Supplemental Material for “DeepLM: Large-scale Nonlinear Least Squares on Deep Learning Frameworks using Stochastic Domain Decomposition”

Anonymous CVPR 2021 submission

Paper ID 615

Dataset	Ceres-CG	PBA	STBA	Ours-G	Ours-BG	Ours-B
Union Square	17.9	5.65	32.5	2.97	1.86	**1.67**
P. del Popolo	8.49	2.13	18.3	**0.90**	1.13	1.09
Ellis Island	18.6	4.67	14.50	1.60	1.63	**1.42**
NYC Library	6.43	1.65	-	2.90	1.45	**1.38**
M. N. Dame	52.1	10.2	37.5	1.76	1.56	**1.21**
Gen. markt	65.0	15.7	95.5	4.05	1.32	**1.06**
Alamo	16.5	4.95	217	1.5	1.37	**1.23**
Yorkminster	36.3	9.66	115	3.76	3.24	**2.95**
Roman Forum	60.0	16.3	167	6.80	6.02	**4.13**
V. Cathedral	36.9	11.5	75.3	4.45	2.58	**2.23**
M. Metropolis	30.1	8.33	216	7.16	3.15	**2.94**
Piccadily	95.2	24.0	828	7.79	4.21	**3.85**
T. of London	377	95.36	1052	45.7	8.74	**5.46**
Trafalgar	188	49.8	6739	34.8	6.79	**5.59**

Table 1. Time (second) takes for different methods to optimize the problems in 1DSFM [3] dataset.

Dataset	Ceres-CG	PBA	STBA	Ours-G	Ours-BG	Ours-B
Union Square	0.094	0.055	0.168	0.044	0.021	**0.006**
P. del Popolo	0.106	0.065	0.190	0.051	0.023	**0.006**
Ellis Island	0.110	0.071	0.172	0.056	0.026	**0.007**
NYC Library	0.131	0.079	-	0.061	0.034	**0.008**
M. N. Dame	0.201	0.143	0.320	0.108	0.040	**0.014**
Gen. markt	0.207	0.148	0.384	0.115	0.041	**0.015**
Alamo	0.228	0.156	0.544	0.121	0.043	**0.015**
Yorkminster	0.238	0.175	0.388	0.136	0.042	**0.018**
Roman Forum	0.311	0.218	0.576	0.176	0.053	**0.022**
V. Cathedral	0.347	0.232	0.608	0.183	0.067	**0.023**
M. Metropolis	0.444	0.314	0.640	0.239	0.094	**0.030**
Piccadily	0.598	0.425	2.360	0.330	0.103	**0.041**
T. of London	1.121	0.739	1.472	0.596	0.237	**0.076**
Trafalgar	1.406	0.963	7.328	0.767	0.256	**0.099**

Table 2. Memory (GB) used by different methods to optimize the problems in 1DSFM [3] dataset.

1. Evaluation

1.1. Evaluation on 1DSFM

We report the time (second) and memory (GB) usage in Table 1 and Table 2 for 1DSFM [3] dataset for different methods including We compare our methods with Ceres [1] using conjugate gradient (Ceres-CG), PBA [4], STBA [5] and different variants of our methods. For our methods, we report “Ours-G” as our Pytorch-based LM solver that optimizes the problem globally as a whole. “Ours-BG” and “Ours-B” represent stochastic domain composition using our whole pipeline with and without global reinitialization. The statistics lead to the same conclusion derived from Section 4.1 (**Efficiency and Memory**) in the main paper.

Among existing global solvers, PBA [4] on GPU is faster than other algorithms run in multicore CPU implementation. Ours-G is even faster than PBA by directly calling an efficient “index_add_” in Pytorch to implement $\mathbf{J}^T\mathbf{r}$.

“Ours-G” uses less memory than other existing solvers. Typically, we use less memory to compute jacobians with our backward jacobian network. By solving subproblems in parallel for “Ours-BG” and “Ours-B”, we significantly reduce the computation time and maximum memory usage for each sub-problem. Therefore, our stochastic solver supports the optimization of problems on a very large scale. “Ours-B” is more efficient than “Ours-BG” considering time and memory since it does not perform global reinitialization by slightly sacrificing the quality. In practice, either of them can be used depending on whether the application prefers quality or speed.

1.2. Solver robustness

We repeat processing Ladybug dataset using “ours-B” for 5 times, and the final losses are 1.131, 1.129, 1.138, 1.133, 1.139. As a result, our stochastic solver shows stability for preserving the quality of the solution.

1.3. Performance Profiling

We follow [5] and report the performance profile of all 19 scenes in Table 2 of the main paper in Figure 1 by setting $\tau = 0.1$. We show significant performance increase among existing methods.

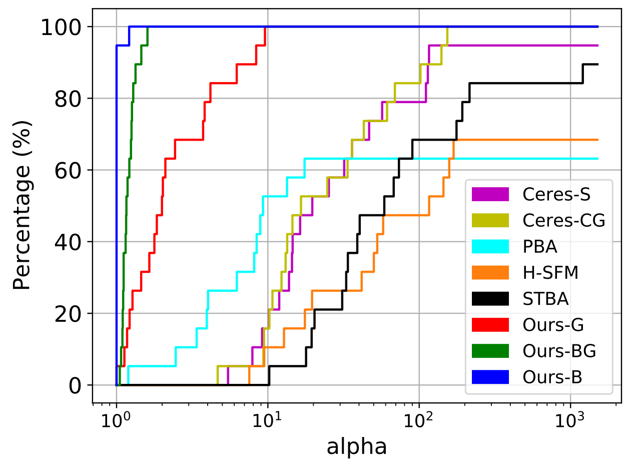

Figure 1. Performance Profile.

2. Implementation Details

2.1. Solver Interface

We aim at providing an general solver with a easy-to-use interface. The interface of our solver is implemented using Python that takes four inputs, including a list of variables $\mathbf{x} = \{\mathbf{v}_i\}$, a list of constants $\mathbf{c}_i$, a list of variable indices $\mathbf{I}_i$, and a user-defined function $\mathbf{f}$. $\mathbf{f}$ takes arguments in the order of $(\mathbf{v}_1(\mathbf{I}_1), ..., \mathbf{v}_m(\mathbf{I}_m), \mathbf{c}_1, ...\mathbf{c}_n)$ and output a residual $\mathbf{r}$. Each input is a tensor with multiple dimensions implemented in Pytorch. The j-th element for the variable indices as $\mathbf{I}_i[j]$ corresponds to the j-th residual $\mathbf{r}_j$. We allow both $\mathbf{I}_i[j]$ and $\mathbf{r}_j$ to have multiple dimensions, but require that indices inside $\mathbf{I}_i[j]$ for each j is unique. As a result, a nonlinear least squares problem can be easily formulated using our solver.

2.2. Jacobian Representation

The loss of backward jacobian network can be implemented by computing residuals $\mathbf{r}$ from inputs defined in Section 2.1. Then, we compute the loss L as the reduced sum of $\mathbf{r}$. It is important that we set $\{\mathbf{v}_i(\mathbf{I}_i)\}$ rather than $\{\mathbf{v}_i\}$ as network parameters, and collect their gradients by calling backward function from L. As a result, we collect a set of gradient tensors $\{\mathbf{g}_i\}$, each of which has the same dimensions with $\mathbf{v}_i(\mathbf{I}_i)$. The final jacobian is stored as a list of tensors as $\{\mathbf{g}_1, ..., \mathbf{g}_m, \mathbf{I}_1, ..., \mathbf{I}_m\}$.

2.3. Kernels

To implement an LM solver in Pytorch, two most important kernels are the computation of $\mathbf{Jx}$ and $\mathbf{J}^T\mathbf{r}$. $\mathbf{Jx}$ can be computed as shown in Equation 1.

$$\mathbf{Jx} = \sum_{i=1}^{m} \mathbf{J}_i \mathbf{v}_i \tag{1}$$

$\mathbf{J}_i$ is represented using $(\mathbf{g}_i, \mathbf{I}_i)$. Specifically, we compute $\mathbf{J}_i\mathbf{v_i}$ as the reduced sum over all dimensions except the first one for $\mathbf{g}_i \circ \mathbf{v}_i(\mathbf{I}_i)$, where $\circ$ represents the element-wise multiplication operator.

$\mathbf{J}^T\mathbf{r}$ leads to a tensor with the same dimensions as the variable $\mathbf{x}$. Therefore, we use a list of m tensors $\{\mathbf{J}_1^T\mathbf{r}, ..., \mathbf{J}_m^T\mathbf{r}\}$ to represent it. To compute $\mathbf{J}_i^T\mathbf{r}$, we first expand $\mathbf{r}$ to $\mathbf{r}_i^*$ which have the same dimension as $\mathbf{g}_i$ does. We compute $\mathbf{g}_i \circ \mathbf{r}_i^*$ and call "index_add_" to compute the reduced sum of $\mathbf{g}_i \circ \mathbf{r}_i^*$ through index $\mathbf{I}_i$.

Combining these components with other basic tensor operators, we implement an LM solver with a preconditioned conjugate gradient linear solver in Pytorch. The linear solver terminates if the relative error is smaller than $1e-3$ or the number of linear steps is larger than 150. Triggs correction [2] can be further integrated as a robust kernel in the system.

References

[1] Sameer Agarwal, Keir Mierle, et al. Ceres solver. 2012. 1

[2] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer, 1999. 2

[3] Kyle Wilson and Noah Snavely. Robust global translations with 1dsfm. In *European Conference on Computer Vision*, pages 61–75. Springer, 2014. 1

[4] Changchang Wu, Sameer Agarwal, Brian Curless, and Steven M Seitz. Multicore bundle adjustment. In *CVPR 2011*, pages 3057–3064. IEEE, 2011. 1

[5] Lei Zhou, Zixin Luo, Mingmin Zhen, Tianwei Shen, Shiwei Li, Zhuofei Huang, Tian Fang, and Long Quan. Stochastic bundle adjustment for efficient and scalable 3d reconstruction. *arXiv preprint arXiv:2008.00446*, 2020. 1