The University of Texas at Austin
Oden Institute for Computational Engineering and Sciences

28 April 2023

PARALLEL SUPPORT VECTOR MACHINES

All relevant code is at link: https://github.com/V-Rang/CSE-392-Parallel-Algorithms-for-Scientific-Computing-Project---Parallel-Support-Vector-Machines

Dilip Geethakrishnan and Venugopal Ranganathan

OUTLINE

1. Problem Statement
2. Sequential – Complexity & Algorithm
3. Parallel – Complexity & Algorithm
4. Implementation Details
5. Results

OUTLINE

1. **Problem Statement**
2. Sequential – Complexity & Algorithm
3. Parallel – Complexity & Algorithm
4. Implementation Details
5. Results

Problem Statement

$$
\begin{aligned}
\textit{minimize} \qquad & f_o(\vec{w}, \vec{\xi}) = \frac{1}{2}\vec{w}^T\vec{w} + C\sum_{i=1}^{n}\xi_i \\
s.t. \qquad & y_i(\vec{w}^T\vec{x}_i + b) \geq 1 - \xi_i, \qquad i = 1, \ldots, n \\
& \xi_i \geq 0, \qquad i = 1, \ldots, n
\end{aligned}
\longrightarrow
\begin{bmatrix} Q & \vec{1}_m \\ \vec{1}_m^T & 0 \end{bmatrix} \cdot \begin{bmatrix} \vec{\alpha} \\ b \end{bmatrix} = \begin{bmatrix} \vec{y} \\ 0 \end{bmatrix},
$$

The aim is to improve/study some performance of SVM's. We do this by the following:

- Parallelize Matrix multiplications on GPUs
- Improve performance of Matrix inversion by using parallelized Randomized SVD.

In this study, we compare the performances of some standard SVM libraries, as well as modify a variant of SVM, called Least-Squared SVM. In Least squared SVM, we arrive at a step where we have a system to solve, of the form Ax=b. We then use the RSVD to find the pseudo inverse.

OUTLINE

1. Problem Statement
2. **Sequential – Complexity & Algorithm**
3. Parallel – Complexity & Algorithm
4. Implementation Details
5. Results

Sequential – Complexity & Algorithm

Matrix-matrix multiplication – O(n^3)

Algorithm 1 Naive matrix-matrix multiplication

Require: $A_{m,p}, B_{p,n}$

```
Initialize C to be a zero matrix of size m × p
for i = 1, . . . , m do
    for j = 1, . . . , p do
        for k = 1, . . . , n do
            C_ij = C_ij + A_ik B_kj
        end for
    end for
end for
Return C
```

Randomized Singular Value Decomposition – O(mnk + nk^2)

ALGORITHM: RSVD — BASIC RANDOMIZED SVD

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = 10$).

Outputs: Matrices $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ in an approximate rank-$(k+p)$ SVD of $\mathbf{A}$ (so that $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, $\mathbf{D}$ is diagonal, and $\mathbf{A} \approx \mathbf{U}\mathbf{D}\mathbf{V}^*$.)

Stage A:

(1) Form an $n \times (k+p)$ Gaussian random matrix $\mathbf{G}$.
`G = randn(n,k+p)`

(2) Form the sample matrix $\mathbf{Y} = \mathbf{A}\,\mathbf{G}$.
`Y = A*G`

(3) Orthonormalize the columns of the sample matrix $\mathbf{Q} = \mathtt{orth}(\mathbf{Y})$.
`[Q,~] = qr(Y,0)`

Stage B:

(4) Form the $(k+p) \times n$ matrix $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$.
`B = Q'*A`

(5) Form the SVD of the small matrix $\mathbf{B}$: $\mathbf{B} = \hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*$.
`[Uhat,D,V] = svd(B,'econ')`

(6) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.
`U = Q*Uhat`

"Finding Structure with Randomness" - Halko, Martinsson, Tropp (2011)

OUTLINE

1. Problem Statement
2. Sequential – Complexity & Algorithm
3. **Parallel – Complexity & Algorithm**
4. Implementation Details
5. Results

Parallel – Complexity & Algorithm

Naïve Gpu Matrix-matrix multiplication shown for analysis.

Work: $O(mn^2)$.
Depth: $O(n)$

Algorithm 2 Naive GPU matmul

Require: $A_{m,p}, B_{p,n}$
  Initialize C to be a zero matrix of size $m \times p$
  parfor $i = 1, \ldots, m$ **do**
    parfor $j = 1, \ldots, p$ **do**
      for $k = 1, \ldots, n$ **do**
        $C_{ij} = C_{ij} + A_{ik} B_{kj}$
      end for
    end parfor
  end parfor
  Return C

Parallel – Complexity & Algorithm

Naïve Gpu Matrix-matrix multiplication shown for analysis. We replace the other operations with parallel variants from various libraries and compare results.
For simplicity, we assume m=n

Steps:
Forming omega: work =O(n*k) Depth = O(n*k)
Y=A*O , O(nk^2), O(k)
Q=QR(Y) O(nk^2)
B=Q^TA O(nk^2) , O(k)
U'DVh=SVD(B) O(nk^2)
U=QU' O(nk^2) ,O(k)

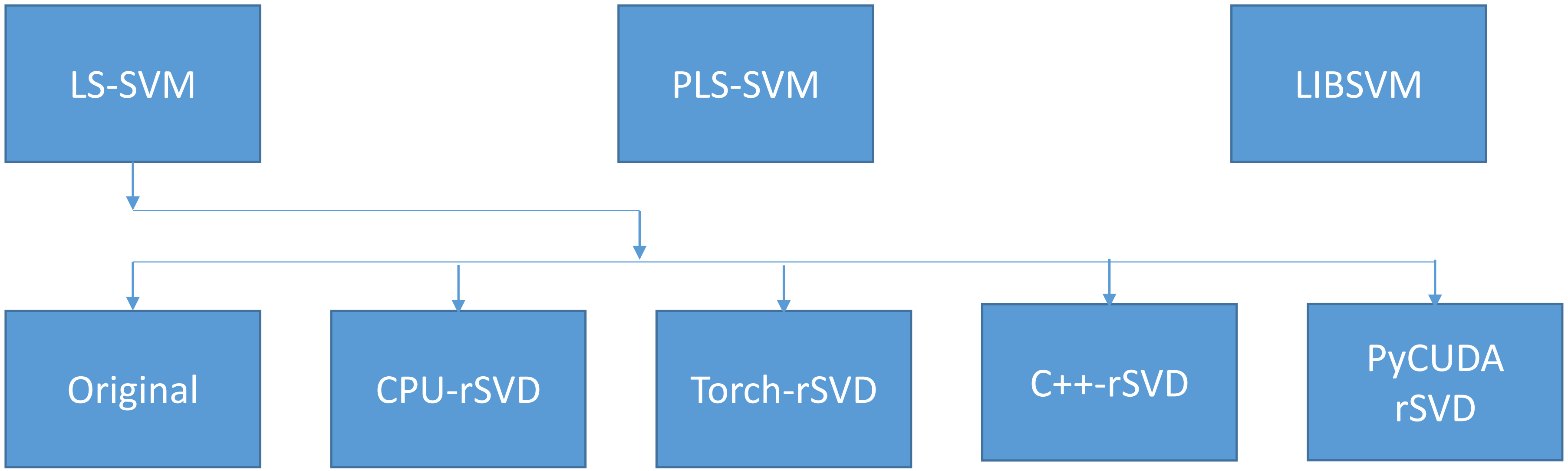
Algorithm 3 Simplified parallel RSVD

Require: $A_{m,n}$ With approx rank=k
Form $\Omega_{n,k}$, a random matrix.
Y=Naive_GPU_matmul(A,Ω)
Q,_=parallel_QR(Y)
B=Naive_GPU_matmul(Q^T,A)
U'DVh=ParallelSVD(B)
U=Naive_GPU_matmul(Q,U')
Return U, D, Vh

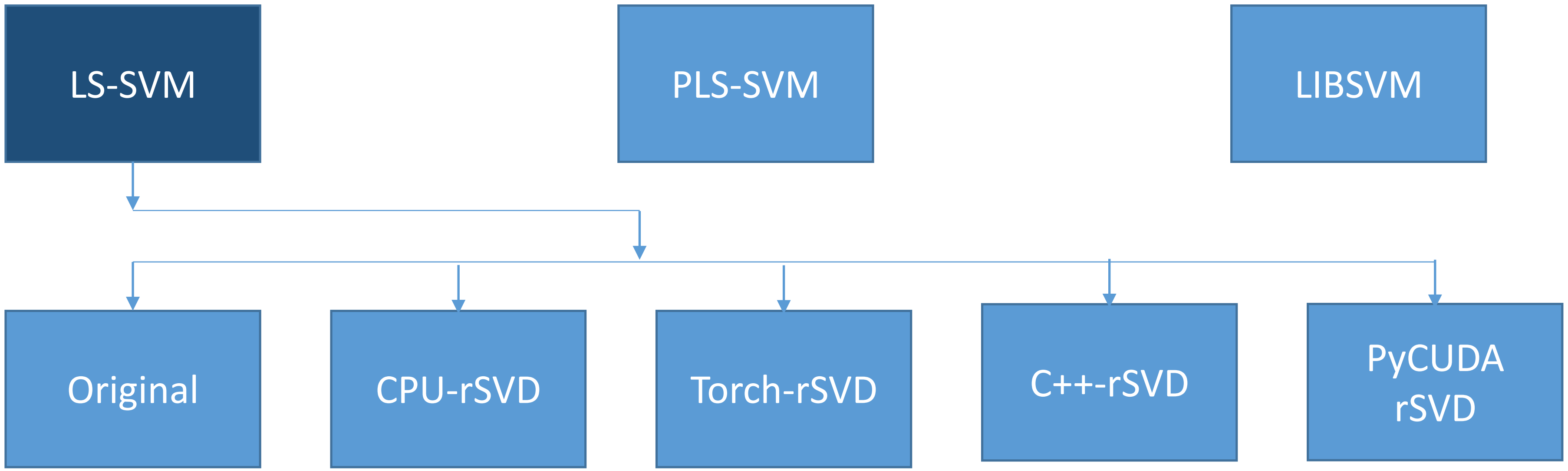
OUTLINE

1. Problem Statement
2. Sequential – Complexity & Algorithm
3. Parallel – Complexity & Algorithm
4. **Implementation Details**
5. Results

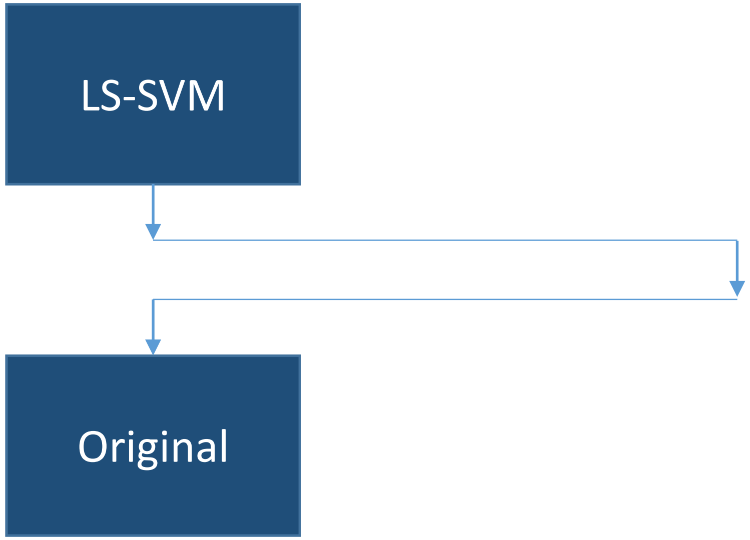
Implementation Details



Implementation Details



Implementation Details



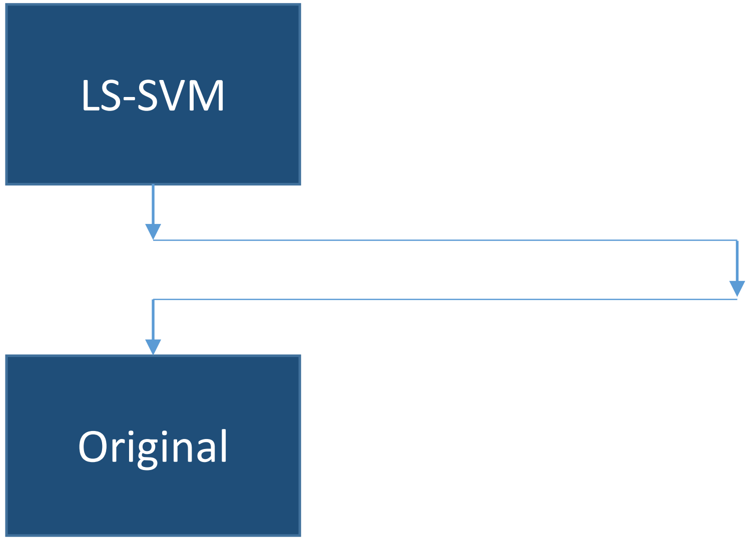
$$
\begin{aligned}
\text{minimize} \quad & f_o(\vec{w}, \vec{\xi}) = \frac{1}{2}\vec{w}^T\vec{w} + C\sum_{i=1}^{n}\xi_i \\
s.t. \quad & y_i(\vec{w}^T\vec{x}_i + b) \geq 1 - \xi_i, \qquad i = 1, \ldots, n \\
& \xi_i \geq 0, \qquad i = 1, \ldots, n
\end{aligned}
$$

$$
\begin{aligned}
\text{minimize} \quad & f_o(\vec{w}, \vec{\xi}) = \frac{1}{2}\vec{w}^T\vec{w} + \gamma\frac{1}{2}\sum_{i=1}^{n}\xi_i^2 \\
s.t. \quad & y_i(\vec{w}^T\vec{x}_i + b) = 1 - \xi_i, \qquad i = 1, \ldots, n
\end{aligned}
$$

$$
\begin{bmatrix} Q & \vec{1}_m \\ \vec{1}_m^T & 0 \end{bmatrix} \cdot \begin{bmatrix} \vec{\alpha} \\ b \end{bmatrix} = \begin{bmatrix} \vec{y} \\ 0 \end{bmatrix},
$$

https://github.com/RomuloDrumond/LSSVM

Implementation Details




```python
def fit(self, X, y):
    """Fits the model given the set of X attribute vectors and y labels.
    - X: ndarray of shape (n_samples, n_attributes)
    - y: ndarray of shape (n_samples,) or (n_samples, n)
        If the label is represented by an array of n elements, the y
        parameter must have n columns.
    """
    y_reshaped = y.reshape(-1,1) if y.ndim==1 else y

    self.sv_x = X
    self.sv_y = y_reshaped
    self.y_labels = np.unique(y_reshaped, axis=0)

    if len(self.y_labels) == 2: # binary classification
        # converting to -1/+1
        y_values = np.where(
            (y_reshaped == self.y_labels[0]).all(axis=1)
            ,-1,+1)[:,np.newaxis] # making it a column vector

        self.b, self.alpha = self._optimize_parameters(X, y_values)
```

```python
def _optimize_parameters(self, X, y_values):
    """Help function that optimizes the dual variables through the
    use of the kernel matrix pseudo-inverse.
    """
    sigma = np.multiply(y_values*y_values.T, self.K(X,X))
    #print(y_values.shape)
    # print(self.K(X,X).shape)
    A = np.block([
        [0, y_values.T],
        [y_values, sigma + self.gamma**-1 * np.eye(len(y_values))]
    ])
    B = np.array([0]+[1]*len(y_values))
```

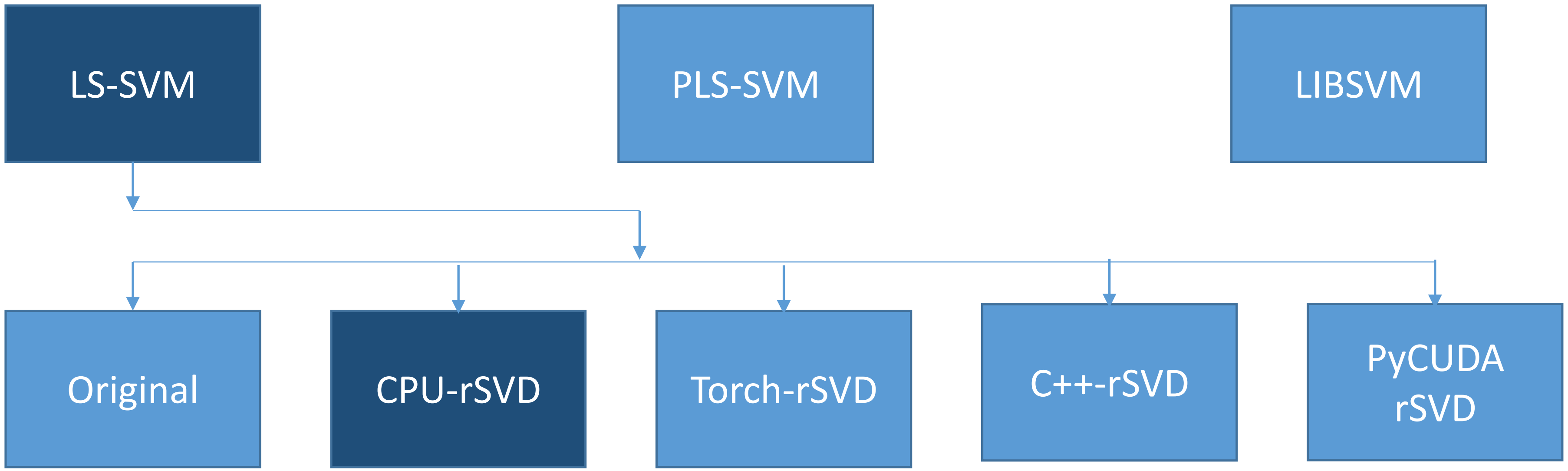
Implementation Details

LS-SVM

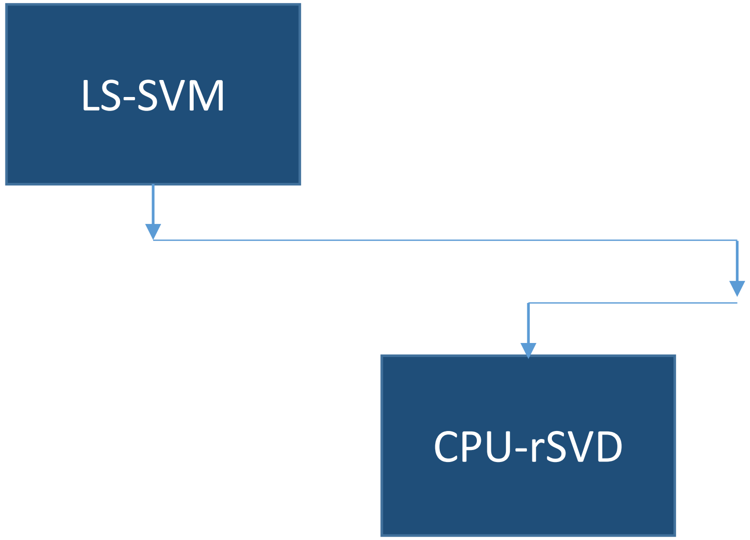
Original

```
if (use_cpu_original):
    print('I Doing CPU')
    U,S,V = np.linalg.svd(A)
    A_cross = V.T@np.linalg.inv(np.diag(S))@U.T
    solution = dot(A_cross, B)
```

Implementation Details



Implementation Details



ALGORITHM: RSVD — BASIC RANDOMIZED SVD

Inputs: An $m \times n$ matrix $\mathbf{A}$, a target rank k, and an over-sampling parameter p (say $p = 10$).
Outputs: Matrices $\mathbf{U}$, $\mathbf{D}$, and $\mathbf{V}$ in an approximate rank-$(k+p)$ SVD of $\mathbf{A}$ (so that $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, $\mathbf{D}$ is diagonal, and $\mathbf{A} \approx \mathbf{U}\mathbf{D}\mathbf{V}^*$.)

Stage A:

(1) Form an $n \times (k+p)$ Gaussian random matrix $\mathbf{G}$.

```
G = randn(n,k+p)
```

(2) Form the sample matrix $\mathbf{Y} = \mathbf{A}\,\mathbf{G}$.

```
Y = A*G
```

(3) Orthonormalize the columns of the sample matrix $\mathbf{Q} = \mathtt{orth}(\mathbf{Y})$.

```
[Q,~] = qr(Y,0)
```

Stage B:

(4) Form the $(k+p) \times n$ matrix $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$.

```
B = Q'*A
```

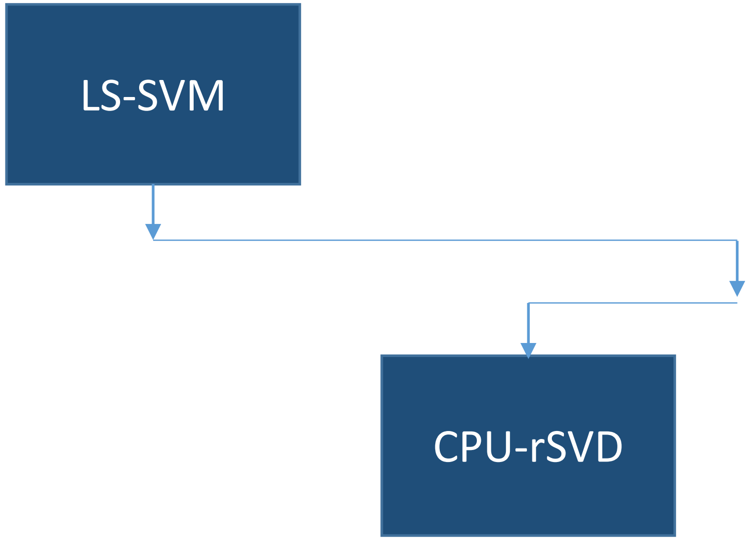
(5) Form the SVD of the small matrix $\mathbf{B}$: $\mathbf{B} = \hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*$.

```
[Uhat,D,V] = svd(B,'econ')
```

(6) Form $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

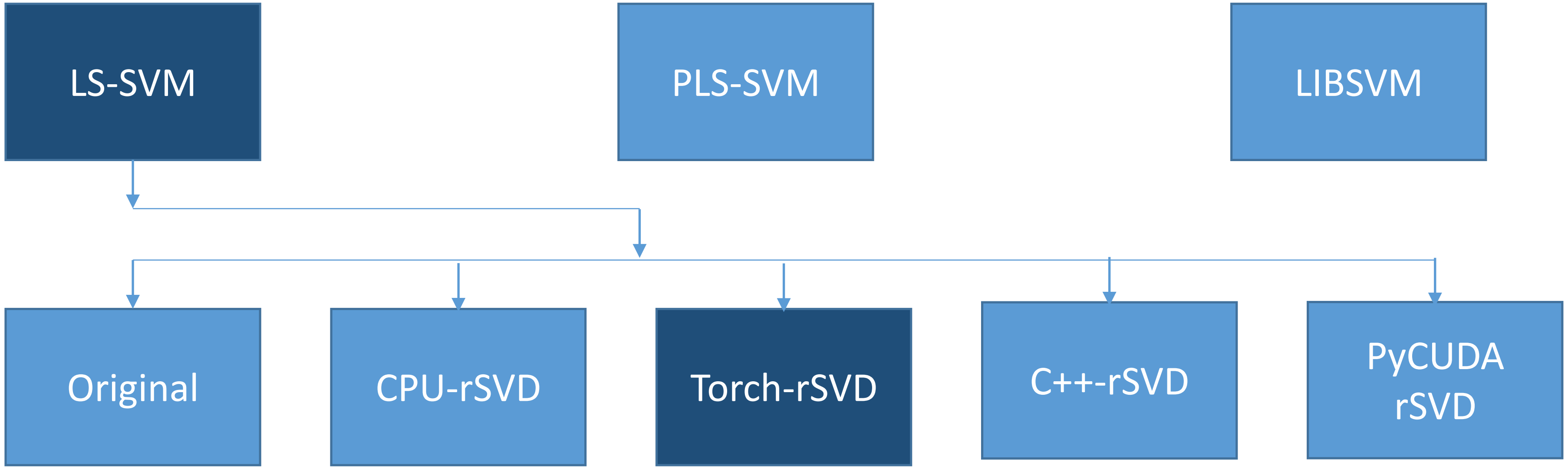
```
U = Q*Uhat
```

Implementation Details

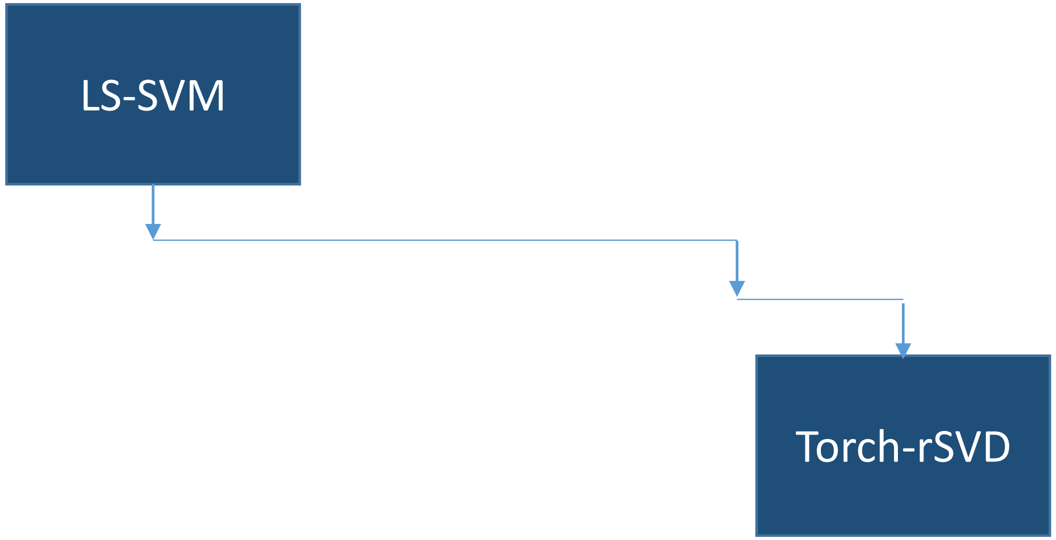



```python
if(use_cpu_rsvd):
    print('cpu_rsvd')

    # tic2=time.perf_counter()
    ##########################################################
    rank = math.ceil(0.25*Aa.shape[1])
    Omega = np.random.randn(Aa.shape[1], rank)
    Y = Aa @ Omega
    Q, _ = np.linalg.qr(Y)
    Bb = Q.T @ Aa
    u_tilde, s, v = np.linalg.svd(Bb, full_matrices = 0)
    u = Q @ u_tilde
    solution = np.dot(u.T, B)
    solution=np.divide(solution,s)
    solution=np.dot(v.T,solution)
    ##########################################################
```

Implementation Details

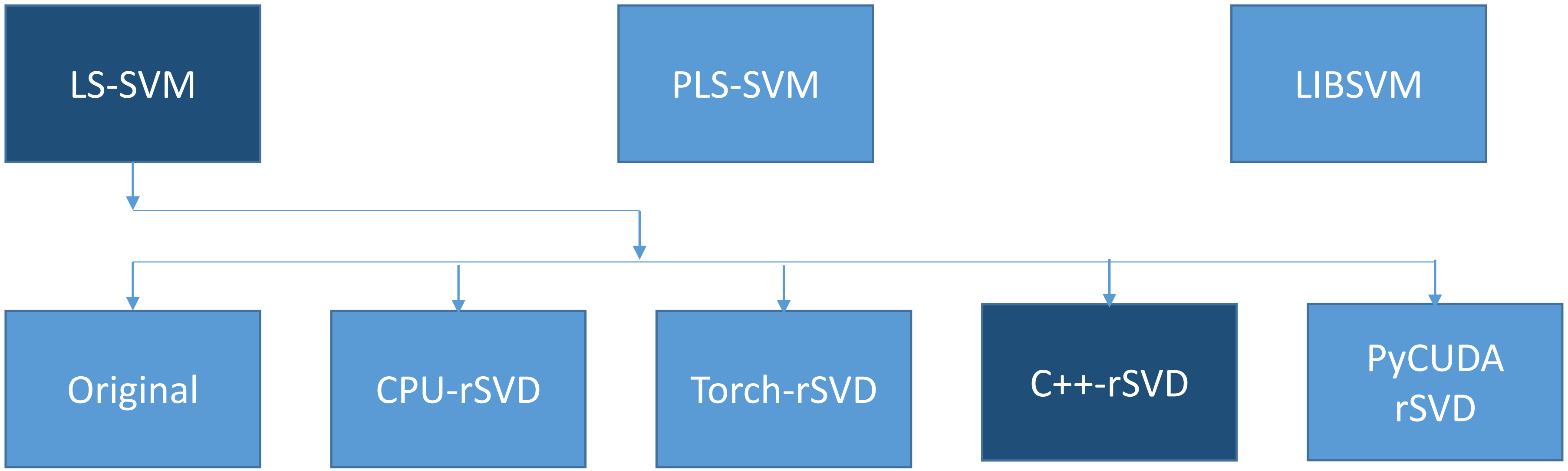


Implementation Details




```
if(use_gpu_rsvd):
    print('i DOing gpu')
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    at = torch.from_numpy(A)

    at=at.float()
    bb=torch.from_numpy(B)

    bb=bb.float()
    tic1=time.perf_counter()
    at=at.cuda()
    bb=bb.cuda()
    with torch.no_grad():

        rank1 = math.ceil(0.75*Aa.shape[1])
        Omega1 = torch.rand(at.shape[1],rank1,device=device)
        Y1=torch.matmul(at,Omega1)
        Q1, _ = torch.linalg.qr(at)
        Bb1=torch.matmul(torch.transpose(Q1,0,1),at)
        U_t,S,Vt=torch.linalg.svd(Bb1,full_matrices=False)
        U=torch.matmul(Q1,U_t)
        V=Vt.mH
        solution = torch.mv(U.mH,bb)
        solution = torch.div(solution,S)
        solution = torch.mv(V,solution)

    solution=solution.detach().cpu().numpy()
```

Implementation Details



Implementation Details

LS-SVM

C++-rSVD

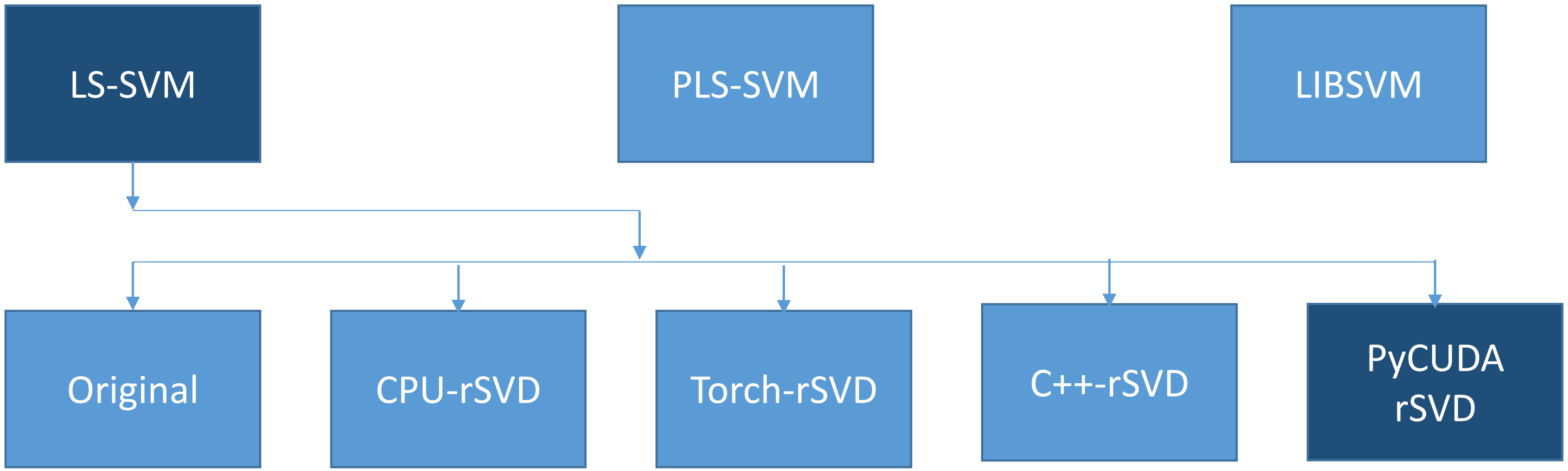
```
c302-005.ls6(128)$ icc -o c_implementation my_mkl.cpp -mkl
```

```
cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,m,n,k,1,A,k,Omega,n,0,C,n);
```

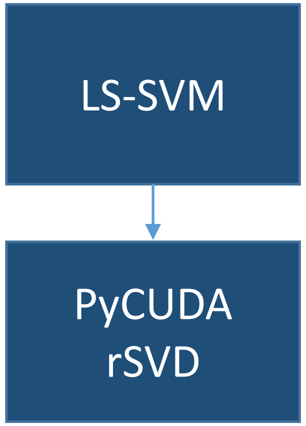
```
info =  LAPACKE_dgeqrf(LAPACK_ROW_MAJOR,m,n,C,n,tau);
```

```
info = LAPACKE_dorgqr(LAPACK_ROW_MAJOR,m,n,n,C,n,tau);
```

Implementation Details



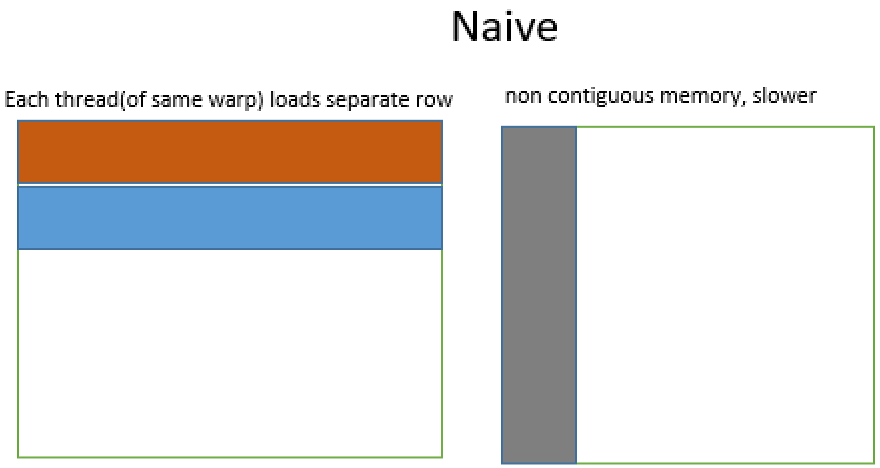
Implementation Details

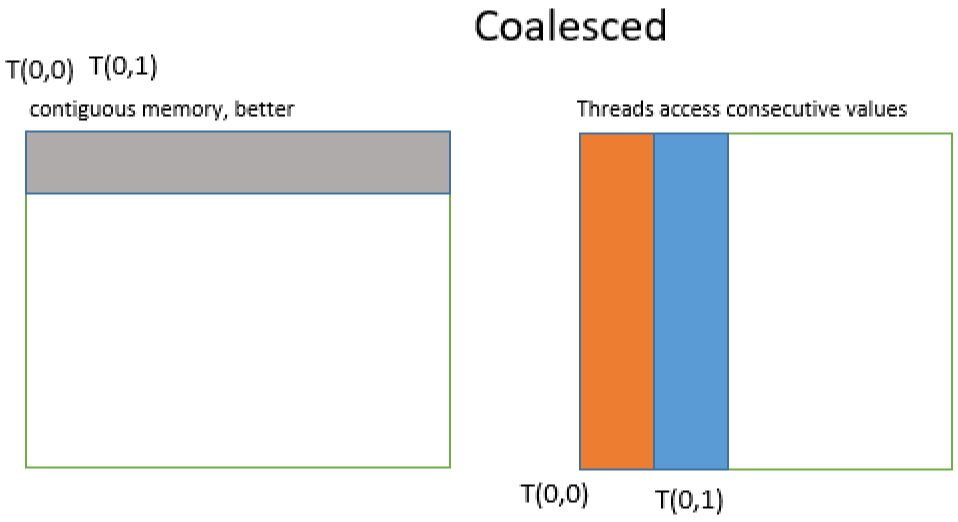


Matrix Multiplication on the GPU:

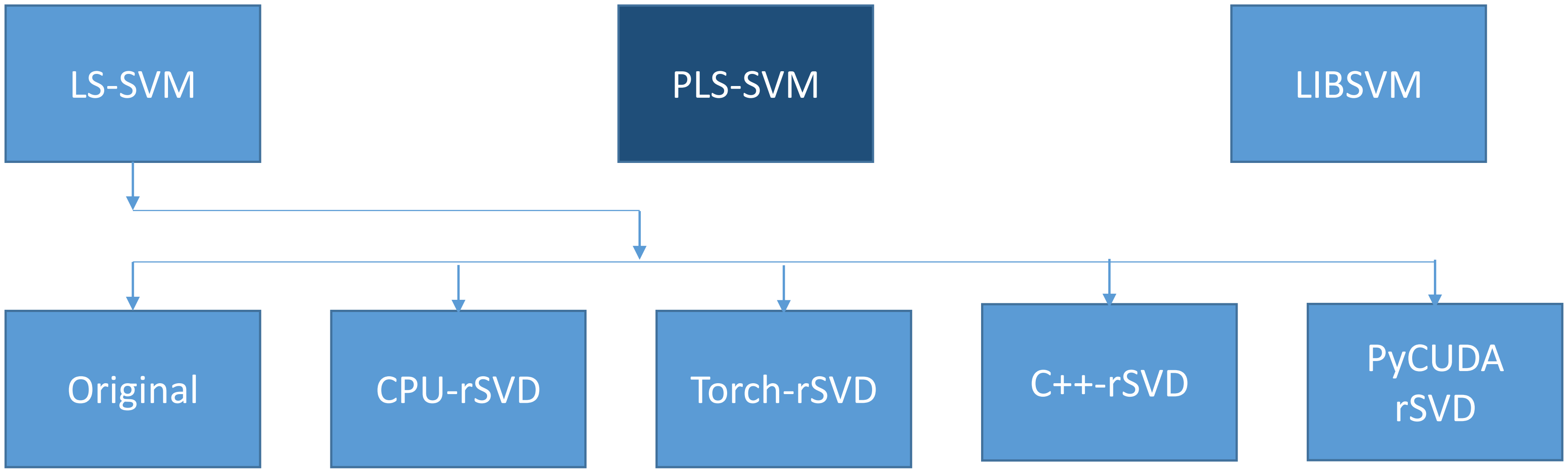
- Started with the Naïve Kernel. First improvement was by Memory Coalescing the access to data. Threads are launched as a set of warps, with T(0,0)→T(0,31). But the Naïve implementation makes the threads of the same warp access different memory locations, reducing efficiency. Code changed to reverse access pattern.
- Shared memory used with Tiling.
- Loop unrolling for the lowest level loop.
- Did some benchmark measurements and approximate FLOP counts for this kernel.

Language: CUDA/C++, also implemented in pycuda.





Implementation Details



Implementation Details

PLS-SVM

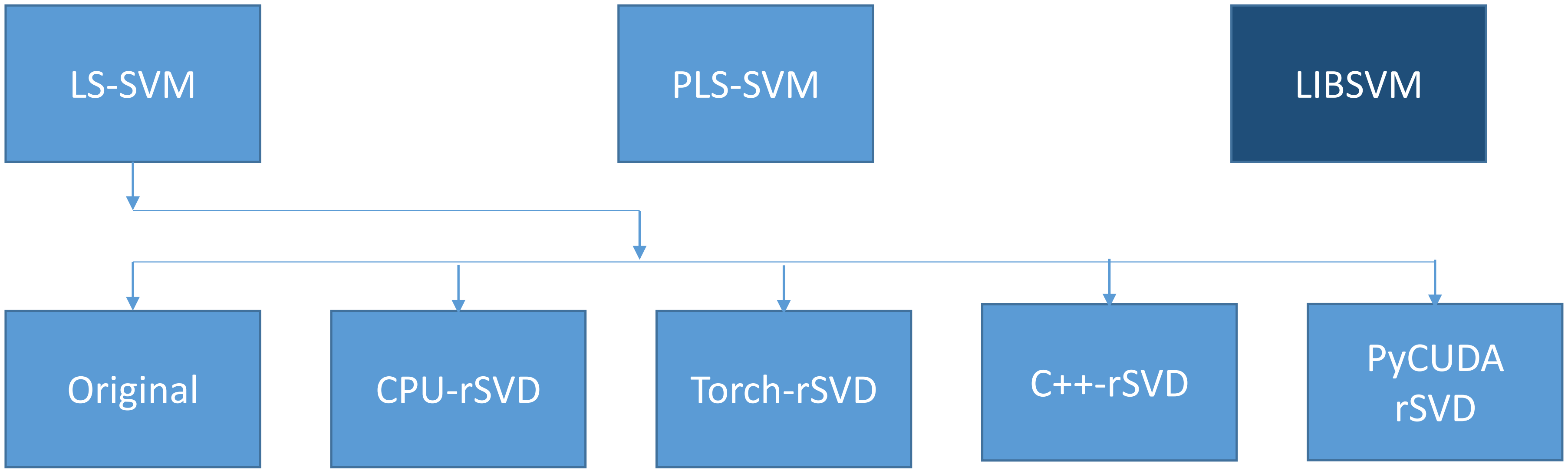
Implementations:
OpenMP
CUDA
HIP
OpenCL
SYCL

Parallel RSVD:

- Used a simple python implementation of LSSVM*
- Started with NumPy implementation for the algorithm, used np.linalg.svd() and then multiplied them correctly to get the pseudo inverse($\mathbf{A}^{+}$), and finally did solution=($\mathbf{A}^{+}*\mathbf{b}$)
- First changed this to the RSVD algorithm and then changed the multiplication order multiply **b** with the svd(A) in the correct order to minimize matrix-matrix multiplies.
- To get an idea of the parallelism we could expect, used torch to transfer the data to the GPU, and implemented the RSVD algorithm on the GPU.
- Implemented a C++ BLAS/LAPACKe RSVD algorithm, using blas libraries for QR, svd, etc.
- Implemented a PyCUDA version of RSVD,(using scikit-cuda for QR)where we also plug in our Matrix-Matrix kernel, and measure different benchmarks.

https://github.com/SC-SGS/PLSSVM

Implementation Details



Implementation Details

LIBSVM

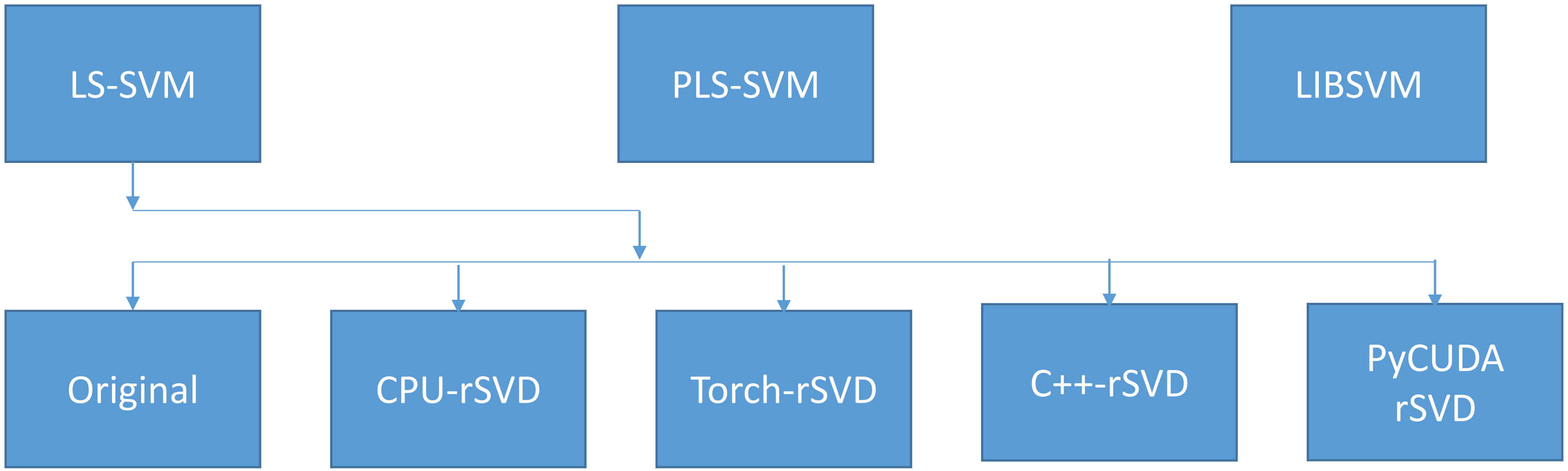
LIBSVM: A Library for Support Vector Machines

Chih-Chung Chang and Chih-Jen Lin
Department of Computer Science
National Taiwan University, Taipei, Taiwan
Email: cjlin@csie.ntu.edu.tw

Initial version: 2001   Last updated: August 23, 2022

svm-train, svm-predict, svm-scale

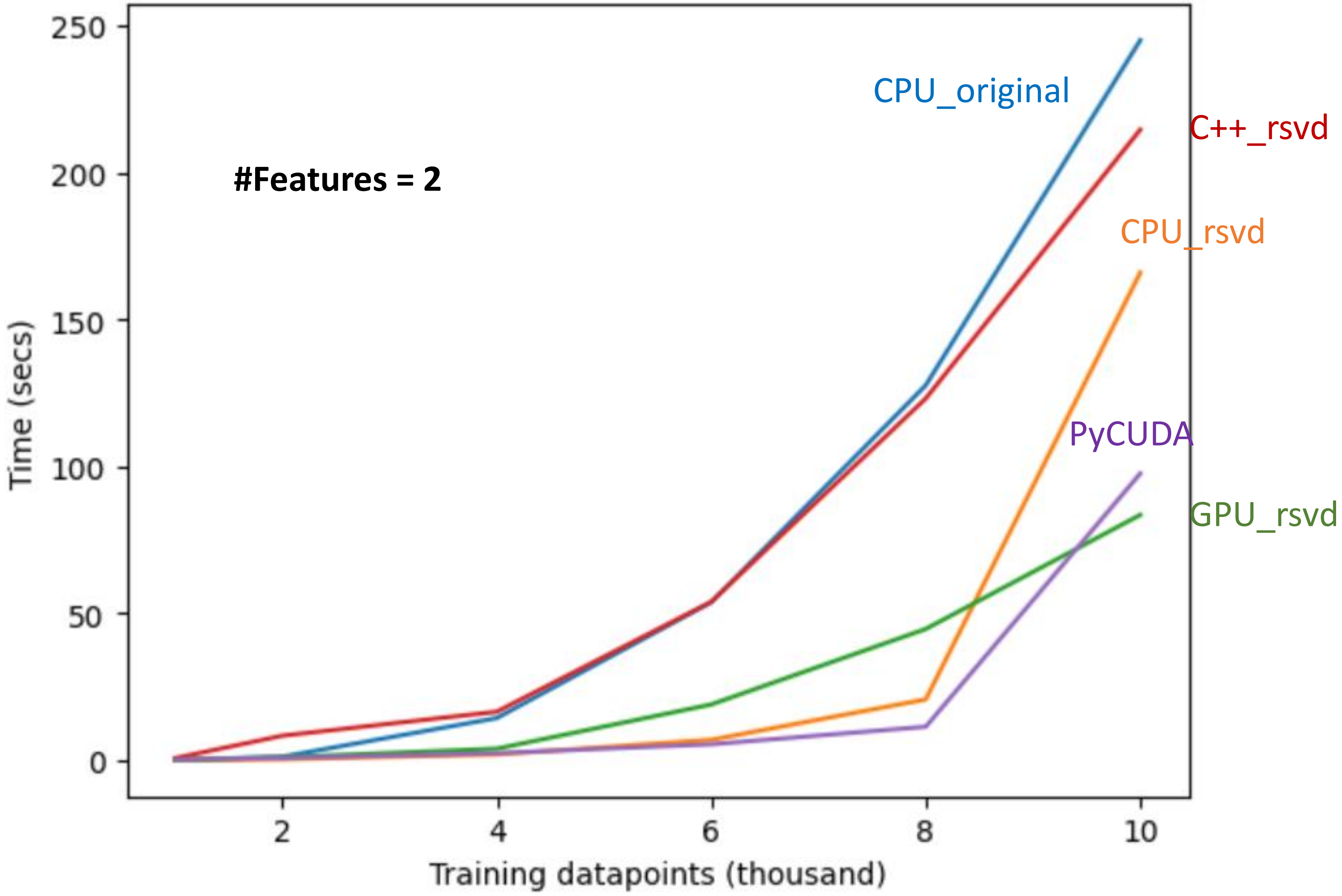
Implementation Details



OUTLINE

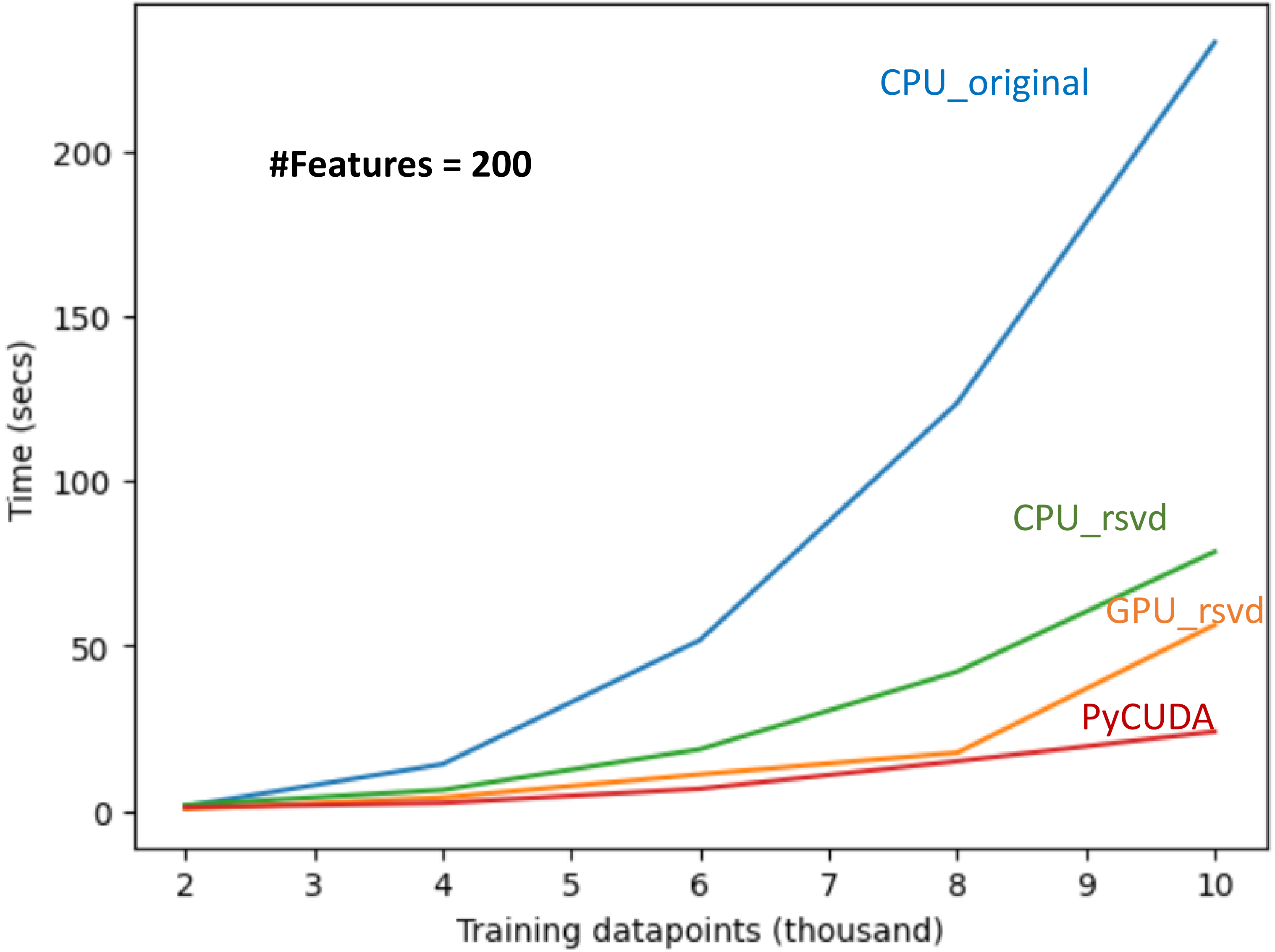
1. Problem Statement
2. Sequential – Complexity & Algorithm
3. Parallel – Complexity & Algorithm
4. Implementation Details
5. **Results**

Results



Time (secs)
250
200
150
100
50
0
2
4
6
8
10
Training datapoints (thousand)
#Features = 2
CPU_original
C++_rsvd
CPU_rsvd
PyCUDA
GPU_rsvd

Results



CPU_original
#Features = 200
CPU_rsvd
GPU_rsvd
PyCUDA
Time (secs)
200
150
100
50
0
2
3
4
5
6
7
8
9
10
Training datapoints (thousand)

OUTLINE

1. Problem Statement
2. Sequential – Complexity & Algorithm
3. Parallel – Complexity & Algorithm
4. Implementation Details
5. Results

END