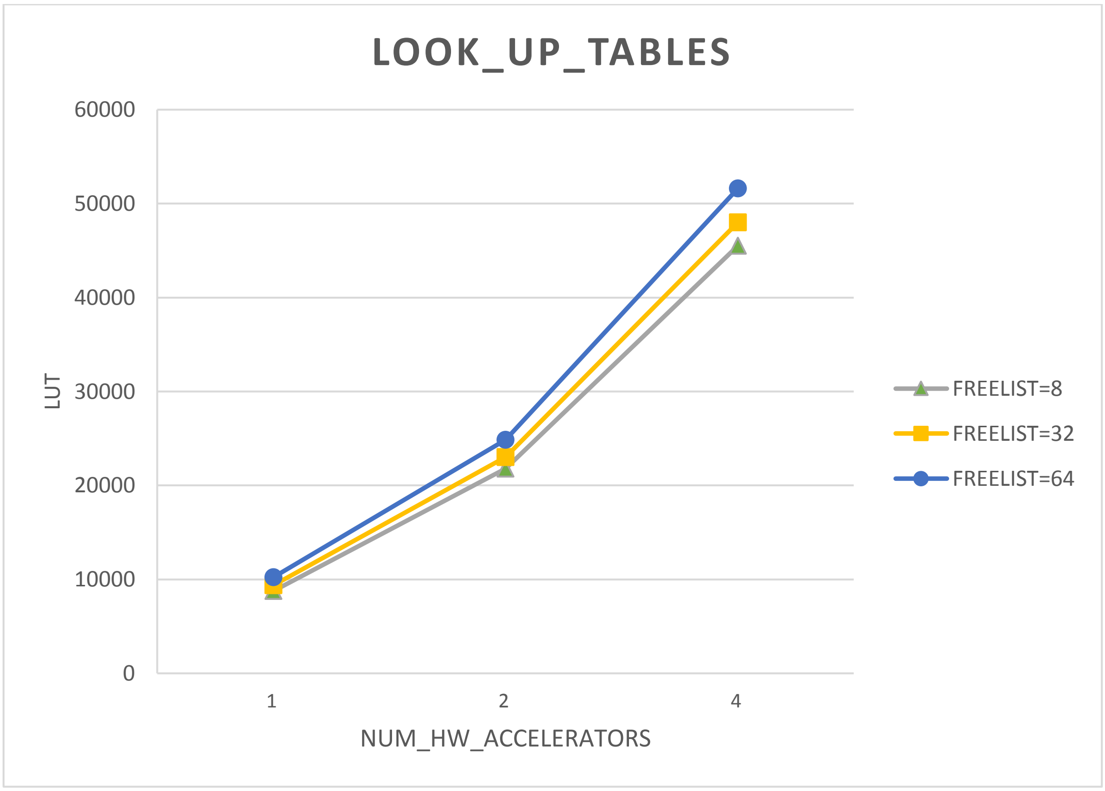


Figure 119: LUT regarding accelerators for Kmeans kernel

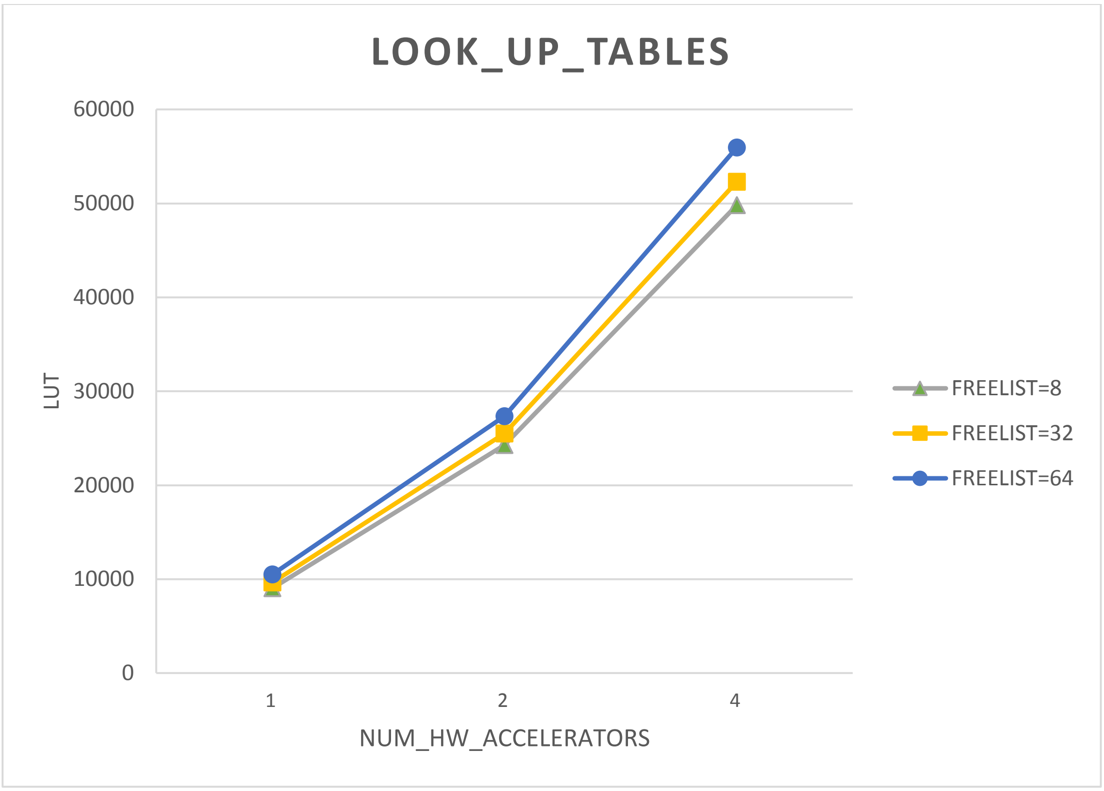


Figure 120: LUT regarding accelerators for Strmatch kernel

As we add more accelerators and heaps, the use of LUTs increases. In addition, from the above figures, it is clear that (as we mentioned before) the LUTs in use depend on the width of freelist array.

4.3 Evaluation of Results

In this Chapter we analyzed two of the main features of the DMM library that we proposed in this diploma thesis, the function inlining and the width size of freelist array. We come to the conclusion that these two features optimize the simulation time of the system, with inline having a greater impact on the system's latency. It is worth noting that both features do not use extra BRAM units to be executed. The utilization of BRAMs depends only on the number of memory heaps and accelerators that compose the whole system.

Besides the DSE that was accomplished using all the above figures, we also used MultiCube tool in order to find the suitable solution area for each application. MultiCube stands for the Multi-objective Design Space Exploration of Multiprocessor SOC Architectures for Embedded Multimedia Applications. "MultiCube project focuses on the definition of an automatic multi-objective Design Space Exploration (DSE) framework to be used to tune the System-on-Chip architecture for the target application evaluating a set of metrics (e.g. energy, latency, throughput, bandwidth, QoS, etc.) for the next generation of embedded multimedia platforms." [16]. The use of MultiCube leads to "a Pareto-optimal set of design alternatives" [16] for every kernel. The figures of MultiCube results can be found below. It is worth mentioning that the experimental data that were given as input to MultiCube came from the execution of the several kernels with small data sets as inputs. (The simulation time has been measured in picoseconds, ps).

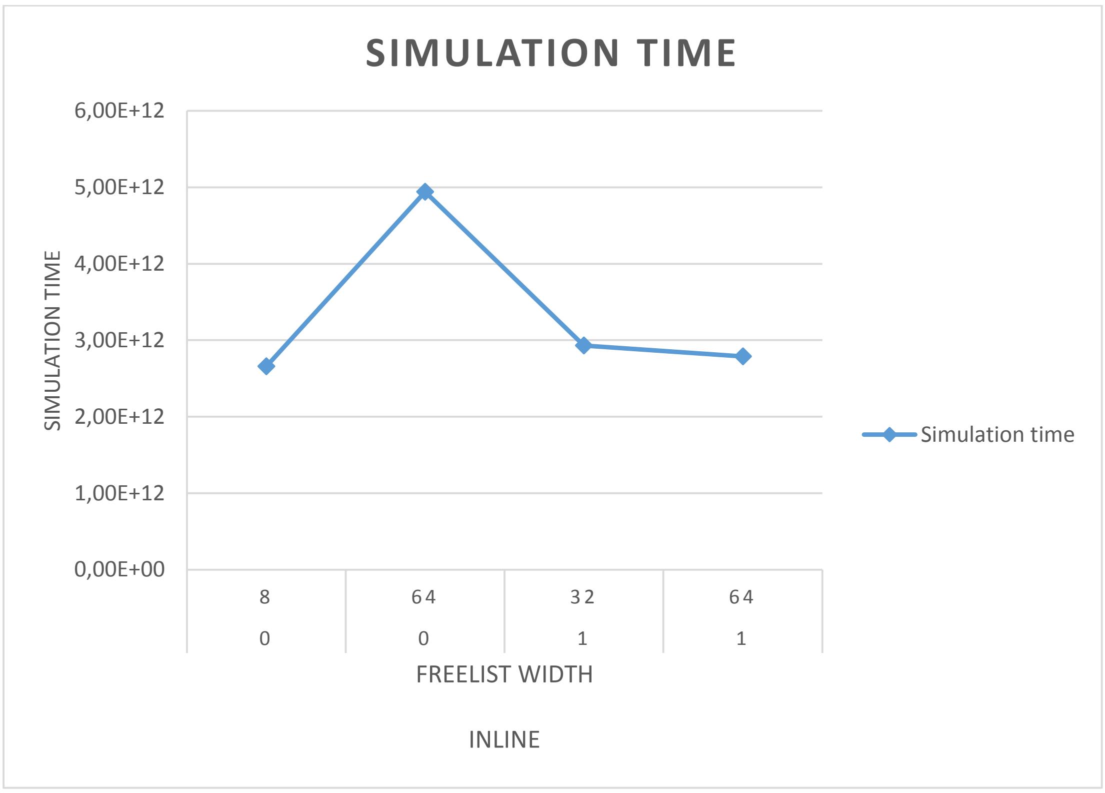


Figure 121: Analysis of Simulation time for Histogram kernel

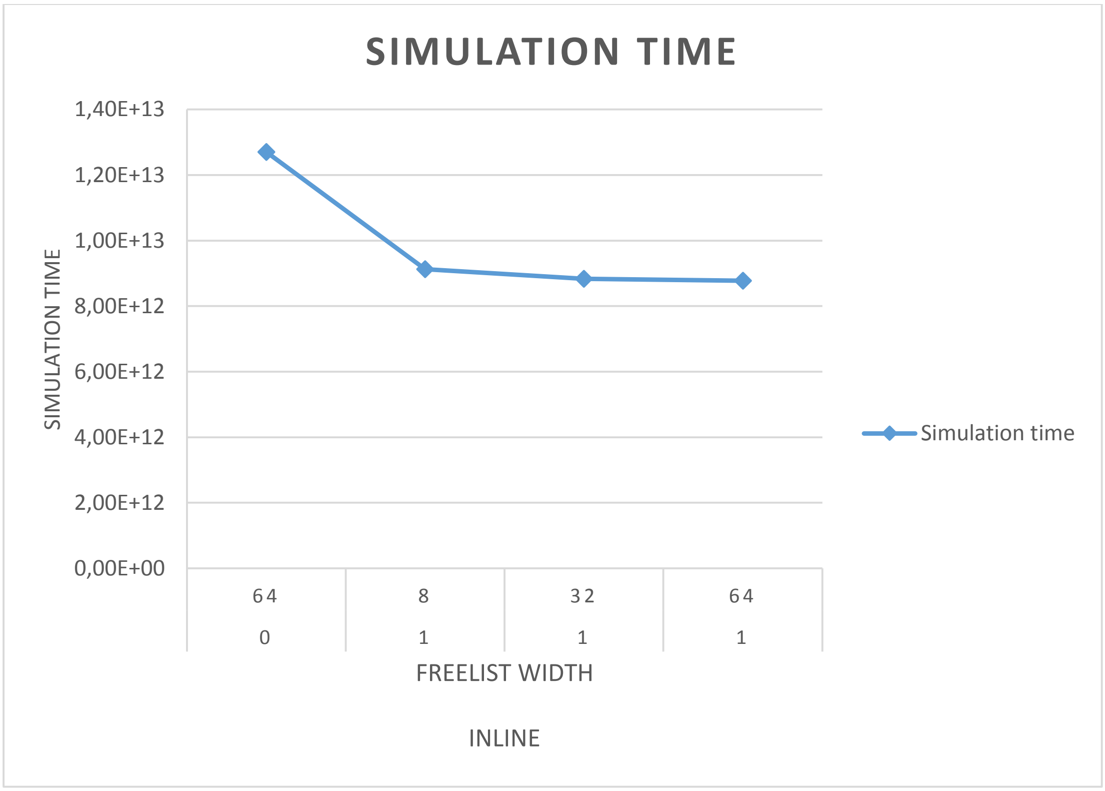


Figure 122: Analysis of Simulation time for MMUL kernel

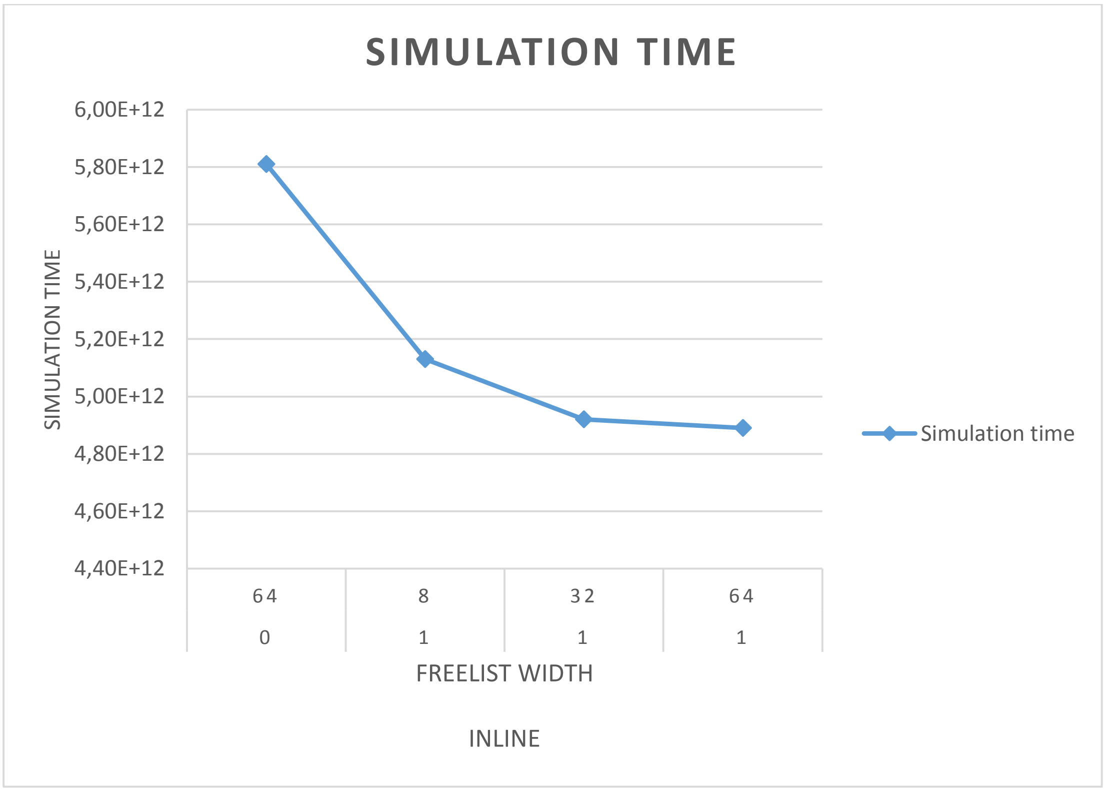


Figure 123: Analysis of Simulation time for PCA kernel

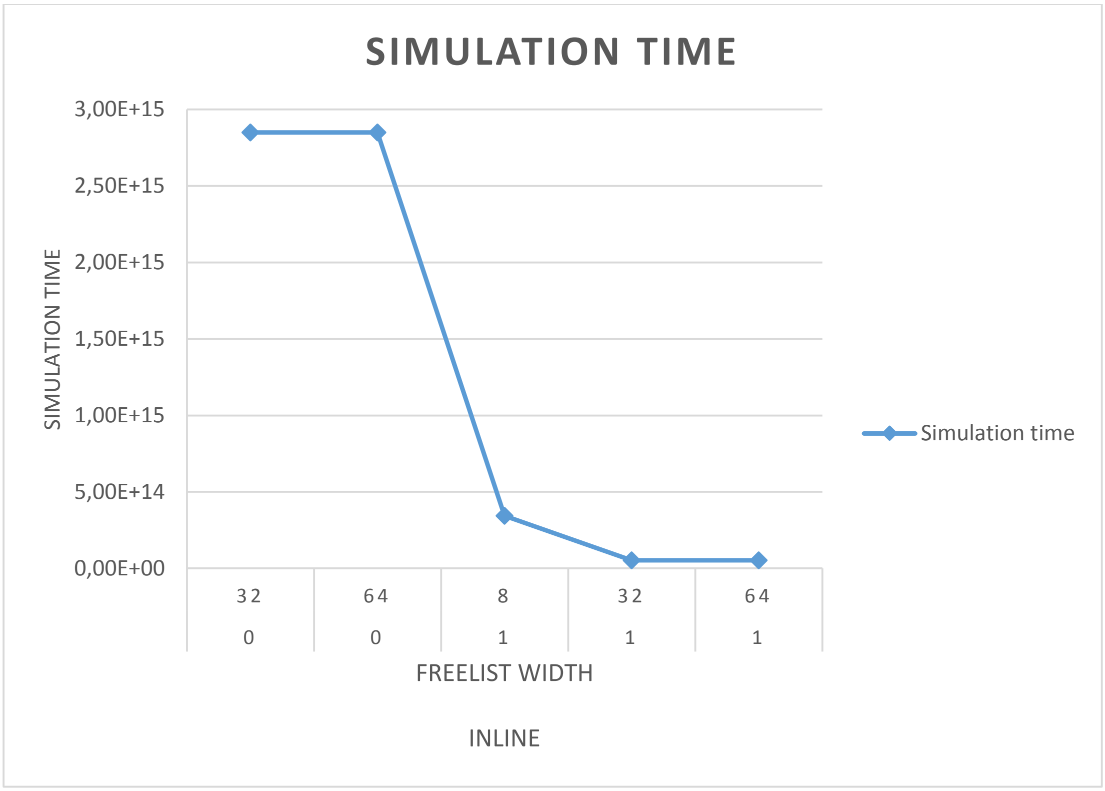


Figure 124: Analysis of Simulation time for Kmeans kernel

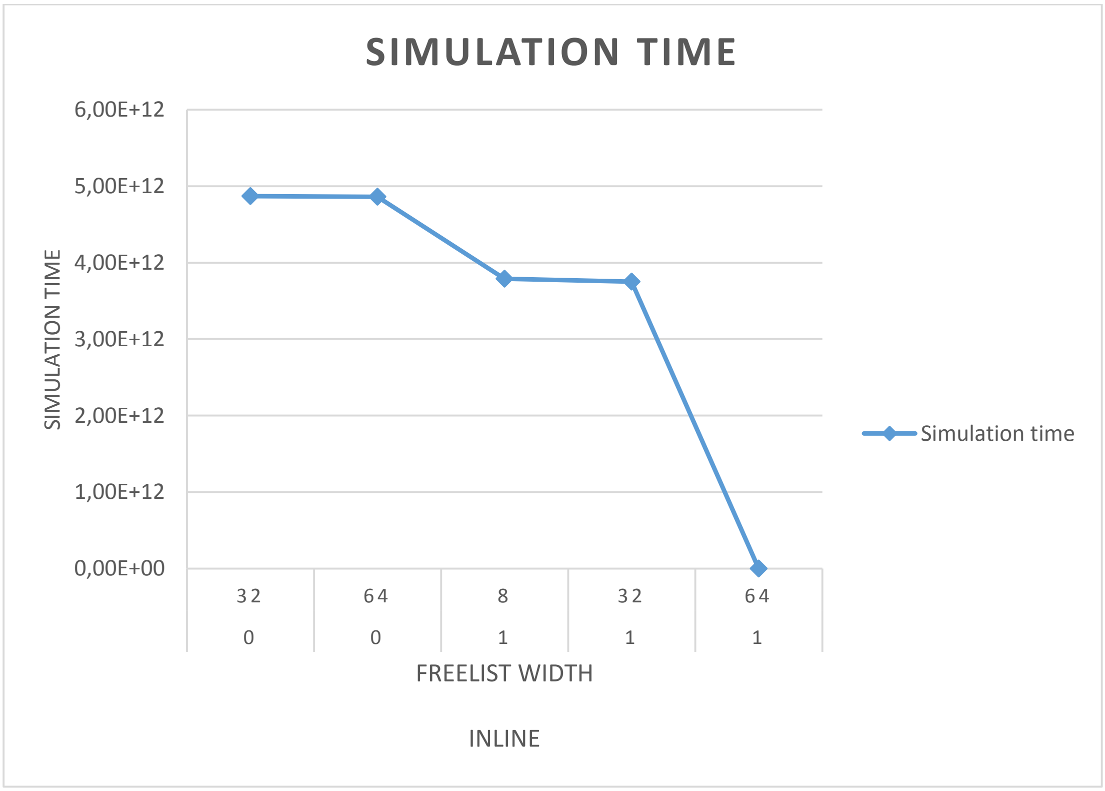


Figure 125: Analysis of Simulation time for Strmatch kernel

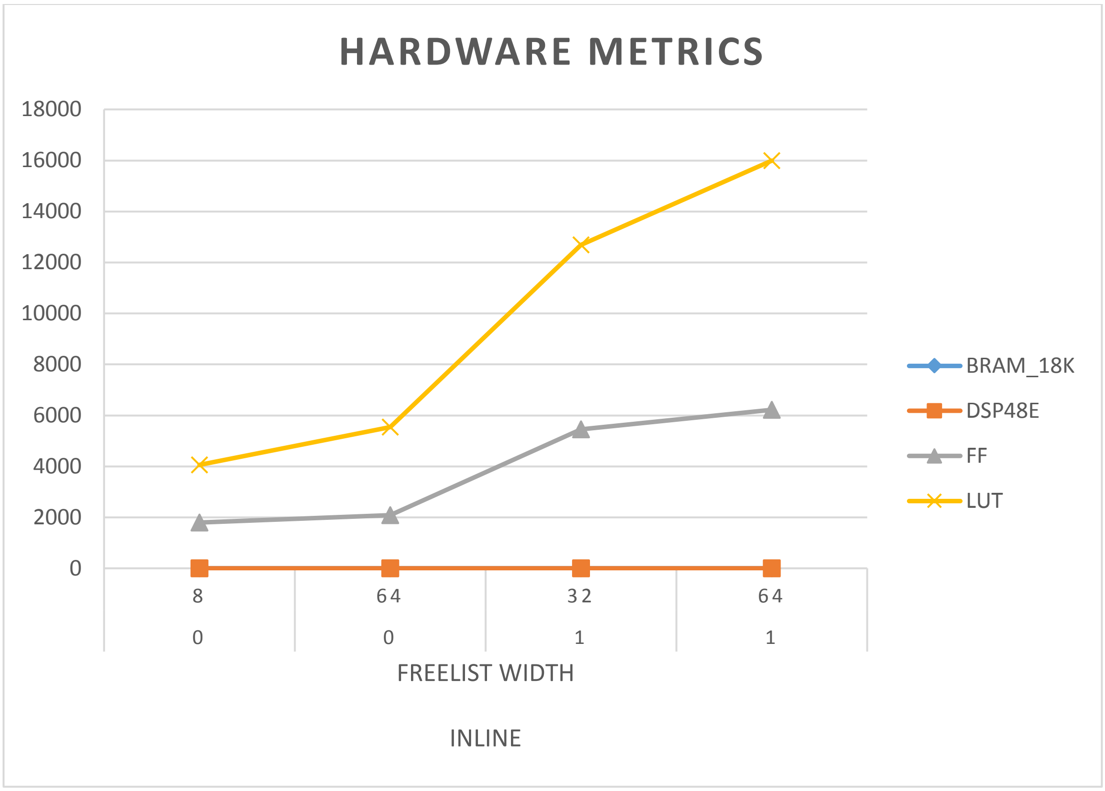


Figure 126: Analysis of Hardware metrics for Histogram kernel

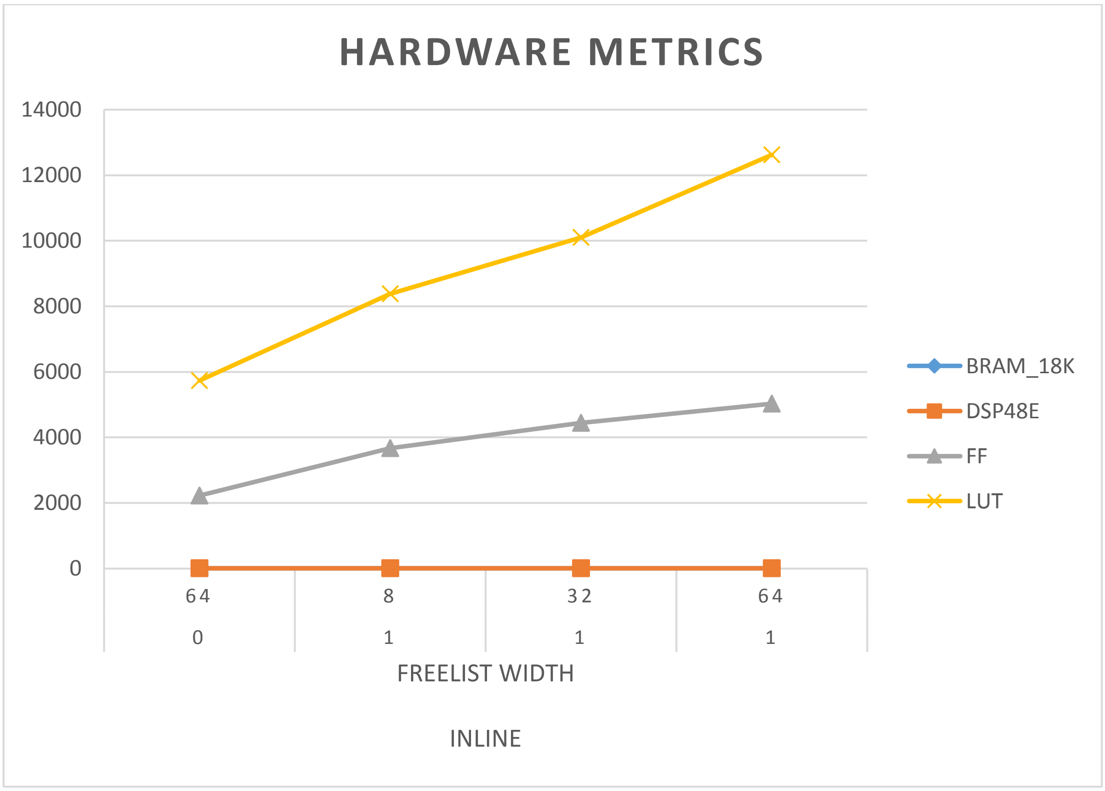


Figure 127: Analysis of Hardware metrics for MMUL kernel

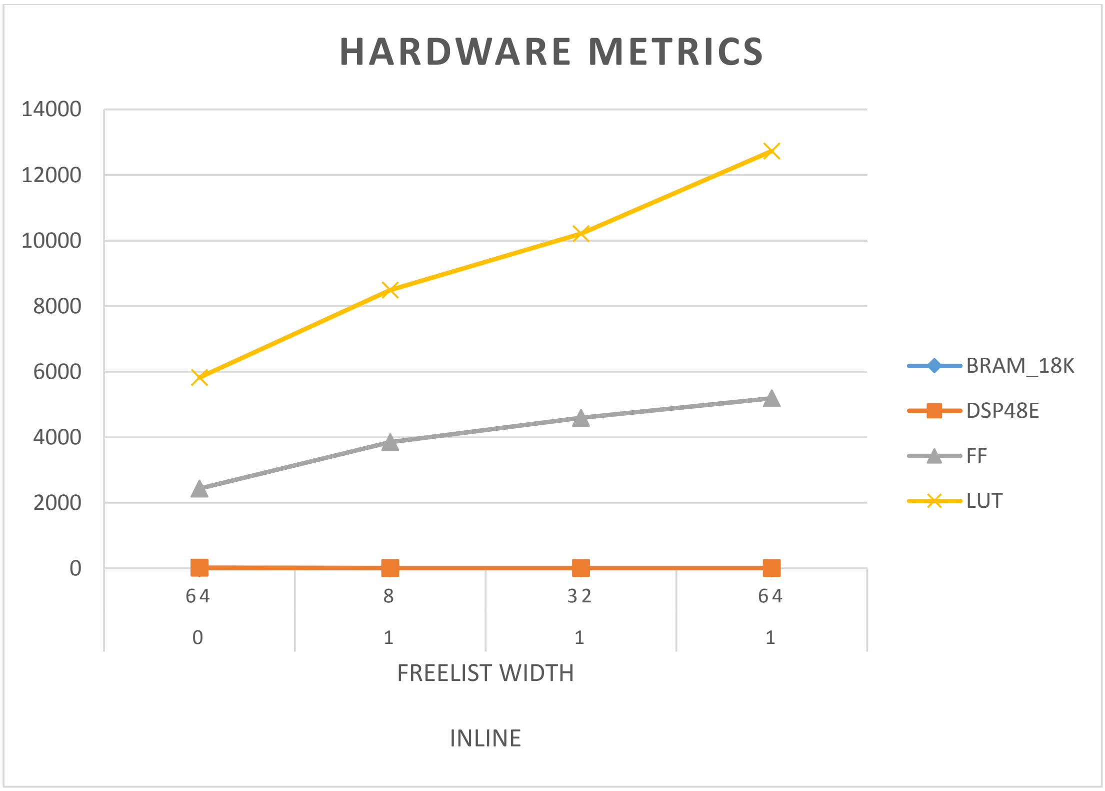


Figure 128: Analysis of Hardware metrics for PCA kernel

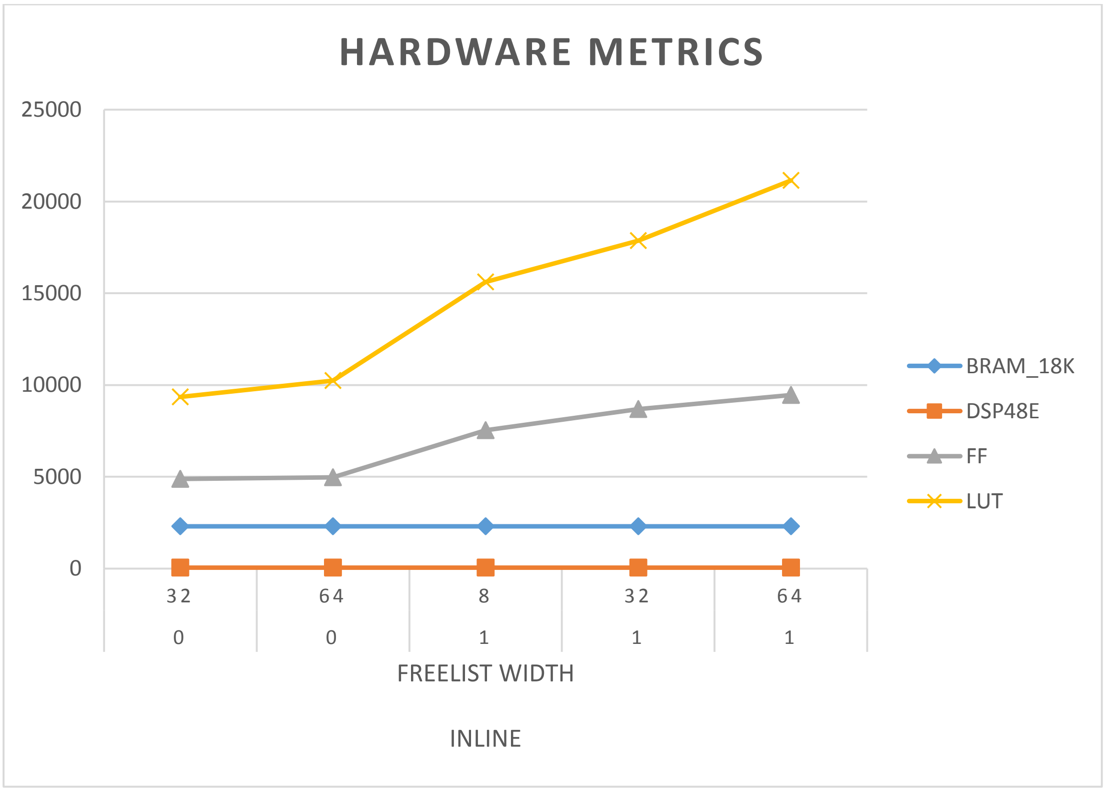


Figure 129: Analysis of Hardware metrics for Kmeans kernel

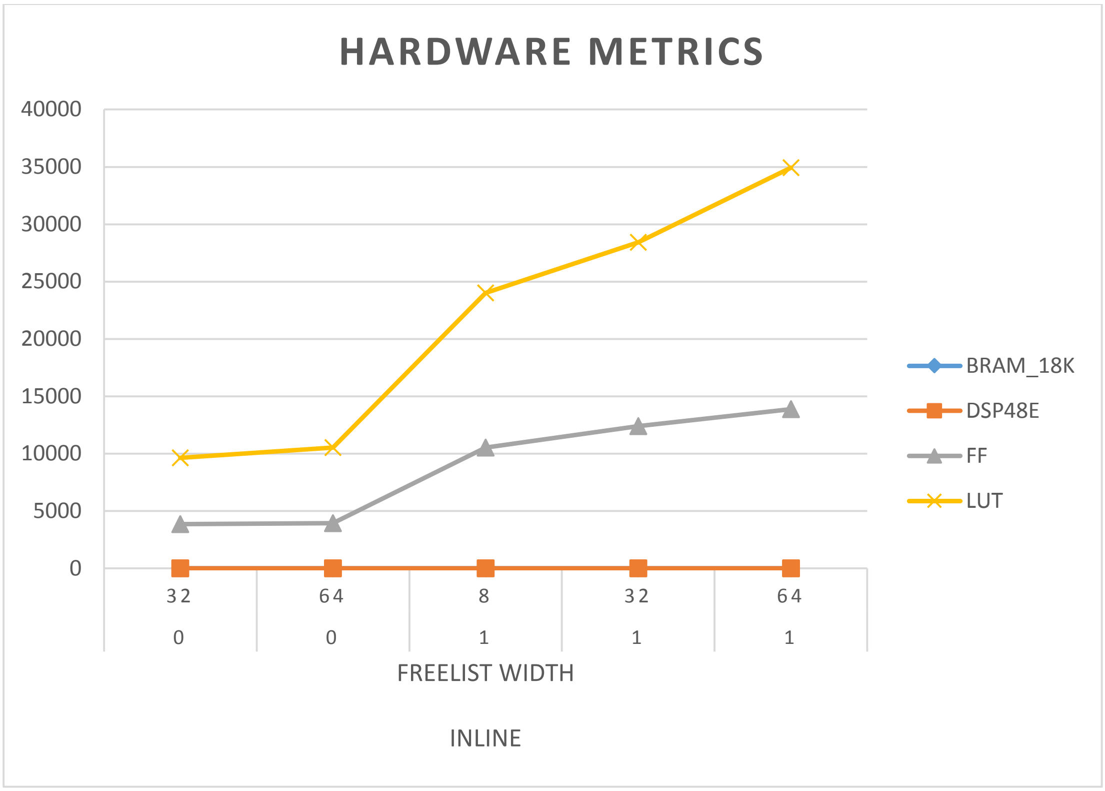


Figure 130: Analysis of Hardware metrics for Strmatch kernel

As we see, MultiCube results confirm our analysis about DMM features regarding simulation time and hardware metrics. We observe some exceptions (for example the simulation time of Histogram kernel) that are due to the small input data sets, as we mentioned before.

Chapter 5

DSE using HLS directives

5.1 Analysis of Histogram and PCA kernels

In this Chapter, we apply some of the HLS directives (those referred in Chapter 3.2) to Histogram and PCA kernels. The software implementation of these two kernels has some characteristics that affect the application and the return of HLS directives. More precisely, we quote the software implementation of Histogram kernel:

```
for (i_HISTOGRAM_k0=0; i_HISTOGRAM_k0 < N_HISTOGRAM_k0; i_HISTOGRAM_k0+=3)
        fdata_HISTOGRAM_k0[i_HISTOGRAM_k0]=(char)RandMinMaxSyn(1,
                                                               (uint_t)i_HISTOGRAM_k0+1,
                                                                &heap_lfsr_ptr,
                                                                1);

for (i_HISTOGRAM_k0=0; i_HISTOGRAM_k0 < N_HISTOGRAM_k0; i_HISTOGRAM_k0+=3) {
        val = (unsigned char *)&(fdata_HISTOGRAM_k0[i_HISTOGRAM_k0]);
        blue_HISTOGRAM_k0[*val]++;
        val = (unsigned char *)&(fdata_HISTOGRAM_k0[i_HISTOGRAM_k0+1]);
        green_HISTOGRAM_k0[*val]++;
        val = (unsigned char *)&(fdata_HISTOGRAM_k0[i_HISTOGRAM_k0+2]);
        red_HISTOGRAM_k0[*val]++;
        main_result_HISTOGRAM_k0+=3;
        }
```

Firstly, we observe the existence of 2 sequential loops. Therefore, we may apply the loop merge directive in Histogram kernel. In addition, we see some true dependencies (RAW) and antidependecies (WAR) into the second for-loop (between 1st and 2nd , and 2nd and 3rd lines of each iteration respectively). We also observe that if we try to unroll the 1st loop, some extra output dependencies (WAW) will occur. From these code indications, we expect that the unroll directive will have a smaller impact on performance than the pipeline directive because of the dependencies.

Furthermore, we quote the software implementation of PCA kernel:

```
generate_points(matrix, num_rows, num_cols, grid_size);
calc_mean(matrix, mean, num_rows, num_cols);
calc_cov(matrix, mean, cov, num_rows, num_cols);
for (i=0; i<num_rows; i++)
        for (j=0; j<num_rows; j++)
                main_result_PCA_k0 += cov[i*num_rows+j];
return (main_result_PCA_k0 + N_PCA_k0);
```

In this kernel, we observe the existence of 2 perfectly nested loops. Thus, we may apply the loop flatten directive in PCA kernel. Moreover, we observe that if we try to unroll the loops, we will face true dependencies (RAW), regarding the main_result_PCA_k0 variable, and input dependencies (RAR), regarding cov array. Therefore, we do not expect great differences concerning performance between unrolling and pipelining, because of the type of the dependencies (RAW stalls unrolling while RAR stalls pipelining).

The figures that are included in the rest of Chapter 5 concern systems with 1 accelerator and 1 memory heap, DMM inline equal to 1 and DMM freelist width equal to 32 bits.

5.2 Loop Unroll

The figures concerning the simulation time and hardware metrics for loop unroll are shown below.

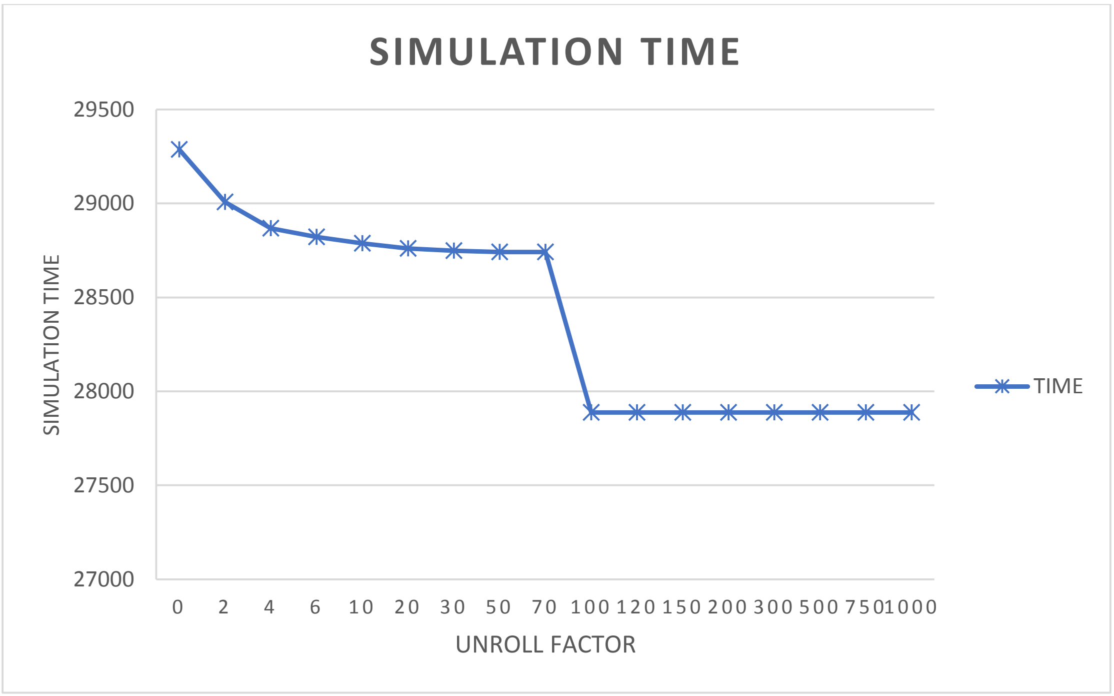


Figure 131: Simulation time of loop unroll for Histogram kernel

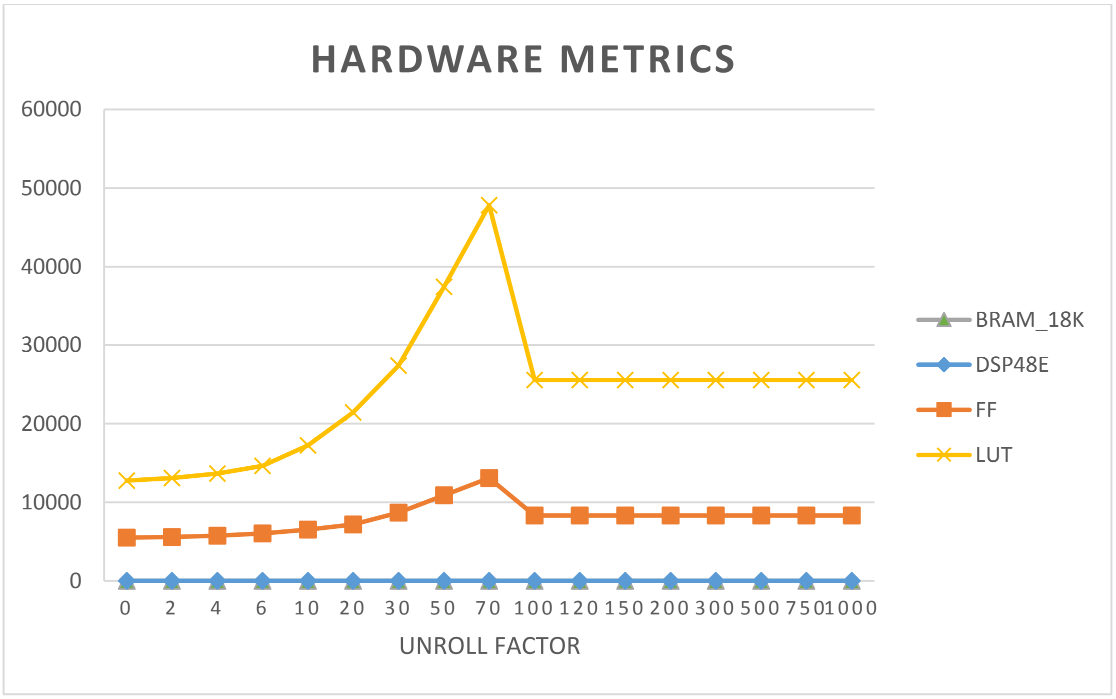


Figure 132: Hardware metrics of loop unroll for Histogram kernel

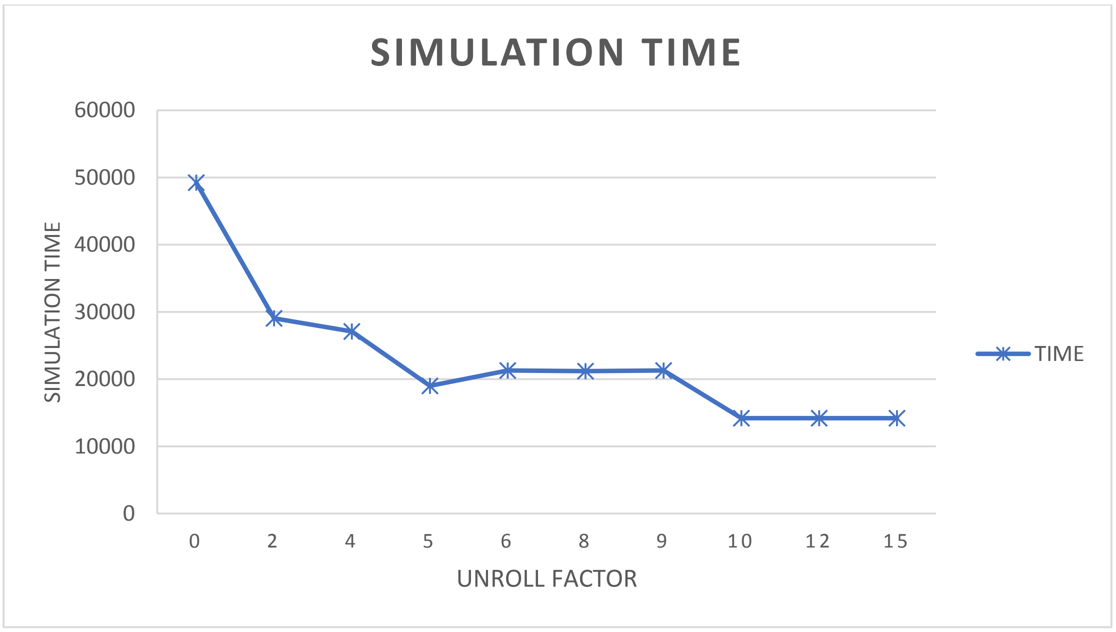


Figure 133: Simulation time of loop unroll for PCA kernel

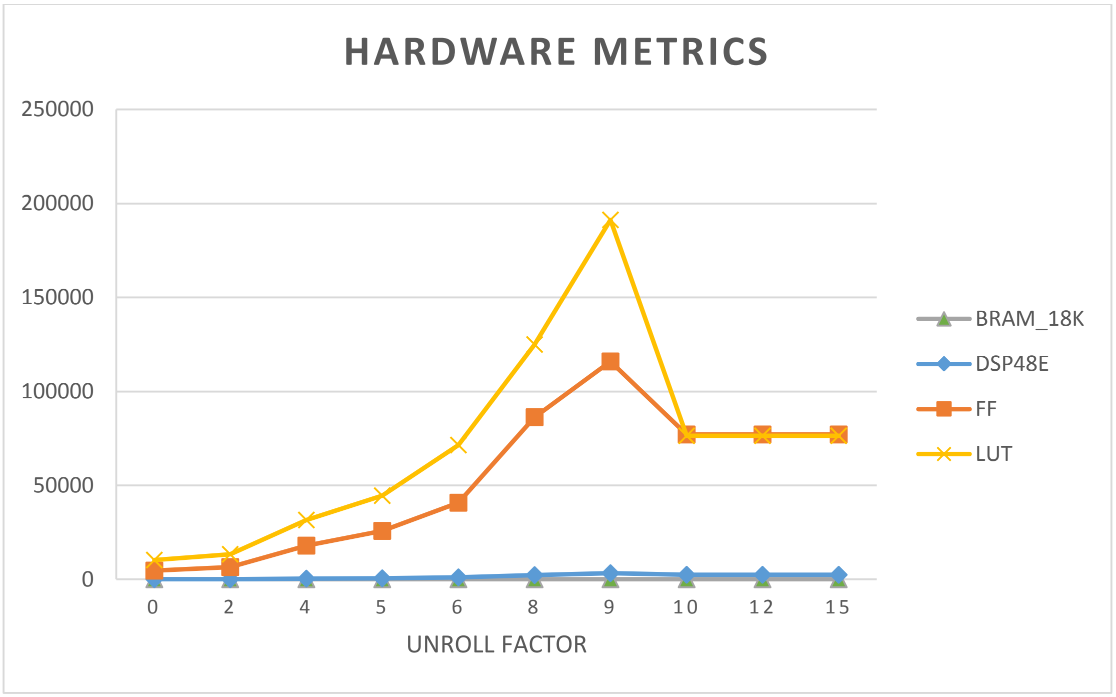


Figure 134: Hardware metrics of loop unroll for PCA kernel

As we expected, we observe the minimum simulation time for unroll factors equal or greater than the overall number of iterations of the unrolled loop (complete unrolling). Concerning the hardware metrics, complete unrolling uses few BRAM and DSP units and a big but acceptable amount of FFs and LUTs. In case of unroll factors greater than the complete unroll, simulation time and hardware metrics are equal to complete unrolling. This is happening because Vivado HLS compares the unroll factor to the complete unroll one and if it is greater, Vivado HLS applies only complete unrolling. Regarding factors smaller than the unroll factor, we observe a gradual reduction of simulation time, having more abrupt reduction around factors that are divisors of the complete unroll factor. As we increase the unroll factor, we see an increment of hardware in use. This is because, as we analyzed in Chapter 3.2.1.1, loop unroll creates as many replicates of the hardware used by the loop as the unroll factor indicates. It is worth noting that if the unroll factor is not a divisor of the complete unroll factor, loop unroll creates extra control circuits in order to ensure that the created hardware is identical to the hardware before unrolling. This fact justifies the peak of hardware metrics just before complete unrolling, when we have the maximum identical groups of hardware resources with extra control circuits for each one.

5.3 Loop Pipeline

The figures concerning the simulation time and hardware metrics for loop pipeline are shown below.

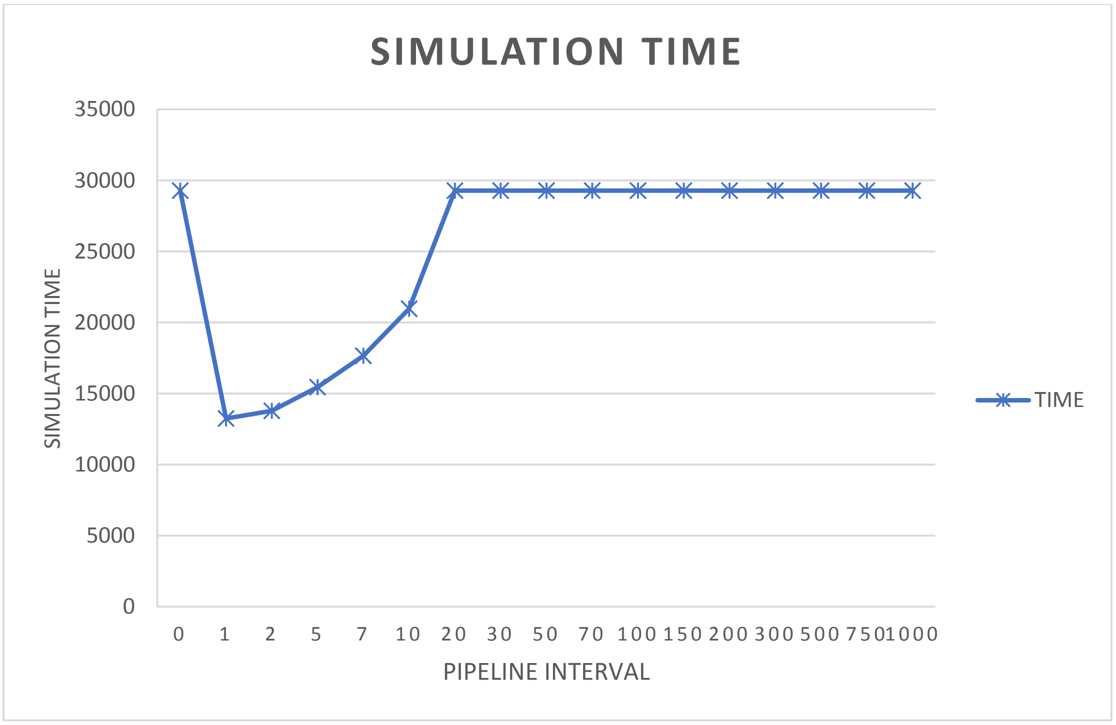


Figure 135: Simulation time of loop pipeline for Histogram kernel

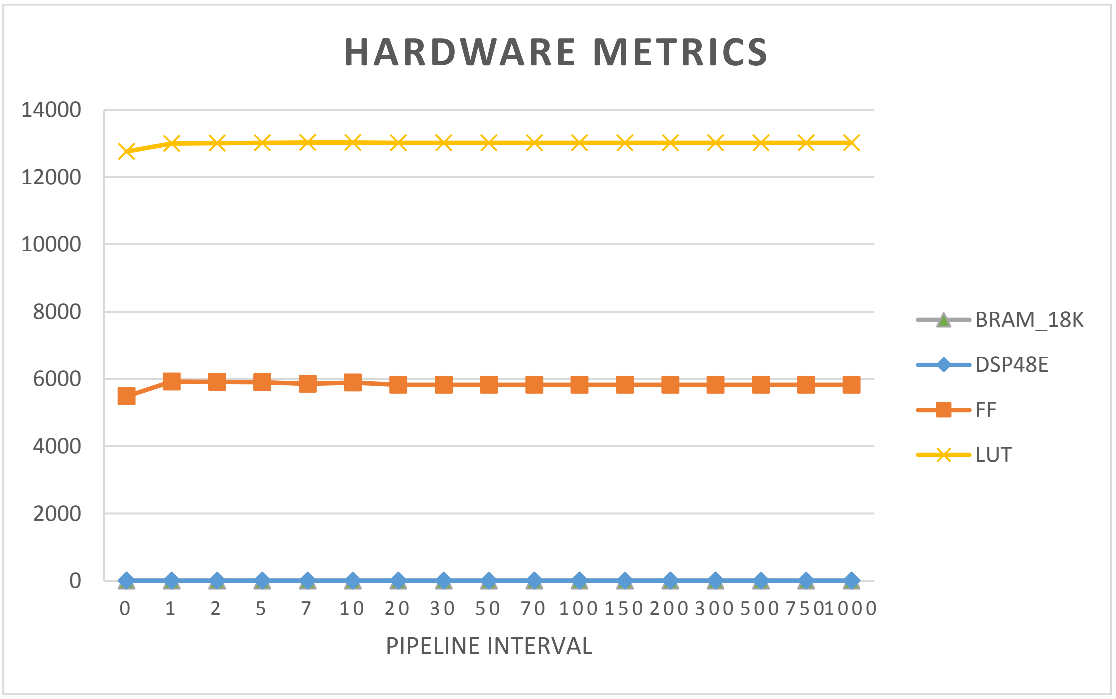


Figure 136: Hardware metrics of loop pipeline for Histogram kernel

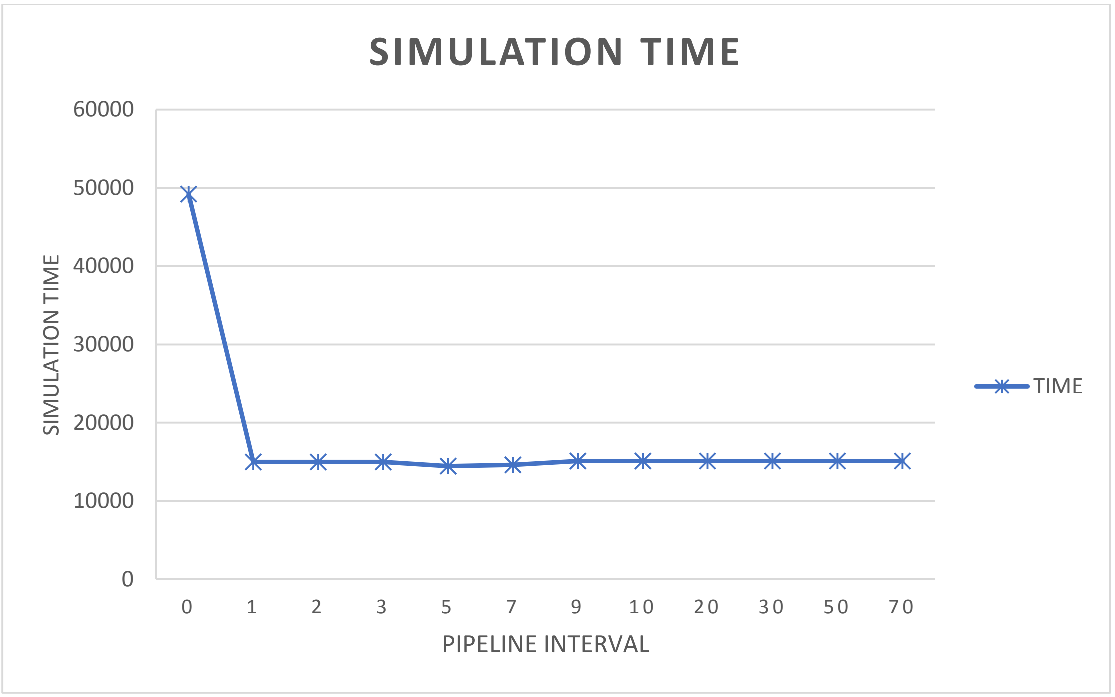


Figure 137: Simulation time of loop pipeline for PCA kernel

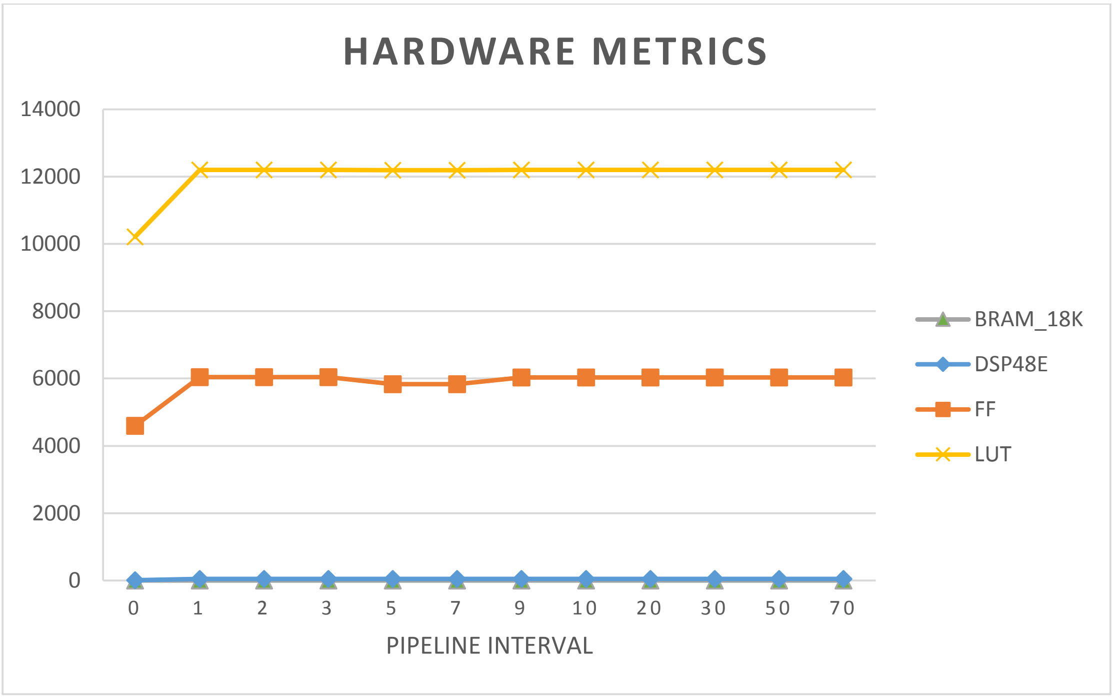


Figure 138: Hardware metrics of loop pipeline for PCA kernel

As we expected, we observe the minimum simulation time pipeline interval equal to 1. Concerning the hardware metrics, we observe that loop pipelining has a small impact on the amount of FFs and LUTs in use.

We have to mention the difference between the 2 figures of simulation time (for Histogram and PCA kernel). We see a different behavior between these kernels as we increase the pipeline interval. We expect that as we increase the pipeline interval the simulation time will also increases. This happens for Histogram kernel but does not happen for PCA.

PCA has a different behavior due to the fact that this kernel includes many nested loops. As we saw in Chapter 3.2.1.2, when we apply loop pipeline to the top loop, all the sub-loops are automatically unrolled. Therefore, what we observe from PCA kernel is a combination of loop pipelining and unrolling. More precisely, loop unroll occurs before pipelining and causes the increment of data dependencies that already exist in the kernel code, as we saw previously at the beginning of Chapter 5. The raise of data dependencies results in stalling the loop pipelining. Thus, Vivado HLS fails to satisfy the demand implied by the pipeline interval and automatically relaxes the optimization target regarding performance and creates a design with lower performance. So, increasing the pipeline interval we end up with the same hardware and time metrics. There is only an issue when pipeline interval is equal to 5, we observe an even better simulation time than the time for pipeline interval equal 1. This is due to some data dependencies that caused a bottleneck during the execution and required a stall of 5cc, even when the pipeline interval was 1cc. The existence of unrolling combined with pipelining, regarding PCA kernel, can be perceived from the hardware metrics figure. We observe a greater increase of FFs and LUTs for the pipelining of PCA kernel compared to Histogram.

Finally regarding the time metrics of Histogram, we see that as we increase the pipeline interval we face an increment of simulation time. After some point, it is clear that Vivado HLS ignores completely the pipeline directive because the interval is too big and this situation is identical of having the kernel without pipeline directive. That is why we end up with the same simulation time as we had without pipelining.

5.4 Loop Merge

As we mentioned before, Histogram kernel includes two sequential loops. We observe that the two loops have the same bounds, so they have the same number of iterations. Also we see that in the second loop, there are some commands of type $a = a + 1$, regarding the blue, green and red array. Thus, we believe that Vivado HLS will not apply loop merge and the time and hardware metrics will not change. As expected, we do not observe any change concerning time and hardware metrics as Vivado HLS was not able to merge the two loops because of the abovementioned commands.

5.5 Loop Flatten

We observed that PCA kernel includes some nested for loops. According to Chapter 3.2.1, we can examine if PCA is eligible for flattening these nested loops. Below, we highlight some code segments of PCA implementation that we studied.

```
void generate_points(int *pts, int rows, int cols, int grid_size) {
  int i, j;
  unsigned short heap_lfsr_ptr;
  heap_lfsr_ptr = 0xACE1u;
  for (i=0; i<rows; i++) {
    for (j=0; j<cols; j++) {
        pts[i*cols+j] = (int)RandMinMaxSyn(1, (uint_t)grid_size, &heap_lfsr_ptr, 1);
    }
  }
}

void calc_mean(int *matrix, int *mean, int rows, int cols) {
  int i, j;
  int sum = 0;
  for (i = 0; i < rows; i++) {
    sum = 0;
    for (j = 0; j < cols; j++) {
      sum += matrix[i*cols+j];
    }
    mean[i] = sum / cols;
  }
}

void calc_cov(int *matrix, int *mean, int *cov, int rows, int cols) {
  int i, j, k;
  int sum;
  for (i = 0; i < rows; i++) {
    for (j = i; j < rows; j++) {
      sum = 0;
      for (k = 0; k < cols; k++) {
        sum = sum + ((matrix[i*cols+k] - mean[i]) * (matrix[j*cols+k] - mean[j]));
      }
      cov[i*rows+j] = cov[j*rows+i] = sum/(cols-1);
    }
  }
}

int PCA_k0(void) {
        int i,j;
        generate_points(matrix, num_rows, num_cols, grid_size);
        calc_mean(matrix, mean, num_rows, num_cols);
        calc_cov(matrix, mean, cov, num_rows, num_cols);
        for (i=0; i<num_rows; i++)
            for (j=0; j<num_rows; j++)
                        main_result_PCA_k0 += cov[i*num_rows+j];
        return (main_result_PCA_k0 + N_PCA_k0);
```

We have only two semi-perfect loops (the nested loops in PCA_k0, generate_points functions) and two imperfect loops (the nested loops in calc_mean and calc_cov functions). We apply the loop flatten directive to all the inner loops (both semi-perfect and imperfect), considering that Vivado HLS will ignore the directive in case of an imperfect loop. As expected, Vivado HLS applies flattening only to semi-perfect loops. We observe a very small increase of FFs and LUTs in use and a small increase in latency and simulation time (0,2μs). The type of the commands inside both the semi-perfect loops causes the increase of simulation time due to flattening. More precisely, the commands:

- pts[i*cols+j] = (int)RandMinMaxSyn(1, (uint_t)grid_size, &heap_lfsr_ptr, 1);
- main_result_PCA_k0 += cov[i*num_rows+j];

include some computation stages and some reads/writes from/to memory. The combination of computations and memory interaction result in more than one cycle of latency between iterations of the outer loop. Thus, loop flatten (which saves clock cycles from collapsing nested loops to a single one) does not improve the time metrics of the PCA kernel.

5.6 Evaluation of Results

In this Chapter we analyzed the directives of unroll, pipeline, merge and flatten through experimental simulations. We stated that as we increase the unroll factor the simulation time decreases but the use of hardware resources raises. We observed also that for pipeline interval equal to one, we took the most optimized simulation time with a small trade off concerning hardware resources. Regarding loop merge and loop flatten, we saw that they have many prerequisites in order to be applicable to the code. We also observed that there are some cases where loop flatten is applicable but does not optimize the code. Generally, we may state that the HLS directives cannot be applied automatically. It is required from the designer to read, analyze and evaluate the functionality and the dependencies of the code before continue with design space exploration using HLS directives.

Chapter 6

Thesis Conclusion

6.1 General Remarks

This diploma thesis aims to optimize the execution of many-accelerators systems in embedded systems, especially FPGAs. To achieve this, we propose the adoption of Dynamic Memory Management concerning on-chip memory. High Level Synthesis is applied to both DMM library and the accelerators' implementation. Afterwards, we explore all the design alternatives for either DMM structures or algorithmic implementations.

We conclude that, concerning memory optimizations, function inlining almost always improves the simulation time, consuming some extra hardware resources. In addition, as we escalate our system, the impact of inline maximizes and the hardware reuse prevails. Also, the selection of the most suitable freelist width reduces simulation time, but we have to study the accelerator's memory pattern first. Another thing that is worth mentioning is that all the DMM functionality is synthesized into RTL implementation using the backend of Vivado HLS. Thus, the impact of all DMM optimizations depends on the scheduling and synthesis of Vivado, and the result is not completely configured by the designer.

Regarding loop optimizations provided by Vivado HLS, they improve the execution time, under specific circumstances. Some of them have many prerequisites in order to be applied correctly (loop merge, loop flatten) and others are too complex to be perceived by the user without analyzing Vivado logs (loop pipeline). Furthermore, these directives do not always improve the execution time, as it depends on the unroll factor or the pipeline interval that we are using. Also they cannot be applied automatically to every loop as they may deteriorate the system's performance. For all these reasons, the designer needs to have all data dependencies analyzed, before applying HLS directives. Thus, there is a tradeoff, concerning time and effort, for the designer in order to take advantage of these directives.

6.2 Future Work

As for future work, an interesting approach regarding DMM library's evolution would be to develop appropriate DMM mechanisms to overcome some Vivado HLS limitations. These HLS restrictions, as the management and compilation of more complex pointers, necessitate the code to be rewritten by the designer in order to be synthesized to RTL. Overcoming these issues would result in a more user-friendly FPGA programming.

Another important issue that needs further investigation is the allocation techniques into DMM structures. The use of multiple freelists in the same DM allocator may facilitate the execution of accelerators that need to allocate variables of different data types. Also, if the memory footprint analysis of the accelerators implemented in the FPGA board is possible and a specific pattern is found, the usage of suballocators will be helpful.

Finally, an extensive study of the combination between HLS directives and DMM features can be performed. HLS directives related to a more flexible scheduling and parallelism could be used in order to support parallelism caused by the adoption of DMM concept in Vivado HLS. This study may result in optimal circuits and even smaller simulation times.

Bibliography

[1] R. Nane, V. M. Sima, C. P. Quoc, F. Goncalves and K. Bertels, "High-Level Synthesis in the Delft Workbench Hardware/Software Co-design Tool-Chain," in *12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, Milano, 2014.

[2] E. Kang, E. Jackson and W. Schulte, "An Approach for Effective Design Space Exploration," in *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, Redmond, Springer Berlin Heidelberg, 2011, pp. 33-54.

[3] J. Wakerly, "Digital Systems Design," 2013. [Online]. Available: http://www.slideshare.net/abhilash128/lec-23.

[4] Xilinx, Introduction to FPGA Design with Vivado High-Level Synthesis, Xilinx, 2013.

[5] D. Diamantopoulos, S. Xydis, K. Siozios and D. Soudris, "Dynamic Memory Management in Vivado-HLS for Scalable Many-Accelerator Architectures," in *Applied Reconfigurable Computing*, Bochum, Springer International Publishing, 2015, pp. 117-128.

[6] D. Diamantopoulos, S. Xydis, K. Siozios and D. Soudris, "Mitigating Memory-Induced Dark Silicon in Many-Accelerator Architectures," in *Computer Architecture Letters (Volume:14 , Issue: 2)*, IEEE, 2015, pp. 136 - 139.

[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *IEEE 13th International Symposium on High Performance Computer Architecture*, Scottsdale, IEEE, 2007, pp. 13 - 24.

[8] University of Miskolc, "Field Programmable Gate Arrays (FPGA)," [Online]. Available: http://mazsola.iit.uni-miskolc.hu/cae/docs/pld1.en.html.

[9] S. KUNZLI, Efficient Design Space Exploration for Embedded Systems, Zurich: Swiss Federal Institute of Technology Zurich, 2006.

[10] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS,* pp. 473-491, 2011.

[11] G. Shute, "Loop Unrolling," University of Minessota Duluth, Duluth.

[12] Xilinx, Vivado Design Suite User Guide, High-Level Synthesis, Xilinx, 2014.

[13] B. Goldberg, "Compiler Optimizations for Modern VLIW/EPIC Architectures," New York University, New York.

[14] M. J. Flynn, O. Mencer, V. Milutinovic, G. Rakocevic, P. Stenstrom, R. Trobec and M. Valero, "Moving from petaflops to petadata," *Communications of the ACM,* vol. 56 , no. 5, pp. 39-42, 2013.

[15] D. Diamantopoulos, S. Xydis, K. Siozios and D. Soudris, "High-Level-Synthesis Extensions for Scalable Single-Chip Many-Accelerators on FPGAs," in *25th International Conference on Field Programmable Logic and Applications (FPL)*, London, 2015.

[16] Politecnico di Milano, "MULTICUBE Leaflet," [Online]. Available: http://www.multicube.eu/.

[17] J. Cong, B. Liu, R. Prabhakar and P. Zhang, "A Study on the Impact of Compiler Optimizations on High-Level Synthesis," in *Languages and Compilers for Parallel Computing*, Tokyo, Springer Berlin Heidelberg, 2013, pp. 143-157.

[18] G. Zhong, V. Venkataraman, Y. Liang, T. Mitra and S. Niar, "Design Space Exploration of Multiple Loops on FPGAs using High Level Synthesis," in *32nd IEEE International Conference on Computer Design (ICCD)*, Seoul, 2014.

[19] B. Zeidman, "All about FPGAs," UBM Canon, 22 March 2006. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1274496&page_number=1. [Accessed 12 12 2015].