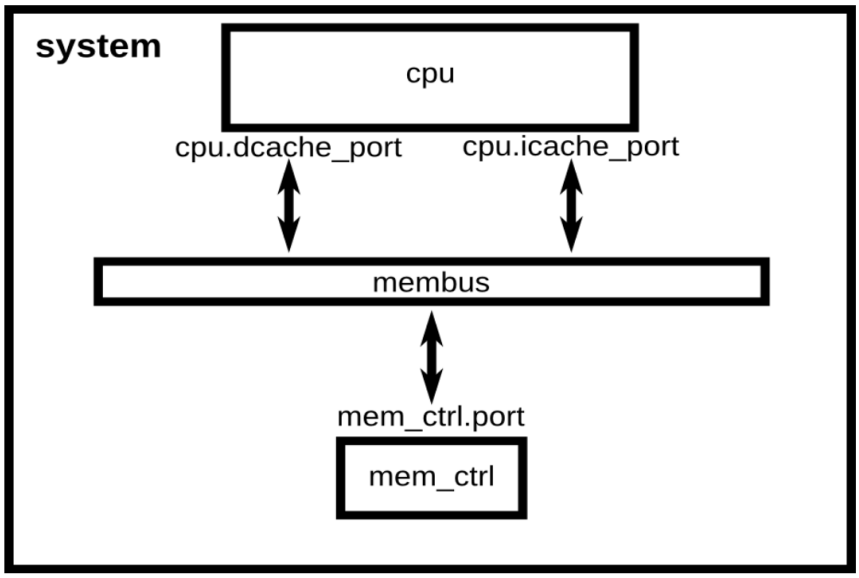
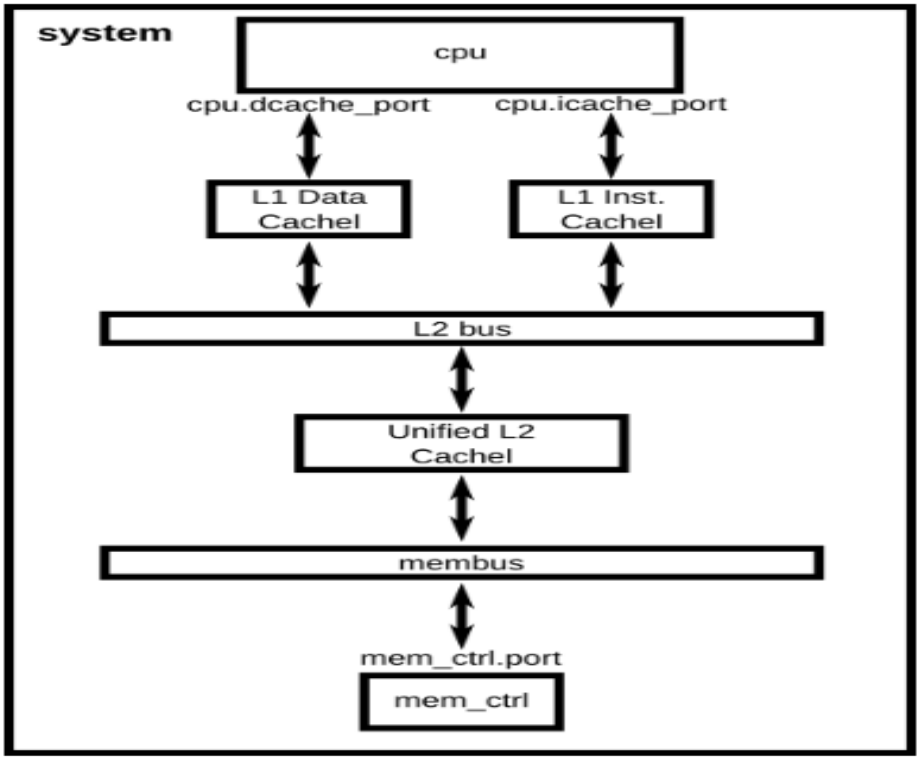
Ζητούμενο 2ο : Βελτιστοποίηση εφαρμογής σε διαφορετικές αρχιτεκτονικές x86

Σε αυτό το ζητούμενο σκοπός είναι να συγκρίνουμε και να βελτιστοποιήσουμε εφαρμογή που εκτελεί προσπέλαση και πράξεις πινάκων προσομοιώνοντας μέσω του Gem5 διαφορετικές αρχιτεκτονικές.

1) Η **πρώτη αρχιτεκτονική** που θα εξετάσουμε φαίνεται στην συνέχεια και όπως βλέπουμε δεν διαθέτει καμία cache memory (μόνο επεξεργαστή και κύρια μνήμη) οπότε περιμένουμε να μην είναι εξαιρετικά αποδοτική αλλά αντιθέτως αρκετά αργή. Αφού προσομοιώσουμε την tables.exe στην πρώτη αρχιτεκτονική βλέπουμε τους κύκλους εκτέλεσης που χρειάζεται. Συγκεκριμένα μέσω της μεταβλητής system.cpu.numCycles μετράμε **857669639 κύκλους** εκτέλεσης.



2) Στην συνέχεια εξετάζουμε την ίδια εφαρμογή όμως τώρα χρησιμοποιώντας την **δεύτερη αρχιτεκτονική** στην οποία όμως προσθέτουμε δύο επίπεδα cache μνήμης την L1 cache (ξεχωριστά για δεδομένα και εντολές) και την L2 cache (ενιαία για δεδομένα και εντολές).Προφανώς περιμένουμε οι κύκλοι εκτέλεσης του προγράμματος να μειωθούν σημαντικά καθώς θα εκμεταλλευτούμε την τοπικότητα δηλαδή η cpu θα αποθηκεύει τα δεδομένα που χρησιμοποιεί σωστά στην cache memory και δεν θα χρειάζεται να διαβάζει συνεχώς από την κύρια μνήμη τα δεδομένα αλλά μόνο όταν αυτά δεν υπάρχουν στην cache. Σημειώνουμε ότι όπως είναι λογικό ο χρόνος προσπέλασης δεδομένων από την κύρια μνήμη είναι σημαντικά μεγαλύτερος σε σχέση με την cache memory. Ομοίως με πριν προσομοιώσουμε την tables.exe μέσω της μεταβλητής system.cpu.numCycles μετράμε **59792116 κύκλους** εκτέλεσης. Παρατηρούμε ότι έχουμε περίπου **93% μείωση** των κύκλων εκτέλεσης που υποδηλώνει ότι η χρήση των cache μνημών είναι απαραίτητη.

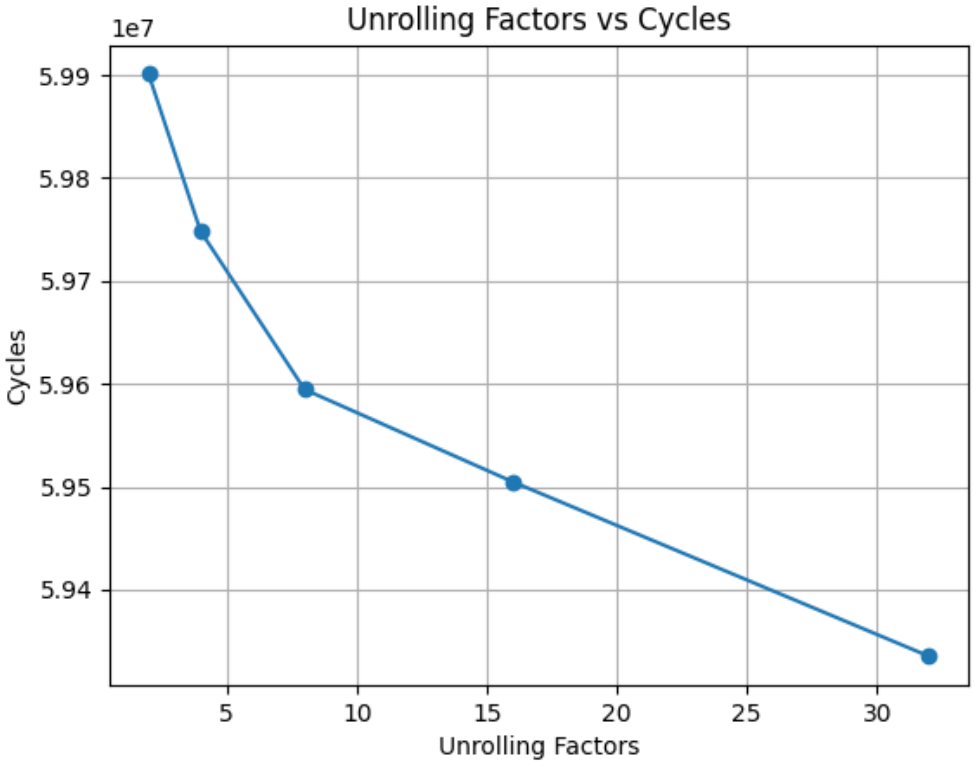


*Σημειώνουμε ότι οι L1 είναι 8KB η κάθε μία και η L2 128KB ,προφανώς η L1 είναι ακόμη ταχύτερη από την L2.

3) Στην συνέχεια για την παραπάνω αρχιτεκτονική (αρχιτεκτονική 2) θα προσομοιώσουμε την εφαρμογή για διαφορετικά unrolling factors.

Δηλαδή να εκτελέσουμε for loops στην εφαρμογή μας με τέτοιο τρόπο που αντί για παράδειγμα να χρειαστούμε δέκα επαναλήψεις να χρειαστούμε μόνο δύο αλλά σε αυτές τις δύο να κάνουμε περισσότερους υπολογισμούς και προσπελάσεις στην μνήμη.

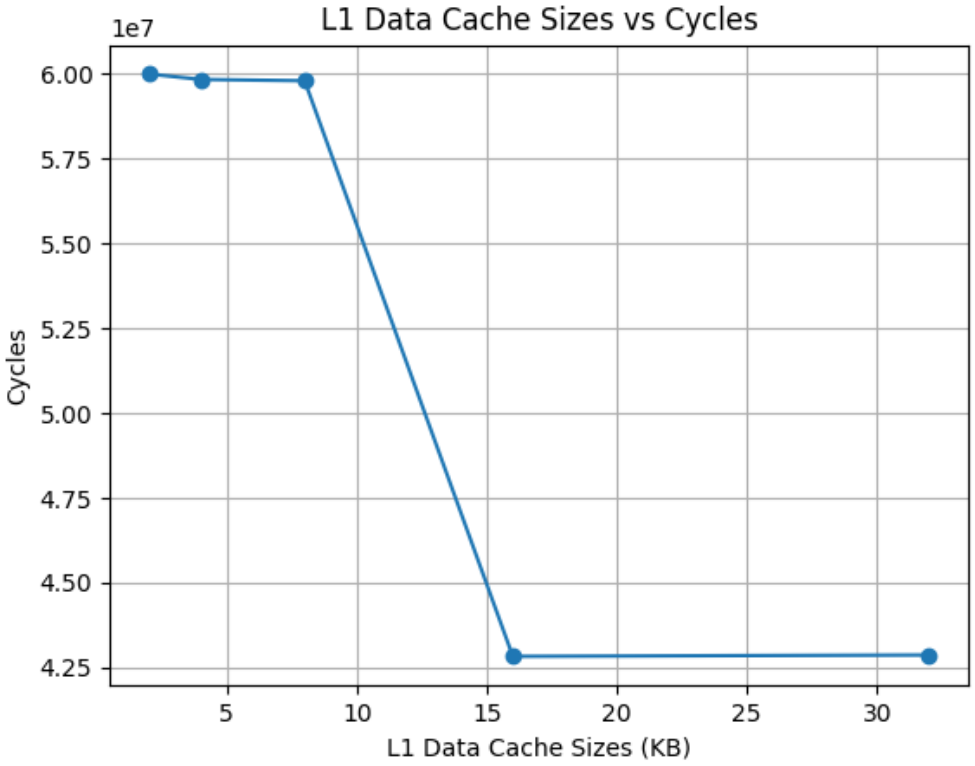
Unrolling Factors	**Cycles**
2	59901259
4	59748430
8	59594513
16	59504767
32	59335106



Παρατηρήσεις : Όσο αυξάνεται το unrolling factor βλέπουμε ότι μειώνονται οι κύκλοι εκτέλεσης της εφαρμογής αυτό οφείλεται στο γεγονός ότι γίνονται πλέον λιγότερες επαναλήψεις συνεπώς λιγότεροι έλεγχοι συνθήκης και σε επίπεδο assembly λιγότερά jumps.Βέβαια το πρόβλημα δεν κλιμακώνει δηλαδή όσο αυξάνει το unrolling factor μειώνεται ο ρυθμός πτώσης των κύκλων εκτέλεσης.

Στην συνέχεια θα τροποποιήσουμε ξανά την δεύτερη αρχιτεκτονική αλλά τώρα θα δοκιμάσουμε τις επιδόσεις και τους κύκλους εκτέλεσης για διάφορες τιμές της L1 data cache.

L1 data cache (KB)	**Cycles**
2	59992651
4	59831183
8	59792116
16	42834630
32	42875295



Παρατηρήσεις :

Αρχικά, παρατηρούμε ότι για μικρές τιμές της L1 Cache (2KB, 4KB, 8KB), ο ρυθμός μείωσης των κύκλων εκτέλεσης είναι χαμηλός. Αυτό υπονοεί ότι η επίδραση της μνήμης cache στην απόδοση είναι περιορισμένη για μικρές μνήμες. Πιθανόν, ο υψηλός χρόνος εκτέλεσης για αυτές τις μικρές τιμές οφείλεται σε συχνά cache misses λόγω περιορισμένου χώρου αποθήκευσης δεδομένων.

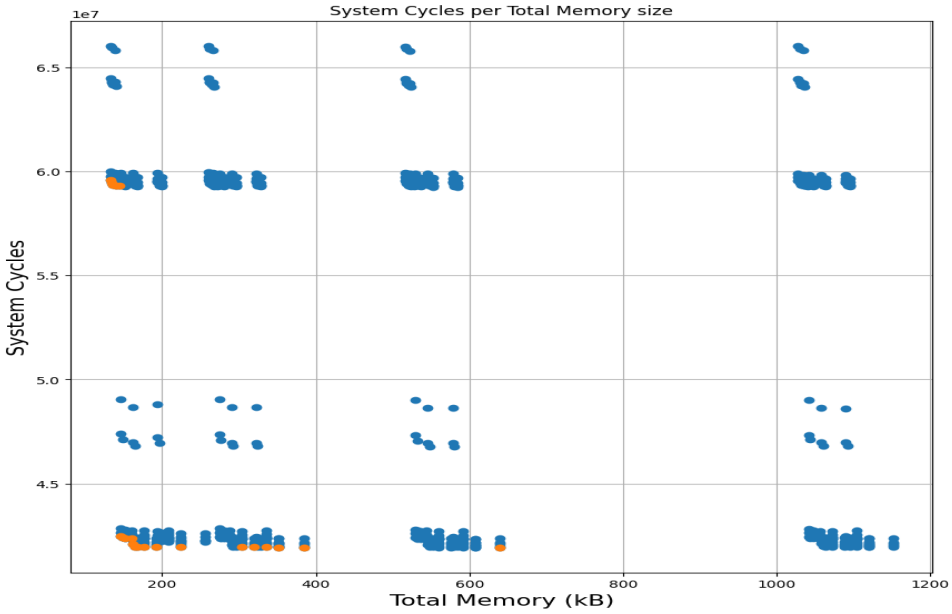
Στη συνέχεια, για μέγεθος cache 16KB, παρατηρούμε σημαντική βελτίωση στον χρόνο εκτέλεσης. Αυτό υποδηλώνει ότι η αύξηση του μεγέθους της cache έχει σημαντικό αντίκτυπο στη μείωση των cache misses. Η εφαρμογή φαίνεται να επωφελείται σημαντικά από την αυξημένη χωρητικότητα της cache για την αποθήκευση και ανάκτηση δεδομένων.

Ωστόσο, για cache μεγέθους 32KB, παρατηρούμε ότι παρά το γεγονός ότι ο χρόνος εκτέλεσης είναι σημαντικά καλύτερος σε σύγκριση με μικρότερα μεγέθη, δεν παρατηρείται περαιτέρω βελτίωση σε σχέση με το 16KB. Αυτό ενδεχομένως υποδηλώνει ότι η εφαρμογή δεν είναι πλέον υποχρεωτικά memory-bound, αλλά και άλλοι παράγοντες, όπως η επεξεργαστική ισχύς, επηρεάζουν την τελική απόδοση.

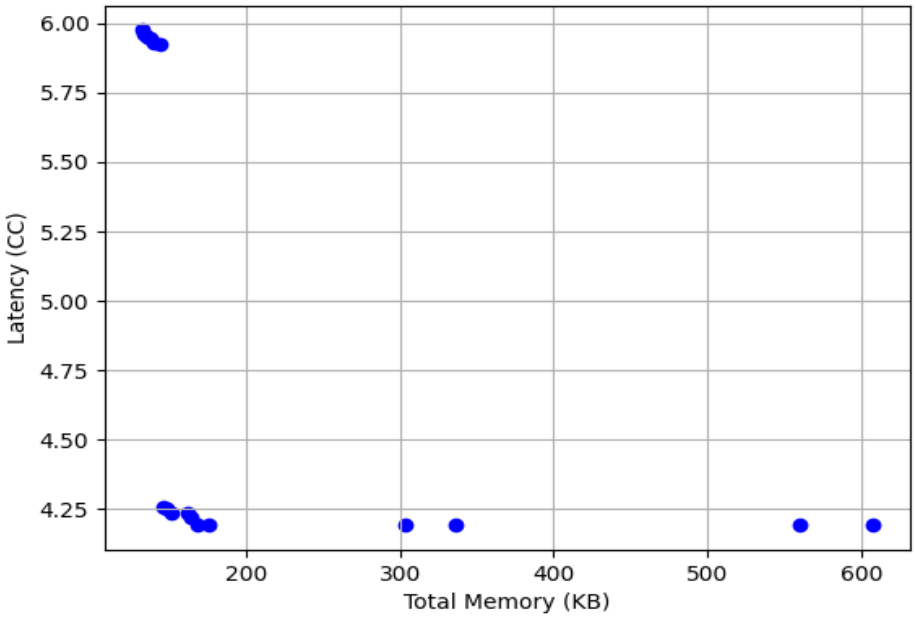
Τέλος, παρατηρούμε μια ελαφρά αύξηση των κύκλων εκτέλεσης από 16KB σε 32KB. Αυτό μπορεί να οφείλεται στο γεγονός ότι η αύξηση του μεγέθους της cache δεν είναι γραμμικά αναλογική με τη μείωση του χρόνου πρόσβασης, και ενδεχομένως για αυξημένο μέγεθος να υπάρχει μια μικρή επιβάρυνση στην πρόσβαση. Που θα χρειαζόμασταν περισσότερη μνήμη τα 32KB θα βελτίωναν ακόμη περισσότερο την επίδοση της εφαρμογής.

4) Στην συνέχεια θα εκτελέσουμε εξαντλητική αναζήτηση για κάθε συνδυασμό των μνήμων L1 data ,L1 instruction ,L2 cache και του unrolling factor.

Σκοπός αυτή της αναζήτησης είναι να εντοπίσουμε το Pareto Frontier και να το σχεδιάσουμε κατάλληλα. Σημειώνουμε ότι πλέον έχουμε πάρα πολλούς συνδυασμούς και συνεπώς χρειάζεται να γράψουμε ένα κατάλληλο bash script ώστε με ένα τετραπλό nested for loop να δοκιμάσουμε όλους τους συνδυασμούς. Στην συνέχεια κρατάμε τους χρόνους εκτέλεσης για την κάθε περίπτωση σε ένα txt αρχείο . Μέσω python (ή με το χέρι) περνάμε αυτές τις τιμές σε ένα csv file. Τέλος με ένα κατάλληλο script στην pyhton έχουμε το pareto frontier (design space and pareto points).



Στην συνέχεια με μπλε χρώμα βλέπουμε το pareto frontier.



Βρήκαμε συνολικά 17 σημεία pareto μέσω της εξαντλητικής αναζήτησης τα οποία φαίνονται στον πίνακα παρακάτω.

Θυμίζουμε ότι εξετάσαμε όλους τους συνδυασμούς για τις τιμές

L1D cache size ∈ [2kB, 4kB, 8kB, 16kB, 32kB, 64kB]

L1I cache size ∈ [2kB, 4kB, 8kB, 16kB, 32kB, 64kB]

L2 cache size ∈ [128kB, 256kB, 512kB, 1024kB]

Unrolling factor ∈ [2, 4, 8, 16, 32]

L1instructions	L1 Data	L2 cache	Total memory	Unrolling Factor	Cycles
2	2	128	132	8	59754653
2	16	128	146	8	42570518
2	32	128	162	8	42340045
4	2	128	134	16	59603493
4	4	128	136	16	59492753
4	16	128	148	16	42497008
4	32	128	164	16	42206541
8	2	128	138	32	59465800
8	4	128	140	32	59304322
8	8	128	144	32	59265171
8	16	128	152	32	42344808
8	32	128	168	32	41954228
16	32	128	176	32	41933696
16	32	256	304	32	41923222
16	32	512	560	32	41916540
16	64	256	336	32	41921152
32	64	512	608	32	41913320

Παρατηρήσεις: Γενικά τα περισσότερα σημεία το pareto frontier έχουν Unrolling factor 32 που σημαίνει ότι αυτό βελτιώνει αρκετά τον χρόνο εκτέλεσης του προγράμματος μας. Παρατηρούμε ότι η L2 cache δεν μειώνει τους κύκλους εκτέλεσης και για αυτό παραμένει συνήθως στα 128KB που είναι η ελάχιστη περίπτωση στις αρχιτεκτονικές που δοκιμάσαμε .Αυτό σημαίνει ότι 128KB συνήθως αρκούν για να αποθηκευόσουν όσα δεδομένα χρειάζεται για να εκμεταλλευτούμε την τονικότητα και να μειώσουμε τα cache misses.Επίσης βλέπουμε ότι για μεγάλη cache L1 γενικά έχουμε βέλτιστους χρόνους και ότι όσο αυξάνουμε την μνήμη συνολικά καταφέρνουμε να μειώσουμε τον χρόνο εκτέλεσης. Είναι θέμα του μηχανικού στην συνέχεια ανάλογα με το πρόβλημα που θέλει να επιλύσει να διαλέξει το κατάλληλο pareto σημείο και συνάμα τις επιμέρους μνήμες.

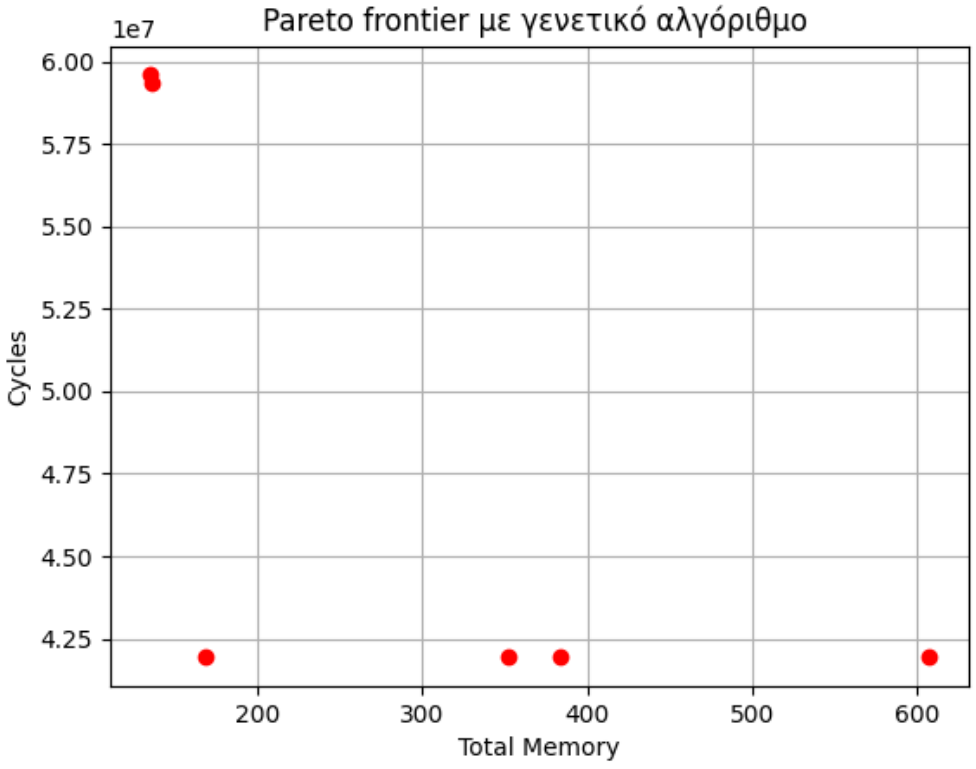
5) Τέλος θα εκτελέσουμε και πάλι εξερεύνηση αλλά τώρα με την χρήση γενετικού αλγορίθμου.

Σημειώνουμε ότι ο αλγόριθμος μας δίνεται έτοιμος και μάλιστα σε μόλις 10 με 15 λεπτά έχουμε έτοιμα τα αποτελέσματα και το pareto frontier που έχει παράξει σε αντίθεση με την εξαντλητική εξερεύνηση που διήρκησε πάνω από 4 ώρες.

```
Pareto Optimal Configurations

1) L1I (KB) = 4 L1D (KB) = 4 L2 (KB) = 128 UF = 16 config. results in Execution Time (CC) = 59348248 and Total Memory (KB) = 136
2) L1I (KB) = 2 L1D (KB) = 4 L2 (KB) = 128 UF = 4 config. results in Execution Time (CC) = 59591662 and Total Memory (KB) = 134
3) L1I (KB) = 32 L1D (KB) = 64 L2 (KB) = 256 UF = 16 config. results in Execution Time (CC) = 41939934 and Total Memory (KB) = 352
4) L1I (KB) = 32 L1D (KB) = 64 L2 (KB) = 512 UF = 16 config. results in Execution Time (CC) = 41934061 and Total Memory (KB) = 608
5) L1I (KB) = 64 L1D (KB) = 64 L2 (KB) = 256 UF = 16 config. results in Execution Time (CC) = 41935983 and Total Memory (KB) = 384
6) L1I (KB) = 8 L1D (KB) = 32 L2 (KB) = 128 UF = 16 config. results in Execution Time (CC) = 41964864 and Total Memory (KB) = 168
```

L1I(KB)	L1D(KB)	L2(KB)	UF	Total Memory	Cycles
4	4	128	16	136	59348248
2	4	128	4	134	59591662
32	64	256	16	352	41939934
32	64	512	16	608	41934006
64	64	256	16	384	41935983
8	32	128	16	168	41964864



Αρχικά παρατηρούμε ότι η εξαντλητική αναζήτηση βρήκε όλα τα pareto points (17 στο σύνολο) ενώ με τον γενετικό αλγόριθμο χάσαμε αρκετά σημεία (μόλις 6 στο σύνολο) , κάτι το οποίο μπορεί να θεωρηθεί και trade-off συγκριτικά με τον χρόνο αναζήτησης και διεξαγωγής αυτών των δεδομένων. Ένα ακόμη βασικό μειονέκτημα του γενετικού αλγόριθμου είναι ότι υπάρχει περίπτωση εκτός από να χαθούν σημεία pareto να θεωρήσει λάθος σημεία ως σημεία pareto.Για παράδειγμα στις δικές μας μετρήσεις παρατηρούμε ότι έχει βγάλει αρκετά σημεία που δεν υπάρχουν στην εξαντλητική αναζήτηση και από το πινακάκι φαίνεται ξεκάθαρα ότι στον γενετικό αλγόριθμο δεν έχουμε UF 32 που κυριαρχούσε στην εξαντλητική αναζήτηση.