Second assignment

Cholesky factorization

Benatti Sebastiano, Carpi Samuele,
Pasquali Mattia

Cholesky decomposition factors a symmetric, positive-definite matrix **A** into

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

where **L is a lower triangular matrix and L^T is the transpose of L.** It provides a numerically efficient way to solve linear systems and is widely used in optimization, statistics, and numerical analysis.

Diagonal elements

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$$

Off-diagonal elements

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right)$$

```
int i, j, k;
DATA_TYPE x;
for (i = 0; i < _PB_N; ++i)
{
  x = A[i][i];
  for (j = 0; j <= i - 1; ++j){
    x -= A[i][j] * A[i][j];
  }
  p[i] = 1.0 / sqrt(x);

  for (j = i + 1; j < _PB_N; ++j)
  {
    x = A[i][j];
    for (k = 0; k <= i - 1; ++k)
      x -= A[j][k] * A[i][k];
    A[j][i] = x * p[i];
  }
}
```

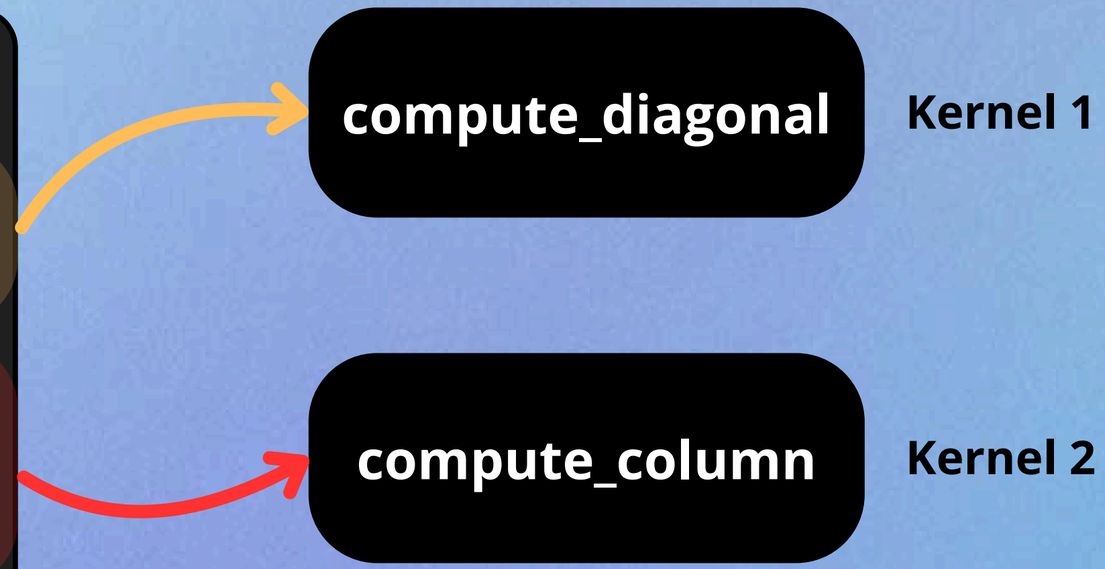
OPTIMIZATION APPROACHES

- UVM
- Pinned memory
- Streams
- **Shared memory**
- **Stride**
- **Tiling**

OUR BEST SOLUTION CODE

Sequential

```
int i, j, k;
DATA_TYPE x;
for (i = 0; i < _PB_N; ++i)
{
  x = A[i][i];
  for (j = 0; j <= i - 1; ++j){
    x -= A[i][j] * A[i][j];
  }
  p[i] = 1.0 / sqrt(x);

  for (j = i + 1; j < _PB_N; ++j)
  {
    x = A[i][j];
    for (k = 0; k <= i - 1; ++k)
      x -= A[j][k] * A[i][k];
    A[j][i] = x * p[i];
  }
}
```



Sequential

```
int i, j, k;
DATA_TYPE x;
for (i = 0; i < _PB_N; ++i)
{
  x = A[i][i];
  for (j = 0; j <= i - 1; ++j){
    x -= A[i][j] * A[i][j];
  }
  p[i] = 1.0 / sqrt(x);

  for (j = i + 1; j < _PB_N; ++j)
  {
    x = A[i][j];
    for (k = 0; k <= i - 1; ++k)
      x -= A[j][k] * A[i][k];
    A[j][i] = x * p[i];
  }
}
```

Optimized

```
__global__ void compute_diagonal(DATA_TYPE* __restrict__ p, DATA_TYPE* __restrict__ A, int n, int i){
    __shared__ DATA_TYPE sharedSum[BLOCK_SIZE];
    int tid = threadIdx.x;

    // 1) ogni thread calcola una somma parziale in parallelo con stride (per i > BLOCK_SIZE)
    DATA_TYPE localSum = 0.0;
    for (int k=tid; k<i; k+=blockDim.x){
      localSum += A[i*n+k]*A[i*n+k];
    }
    sharedSum[tid] = localSum;
    __syncthreads();

    // 2) somma parziale parallela riducendo da BLOCK_SIZE a 32 valori finali (warp size per sincronizzazione implicita)
    if (tid<32){
      DATA_TYPE partialSum = 0.0;
      for (int t=tid; t<blockDim.x; t+=32){
        partialSum += sharedSum[t];
      }
      sharedSum[tid] = partialSum;
    }

    // 3) somma finale dei 32 valori rimanenti seqenzialmente nel thread 0
    if (tid == 0){
      DATA_TYPE sum = 0.0;
      for (int t=0; t<32; t++){
        sum += sharedSum[t];
      }
      p[i] = 1.0/sqrt(A[i*n+i] - sum);
    }
  }
```

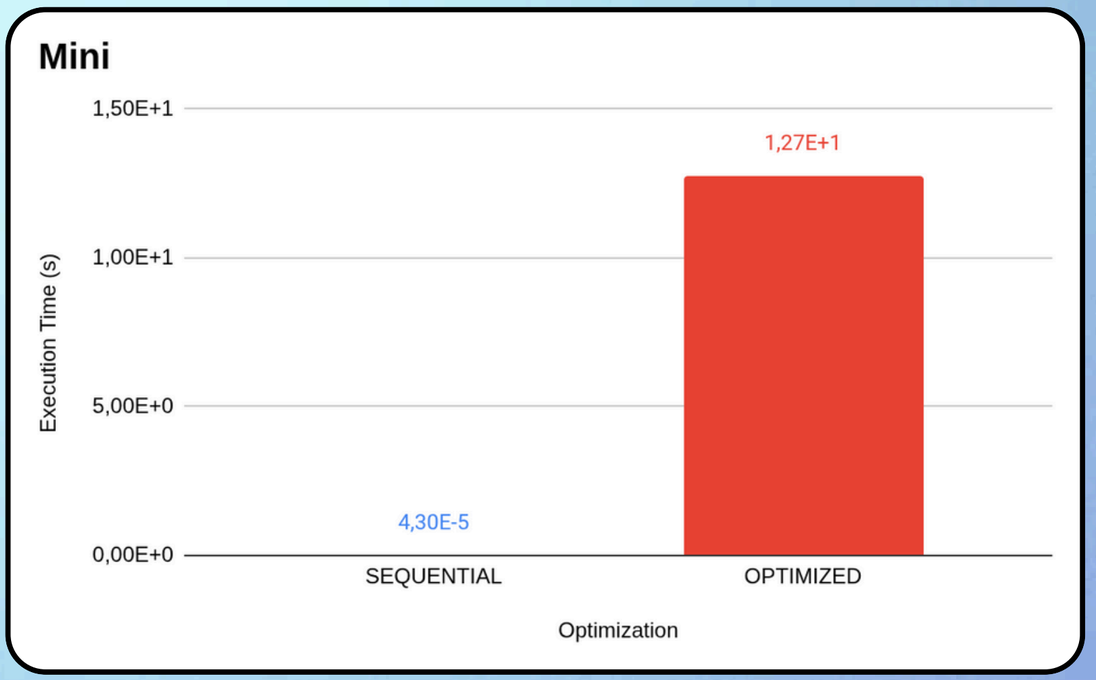
Sequential

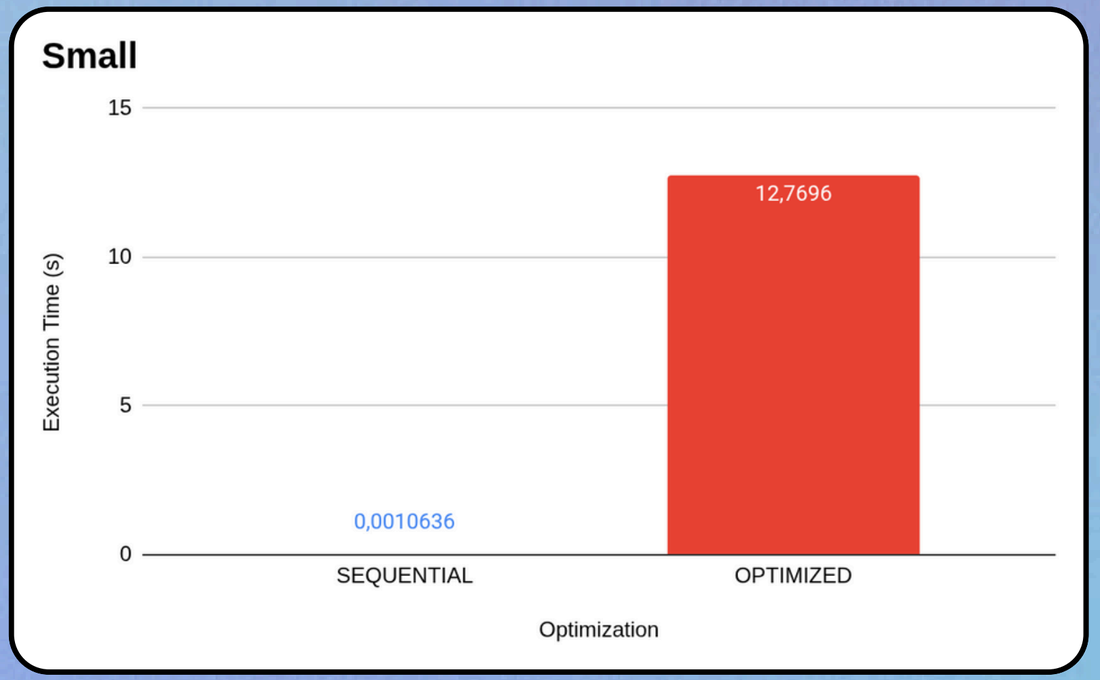
```
int i, j, k;
DATA_TYPE x;
for (i = 0; i < _PB_N; ++i)
{
  x = A[i][i];
  for (j = 0; j <= i - 1; ++j){
    x -= A[i][j] * A[i][j];
  }
  p[i] = 1.0 / sqrt(x);

  for (j = i + 1; j < _PB_N; ++j)
  {
    x = A[i][j];
    for (k = 0; k <= i - 1; ++k)
      x -= A[j][k] * A[i][k];
    A[j][i] = x * p[i];
  }
}
```

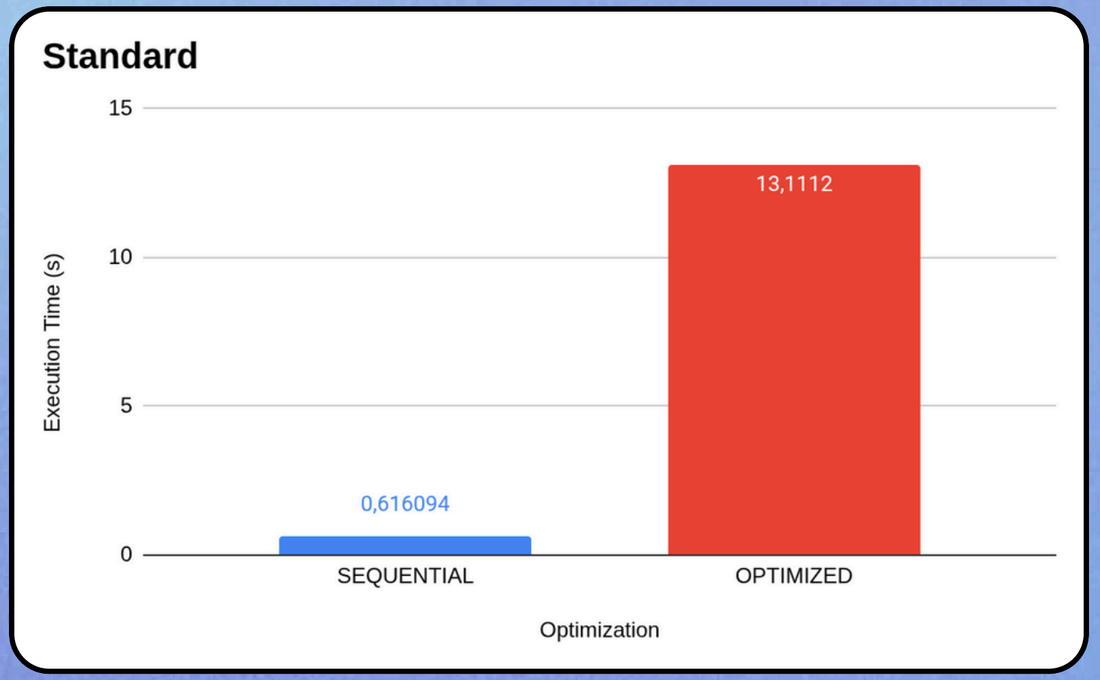
Optimized

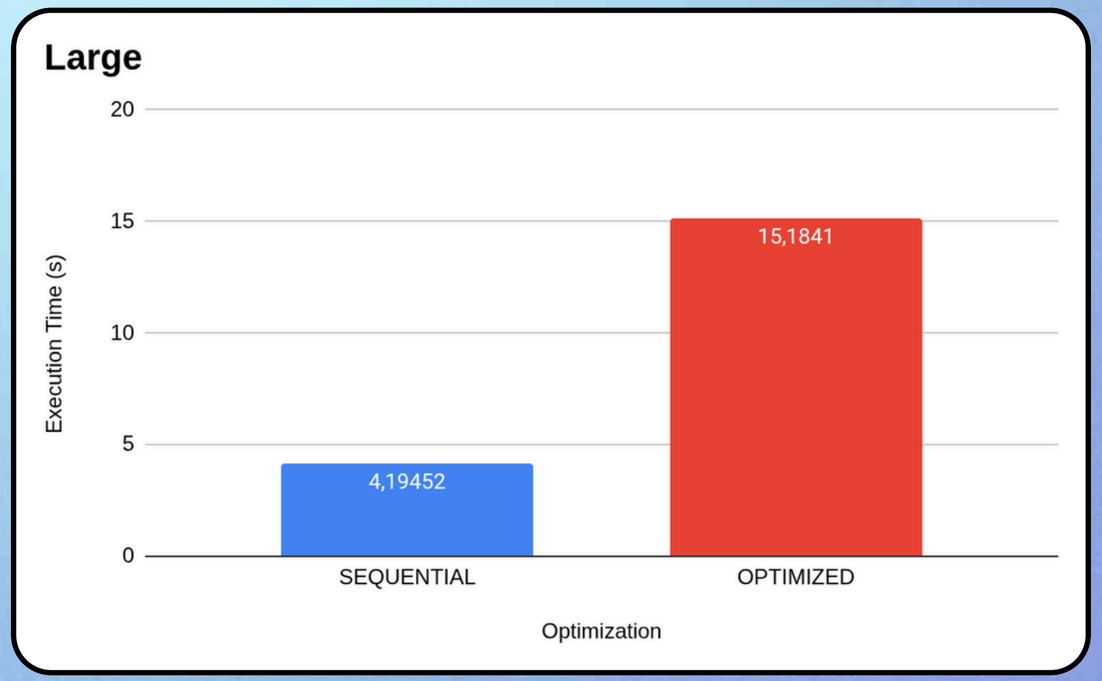
```
__global__ void compute_column(DATA_TYPE* __restrict__ p, DATA_TYPE* __restrict__ A, int n, int i){
    int j = blockIdx.x*blockDim.x+threadIdx.x;  // Ogni thread elabora la riga j
    int tid = threadIdx.x;
    __shared__ DATA_TYPE sharedPivotRow[BLOCK_SIZE];  // Shared memory per il tile della riga pivot A[i][0..i-1]
    __shared__ DATA_TYPE sharedP;                     // Shared memory per p[i]

    // 1) Thread 0 carica p[i] in shared memory
    if (tid == 0){
      sharedP = p[i];
    }

    // 2) x = A[i][j] inizializzato per il triangolo inferiore
    DATA_TYPE x = 0.0;
    if (j>i && j<n){
      x = A[i*n+j];
    }

    // Tiling della riga pivot di BLOCK_SIZE elementi.
    for (int tile=0; tile<i; tile+=BLOCK_SIZE){

      // 3) Caricamento del tile in shared memory
      int k_curr = tile+tid; // Indice globale dell'elemento da caricare
      if (k_curr<i){
        sharedPivotRow[tid] = A[i*n+k_curr];
      }else{
        sharedPivotRow[tid] = 0.0; // Padding per evitare accessi fuori limite
      }
      __syncthreads();  // Aspetta che tutti i thread abbiano caricato il loro elemento

      // 4) Calcolo del prodotto scalare parziale
      if (j>i && j<n){
        int pivotRowSize = min(BLOCK_SIZE,i-tile); // Elementi validi nel tile corrente
        for (int k=0; k<pivotRowSize; ++k){
          x -= A[j*n+(tile+k)]*sharedPivotRow[k]; // A[j][k] in global memory, A[i][k] in shared memory
        }
      }
      __syncthreads();  // Aspetta prima di sovrascrivere sharedPivotRow nel prossimo tile
    }

    // 5) Scrittura del risultato finale
    if (j>i && j<n){
      A[j*n+i] = x*sharedP;
    }
}
```

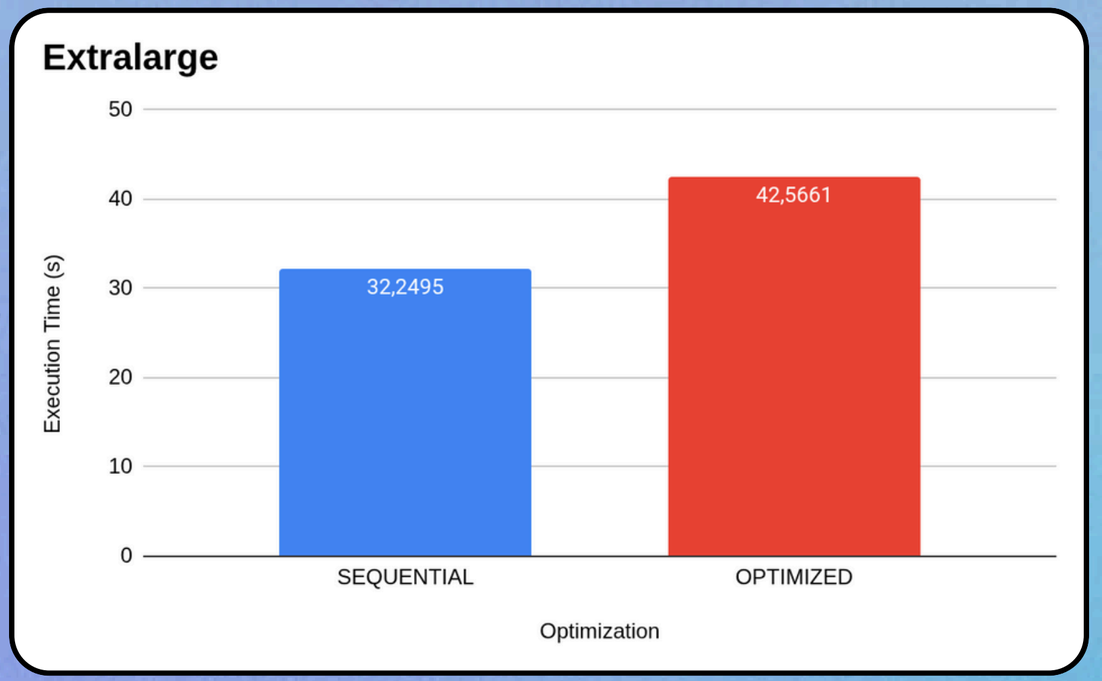
EXPERIMENTAL RESULTS

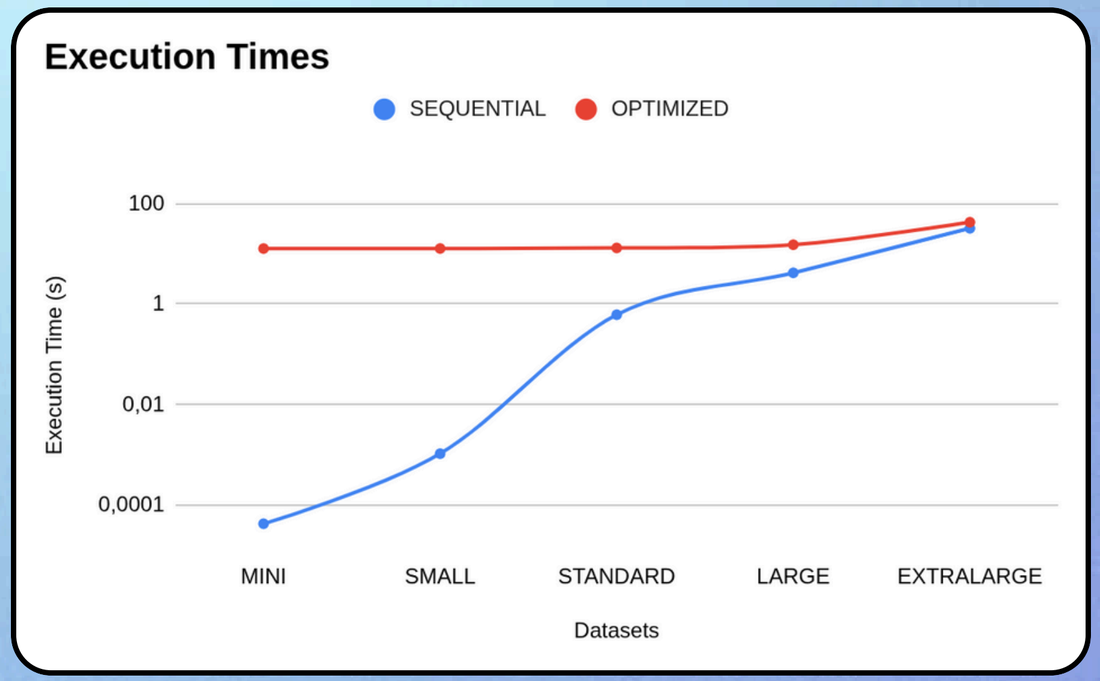

Mini
1,50E+1
1,00E+1
5,00E+0
0,00E+0
Execution Time (s)
1,27E+1
4,30E-5
SEQUENTIAL
OPTIMIZED
Optimization

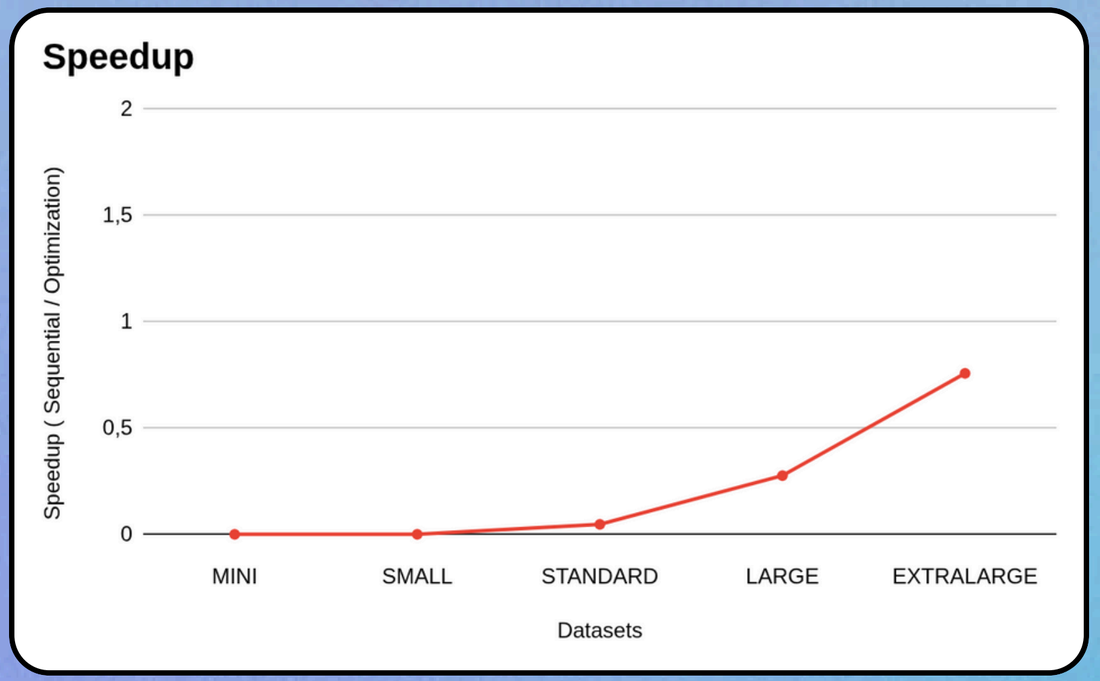

Small
15
10
5
0
Execution Time (s)
12,7696
0,0010636
SEQUENTIAL
OPTIMIZED
Optimization


Standard
15
10
5
0
Execution Time (s)
13,1112
0,616094
SEQUENTIAL
OPTIMIZED
Optimization


Large
20
15
10
5
0
Execution Time (s)
4,19452
15,1841
SEQUENTIAL
OPTIMIZED
Optimization


Extralarge
50
40
30
20
10
0
Execution Time (s)
32,2495
42,5661
SEQUENTIAL
OPTIMIZED
Optimization


Execution Times
SEQUENTIAL
OPTIMIZED
100
1
0,01
0,0001
Execution Time (s)
MINI
SMALL
STANDARD
LARGE
EXTRALARGE
Datasets


Speedup
2
1,5
1
0,5
0
Speedup (Sequential / Optimization)
MINI
SMALL
STANDARD
LARGE
EXTRALARGE
Datasets

CONCLUSIONS

- It **does not** make sense use **CUDA** for Cholesky Algorithm because of **data dependencies** between for loops

- But if the matrix is **large enough**, the execution time would be **similar** or better than the **sequential** version