C++ script that contains 2 phases:

Learning Phase:

* Read training dataset in the file 'power.csv' that has:
  + 1 target variable as the first column (float Power),
  + string Graph (factor of 5 levels ‘a’, ‘b’, ‘c’, ‘d’, ‘e’),
  + float FPS, float Latency,
  + int BigFrequency {500000, 667000, 1000000, 1200000, 1398000, 1512000, 1608000, 1704000, 1800000, 1908000, 2016000, 2100000, 2208000},
  + int LittleFrequency {500000, 667000, 1000000, 1200000, 1398000, 1512000, 1608000, 1704000, and 1800000},
  + int TotalParts, int PartitionPoint1, int PartitionPoint2,
  + string Order (factor of 6 levels 'G-B-L', 'G-L-B', 'B-G-L', 'B-L-G', 'L-G-B', 'L-B-G'),
  + float stage1\_inference\_time, float stage2\_inference\_time, float stage3\_inference\_time
* Establish the relationship between Power and the 5 variables BigFrequency, LittleFrequency, PartitionPoint1, PartitionPoint2, Order for each level in the factor Graph.
* Read another training dataset in the file ‘constraints.csv’ that has:
  + 2 target variables float FPS, float Latency
  + String Graph (factor of 5 levels ‘a’, ‘b’, ‘c’, ‘d’, ‘e’)
  + int TotalParts, int PartitionPoint1, int PartitionPoint2
  + string Order (factor of 6 levels 'G-B-L', 'G-L-B', 'B-G-L', 'B-L-G', 'L-G-B', 'L-B-G'),
  + float stage1\_inference\_time, float stage2\_inference\_time, float stage3\_inference\_time
* Establish relationship between the target variables and the other variables for each level in the factor Graph.

Execution Phase in main (int argc, char \*argv[]):

* Take in 4 inputs (string Graph, int TotalParts, float TargetFPS, float TargetLatency)
* Find a suitable design point that contains int BigFrequency, int LittleFrequency, int PartitionPoint1, PartitionPoint2, string Order based on the Learning Phase that satisfy 1 objective of minimizing Power and 4 constraints:
  + PartitionPoint1 <= PartitionPoint2 <= TotalParts
  + Graph is used
  + The design point formed by the outputs needs to result in Latency <= TargetLatency
  + The design point formed by the outputs needs to result in FPS >= TargetFPS
* To obtain the result of the design point, run sprintf(Run\_Command,"./%s --threads=4 --threads2=2 --target=NEON --n=%d --partition\_point=%d --partition\_point2=%d --order=%s > output.txt", graph.c\_str(), N\_Frames, PartitionPoint1, PartitionPoint2, Order.c\_str()); system(Run\_Command); ParseResults();
* While FPSCondition != 1 && LatencyCondition != 1:
  + Update learned knowledge from the new result obtained
  + Create a new design point based on the updated knowledge
  + Regenerate result by running again run sprintf(Run\_Command,"./%s --threads=4 --threads2=2 --target=NEON --n=%d --partition\_point=%d --partition\_point2=%d --order=%s > output.txt", graph.c\_str(), N\_Frames, PartitionPoint1, PartitionPoint2, Order.c\_str()); system(Run\_Command); ParseResults();
* Return 5 outputs BigFrequency, LittleFrequency, PartitionPoint1, PartitionPoint2, Order from the optimal design point

­------------------------------------------------------------------------------------------------------------------------------------------

void balanceLoad(int &PartitionPoint1, int &PartitionPoint2, string Order) {  
 static bool considered\_point1 = false;  
 static bool considered\_point2 = false;  
 static int point\_min = 0;  
 static int point\_max = TotalParts;  
  
 if ((considered\_point1 && considered\_point2) || *// achieved both 'optimal' points* p1 == TotalParts) *// 1 processor* return;  
  
 if (p2 == TotalParts) { *// 2 processors* if (considered\_point1)  
 return;  
 if (! considered\_point2)  
 considered\_point2 = true;  
 if (t1 < StageTwoInferenceTime && point\_min != p1) {  
 point\_min = p1;  
*// if (StageTwoInferenceTime - StageOneInferenceTime > timeDifference\_big)* p1 = floor((p1 + point\_max)/2);  
 return;  
 }  
 else if (1 > StageTwoInferenceTime && point\_max > p1 + 1) {  
 point\_max = p1;  
 p1 = floor((p1 + point\_min)/2);  
 return;  
 }  
 else {  
 considered\_point1 = true;  
 return;  
 }  
 }  
  
 */\* 3 processors \*/* float t = min({t1, StageTwoInferenceTime, StageThreeInferenceTime});  
  
 if (t == StageOneInferenceTime) { */\* StageOneInferenceTime is smallest -> increase p1 \*/* if (p1 != p2 - 1) {  
 point\_min = p1;  
 p1 = floor((p1 + p2) / 2);  
 return;  
 } else {  
 considered\_point1 = true;  
 return;  
 }  
 }  
 else if (t == StageTwoInferenceTime) { */\* StageTwoInferenceTime is smallest \*/* float d1 = StageOneInferenceTime - StageTwoInferenceTime;  
 float d2 = StageThreeInferenceTime - StageTwoInferenceTime;  
 if (d1 > d2 && point\_min != p1) { */\* -> decrease p1 \*/* p1 = floor((p1 + point\_min) / 2);  
 } else if (d1 < d2 && point\_max > p1 + 1) { */\* -> increase p2 \*/* p2 = floor((p2 + point\_max) / 2);  
 } else { */\* d1 = d2 (unlikely, but if not implemented right, this'll be the error) \*/* p1 = p1 - 1;  
 p2 = p2 + 1;  
 return;  
 }  
 }  
 else if (t == StageThreeInferenceTime) { */\* StageThreeInferenceTime is smallest -> decrease p2 \*/* if (p1 != p2 + 1) {  
 point\_max = p2;  
 p2 = floor((p1 + p2) / 2);  
 return;  
 } else {  
 considered\_point2 = true;  
 return;  
 }  
 }  
}