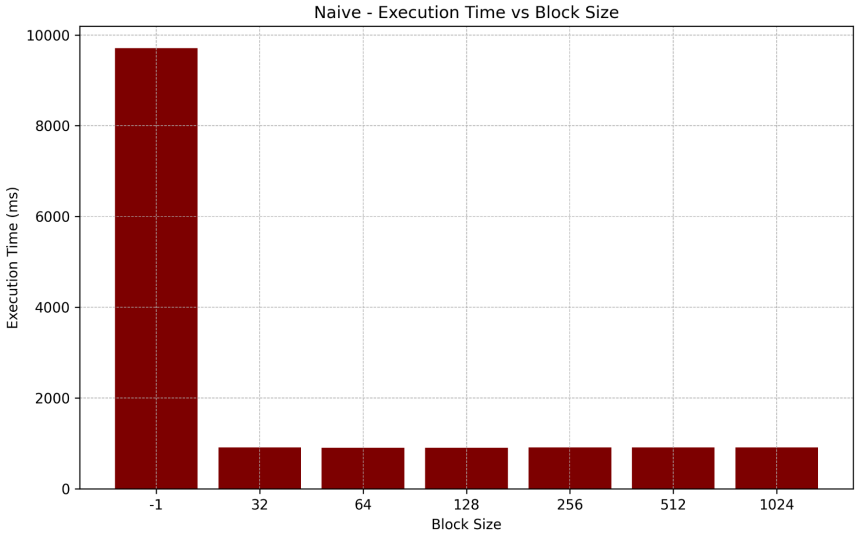
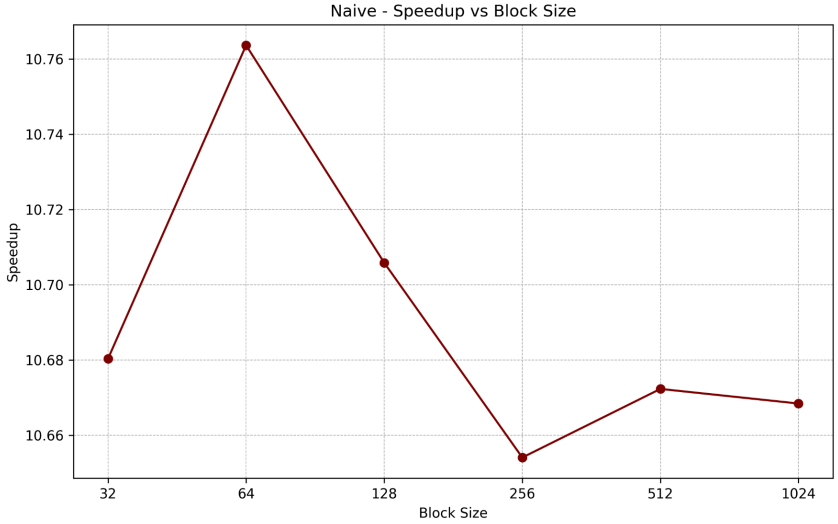
4. Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Μέχρι και bottleneck analysis έχουμε το default Config:{256,2,16,10}

Naive Implementation:





Για τη παράλληλη υλοποίηση αναθέτουμε στη GPU τον υπολογισμό των αποστάσεων των objects απο τα κεντρα των clusters, αναθέτουμε σε κάθε thread ενα object. Παρατηρουμε στο διαγραμμα του χρόνου εκτέλεσης τεράστια διαφορά μεταξύ σειριακής υλοποίησης (ορισμένη αυθαίρετα με block_size = -1) και παράλληλης υλοποίησης. Η παράλληλη υλοποίηση είναι μια τάξη (~10) μεγέθους μικρότερη σε χρόνο.
Το block size δε φαίνεται να παίζει ιδιαίτερο ρόλο για αυτήν την έκδοση. Για να έχουμε καλύτερη εικόνα για το πως επηρεάζει το block size οφείλουμε να γνωρίζουμε παραπάνω πληροφορίες για το hardware στο οποίο τρέχει η εφαρμογή.
Με το παρακάτω C++ αρχείο και με τη βοήθεια του CUDA API που παρέχει τη συνάρτηση cudaGetDeviceProperties() λαμβάνουμε πληροφορίες για πληροφορίες όπως maxThreadsPerBlock ώστε να σχηματίσουμε πληρέστερη εικόνα:

```
#include <iostream>
using namespace std;

int main()
{

   cout <<"=========" << endl << endl;
   cout << "CUDA version:   v" << CUDART_VERSION << endl;



   int devCount;
   const int kb = 1024;
   const int mb = kb * kb;
   cudaGetDeviceCount(&devCount);
   cout << "CUDA Devices: " << endl << endl;

   for(int i = 0; i < devCount; ++i)
   {
       cudaDeviceProp props;
       cudaGetDeviceProperties(&props, i);
       cout << i << ": " << props.name << ": " << props.major << "." <<
props.minor << endl;
       cout << "  Global memory:   " << props.totalGlobalMem / mb << "mb" <<
endl;
       cout << "  Shared memory:   " << props.sharedMemPerBlock / kb << "kb" <<
endl;
       cout << "  Constant memory: " << props.totalConstMem / kb << "kb" <<
endl;
       cout << "  Block registers: " << props.regsPerBlock << endl << endl;
```

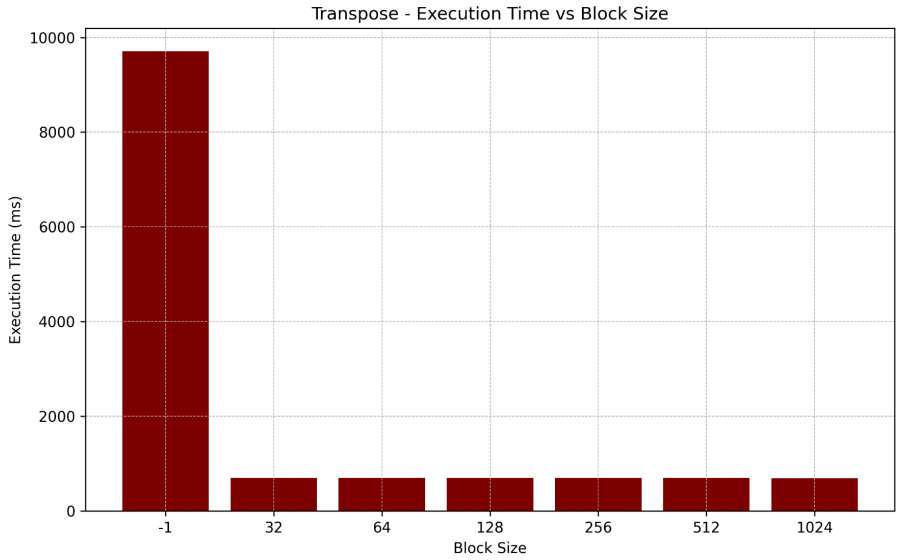
```
      cout << "  Max threads per Multiprocessor: " <<
props.maxThreadsPerMultiProcessor << endl;
      cout << "  Max threads per Block: " << props.maxThreadsPerBlock << endl;
      cout << "  Number of Multiprocessors: " << props.multiProcessorCount <<
endl << endl;


      cout << "  Warp size:         " << props.warpSize << endl;
      cout << "  Threads per block: " << props.maxThreadsPerBlock << endl;
      cout << "  Max block dimensions: [ " << props.maxThreadsDim[0] << ", "
<< props.maxThreadsDim[1]  <<
", " << props.maxThreadsDim[2] << " ]" << endl;
      cout << "  Max grid dimensions:  [ " << props.maxGridSize[0] << ", " <<
props.maxGridSize[1]  << ", "
<< props.maxGridSize[2] << " ]" << endl;
      cout << endl;
   }
}
```

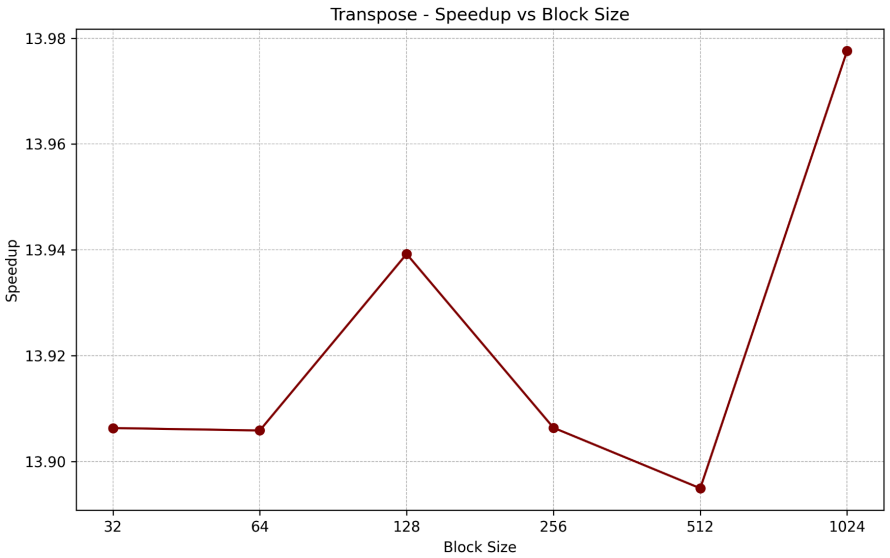
Χρησιμοποιούμε ως baseline μια Tesla V100-SXM2-32GB GPU για όλες τις μετρήσεις μας:

```
1: Tesla V100-SXM2-32GB: 7.0
  Global memory:   32501mb
  Shared memory:   48kb
  Constant memory: 64kb
  Block registers: 65536

  Max threads per Multiprocessor: 2048
  Max threads per Block: 1024
  Number of Multiprocessors: 80

  Warp size:         32
  Threads per block: 1024
  Max block dimensions: [ 1024, 1024, 64 ]
  Max grid dimensions:  [ 2147483647, 65535, 65535 ]
```

Στη Naive υλοποίηση έχουμε peak performance για block size 128, χωρίς όμως να διαφέρει ιδιαίτερα το speedup συγκριτικά με τα υπόλοιπα block_sizes > 32. Επομένως φαίνεται να μην έχουμε αξιοσημείωτο hit στην απόδοση παίζοντας με το block size.

Transpose:





Στη Transpose version αλλάζει μόνο η δομή των clusters πινάκων:

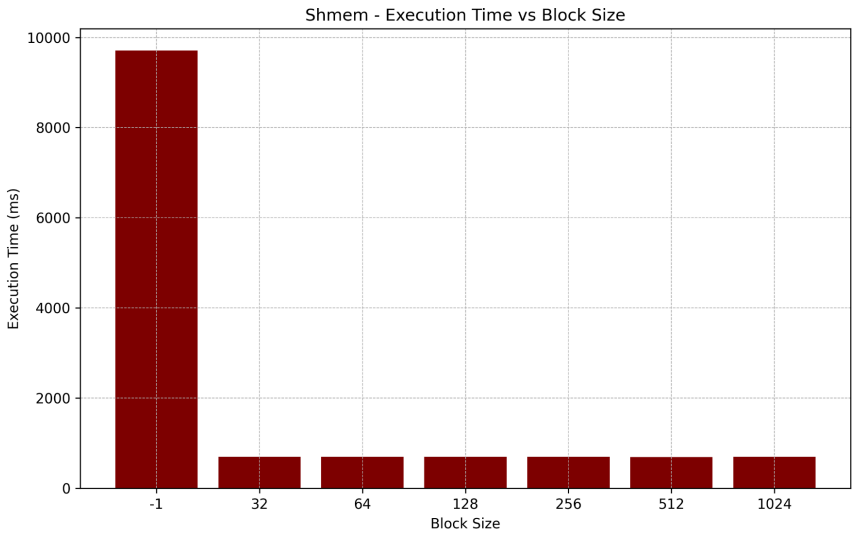
```
double *clusters,     // [numCoords][numClusters]
```

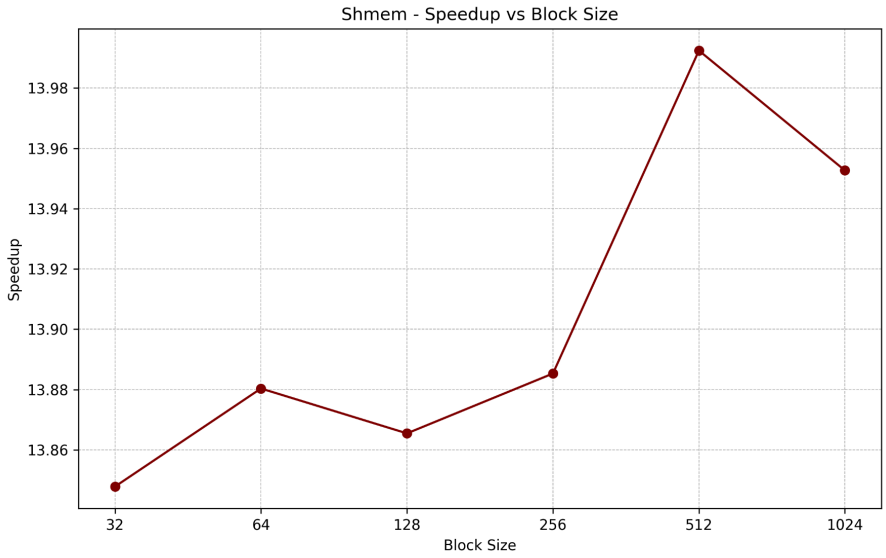
και πλέον έχουμε την ανάστροφη μορφή της Naive:

```
double *clusters,     // [numClusters][numCoords]
```

Αυτή αλλαγή ωφελεί το access στη RAM όταν επιθυμούμε να φορτώσουμε και να μεταφέρουμε δεδομένα στη GPU. Τα coordinates αυτά, χάρη στην transpose συνάρτηση, βρίσκονται σε γειτονικές θέσεις μνήμης (στη naive βρίσκονται σε γειτονικές θέσεις τα διαφορετικά coordinates του ίδιου object) και μπορεί να τα πάρει μαζικά και να τα διαμοιράσει στα threads που είναι υπεύθυνα για τον υπολογισμό. Έτσι μειώνουμε τη καθυστέρηση λόγω accesses καθώς μπορεί η GPU να μαζέψει συνολικά τα objects με ένα access. Ως προς τη αλλαγή συμπεριφοράς με block size δεν αλλάζει κάτι.

Shared:





Στη shared υλοποίηση αποθηκεύουμε τα clusters στην κοινή μνήμη της GPU. Με αυτόν τον τρόπο αποφεύγονται τα accesses στην κύρια μνήμη για τον υπολογισμό της απόστασης του κάθε object από όλα τα clusters, τα οποία στοιχίζουν χρόνο λόγω του latency. Δεσμεύουμε λοιπόν μνήμη μεγέθους `numCoords*numClusters*sizeof(double)`

Παρατηρούμε γρηγορότερη ή και ίση σε κάποιες φάσεις) εκτέλεση από τις Naive και Transpose υλοποιήσεις, αναμενόμενο λόγω της γρήγορης shared μνήμης χωρίς την ανάγκη για πολλα accesses στη κύρια μνήμη.

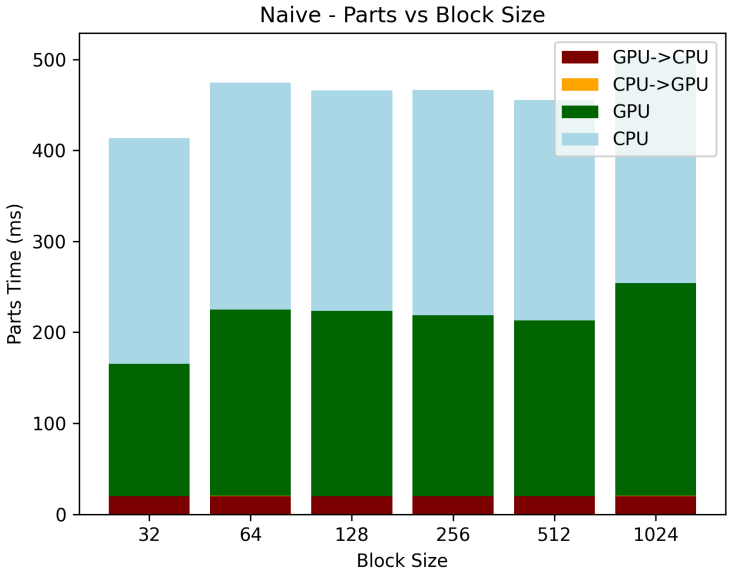
Γενικές Παρατηρήσεις:

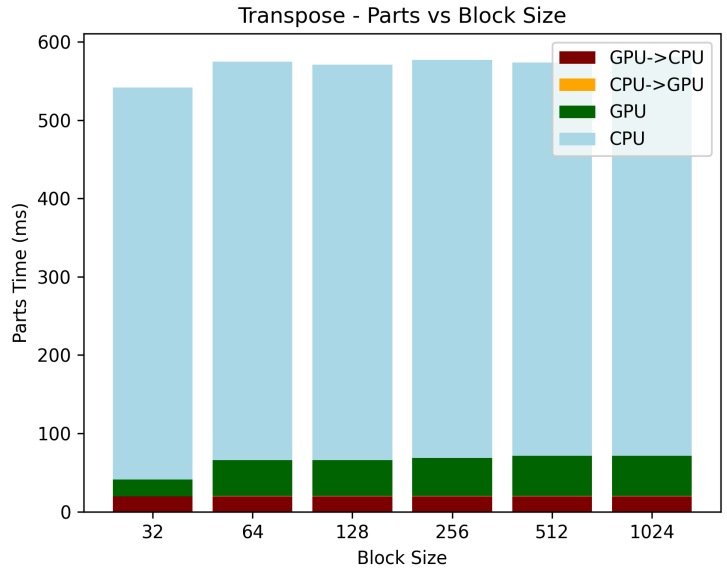
Το speedup αυξάνεται σημαντικά σε σχέση με την naive και την transpose εκδοση. Αυτό συμβαίνει διότι κάθε thread εκτελεί προσβάσεις σε όλα τα clusters, επομένως έχοντας τα στην κοινή μνήμη στην οποία τα φέρνουμε μία φορά όλα μαζί χρησιμοποιώντας πολλά threads γλιτώνουμε πολλά χρονοβόρα accesses.

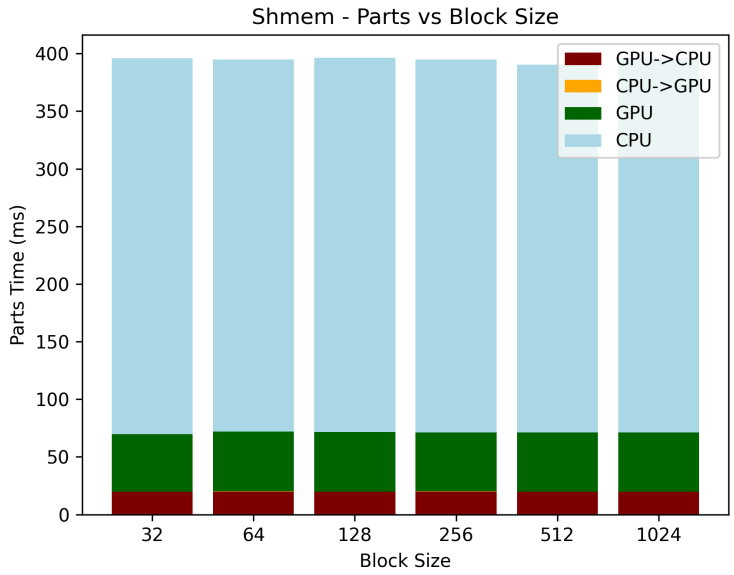
Σχετικά με block size θα περιμέναμε η βέλτιστη επιτάχυνση να επιτυγχάνεται ούτε σε πολύ μεγάλο ούτε σε πολύ μικρό block size, στο πρώτο μπορεί να οδηγουμαστε σε memory bank conflicts καθώς όταν πολλά threads δουλεύουν στο ίδιο memory bank οδηγούμαστε σε σειριοποίηση κάποιων από αυτά, στο δευτερο θα είχαμε πάρα πολλές αντιγραφές αυξάνοντας το latency. Αν κοιτάξουμε τα διαγράμματα speedup για όλες τις περιπτώσεις φαίνεται το βέλτιστο block size να είναι το 128.

Bottleneck Analysis

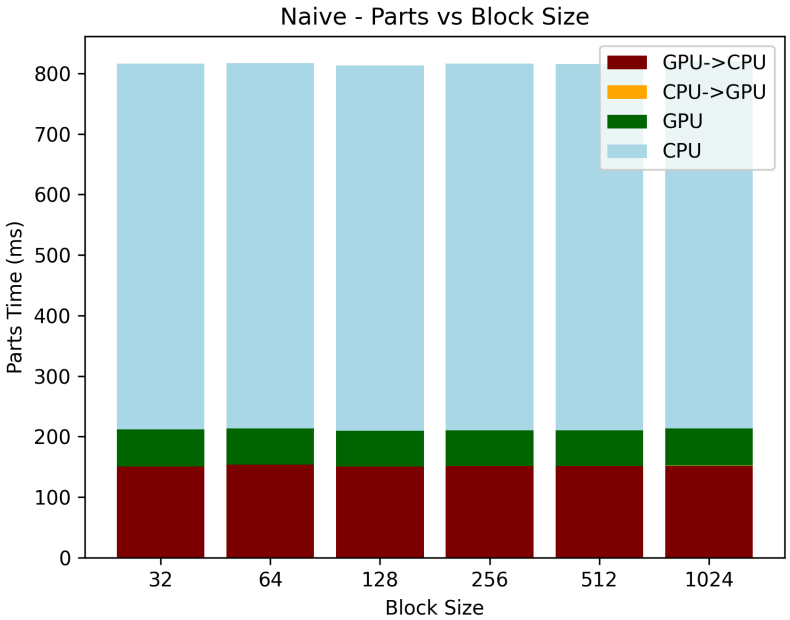
Config {256,16,16,10}

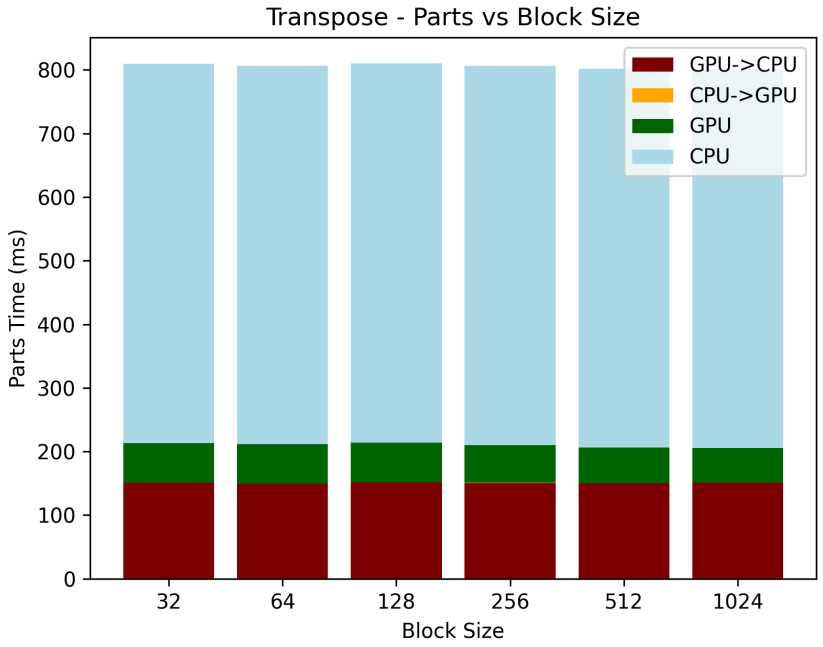


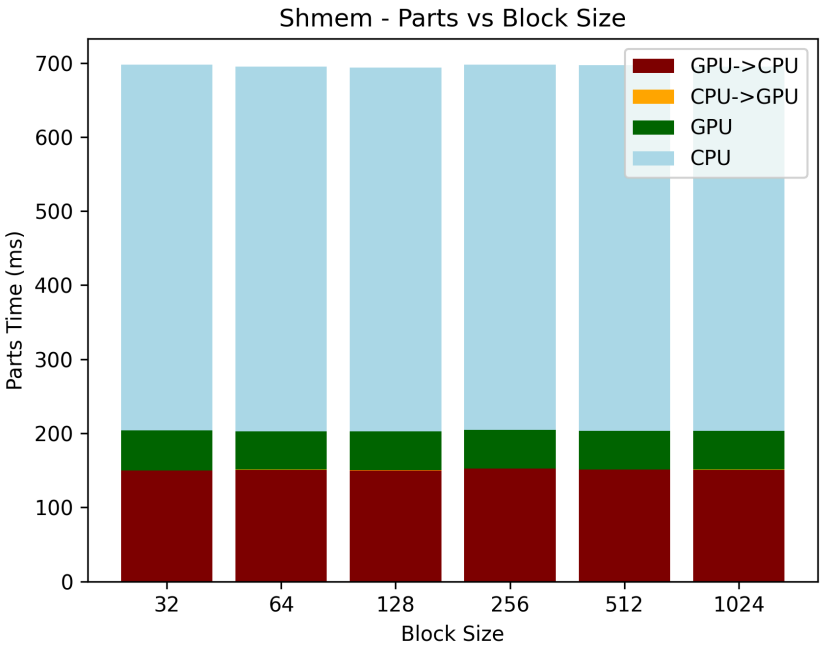




Config {256,2,16,10}



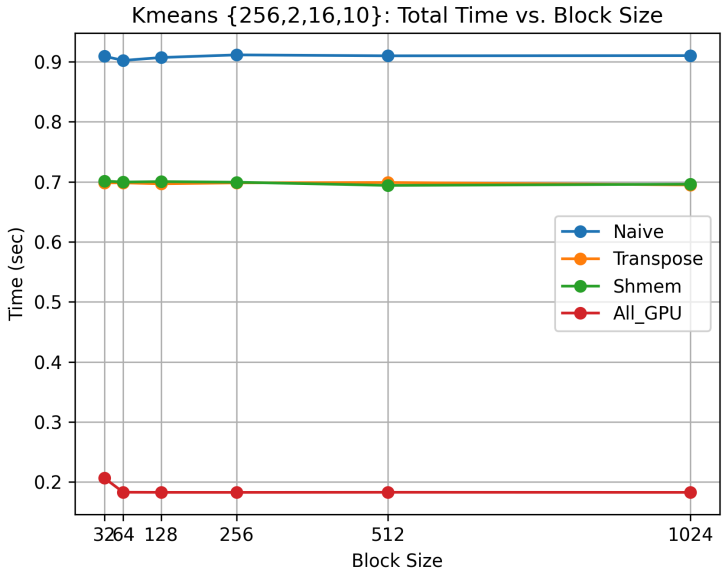

Transpose - Parts vs Block Size
GPU->CPU
CPU->GPU
GPU
CPU
Parts Time (ms)
0
100
200
300
400
500
600
700
800
32
64
128
256
512
1024
Block Size

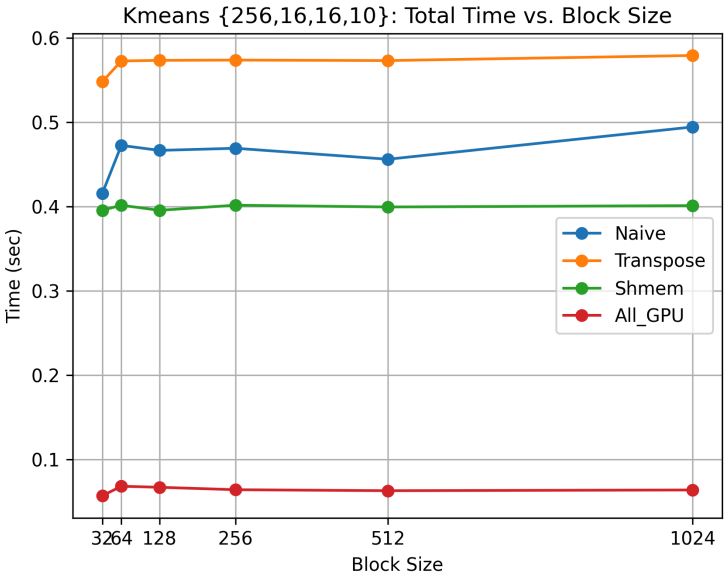

Shmem - Parts vs Block Size
GPU->CPU
CPU->GPU
GPU
CPU
Parts Time (ms)
0
100
200
300
400
500
600
700
32
64
128
256
512
1024
Block Size

Τοποθετούμε έξτρα time counters κατάλληλα στο κώδικα και αντλούμε τέσσερα επιπλέον χρονικά διαστήματα για να εκτιμήσουμε χρόνο μεταφορών απο CPU προς GPU και τανάπαλιν, ενώ μετράμε και χρόνους αμιγώς CPU ή GPU.
Αφού αντιστρέφουμε τις διαστάσεις των πινάκων, καταστρέφεται η τοπικότητα με συνέπεια να μειώνει την απόδοση της CPU η οποία υποβοηθείται με τις caches της. Συμπεραίνουμε λοιπόν πως το κυρίαρχο bottleneck είναι η CPU.

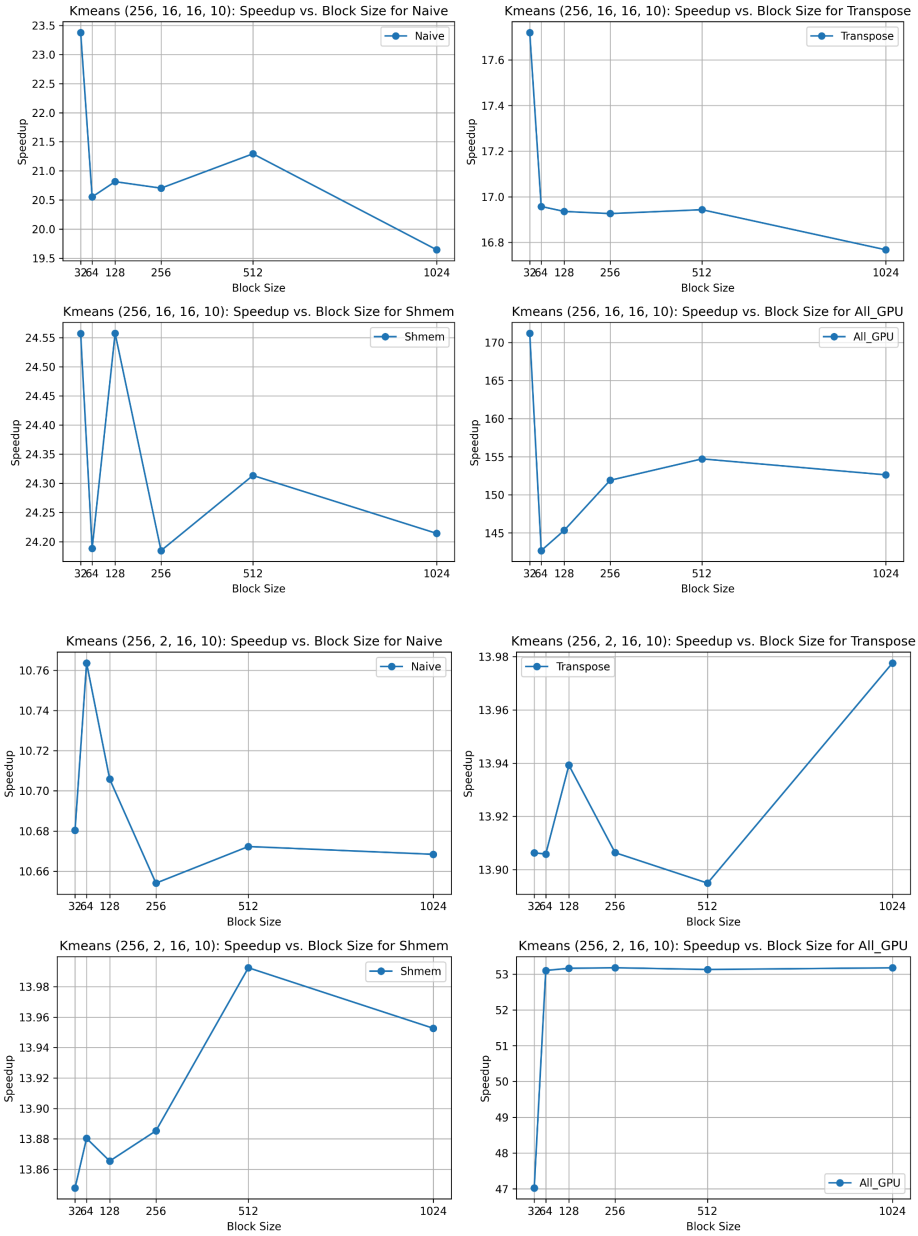
Είναι εμφανές ότι οι βελτιστοποιημένες υλοποιήσεις υποφέρουν στον χρόνο της CPU και όχι σε αυτόν που δαπανάται στους kernels ή στις μεταφορές, καθώς το μεγαλύτερο μερίδιο χρόνου δαπανάται σε CPU και GPU time.

Bonus ερώτημα: Full-GPU Offload


Kmeans {256,2,16,10}: Total Time vs. Block Size
Time (sec)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
32
64
128
256
512
1024
Block Size
Naive
Transpose
Shmem
All_GPU


Kmeans {256,16,16,10}: Total Time vs. Block Size
Time (sec)
0.6
0.5
0.4
0.3
0.2
0.1
32
64
128
256
512
1024
Block Size
Naive
Transpose
Shmem
All_GPU

Speedup Plots for both Configurations:



Σχόλια - Παρατηρήσεις: Όπως επιθυμούμε με βάση το bottleneck των προηγούμενων ερωτημάτων επιλύθηκε η τεράστια καθυστέρηση που επέβαλε η επεξεργασία απο τη CPU. Απο τα διαγράμματα speedup παρατηρούμε πως σε σχέση με το sequential implementation που πετυχαίνει χρόνο εκτέλεσης 9.71 seconds, τη μεγαλύτερη επιτάχυνση απολαμβάνει το configuration {256,16,16,10}.

Για την επίτευξη του full GPU implementation, έγιναν αρκετές αλλαγές τόσο στον υφιστάμενο πυρήνα (kernel) ενώ προέκυψε και η δημιουργία ενός καινούργιου (update_centroids). Τώρα, ο kernel που καθορίζει σε ποιο cluster ανήκει κάθε αντικείμενο πρέπει επίσης να συγκεντρώνει τις συντεταγμένες του αντικειμένου που ανήκει στο συγκεκριμένο cluster και να αυξάνει το μέγεθός του κατά ένα.

Tail of find_nearest_cluster kernel:

```
        }

        if (deviceMembership[tid] != index) {
            /* TODO: Maybe something is missing here... is this write safe? */\
            atomicAdd(devdelta,1.0);
            /*(*devdelta)+= 1.0;*/
        }

        /* assign the deviceMembership to object objectId */
        deviceMembership[tid] = index;


        /* TODO: additional steps for calculating new centroids in GPU? */
        // __syncthreads();

          // Atomic operations for updating cluster size and centroid in global memory
        atomicAdd(&(devicenewClusterSize[index]), 1);

        for(int i = 0; i < numCoords; i++) {
            atomicAdd(&(devicenewClusters[i * numClusters + index]), deviceobjects[i * numObjs + tid]);
        }


    }

```

update_centroids kernel:

```


__global__ static
void update_centroids(int numCoords,
                      int numClusters,
                      int *devicenewClusterSize,           //  [numClusters]
                      double *devicenewClusters,    //  [numCoords][numClusters]
                      double *deviceClusters)    //  [numCoords][numClusters])
{


    int tid = get_tid();
    if (tid < numClusters*numCoords) {
        int coordId = tid % numCoords;
        int clusterId = tid / numCoords;

        if (devicenewClusterSize[clusterId] > 0)
            deviceClusters[coordId*numClusters + clusterId] = devicenewClusters[coordId*numClusters + clusterId] /
        devicenewClusters[coordId*numClusters + clusterId] = 0;
    }
    __syncthreads();
    if (tid < numClusters) devicenewClusterSize[tid] = 0;
}


```

Αυτές οι τιμές θα χρησιμοποιηθούν αργότερα από άλλον kernel (update_centroids) για τον άμεσο υπολογισμό των νέων clusters. Το πρόβλημα είναι ότι οι προσβάσεις στους πίνακες newClusterSize και newClusters πρέπει να γίνονται με atomic instructions και είναι τυχαίες, καθώς κάθε αντικείμενο μπορεί να ανήκει σε οποιοδήποτε cluster. Κάτι τέτοιο πάει κόντρα στην ιδανική δομή μνήμης για μια GPU η οποία θα ήταν συγκεντρωμένη και “στοιχισμένη” (coalesced), επιπρόσθετα τα atomic access καταστρατηγούν τελείως την έννοια της παραλληλοποίησης. Επομένως το update_centroids δεν είναι ιδανικό για GPU. Για να αντιμετωπιστεί αυτό το πρόβλημα, τοποθετήσαμε αυτούς τους πίνακες στην κοινή μνήμη, εξαλείφοντας έτσι το πρόβλημα των μη συγχρονισμένων τυχαίων προσβάσεων, που δεν υπάρχει στην κοινή μνήμη, και μειώνοντας τις συγκρούσεις στις ατομικές εγγραφές.

Το block size φαινεται να μην επηρεάζει ιδιαίτερα την επίδοση στο config με 2 coordinates. Η διαφορά επίδοσης ανάμεσα σε Coord 2 και Coord 16 config μάλλον οφείλεται στο μεγαλύτερο shared memory size του Coord 16 το οποίο μειώνει την ανάγκη να ανατρέχουμε συχνά στο global memory για να κάνουμε fetch δεδομένα.

Το initialization των kernels είναι το ακόλουθο:

```
/* TODO: change invocation if extra parameters needed  */
find_nearest_cluster
      <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
    (numCoords, numObjs, numClusters,
     deviceObjects,devicenewClusters, devicenewClusterSize,deviceClusters, deviceMembership, dev_delta_ptr);
```

```
const unsigned int update_centroids_block_sz = (numCoords* numClusters > blockSize) ? blockSize: numCoords* numClusters;  /* TODO:
const unsigned int update_centroids_dim_sz =  (numCoords*numClusters + update_centroids_block_sz - 1) / update_centroids_block_sz;
update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0 >>>
    (numCoords, numClusters, devicenewClusterSize, devicenewClusters, deviceClusters);
```

Τέλος, με full GPU implementation έχουμε εξαφανίσει χρόνους μεταφορών host to device και device to host μαζί με το CPU time, εκτός μόνο από το τελικό κομμάτι που η CPU μαζευει τις μετρήσεις από dimClusters σε clusters.