1η Άσκηση – Σχεδιασμός Ενσωματωμένων Συστημάτων

Ομάδα 12
Κυριακοπούλου Φωτεινή AM 0310282
Μπουφίδης Ιωάννης AM 03120162

Ζητούμενο 1ο
*Για τον σχεδιασμό των παρακάτω plot χρησιμοποιήθηκε το συνημμένο αρχείο embedded_1_1.ipynb

1.1)

Μέγεθος RAM και Flash Μνημών
Μέγεθος RAM: 640kB
Μέγεθος standby SRAM: 8kB
Μέγεθος Code Flash Memory: 4MB
Μέγεθος Data Flash Memory: 64kB

Κρυφές Μνήμες
Instruction Cache: Χρησιμοποιείται για την προσωρινή αποθήκευση εντολών και επιτρέπει την ταχύτερη εκτέλεση προγραμμάτων μειώνοντας τους κύκλους προσπέλασης της Flash μνήμης.

Data Cache: Χρησιμοποιείται για την προσωρινή αποθήκευση δεδομένων, βελτιώνοντας την ταχύτητα πρόσβασης στα δεδομένα που βρίσκονται στη μνήμη RAM ή Flash.

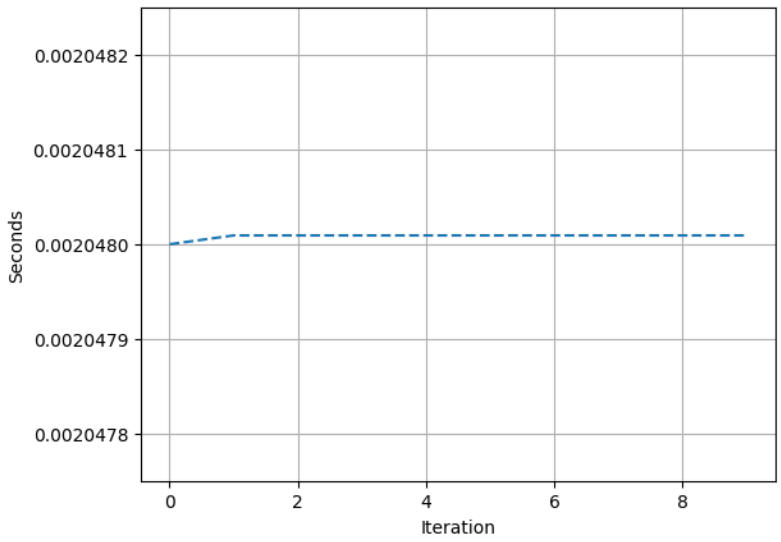
Συχνότητα Ρολογιού
Στο e2 studio σε debugger mode χρησιμοποιούμε ένα watch expression για το SystemCoreClock. Μέσω αυτού, βρίσκουμε ότι η συχνότητα ρολογιού στην οποία θα προγραμματιστεί το board είναι 216 MHz.

1.2) Τροποποιούμε τον κώδικα της hal_entry ως εξής:

```
#define CPU_FREQUENCY_HZ 216000000

void hal_entry(void) {

    // Code to initialize the DWT->CYCCNT register
    CoreDebug->DEMCR |= 0x01000000;
    ITM->LAR = 0xC5ACCE55;
    DWT->CYCCNT = 0;
    DWT->CTRL |= 1;
    // Initialize your variables here
    int motion_vectors_x[N/B][M/B], motion_vectors_y[N/B][M/B], i, j;
    uint32_t start_cycles, end_cycles;
    float execution_time, total_time = 0;
    float max_time = 0, min_time = 1e6;  // Initialize min_time to a large number
```

```
    for (int iter = 0; iter < 10; iter++) {
      // Start the timer
      start_cycles = DWT->CYCCNT;
      // Call the motion estimation function
      phods_motion_estimation(current, previous, motion_vectors_x, motion_vectors_y);

      // Read the final cycle count
      end_cycles = DWT->CYCCNT;

      execution_time = (float)(end_cycles - start_cycles) / CPU_FREQUENCY_HZ;
      total_time += execution_time;


      // Update max and min times
      if (execution_time > max_time) {
        max_time = execution_time;
      }
      if (execution_time < min_time) {
        min_time = execution_time;
      }
    }

    // Calculate average time
    float average_time = total_time / 10;

    while(1){}
}
```

Προκειμένου να παρατηρήσουμε τις τιμές των μεταβλητών καθ' όλη την διάρκεια της εκτέλεσης του προγράμματος, χρησιμοποιήσαμε Watch Expressions και Breakpoints στα κατάλληλα σημεία.
Στο παρακάτω διάγραμμα φαίνονται οι μετρήσεις μας για τις 10 εκτελέσεις του **`phods_motion_estimation()`**



Από τις μετρήσεις προκύπτουν τα εξής:

- Μέσος Όρος: 0.00204800837 sec
- Μέγιστη τιμή: 0.00204800931 sec
- Ελάχιστη τιμή: 0.00204799999 sec

1.3) Οι μετασχηματισμοί βρόχων που κάναμε στην συνάρτηση **phods_motion_estimation()** κατά σειρά είναι οι εξής:

1. Loop interchange (“inter”): Αντιστρέψαμε την σειρά κάποιων βρόχων σε περίπτωση που βελτιώνει την τοπικότητα των δεδομένων και μειώνει τα cache misses. Συγκεκριμένα, δοκιμάσαμε το εξής:

```
for(k=0; k<B; k++) {
   for(l=0; l<B; l++) {...} }

for(k=0; k<B; k++) {
   for(l=0; l<B; l++) {...} }
```

→

```
for(l=0; l<B; l++) {
   for(k=0; k<B; k++) {...} }

for(l=0; l<B; l++) {
   for(k=0; k<B; k++) {...} }
```

2. Loop Merging (“merging”): Στην συνέχεια, παρατηρήσαμε ότι τα 2 παραπάνω loops έχουν παρόμοιους κώδικες, οι οποίοι μπορούν να συγχωνευθούν σε έναν βρόχο, ώστε να μειωθούν οι περιττές προσπελάσεις στην μνήμη. Ο νέος κώδικας που προέκυψε είναι ο εξής:

```
while(S > 0) {
   min1 = 255*B*B;
   min2 = 255*B*B;

   /*For all candidate blocks*/
   for(i=-S; i<S+1; i+=S) {
         distx = 0;
         disty = 0;
         /*For all pixels in the block*/
         for(l=0; l<B; l++) {
            for(k=0; k<B; k++) {
               p1 = current[B*x+k][B*y+l];
               if((B*x + vectors_x[x][y] + i + k) < 0 ||
                  (B*x + vectors_x[x][y] + i + k) > (N-1) ||
                  (B*y + vectors_y[x][y] + l) < 0 ||
                  (B*y + vectors_y[x][y] + l) > (M-1))
                 { p2 = 0; }
                 else {
                   p2 = previous[B*x+vectors_x[x][y]+i+k][B*y+vectors_y[x][y]+l];
                 }

                 if(p1-p2>0)
                     distx += p1-p2;
                 else
                     distx += p2-p1;

                 if((B*x + vectors_x[x][y] + k) <0 ||
                    (B*x + vectors_x[x][y] + k) > (N-1) ||
                    (B*y + vectors_y[x][y] + i + l) < 0 ||
                    (B*y + vectors_y[x][y] + i + l) > (M-1))
                { q2 = 0; }
                 else {
```

```
                q2 = previous[B*x+vectors_x[x][y]+k][B*y+vectors_y[x][y]+i+l];
              }

              if(p1-q2>0)
                   disty += p1-q2;
              else
                   disty += q2-p1;
           }
        }

        if(disty < min2) {
           min2 = disty; besty = i;
        }

        if(distx < min1) {
          min1 = distx; bestx = i;
        }

   }

   S = S/2;
   vectors_x[x][y] += bestx;
   vectors_y[x][y] += besty;
}
```

3. Loop interchange (“merge_int”): Στον νέο κώδικα που προέκυψε από το merge αντιστρέψαμε πάλι την σειρά των βρόχων του βήματος (1), φέρνοντας τους στην αρχική τους μορφή.

4. Invariants Extraction (“inv_extr”): Παρατηρήσαμε ότι σε κάθε επανάληψη του while-loop, υπολογίζεται το 255*B*B, το οποίο ανατίθεται στις μεταβλητές min1 και min2. Εφόσον το αποτέλεσμα αυτής της πράξης δεν αλλάζει κατά την εκτέλεση του προγράμματος, μπορούμε να το υπολογίζουμε εξ’ αρχής. Αυτό μπορεί να γίνει ως εξής:

```
while(S > 0) {
   min1 = 255*B*B;
   min2 = 255*B*B;
   … }
```

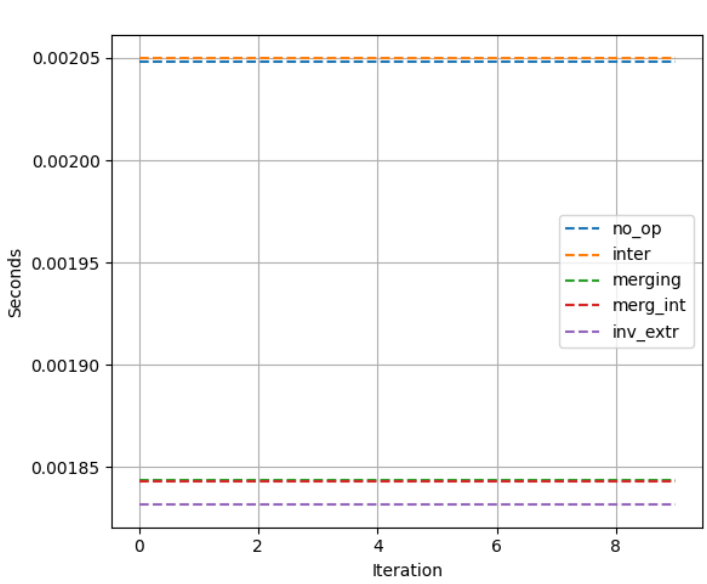
→

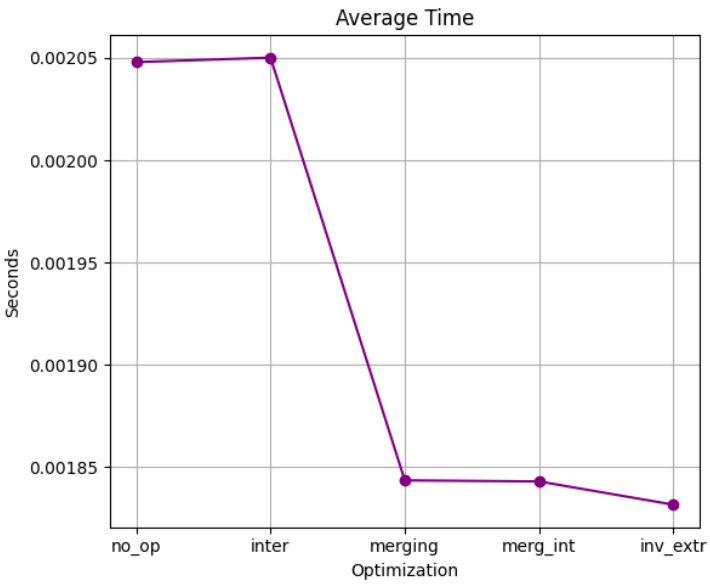
```
const int bb = 255*B*B;
for(x=0; x<N/B; x++) {
  for(y=0; y<M/B; y++) {
    S = 4;
    while(S > 0) {
      min1 = bb;
      min2 = bb;
      … } } }
```

5. Loop Unrolling: Η συγκεκριμένη τεχνική δεν δοκιμάστηκε, καθώς δεν περιμένουμε να έχει μεγάλη επίδραση στην απόδοση του προγράμματος λόγω της μορφής του κώδικα.

1η Άσκηση – Σχεδιασμός Ενσωματωμένων Συστημάτων

Τα αποτελέσματα των επιμέρους εκτελέσεων των παραπάνω βελτιστοποιήσεων και οι μέσοι όροι τους παρουσιάζονται συγκεντρωτικά στα παρακάτω διαγράμματα:



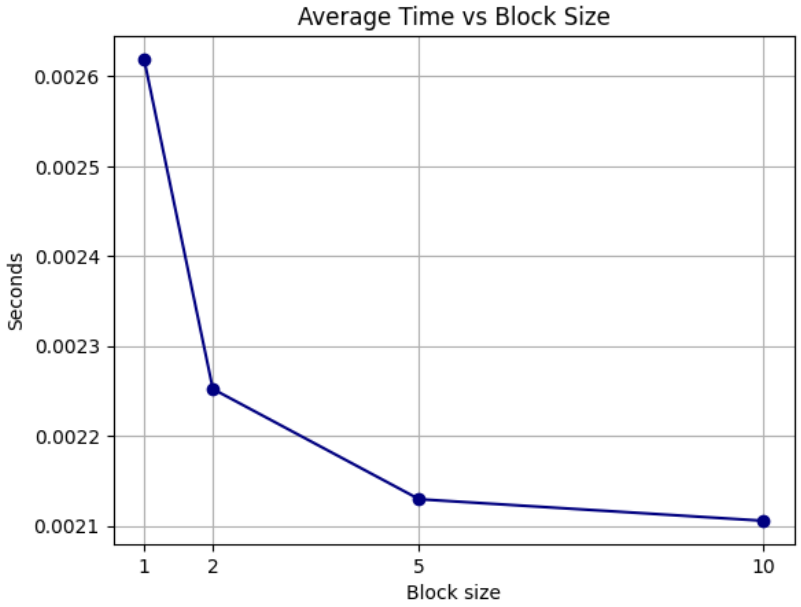


1.4) Πραγματοποιήσαμε Design Space Exploration για την εύρεση του βέλτιστου μεγέθους block B. Προκειμένου να πραγματοποιήσουμε εξαντλητική αναζήτηση, τροποποιήσαμε τον κώδικα ως εξής:

```
int B;

void hal_entry(void) {
    ...
    /* Initialize your variables here */
    const int block_size[4]={1,2,5,10};
    uint32_t start_cycles, end_cycles;
    float execution_time, total_time;
    float max_time, min_time;
    for (int index = 0; index < 4; index++) {
      B = block_size[index];
      int motion_vectors_x[N/B][M/B], motion_vectors_y[N/B][M/B], i, j;
      /* Initialize min_time to a large number */
      total_time = 0; max_time = 0; min_time = 1e6;

      for (int iter = 0; iter < 10; iter++) {...}
      ...
    }
    while(1){}
}
```

Τα αποτελέσματα της αναζήτησης φαίνονται στο διπλανό διάγραμμα.

Η τιμή του B για την οποία πετυχαίνουμε την βέλτιστη απόδοση είναι B=10, ενώ όταν το B ισούται με 1 ή με 2 η επίδοση του προγράμματος φαίνεται να επιδεινώνεται σημαντικά.



Όταν το μέγεθος του μπλοκ B ευθυγραμμίζεται καλά με το μέγεθος των cache lines, η προσπέλαση των δεδομένων γίνεται πιο αποτελεσματική. Εφόσον το B=10 παρέχει την καλύτερη απόδοση, αυτό υποδηλώνει ότι ίσως ταιριάζει καλύτερα με τη δομή των cache lines, αποφεύγοντας αχρείαστα cache misses και μεγιστοποιώντας την επαναχρησιμοποίηση των δεδομένων εντός της data cache.

Παρατηρούμε μια αύξηση στον χρόνο εκτέλεσης σε σχέση με το 1.3, καθώς η μεταβλητή B δεν είναι const πλέον, με αποτέλεσμα να μην μπορεί να αποθηκευθεί στην cache.

1.5) Πραγματοποιήσαμε Design Space Exploration για την εύρεση του βέλτιστου μεγέθους ορθογώνιου block Bx*By. Προκειμένου να πραγματοποιήσουμε εξαντλητική αναζήτηση, τροποποιήσαμε τον κώδικα ως εξής:

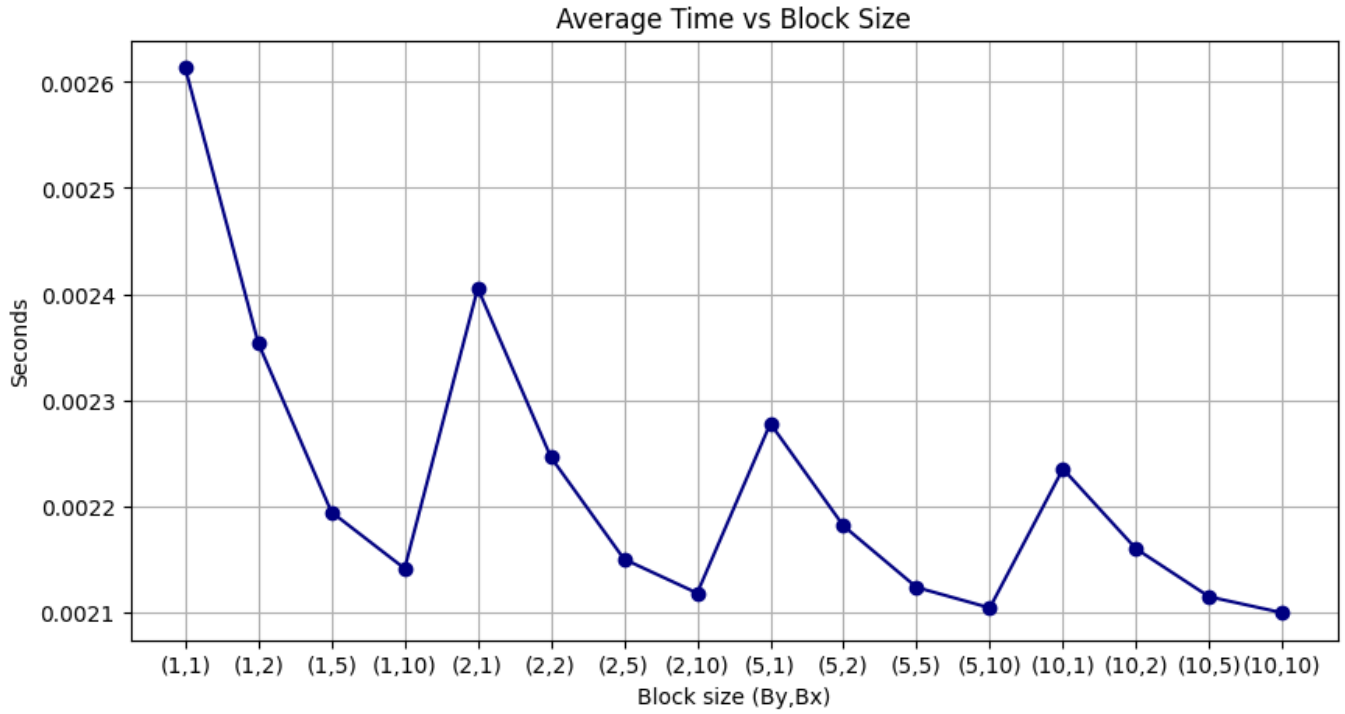
```
...
int Bx, By;

void phods_motion_estimation(const int current[N][M], const int previous[N][M],
        int vectors_x[N/Bx][M/By],int vectors_y[N/Bx][M/By])
{

  ...

  /*Initialize the vector motion matrices*/
  for(i=0; i<N/Bx; i++) {
    for(j=0; j<M/By; j++) {
      vectors_x[i][j] = 0;
      vectors_y[i][j] = 0;
    }
  }

  int bb = 255*Bx*By;
  /*For all blocks in the current frame*/
  for(x=0; x<N/Bx; x++) {
    for(y=0; y<M/By; y++) {
      ...
```

```
      while(S > 0) {
        ...
        for(i=-S; i<S+1; i+=S) {
          ...
          for(k=0; k<Bx; k++) {
            for(l=0; l<By; l++) {
              p1 = current[Bx*x+k][By*y+l];

              if((Bx*x + vectors_x[x][y] + i + k) < 0 ||
                  (Bx*x + vectors_x[x][y] + i + k) > (N-1) ||
                  (By*y + vectors_y[x][y] + l) < 0 ||
                  (By*y + vectors_y[x][y] + l) > (M-1))
              {...}
              else {
                p2 = previous[Bx*x+vectors_x[x][y]+i+k][By*y+vectors_y[x][y]+l];
              }

              ...

              if((Bx*x + vectors_x[x][y] + k) <0 ||
                  (Bx*x + vectors_x[x][y] + k) > (N-1) ||
                  (By*y + vectors_y[x][y] + i + l) < 0 ||
                  (By*y + vectors_y[x][y] + i + l) > (M-1))
              {...}
              else {
                q2 = previous[Bx*x+vectors_x[x][y]+k][By*y+vectors_y[x][y]+i+l];
              }

              ...
            }
          }
          ...
        }
        ...
      }
    }
  }

void hal_entry(void) {
  ...
  /* Initialize your variables here */
  const int block_size[4]={1,2,5,10};
  uint32_t start_cycles, end_cycles;
  float execution_time, total_time;
  float max_time, min_time;

  for (int ix = 0; ix < 4; ix++) {
```

1η Άσκηση – Σχεδιασμός Ενσωματωμένων Συστημάτων

```
    Bx = block_size[ix];
    for (int iy = 0; iy < 4; iy++) {
      By = block_size[iy];
      int motion_vectors_x[N/Bx][M/By], motion_vectors_y[N/Bx][M/By], i, j;
      // Initialize min_time to a large number
      total_time = 0; max_time = 0; min_time = 1e6;

      for (int iter = 0; iter < 10; iter++) {...}
      ...
    }
  }
  while(1){}
}
```

Τα αποτελέσματα της αναζήτησης φαίνονται στο παρακάτω διάγραμμα



Παρατηρούμε ότι η καλύτερη επίδοση εμφανίζεται για Bx=By=10, ενώ για By=5,Bx=10 η επίδοση είναι ελάχιστα χειρότερη. Ένας ευρυστικός τρόπος προκειμένου να περιόρισουμε την αναζήτηση είναι να εξετάσουμε αρχικά τις τετραγωνικές ή σχεδόν τετραγωνικές διαστάσεις (Bx ≈ By), καθώς αυτά τα block προσφέρουν συχνά καλύτερη απόδοση. Επίσης, μπορούμε να περιορίσουμε την αναζήτηση σε ένα διάστημα τιμών για τα Bx, By αποφεύγοντας πολύ μεγάλα και πολύ μικρά block. Στην συνέχεια, με βάση τα αποτελέσματα, μπορούμε να επεκτείνουμε ή να περιορίσουμε το εύρος αναζήτησης ανάλογα με το που φαίνεται να υπάρχει καλύτερη απόδοση.

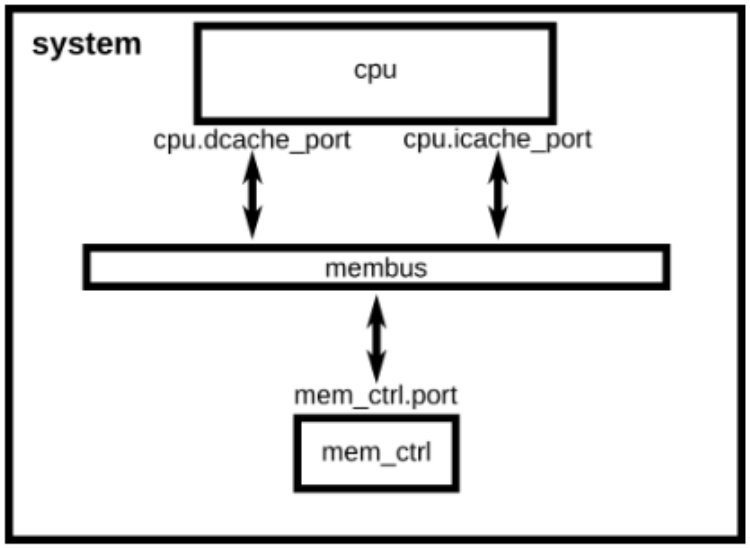
1η Άσκηση – Σχεδιασμός Ενσωματωμένων Συστημάτων

Ζητούμενο 2ο

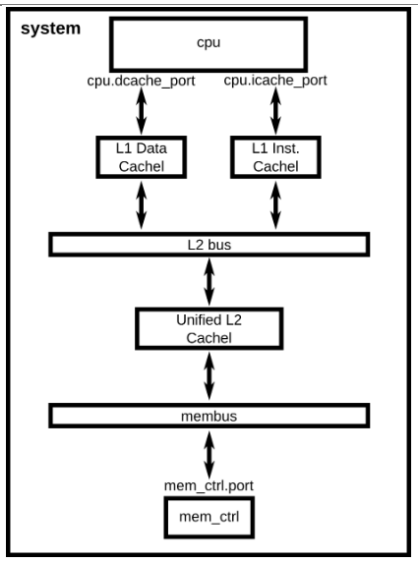
*Για τον σχεδιασμό των παρακάτω plot χρησιμοποιήθηκε το συνημμένο αρχείο embedded_1_2.ipynb, ενώ για την εκτέλεση του 2.4 και 2.5, το script.sh και το script2.sh αντίστοιχα.

**Η προσομοίωση της εφαρμογής tables.exe θα εκτελεστεί για τις 2 παρακάτω αρχιτεκτονικές

Αρχιτεκτονική Α'



Αρχιτεκτονική Β'

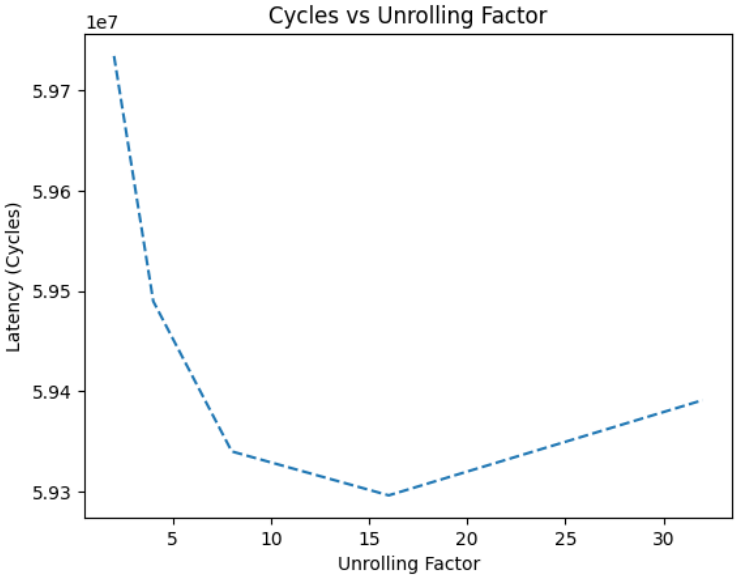


2.1) Η εκτέλεση της προσομοίωσης στην πρώτη αρχιτεκτονική απαιτεί **858973820** κύκλους.

2.2) Η εκτέλεση της προσομοίωσης στην δεύτερη αρχιτεκτονική απαιτεί **59787087** κύκλους.

Παρατηρούμε ότι η πρώτη αρχιτεκτονική χρειάζεται πολλούς περισσότερους κύκλους για να ολοκληρώσει την εκτέλεση του προγράμματος. Αυτό συμβαίνει, καθώς στην αρχιτεκτονική χωρίς cache, κάθε φορά που ο επεξεργαστής χρειάζεται δεδομένα ή εντολές, πρέπει να αναζητήσει την πληροφορία απευθείας στην κύρια μνήμη. Στη δεύτερη αρχιτεκτονική, τα δεδομένα που χρησιμοποιούνται αποθηκεύονται στα δύο επίπεδα της cache. Έτσι, αν τα δεδομένα αυτά χρειαστούν ξανά, ο επεξεργαστής μπορεί να τα βρει στις caches χωρίς να χρειάζεται να τα αναζητήσει στην πιο αργή κύρια μνήμη.

2.3)

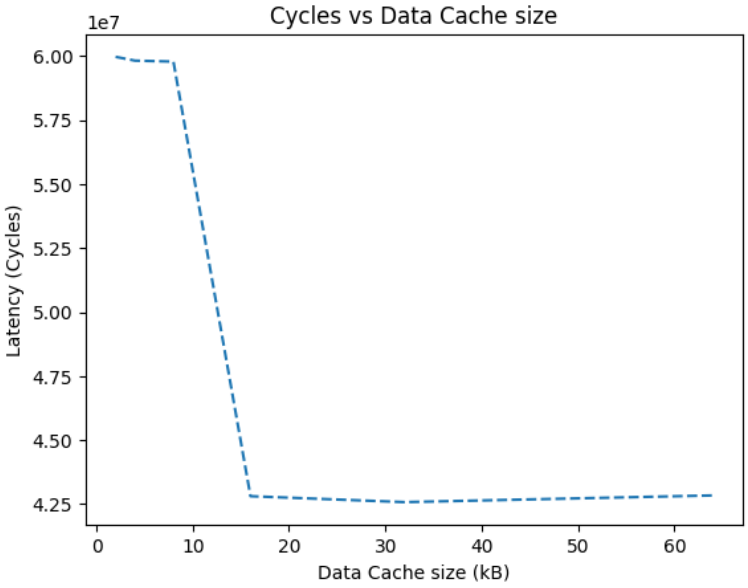


Στο διπλανό διάγραμμα παρουσιάζεται το πλήθος των κύκλων που απαιτούνται συναρτήσει του unrolling factor. Παρατηρούμε ότι η βέλτιστη επίδοση εμφανίζεται για unrolling factor ίσο με το 16, καθώς για unrolling factor = 32 οι κύκλοι εκτέλεσης αυξάνονται.

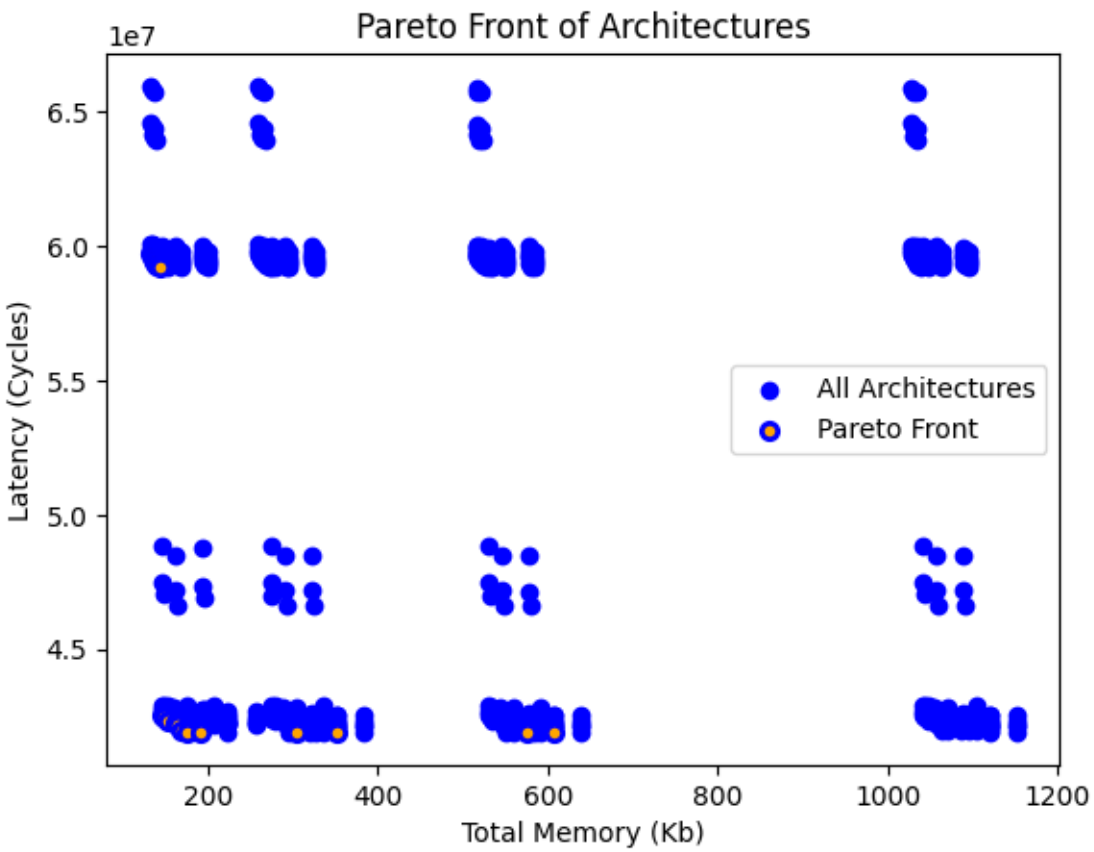
Η μείωση των κύκλων για unrolling factor ≤ 16 μπορεί να οφείλεται στην μείωση των εντολών ελέγχου και στον παραλληλισμό των εντολών (στην περίπτωση που υπάρχουν πολλές ανεξάρτητες πράξεις). Από την άλλη, το μεγάλο unrolling factor αυξάνει σημαντικά τον κώδικα του βρόχου, με αποτέλεσμα να μην εκμεταλλευόμαστε το pipeline ή να απαιτούνται συνεχώς νέα δεδομένα του πίνακα στις caches.

Από το διπλανό διάγραμμα, παρατηρούμε μια απότομη βελτίωση της επίδοσης για μέγεθος cache ίσο με 16kB, ενώ η αύξηση της μνήμης μετά από αυτό το «κατώφλι» δεν μειώνει τους κύκλους που απαιτούνται.

Αυτό σημαίνει ότι μια cache μεγέθους 16kB είναι αρκετή για να φιλοξενήσει όλα τα δεδομένα των πινάκων που χρειάζεται η συγκεκριμένη εφαρμογή.



2.4) Εκτελώντας εξαντλητική αναζήτηση (συνολική διάρκεια: **2hr 45min 26sec**) στο χώρο που μας δίνεται, προκύπτει το πλήθος των κύκλων που απαιτείται για την εκτέλεση της εφαρμογής για κάθε συνδυασμό (**results.csv**). Στην συνέχεια, μέσω του embedded_1_2.ipynb, εντοπίζουμε το Pareto Frontier μαζί με τα στοιχεία της αρχιτεκτονικής που οδήγησαν στο εκάστοτε Pareto optimal σημείο (**pareto_front.csv**). Το Pareto Set μαζί με το Pareto Frontier φαίνεται στο παρακάτω διάγραμμα.



2.5) Ο γενετικός αλγόριθμος του Python script ολοκληρώνει την αναζήτηση μέσα σε **8min 50sec**, εκτελώντας μόνο **30 evaluations** για να βρει μια αποδεκτή λύση. Αντίθετα, η εξαντλητική αναζήτηση χρειάζεται **2hr 45min 26sec** για να ολοκληρωθεί και απαιτεί **720 αξιολογήσεις**, καθώς εξετάζει όλες τις πιθανές αρχιτεκτονικές. Παρατηρώντας το Pareto Front που προκύπτει από τον γενετικό αλγόριθμο (**genOptimizer_PO.csv**), μπορεί κανείς να αντιληφθεί ότι αποτελείται από αρκετά λιγότερα optimal σημεία σε σχέση με το Pareto Frontier που προέκυψε από την εξαντλητική αναζήτηση (**4 σημεία έναντι 18**). Ωστόσο, τα optimal σημεία του **genOptimizer_PO.csv** φαίνεται να μην εμφανίζονται (με εξαίρεση 1) στο Pareto Frontier της εξαντλητικής αναζήτησης.