Final Project: GPU Accelerated Support Vector Machines via Quadratic Programming

Armaan Kohli - ECE453 Advanced Computer Architecture
Spring 2020

Remarks

We implemented a fast support vector machine (SVM) classifier that leverages GPU architecture to achieve performance gains for large datasets. Specifically, our project formulates a SVM classifier as a quadratic program, which we can solve using the alternating direction method of multipliers (ADMM) with either LDL factorisation on the CPU or preconditioned conjugate gradient (PCG) on the GPU. Our project is an extension of the the `osqp` API [1], having added additional tools for SVM problem generation and linear system solving using PCG and CUDA to target an Nvidia Jetson Nano. We compare the results of our GPU implementation to the benchmarks provided by `osqp`.

Support Vector Machines

The support vector machine (SVM) is an example of a convex optimization problem w/ quadratic criteria. SVM is a technique for classifying data that may not be linearly separable. This is accomplished by performing a linear classification problem in a higher dimensional feature space where the data is linearly separable.

First, let's assume we have N tuples, (x_i, y_i), where $x_i \in R^m$ represent feature vectors, and y_i represents the true class, $y_i \in \{-1, 1\}$. For instance, in the Fig 1, the blue dots correspond to x_i from $y_i = -1$, and the red dots correspond to features from $y_i = 1$. Let's also assume we've already done some kind of projection into a space where the classes are linearly separable.

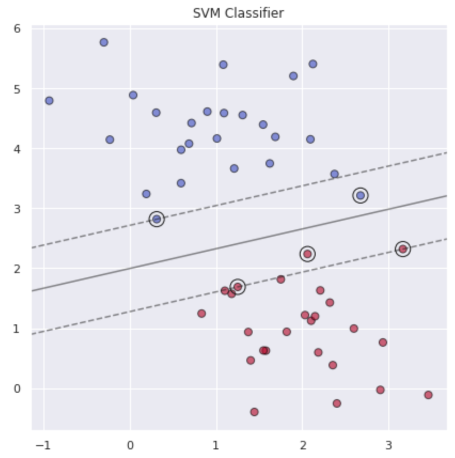

Figure 1: An illustrative example of an SVM classifier. Most of the construction of the SVM section is based on *The Elements of Statistical Learning* [2].

Now, we only have access to each value of x_i, the class they belong to, the corresponding y_i, is unknown. Let's define a hyperplane by:

$$\{x : f(x) = x^T\beta + 1 = 0\} \tag{1}$$

A classification rule induced by this hyperplane $f(x)$ is:

$$G(x) = sign[x^T\beta + 1] \tag{2}$$

Since we are in a space where the classes are linearly separable, we can find a function $f(x) = x^T\beta + 1$ with $y_i f(x_i) > 0 \quad \forall\ i$. Hence, we are able to find a hyperplane that creates the largest margin between the two classes. This is what the SVM accomplishes.
The circled points on the dotted lines are the so-called support vectors. The points between the dashed line and the decision boundary are within the margin. The SVM seeks to find the hyperplane that creates the largest margin, subject to the constraint to minimize the total distance of points on the wrong side of the margin.

We can formulate an SVM as a quadratic program using the method described in [2]. In this form, we can use an algorithm called alternating direction method of multipliers (ADMM).

Implementation Details

The osqp paper presents an two new ways of solving ADMM, a direct method an indirect method. The algorithm in full-form is presented below:

```
Algorithm 1: ADMM algorithm as presented in [1]
[ given: $x^0, z^0, y^0$ and parameters $\rho > 0,\ \sigma > 0,\ \alpha \in [0,2]$
while not terminated do
    $(\tilde{x}^{k+1}, v^{k+1}) \leftarrow \begin{bmatrix} P+\sigma I & A^T \\ A & -\rho^{-1}I \end{bmatrix} \begin{bmatrix} \tilde{x}^{k+1} \\ v^{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \rho^{-1}y^k \end{bmatrix}$
    $\tilde{z}^{k+1} \leftarrow z^k + \rho^{-1}(v^{k+1} - y^k)$
    $x^{k+1} \leftarrow \alpha\tilde{x}^{k+1} + (1-\alpha)x^k$
    $z^{k+1} \leftarrow \prod \left(\alpha\tilde{z}^{k+1} + (1-\alpha)z^k + \rho^{-1}y^k\right)$
    $y^{k+1} \leftarrow y^k + \rho(\alpha\tilde{z}^{k+1} + (1-\alpha)z^k - z^{k+1}$
end
```

Using this algorithm, we can very quickly compute solutions to SVM problems. However, there is one big issue with the algorithm, namely the matrix inversion step, which must be done every iterations until convergence. This is what we'll exploit to get performance gains. The CPU-based algorithm that osqp presents uses LDL matrix factorization to solve this linear system, called the KKT matrix. LDL factorization can be costly as the size of the problem increases, so for large datasets, we can leverage parallelism, and implement an indirect solver for this KKT matrix using PCG. As opposed to a direct method such as KKT, PCG can be efficiently parallelized.

Preconditioned Conjugate Gradient

In cases where the number of datapoints or the dimensionality of the space is too large for LDL factorization to work effectively, we can instead use an indirect method for solving this KKT linear system. PCG was suggested as a solution in [3]. This is accomplished by re-writing the KKT constraints as in the following form:

$$(P + \sigma I + A^T R A)\tilde{x}^{k+1} = \sigma x^k - q + A^T(Rz^k - y^k) \tag{3}$$

Which can be solved using the PCG algorithm below.

GPU Optimizations

In order to write an efficient GPU implementation of PCG, we used the following CUDA Toolkit libraries: `Thrust`, `cuBLAS` and `cuSPARSE`. `Thrust` provides a high-level interface for essential data parallel primitives, such as scan and sort operations. `cuBLAS` is a CUDA implementation of BLAS (Basic Linear Algebra Subprograms). We use only level-1 cuBLAS API functions that implement the inner product, axpyoperation, scalar-vector multiplication, and computation of norms. `cuSPARSE` is a CUDA library that contains a set of linear algebra subroutines for handling sparse matrices. Throughout the GPU codebase, we use a CSR format of sparse

Algorithm 2: PCG algorithm as presented in [3]

```
[ initialise: $r^0 = Kx^0 - b,\ y^0 = M^{-1}r^0,\ p^0 = -y^0,\ k = 0$
while $||r^k|| > \epsilon||b||$ do
    $\alpha^k \leftarrow -\frac{(r^k)^T y^k}{(p^k)^T K p^k}$
    $x^{k+1} \leftarrow x^k + \alpha^k p^k$
    $r^{k+1} \leftarrow r^k + \alpha^k K p^k$
    $y^{k+1} \leftarrow M^{-1} r^{k+1}$
    $\beta^{k+1} \leftarrow -\frac{(r^{k+1})^T y^{k+1}}{(r^k)^T y^k}$
    $p^{k+1} \leftarrow -y^{k+1} + \beta^{k+1} p^k$
    $k \leftarrow k + 1$
end
```

matricies, since that is the format that `cuSPARSE` uses. The rest of the `osqp` api uses CSC representation, so additional code was needed to perform conversion.

For the kernel functions, we used 64 elements per thread and 1024 threads per block. We found that this gave the best performance empirically, through a more thorough analysis or parameter search would be required to test for optimal settings.

Results & Discussion

In order to test the performance of our implementation, we generated random SVM problems of various dimensions and computed the run time. To create the CPU benchmarks, we use the CPU code provided by `osqp`. To test the GPU code, we used our own PCG implementation that we then interfaced with the `osqp` api. We generated the SVM problems using a simple `python` script and used the provided code generation tools provided by `osqp` to generate header files containing the data. Below in Fig. 2 is the performance of the SVM classifier according to the results from [3].

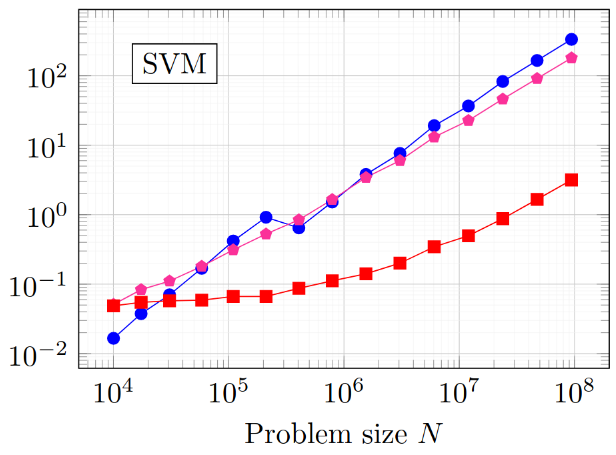

Figure 2: This a plot replicated from [3]. The red line on the right-hand graph shows the performance of the algorithm from [3] in terms of runtime measured in seconds versus problem size. The blue line is the CPU implementation from [1].

They found that using a GPU implementation for SVM solving didn't give massive improvements. However the CPU implementation scales much faster than the GPU implementation as the problem size increases. Furthermore, latency between the host and target devices meant that for smaller problems, the CPU algorithm was superior to the GPU algorithm. But as the problem size increased, the overhead was overcome and the GPU achieved better performance.

For our implementation, we were unable to generate problems greater than 10^4. However, we did see similar performance improvements, our results are pictured in Fig. 3. As was the case for the official implementation, we found that for small problem size, the overhead made the CPU more efficient than the GPU. But, as the problem size increased, the GPU gave superior performance. And, as the problem size increased, the GPU runtime increased at a much slower rate than the CPU runtime.

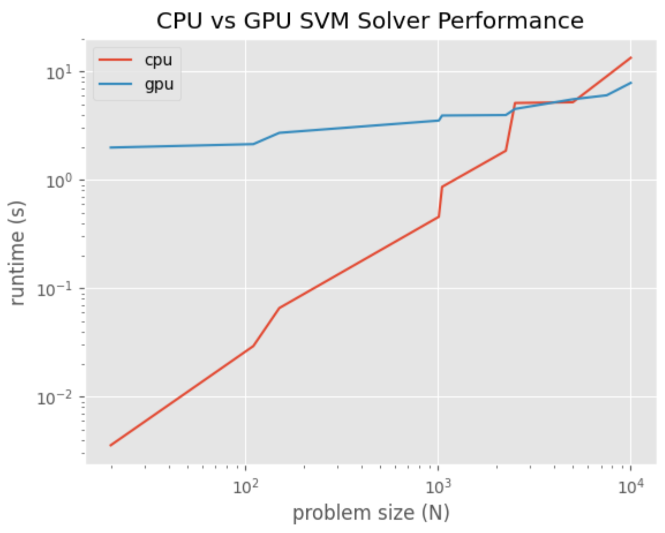

Figure 3: This a plot of our SVM solver results. Unlike the results presented by [3], we were not able to generate problems larger than 10^4. However, the trends in the graphs are the same. For larger problem sizes, GPU outperforms CPU, and GPU performance scales better for larger problems. These metrics were compiled by generating random SVM problems of different sizes, then averaging over several problems. All experiments were run on a Jetson Nano.

Conclusion

In conclusion, we were able to develop our own version of a GPU implementation of ADMM for SVM classification using the `osqp` api. We also successfully verified the results from [3] for small scale problems.

References

[1] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An operator splitting solver for quadratic programs," *Mathematical Programming Computation*, 2020. [Online]. Available: https://doi.org/10.1007/s12532-020-00179-2

[2] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[3] M. Schubiger, G. Banjac, and J. Lygeros, "GPU acceleration of ADMM for large-scale quadratic programming," *arXiv:1912.04263*, 2019.

Appendix A: Code

The code is used to generate SVM classification problems `dmm.py`, the main model code.

The code below contains all of the main `CUDA` routines for the PGC algorithm.

The code below wraps the `CUDA` functions into a `linsys` solver module specified by the `osqp` api so that we can actually solve a linear system via PCG.