fpie

Release 0.2.4

Jiayi Weng

Nov 07, 2022

CONTENTS

Poisson Image Editing is a technique that can blend two images together without artifacts. Given a source image and its corresponding mask, and a coordination on target image, this algorithm can always generate amazing result.

This project aims to provide a fast poisson image editing algorithm (based on Jacobi Method) that can utilize multi-core CPU or GPU to handle a high-resolution image input.

CHAPTER

ONE

GET START

1.1 Installation

1.1.1 Linux/macOS

```
# install cmake >= 3.4
# if you don't have sudo (like GHC), install cmake from source
# on macOS, type `brew install cmake`
$ pip install fpie

# or install from source
$ pip install .
```

1.1.2 Extensions

We provide 7 backends:

- NumPy, `pip install numpy`;
- Numba, `pip install numba`;
- GCC, needs cmake and gcc;
- OpenMP, needs cmake and gcc (on macOS you need to change clang to gcc-11);
- CUDA, needs nvcc;
- MPI, needs mpicc (on macOS: `brew install open-mpi`) and `pip install mpi4py`;
- Taichi, `pip install taichi`.

Please refer to *Backend* for various usages.

After installation, you can use `--check-backend` option to verify:

```
$ fpie --check-backend
['numpy', 'numba', 'taichi-cpu', 'taichi-gpu', 'gcc', 'openmp', 'mpi', 'cuda']
```

The above output shows all extensions have successfully installed.

1.2 Usage

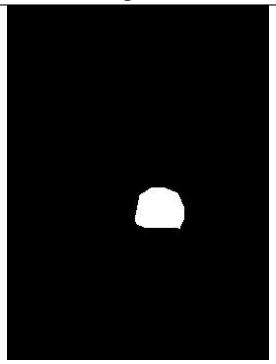
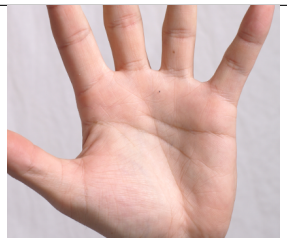
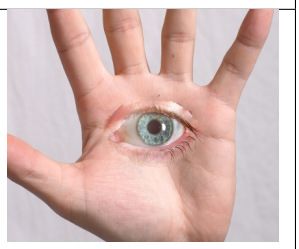
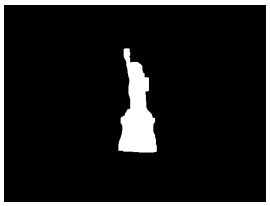
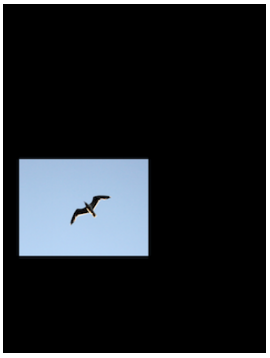
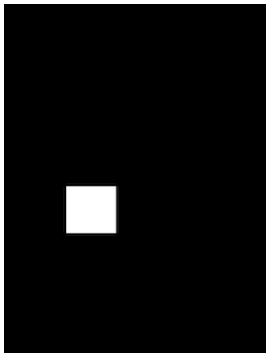
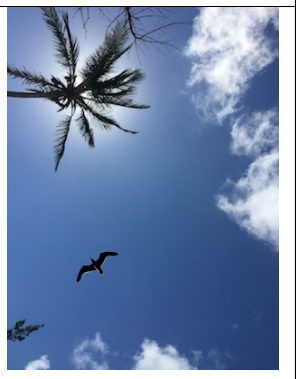
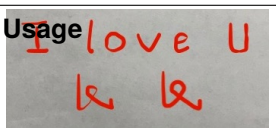
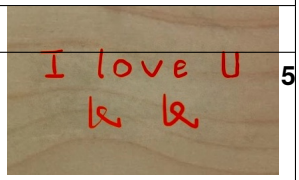
We have prepared the test suite to run:

```
$ cd tests && ./data.py
```

This script will download 8 tests from GitHub, and create 10 images for benchmarking (5 circle, 5 square). To run:

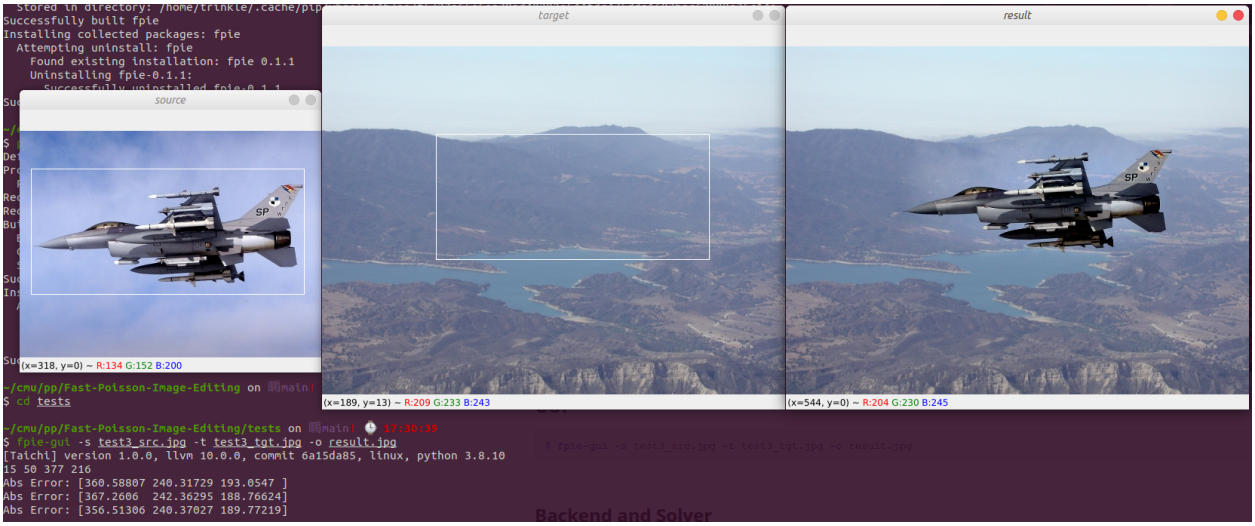
```
$ fpie -s test1_src.jpg -m test1_mask.jpg -t test1_tgt.jpg -o result1.jpg -h1 -150 -w1 -
↪50 -n 5000 -g max
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result2.jpg -h1 130 -w1␣
↪130 -n 5000 -g src
$ fpie -s test3_src.jpg -m test3_mask.jpg -t test3_tgt.jpg -o result3.jpg -h1 100 -w1␣
↪100 -n 5000 -g max
$ fpie -s test4_src.jpg -m test4_mask.jpg -t test4_tgt.jpg -o result4.jpg -h1 100 -w1␣
↪100 -n 5000 -g max
$ fpie -s test5_src.jpg -m test5_mask.png -t test5_tgt.jpg -o result5.jpg -h0 -70 -w0 0 -
↪h1 50 -w1 0 -n 5000 -g max
$ fpie -s test6_src.png -m test6_mask.png -t test6_tgt.png -o result6.jpg -h1 50 -w1 0 -
↪n 5000 -g max
$ fpie -s test7_src.jpg -t test7_tgt.jpg -o result7.jpg -h1 50 -w1 30 -n 5000 -g max
$ fpie -s test8_src.jpg -t test8_tgt.jpg -o result8.jpg -h1 90 -w1 90 -n 10000 -g max
```

Here are the results:

#	Source image	Mask image	Target image	Result image
1				
2				
3				
4				
5				
6				
7		/		

1.2.1 GUI

```
$ fpie-gui -s test3_src.jpg -t test3_tgt.jpg -o result.jpg -b cuda -n 10000
```



We provide a simple GUI for real-time seamless cloning. You need to use your mouse to draw a rectangle on top of the source image, and click a point in target image. After that the result will automatically be generated. In the end, you can press ESC to terminate the program.

1.2.2 Backend and Solver

We have provided 7 backends. Each backend has two solvers: EquSolver and GridSolver. You can find the difference between these two solvers in the next section.

For different backend usage, please check out the related documentation under docs/backend.md.

For other usage, please run `fpie -h` or `fpie-gui -h` to see the hint.

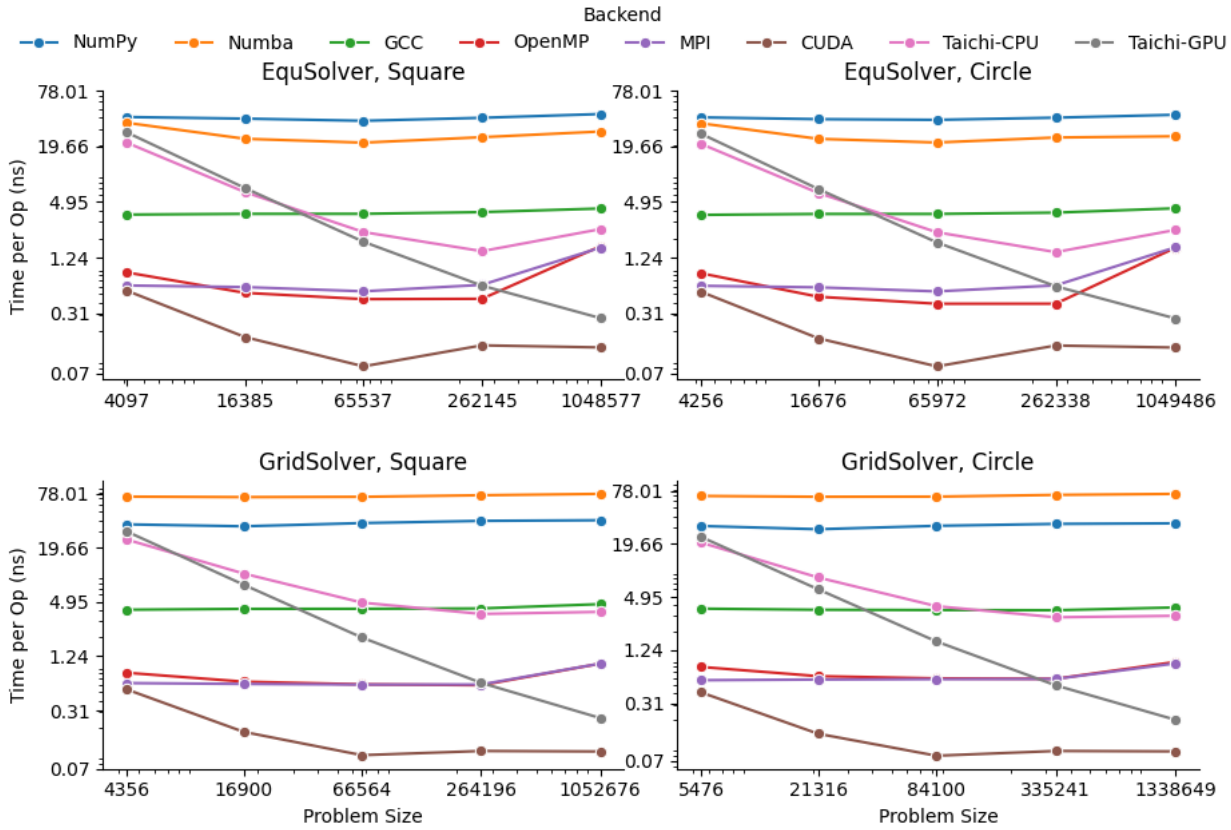
```
$ fpie -h
usage: fpie [-h] [-v] [--check-backend] [-b {numpy,numba,taichi-cpu,taichi-gpu,gcc,
↪openmp,mpi,cuda}] [-c CPU] [-z BLOCK_SIZE]
            [--method {equ,grid}] [-s SOURCE] [-m MASK] [-t TARGET] [-o OUTPUT] [-h0 H0]␣
↪[-w0 W0] [-h1 H1] [-w1 W1] [-g {max,src,avg}]
            [-n N] [-p P] [--mpi-sync-interval MPI_SYNC_INTERVAL] [--grid-x GRID_X] [--
↪grid-y GRID_Y]

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show the version and exit
  --check-backend       print all available backends
  -b {numpy,numba,taichi-cpu,taichi-gpu,gcc,openmp,mpi,cuda}, --backend {numpy,numba,
↪taichi-cpu,taichi-gpu,gcc,openmp,mpi,cuda}
                        backend choice
  -c CPU, --cpu CPU     number of CPU used
  -z BLOCK_SIZE, --block-size BLOCK_SIZE
                        cuda block size (only for equ solver)
  --method {equ,grid}   how to parallelize computation
```

(continues on next page)

(continued from previous page)

```
  -s SOURCE, --source SOURCE
                        source image filename
  -m MASK, --mask MASK  mask image filename (default is to use the whole source image)
  -t TARGET, --target TARGET
                        target image filename
  -o OUTPUT, --output OUTPUT
                        output image filename
  -h0 H0                mask position (height) on source image
  -w0 W0                mask position (width) on source image
  -h1 H1                mask position (height) on target image
  -w1 W1                mask position (width) on target image
  -g {max,src,avg}, --gradient {max,src,avg}
                        how to calculate gradient for PIE
  -n N                  how many iteration would you perfer, the more the better
  -p P                  output result every P iteration
  --mpi-sync-interval MPI_SYNC_INTERVAL
                        MPI sync iteration interval
  --grid-x GRID_X       x axis stride for grid solver
  --grid-y GRID_Y       y axis stride for grid solver
```

1.3 Benchmark Result



Please refer to *Benchmark* for detail.

1.4 Algorithm Detail

The general idea is to keep most of gradient in source image, while matching the boundary of source image and target image pixels.

The gradient is computed by

$\nabla(x, y) = 4I(x, y) - I(x-1, y) - I(x, y-1) - I(x+1, y) - I(x, y+1)$

After computing the gradient in source image, the algorithm tries to solve the following problem: given the gradient and the boundary value, calculate the approximate solution that meets the requirement, i.e., to keep target image's gradient as similar as the source image.

This process can be formulated as $(4 - A)\vec{x} = \vec{b}$, where $A \in \mathbb{R}^{N \times N}$, $\vec{x} \in \mathbb{R}^N$, $\vec{b} \in \mathbb{R}^N$, N is the number of pixels in the mask, A is a giant sparse matrix because each line of A only contains at most 4 non-zero value (neighborhood), $\vec{b}$ is the gradient from source image, and $\vec{x}$ is the result value.

N is always a large number, i.e., greater than 50k, so the Gauss-Jordan Elimination cannot be directly applied here because of the high time complexity $O(N^3)$. People use Jacobi Method to solve the problem. Thanks to the sparsity of matrix A, the overall time complexity is $O(MN)$ where M is the number of iteration performed by poisson image editing.

This project parallelizes Jacobi method to speed up the computation. To our best knowledge, there's no public project on GitHub that implements poisson image editing with either OpenMP, or MPI, or CUDA. All of them can only handle a small size image workload.

1.4.1 EquSolver vs GridSolver

Usage: `--method {equ,grid}`

EquSolver directly constructs the equations $(4 - A)\vec{x} = \vec{b}$ by re-labeling the pixel, and use Jacobi method to get the solution via $\vec{x}' = (A\vec{x} + \vec{b})/4$.

GridSolver uses the same Jacobi iteration, however, it keeps the 2D structure of the original image instead of re-labeling the pixel in the mask. It may take some advantage when the mask region covers all of the image, because in this case GridSolver can save 4 read instructions by directly calculating the neighborhood's coordinate.

If the GridSolver's parameter is carefully tuned (`--grid-x` and `--grid-y`), it can always perform better than EquSolver with different backend configuration.

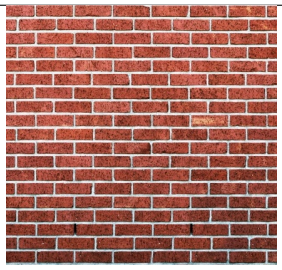
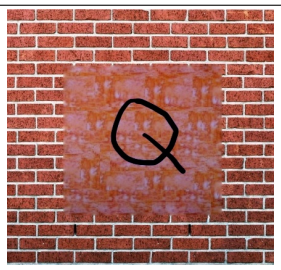
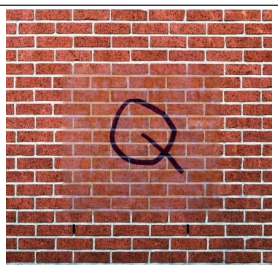
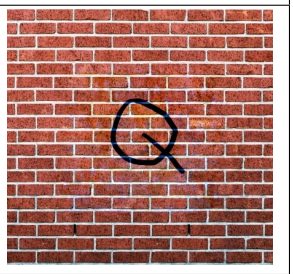
1.4.2 Gradient for PIE

Usage: `-g {max,src,avg}`

The PIE paper states some variant of gradient calculation such as Equ. 12: using the maximum gradient to perform "mixed seamless cloning". We also provide such an option in our program:

- `src`: only use the gradient from source image
- `avg`: use the average gradient of source image and target image
- `max`: use the max gradient of source and target image

The following example shows the difference between these three methods:

#	target image	–gradient=src	–gradient=avg	–gradient=max
3				
4				
8				

1.5 15-618 Course Project Final Report

Please refer to *Final Report*.

CHAPTER

TWO

BACKEND

To specify backend, simply typing `-b cuda` or `--backend openmp`, together with other parameters described below.

Feel free to play `fpie` with other arguments!

2.1 GridSolver

GridSolver keeps most of the 2D structure of the image, instead of relabeling pixels as EquSolver. To use GridSolver in some of the following backends, you need to specify `--grid-x` and `--grid-y` to determine the access pattern of the large 2D array.

Here is a Python pseudocode to show how it works:

```
arr = np.random.random(size=[N, M])
# here is a sequential scan:
for i in range(N):
    for j in range(M):
        func(arr[i, j])
# however, we can use block-level access pattern to improve the cache hit rate:
for i in range(N // grid_x):
    for j in range(M // grid_y):
        # the grid size is (grid_x, grid_y)
        for x in range(grid_x):
            for y in range(grid_y):
                func(arr[i * grid_x + x, j * grid_y + y])
```

2.2 NumPy

This backend uses NumPy vectorized operation for parallel computation.

There's no extra parameter for NumPy EquSolver:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b numpy --method equ
Successfully initialize PIE equ solver with numpy backend
# of vars: 12559
Iter 5000, abs error [450.09415 445.24747 636.1397 ]
Time elapsed: 3.26s
Successfully write image to result.jpg
```

There's no extra parameter for NumPy GridSolver:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b numpy --method grid
Successfully initialize PIE grid solver with numpy backend
# of vars: 17227
Iter 5000, abs error [450.07922 445.27014 636.1374 ]
Time elapsed: 3.09s
Successfully write image to result.jpg
```

2.3 Numba

This backend use NumPy vectorized operation together with numba jit function for parallel computation.

There's no extra parameter for Numba EquSolver:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b numba --method equ
Successfully initialize PIE equ solver with numba backend
# of vars: 12559
Iter 5000, abs error [449.83978128 445.02560616 635.9542823 ]
Time elapsed: 1.5883s
Successfully write image to result.jpg
```

There's no extra parameter for Numba GridSolver:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b numba --method grid
Successfully initialize PIE grid solver with numba backend
# of vars: 17227
Iter 5000, abs error [449.89603 445.08475 635.89545]
Time elapsed: 5.6462s
Successfully write image to result.jpg
```

2.4 GCC

This backend uses a single thread C++ program to perform computation.

There's no extra parameter for GCC EquSolver:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b gcc --method equ
Successfully initialize PIE equ solver with gcc backend
# of vars: 12559
Iter 5000, abs error [ 5.179281   6.6939087 11.006622 ]
Time elapsed: 0.29s
Successfully write image to result.jpg
```

For GCC GridSolver, you need to specify `--grid-x` and `--grid-y` described in the first section:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b gcc --method grid --grid-x 8 --grid-y 8
Successfully initialize PIE grid solver with gcc backend
# of vars: 17227
Iter 5000, abs error [ 5.1776047  6.69458   11.001862 ]
Time elapsed: 0.36s
Successfully write image to result.jpg
```

2.5 Taichi

Taichi is an open-source, imperative, parallel programming language for high-performance numerical computation. We provide 2 choices: `taichi-cpu` for CPU-level parallelization, `taichi-gpu` for GPU-level parallelization. You can install taichi via `pip install taichi`.

- For `taichi-cpu`: use `-c` or `--cpu` to determine how many CPUs it will use;
- For `taichi-gpu`: use `-z` or `--block-size` to determine the number of threads used in a block.

The parallelization strategy for Taichi backend is written by Taichi itself.

There's no other parameters for Taichi EquSolver:

```
# taichi-cpu
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b taichi-cpu --method equ -c 6
[Taichi] version 0.9.2, llvm 10.0.0, commit 7a4d73cd, linux, python 3.8.10
[Taichi] Starting on arch=x64
Successfully initialize PIE equ solver with taichi-cpu backend
# of vars: 12559
Iter 5000, abs error [ 5.1899223  6.708023  11.034821 ]
Time elapsed: 0.57s
Successfully write image to result.jpg
```

```
# taichi-gpu
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b taichi-gpu --method equ -z 1024
[Taichi] version 0.9.2, llvm 10.0.0, commit 7a4d73cd, linux, python 3.8.10
[Taichi] Starting on arch=cuda
Successfully initialize PIE equ solver with taichi-gpu backend
# of vars: 12559
Iter 5000, abs error [37.35366  46.433205 76.09506 ]
Time elapsed: 0.60s
Successfully write image to result.jpg
```

For Taichi GridSolver, you also need to specify `--grid-x` and `--grid-y` described in the first section:

```
# taichi-cpu
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b taichi-cpu --method grid --grid-x 16 --grid-y 16 -c 12
[Taichi] version 0.9.2, llvm 10.0.0, commit 7a4d73cd, linux, python 3.8.10
[Taichi] Starting on arch=x64
Successfully initialize PIE grid solver with taichi-cpu backend
# of vars: 17227
```

(continues on next page)

(continued from previous page)

```
Iter 5000, abs error [ 5.310623   6.8661118 11.2751465]
Time elapsed: 0.73s
Successfully write image to result.jpg
```

```
# taichi-gpu
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b taichi-gpu --method grid --grid-x 8 --grid-y 8 -z 64
[Taichi] version 0.9.2, llvm 10.0.0, commit 7a4d73cd, linux, python 3.8.10
[Taichi] Starting on arch=cuda
Successfully initialize PIE grid solver with taichi-gpu backend
# of vars: 17227
Iter 5000, abs error [37.74704  46.853233 74.741455]
Time elapsed: 0.63s
Successfully write image to result.jpg
```

2.6 OpenMP

OpenMP backend needs to specify the number of CPU cores it can use, with `-c` or `--cpu` option (default choice is to use all CPU cores).

There's no other parameters for OpenMP EquSolver:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b openmp --method equ -c 6
Successfully initialize PIE equ solver with openmp backend
# of vars: 12559
Iter 5000, abs error [ 5.2758713  6.768402  11.11969  ]
Time elapsed: 0.06s
Successfully write image to result.jpg
```

For OpenMP GridSolver, you also need to specify `--grid-x` and `--grid-y` described in the first section:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b openmp --method grid --grid-x 8 --grid-y 8 -c 6
Successfully initialize PIE grid solver with openmp backend
# of vars: 17227
Iter 5000, abs error [ 5.187172  6.701462 11.020264]
Time elapsed: 0.10s
Successfully write image to result.jpg
```

2.6.1 Parallelization Strategy

For EquSolver, it first groups the pixels into two folds by `(x + y) % 2`, then parallelizes per-pixel iteration inside a group in each step. This strategy can utilize the thread-local assessment.

For GridSolver, it parallelizes per-grid iteration in each step, where the grid size is `(grid_x, grid_y)`. It simply iterates all pixels in each grid.

2.7 MPI

To run with MPI backend, you need to install both mpicc and mpi4py (`pip install mpi4py`).

Different from other methods, you need to use `mpiexec` or `mpirun` to launch MPI service instead of directly calling `fpie` program. `-np` option is to indicate the number of process it will launch.

Apart from that, you need to specify the synchronization interval for MPI backend with `--mpi-sync-interval`. If this number is too small, it will cause a large amount of overhead of synchronization; however, if it is too large, the quality of solution drops down dramatically.

MPI EquSolver and GridSolver don't have any other arguments because of the parallelization strategy we used, see the next section.

```
$ mpiexec -np 6 fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -
↪h1 130 -w1 130 -n 5000 -g src -b mpi --method equ --mpi-sync-interval 100
Successfully initialize PIE equ solver with mpi backend
# of vars: 12559
Iter 5000, abs error [264.6767  269.55304 368.4869 ]
Time elapsed: 0.10s
Successfully write image to result.jpg
```

```
$ mpiexec -np 6 fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -
↪h1 130 -w1 130 -n 5000 -g src -b mpi --method grid --mpi-sync-interval 100
Successfully initialize PIE grid solver with mpi backend
# of vars: 17227
Iter 5000, abs error [204.41124 215.00548 296.4441 ]
Time elapsed: 0.13s
Successfully write image to result.jpg
```

2.7.1 Parallelization Strategy

MPI cannot use share-memory program model. We need to reduce the amount of data communicated while maintaining the quality of the solution.

Each MPI process is only responsible for a part of computation, and synchronized with other process per `mpi_sync_interval` steps, denoted as S here. When S is too small, the synchronization overhead dominates the computation; when S is too large, each process computes solution independently without global information, therefore the quality of the solution gradually deteriorates.

For EquSolver, it's hard to say which part of the data should be exchanged to other process, since it relabels all pixels at the very beginning of this process. We use `MPI_Bcast` to force sync all data per S iterations.

For GridSolver, we use line partition: process `i` exchanges its first and last line data with process `i-1` and `i+1` separately per S iterations. This strategy has a continuous memory layout to exchange, thus has less overhead comparing with block-level partition.

2.8 CUDA

CUDA EquSolver needs to specify the number of threads in one block it will use, with `-z` or `--block-size` option (default value is 1024):

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b cuda --method equ -z 256
-------------------------------------------------------------
Found 1 CUDA devices
Device 0: NVIDIA GeForce GTX 1060
   SMs:        10
   Global mem: 6078 MB
   CUDA Cap:   6.1
-------------------------------------------------------------
Successfully initialize PIE equ solver with cuda backend
# of vars: 12559
Iter 5000, abs error [37.63664 48.39614 79.6199 ]
Time elapsed: 0.06s
Successfully write image to result.jpg
```

For CUDA GridSolver, you also need to specify `--grid-x` and `--grid-y` described in the first section, instead of `-z`:

```
$ fpie -s test2_src.png -m test2_mask.png -t test2_tgt.png -o result.jpg -h1 130 -w1 130␣
↪-n 5000 -g src -b cuda --method grid --grid-x 4 --grid-y 128
-------------------------------------------------------------
Found 1 CUDA devices
Device 0: NVIDIA GeForce GTX 1060
   SMs:        10
   Global mem: 6078 MB
   CUDA Cap:   6.1
-------------------------------------------------------------
Successfully initialize PIE grid solver with cuda backend
# of vars: 17227
Iter 5000, abs error [37.50096  48.061874 79.06909 ]
Time elapsed: 0.07s
Successfully write image to result.jpg
```

2.8.1 Parallelization Strategy

The strategy used on the CUDA backend is quite similar to OpenMP.

For EquSolver, it performs equation-level parallelization.

For GridSolver, each grid with size `(grid_x, grid_y)` will be in the same block. A thread in a block performs iteration only for a single pixel.

CHAPTER

THREE

BENCHMARK

3.1 Environment configuration

OS: Red Hat Enterprise Linux Workstation 7.9 (Maipo)

CPU: 8x Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz

GPU: GeForce RTX 2080 8G

Python: 3.6.8

Python package version:

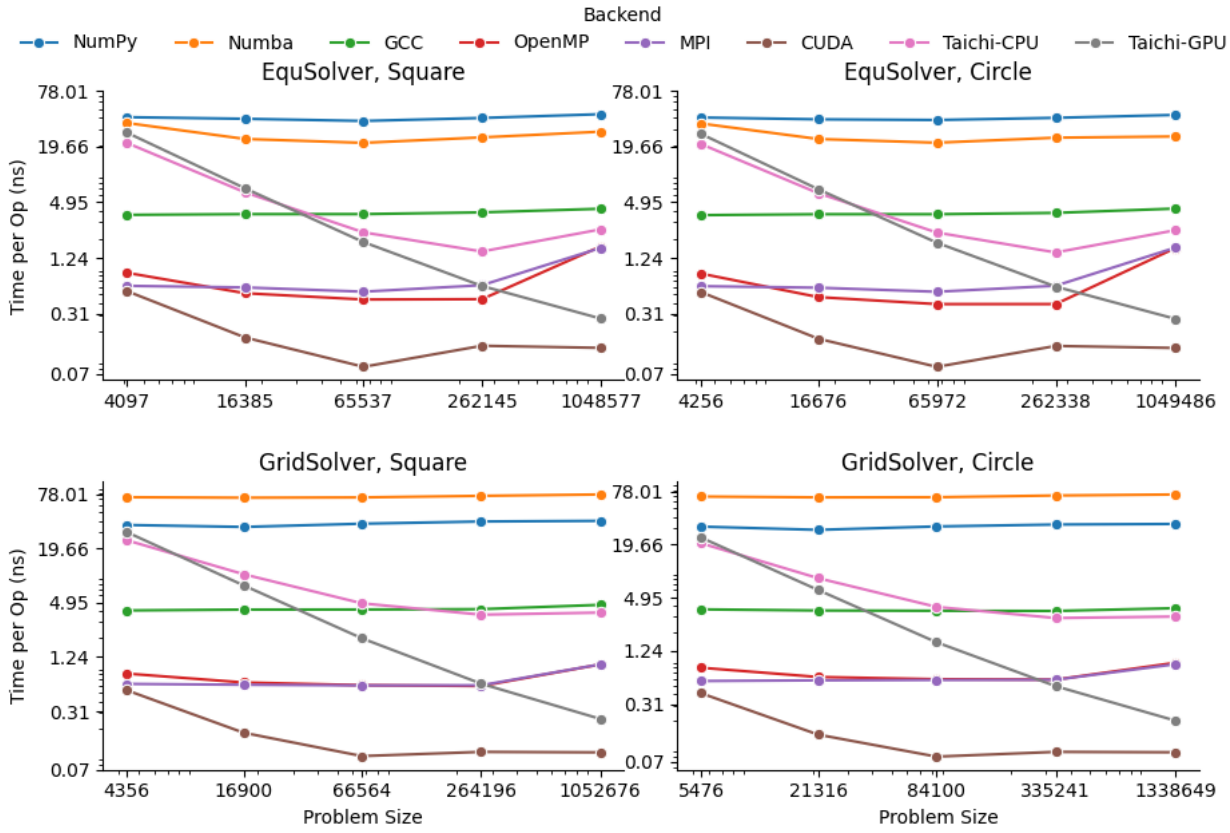
- numpy==1.19.5
- opencv-python==4.5.5.64
- mpi4py==3.1.3
- numba==0.53.1
- taichi==1.0.0

3.2 Problem size vs backend

To run and get the time spend:

```
$ fpie -s $NAME.png -t $NAME.png -m $NAME.png -o result.png -n 5000 -b $BACKEND --method
↪$METHOD ...
```

The following table shows the best performance of corresponding backend choice, i.e., tuning other parameters on square10/circle10 and apply them to other tests, instead of using the default value.



The above plots are generated by per-pixel operation time cost.

3.2.1 EquSolver

The benchmark commands for squareX and circleX:

```
# numpy
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b numpy --
↪method equ
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b numpy --
↪method equ
# numba
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b numba --
↪method equ
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b numba --
↪method equ
# gcc
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b gcc --
↪method equ
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b gcc --
↪method equ
# openmp
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b openmp --
↪method equ -c 8
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b openmp --
```

(continues on next page)

(continued from previous page)

```
↪method equ -c 8
# mpi
mpiexec -np 8 fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000␣
↪-b mpi --method equ --mpi-sync-interval 100
mpiexec -np 8 fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000␣
↪-b mpi --method equ --mpi-sync-interval 100
# cuda
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b cuda --
↪method equ -z 256
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b cuda --
↪method equ -z 256
# taichi-cpu
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b taichi-cpu␣
↪--method equ -c 8
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b taichi-cpu␣
↪--method equ -c 8
# taichi-gpu
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b taichi-gpu␣
↪--method equ -z 1024
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b taichi-gpu␣
↪--method equ -z 1024
```

EquSolver	square6	square7	square8	square9	square10
# of vars	4097	16385	65537	262145	1048577
NumPy	0.8367s	3.2142s	12.1836s	52.4939s	230.5375s
Numba	0.7257s	1.9472s	7.0761s	32.4084s	149.3390s
GCC	0.0740s	0.3013s	1.2061s	5.0351s	22.0276s
OpenMP	0.0176s	0.0423s	0.1447s	0.5835s	8.6203s
MPI	0.0127s	0.0488s	0.1757s	0.8253s	8.3310s
CUDA	0.0112s	0.0141s	0.0272s	0.1835s	0.6967s
Taichi-CPU	0.4437s	0.5178s	0.7667s	1.9061s	13.2009s
Taichi-GPU	0.5730s	0.5727s	0.6022s	0.8101s	1.4430s

EquSolver	circle6	circle7	circle8	circle9	circle10
# of vars	4256	16676	65972	262338	1049486
NumPy	0.8618s	3.2280s	12.5615s	52.7161s	226.5578s
Numba	0.7430s	1.9789s	7.1499s	32.1932s	132.7537s
GCC	0.0764s	0.3062s	1.2115s	4.9785s	22.1516s
OpenMP	0.0179s	0.0391s	0.1301s	0.5177s	8.2778s
MPI	0.0131s	0.0494s	0.1767s	0.8155s	8.3823s
CUDA	0.0113s	0.0139s	0.0274s	0.1831s	0.6966s
Taichi-CPU	0.4461s	0.5148s	0.7687s	1.8646s	12.9343s
Taichi-GPU	0.5735s	0.5679s	0.5971s	0.7987s	1.4379s

3.2.2 GridSolver

The benchmark commands for squareX and circleX:

```
# numpy
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b numpy --
↪method grid
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b numpy --
↪method grid
# numba
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b numba --
↪method grid
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b numba --
↪method grid
# gcc
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b gcc --
↪method grid --grid-x 8 --grid-y 8
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b gcc --
↪method grid --grid-x 8 --grid-y 8
# openmp
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b openmp --
↪method grid -c 8 --grid-x 2 --grid-y 16
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b openmp --
↪method grid -c 8 --grid-x 2 --grid-y 16
# mpi
mpiexec -np 8 fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000␣
↪-b mpi --method grid --mpi-sync-interval 100
mpiexec -np 8 fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000␣
↪-b mpi --method grid --mpi-sync-interval 100
# cuda
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b cuda --
↪method grid --grid-x 2 --grid-y 128
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b cuda --
↪method grid --grid-x 2 --grid-y 128
# taichi-cpu
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b taichi-cpu␣
↪--method grid -c 8 --grid-x 8 --grid-y 128
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b taichi-cpu␣
↪--method grid -c 8 --grid-x 8 --grid-y 128
# taichi-gpu
fpie -s square10.png -t square10.png -m square10.png -o result.png -n 5000 -b taichi-gpu␣
↪--method grid -z 1024 --grid-x 16 --grid-y 64
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b taichi-gpu␣
↪--method grid -z 1024 --grid-x 16 --grid-y 64
```

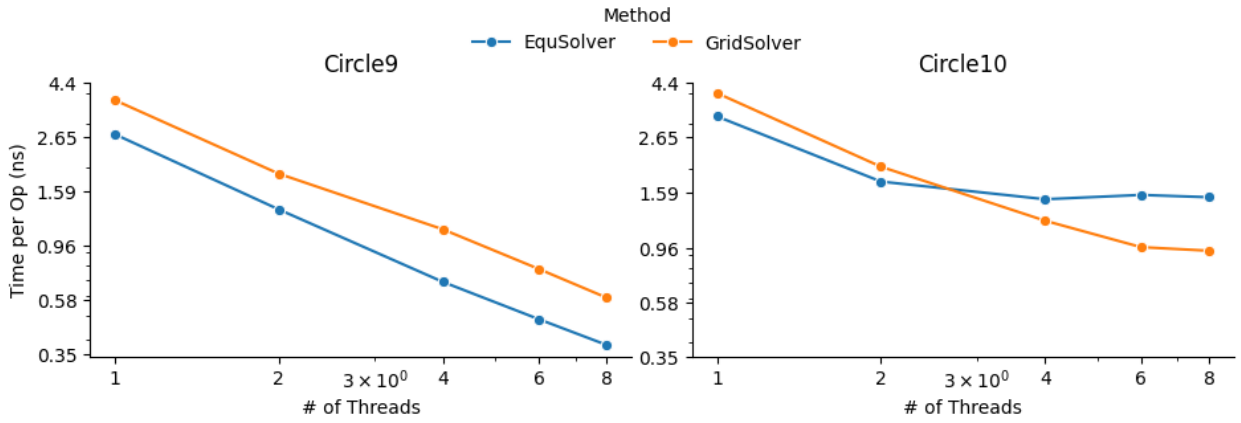
GridSolver	square6	square7	square8	square9	square10
# of vars	4356	16900	66564	264196	1052676
NumPy	0.7809s	2.8823s	12.3242s	51.7496s	209.5504s
Numba	1.5838s	6.0720s	24.0901s	99.5048s	410.6119s
GCC	0.0884s	0.3504s	1.3832s	5.5402s	24.6482s
OpenMP	0.0177s	0.0547s	0.2011s	0.7805s	5.4012s
MPI	0.0136s	0.0516s	0.1999s	0.7956s	5.4109s
CUDA	0.0116s	0.0152s	0.0330s	0.1458s	0.5738s
Taichi-CPU	0.5308s	0.8638s	1.6196s	4.8147s	20.2245s
Taichi-GPU	0.6538s	0.6505s	0.6638s	0.8298s	1.3439s

GridSolver	circle6	circle7	circle8	circle9	circle10
# of vars	5476	21316	84100	335241	1338649
NumPy	0.8554s	3.0602s	13.1915s	55.3018s	224.0399s
Numba	1.8680s	7.1174s	28.1826s	117.5155s	481.5718s
GCC	0.0997s	0.3768s	1.4753s	5.8558s	25.1236s
OpenMP	0.0219s	0.0670s	0.2498s	0.9838s	6.0868s
MPI	0.0155s	0.0614s	0.2446s	0.9810s	5.8527s
CUDA	0.0113s	0.0150s	0.0334s	0.1507s	0.5954s
Taichi-CPU	0.5558s	0.8727s	1.6317s	4.8740s	20.2178s
Taichi-GPU	0.6447s	0.6418s	0.6521s	0.8309s	1.3578s

3.3 Per backend performance

In this section, we will perform ablation studies with OpenMP/MPI/CUDA backend. We use circle9/10 with 5000 iterations as the experiment setting.

3.3.1 OpenMP



Command to run:

```
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b openmp --
↪method equ -c 8
```

(continues on next page)

(continued from previous page)

```
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b openmp --
↪method grid -c 8 --grid-x 2 --grid-y 16
```

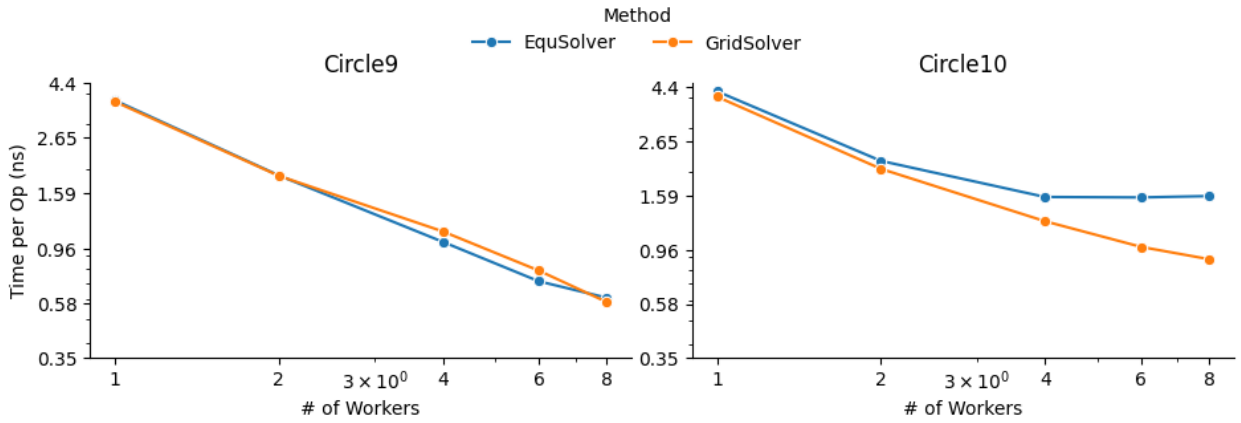
circle9	1	2	4	6	8
# of vars	262338	262338	262338	262338	262338
EquSolver	3.5689s	1.7679s	0.8987s	0.6344s	0.4982s

circle9	1	2	4	6	8
# of vars	335241	335241	335241	335241	335241
GridSolver	6.2717s	3.1530s	1.8758s	1.2955s	0.9897s

circle10	1	2	4	6	8
# of vars	1049486	1049486	1049486	1049486	1049486
EquSolver	16.9218s	9.2764s	7.8828s	8.2016s	8.0285s

circle10	1	2	4	6	8
# of vars	1338649	1338649	1338649	1338649	1338649
GridSolver	26.7571s	13.5669s	8.2486s	6.4654s	6.2539s

3.3.2 MPI



Command to run:

```
mpiexec -np 8 fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000␣
↪-b mpi --method equ --mpi-sync-interval 100
mpiexec -np 8 fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000␣
↪-b mpi --method grid --mpi-sync-interval 100
```

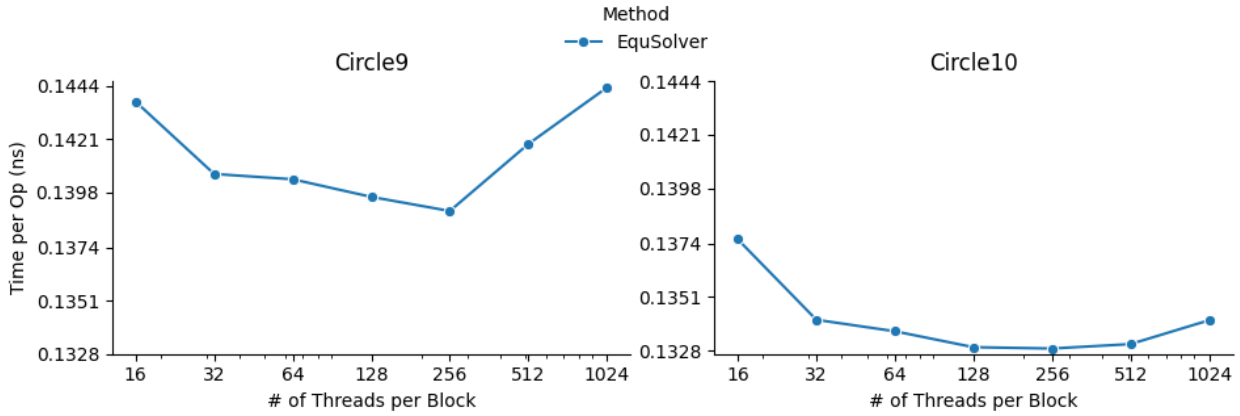
circle9	1	2	4	6	8
# of vars	262338	262338	262338	262338	262338
EquSolver	4.9217s	2.4655s	1.3378s	0.9310s	0.7996s

circle9	1	2	4	6	8
# of vars	335241	335241	335241	335241	335241
GridSolver	6.2136s	3.1381s	1.8817s	1.3124s	0.9822s

circle10	1	2	4	6	8
# of vars	1049486	1049486	1049486	1049486	1049486
EquSolver	22.1275s	11.5566s	8.2541s	8.2208s	8.3238s

circle10	1	2	4	6	8
# of vars	1338649	1338649	1338649	1338649	1338649
GridSolver	26.8360s	13.6866s	8.3945s	6.6107s	5.8929s

3.3.3 CUDA



Command to run:

```
fpie -s circle10.png -t circle10.png -m circle10.png -o result.png -n 5000 -b cuda --
↪method equ -z 1024
```

circle9	16	32	64	128	256	512	1024
# of vars	262338	262338	262338	262338	262338	262338	262338
EquSolver	0.1885s	0.1844s	0.1841s	0.1831s	0.1823s	0.1861s	0.1893s

circle10	16	32	64	128	256	512	1024
# of vars	1049486	1049486	1049486	1049486	1049486	1049486	1049486
EquSolver	0.7220s	0.7038s	0.7012s	0.6976s	0.6973s	0.6983s	0.7037s

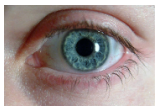
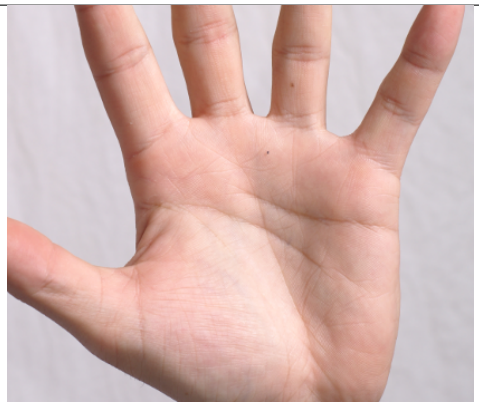
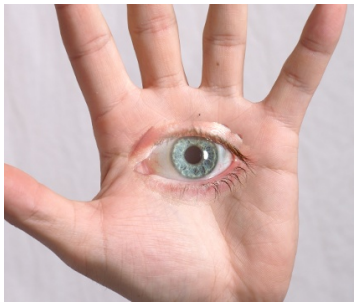
CHAPTER

FOUR

FINAL REPORT

4.1 Summary

We have implemented a parallelized Poisson image editor with Jacobi method. It can compute results using seven extensions: NumPy, Numba, Taichi, single-threaded c++, OpenMP, MPI, and CUDA. In terms of performance, we have a detailed benchmarking result where the CUDA backend can achieve 31 to 42 times speedup on GHC machines compared to the single-threaded c++ implementation. In terms of user-experience, we have a simple GUI to demonstrate the results interactively, released a standard PyPI package, and provide a website for project documentation.

Source image	Mask image	Target image	Result image
			

4.2 Background

4.2.1 Poisson Image Editing

Poisson Image Editing is a technique that can fuse two images together without producing artifacts. Given a source image and its corresponding mask, as well as a coordination on the target image, the algorithm always yields amazing result. The general idea is to keep most of gradient in source image unchanged, while matching the boundary of source image and target image pixels.

The gradient per pixel is computed by

$$\nabla(x, y) = 4I(x, y) - I(x-1, y) - I(x, y-1) - I(x+1, y) - I(x, y+1)$$

After calculating the gradient in source image, the algorithm tries to solve the following problem: given the gradient and the boundary value, calculate the approximate solution that meets the requirement, i.e., to keep target image's gradient as similar as the source image.

This process can be formulated as $(4-A)\vec{x}=\vec{b}$, where $A \in \mathbb{R}^{N\times N}$, $\vec{x} \in \mathbb{R}^N$, $\vec{b} \in \mathbb{R}^N$, N is the number of pixels in the mask, A is a giant sparse matrix because each line of A only contains at most 4 non-zero value (neighborhood), $\vec{b}$ is the gradient from source image, and $\vec{x}$ is the result value.

N is always a large number, i.e., greater than 50k, so the Gauss-Jordan Elimination cannot be directly applied here because of the high time complexity $O(N^3)$. People use Jacobi Method to solve the problem. Thanks to the sparsity of matrix A, the overall time complexity is $O(MN)$ where M is the number of iteration performed by Poisson image editing. The iterative equation is $\vec{x}' \leftarrow (A\vec{x}+\vec{b})/4$.

This project parallelizes Jacobi method to speed up the computation. To our best knowledge, there's no public project on GitHub that implements Poisson image editing with either OpenMP, or MPI, or CUDA. All of them can only handle a small size image workload (link).

4.2.2 PIE Solver

We implemented two different solvers: EquSolver and GridSolver.

EquSolver directly constructs the equations $(4-A)\vec{x}=\vec{b}$ by relabeling the pixel, and use Jacobi method to get the solution via $\vec{x}' \leftarrow (A\vec{x}+\vec{b})/4$.

```
""" EquSolver pseudocode."""

# pre-process
src, mask, tgt = read_images(src_name, mask_name, tgt_name)
A = build_A(mask)            # shape: (N, 4), dtype: int
X = build_X(tgt, mask)       # shape: (N, 3), dtype: float
B = build_B(src, mask, tgt)  # shape: (N, 3), dtype: float

# major computation, can be parallelized
for _ in range(n_iter):
    X = (X[A[:, 0]] + X[A[:, 1]] + X[A[:, 2]] + X[A[:, 3]] + B) / 4.0

# post-process
out = merge(tgt, X, mask)
write_image(out_name, out)
```

GridSolver uses the same Jacobi iteration, however, it keeps the 2D structure of the original image instead of relabeling all pixels in the mask. It may have some advantages when the mask region covers the whole image, because in this case GridSolver can save 4 read instructions by calculating the coordinates of the neighborhood directly. Also, if we set the access pattern appropriately (which will be discussed in Section *Access Pattern*), it has better locality in getting the required data in each iteration.

```
""" GridSolver pseudocode."""

# pre-process
src, mask, tgt = read_images(src_name, mask_name, tgt_name)
# mask: shape: (N, M), dtype: uint8
grad = calc_grad(src, mask, tgt)  # shape: (N, M, 3), dtype: float
x, y = np.nonzero(mask)  # find element-wise pixel index of mask array
```

(continues on next page)

(continued from previous page)

```
# major computation, can be parallelized
for _ in range(n_iter):
    tgt[x, y] = (tgt[x - 1, y] + tgt[x, y - 1] + tgt[x, y + 1] + tgt[x + 1, y] + grad[x,␣
↪y]) / 4.0

# post-process
write_image(out_name, tgt)
```

The bottleneck for both solvers is the for-loop and can be easily parallelized. The implementation detail and parallelization strategies will be discussed in Section *Parallelization Strategy*.

4.3 Method

4.3.1 Language and Hardware Setup

We start building PIE with the help of pybind11 as we aim to benchmark multiple parallelization methods, including hand-written CUDA code and other 3rd-party libraries such as NumPy.

One of our project goal is to let the algorithm run on any *nix machine and can have a real-time interactive result demonstration. For this reason, we didn't choose a supercomputing cluster as our hardware setup. Instead, we choose GHC machine to develop and measure the performance, which has 8x i7-9700 CPU cores and an Nvidia RTX 2080Ti.

4.3.2 Access Pattern

For EquSolver, we can reorganize the order of pixels to obtain better locality when performing parallel operations. Specifically, we can divide all pixels into two folds by `(x + y) % 2`. Here is a small example:

```
# before
x1   x2   x3   x4   x5
x6   x7   x8   x9   x10
x11  x12  x13  x14  x15
...

# reorder
x1   x10  x2   x11  x3
x12  x4   x13  x5   x14
x6   x15  x7   x16  x8
...
```

This results in a tighter relationship between the 4 neighbors of each pixel. The ideal access pattern is to iterate over these two groups separately, i.e.

```
for _ in range(n_iter):
    parallel for i in range(1, p):
        # i < p, neighbor >= p
        x_[i] = calc(b[i], neighbor(x, i))

    parallel for i in range(p, N):
        # i >= p, neighbor < p
        x[i] = calc(b[i], neighbor(x_, i))
```

Unfortunately, we only observe a clear advantage with OpenMP EquSolver. For other backend, the sequential ID assignment is much better than reordering. See the section *Parallelization Strategy - OpenMP* for a related discussion.

For GridSolver, since it retains most of the 2D structure of the image, we can use block-level access pattern instead of sequential access pattern to improve cache hit rate. Here is a Python pseudocode to show how it works:

```
N, M = tgt.shape[:2]
# here is a sequential scan:
parallel for i in range(N):
    parallel for j in range(M):
        if mask[i, j]:
            tgt_[i, j] = calc(grad[i, j], neighbor(tgt, i, j))
# however, we can use block-level access pattern to improve the cache hit rate:
parallel for i in range(N // grid_x):
    parallel for j in range(M // grid_y):
        # the grid size is (grid_x, grid_y)
        for x in range(i * grid_x, (i + 1) * grid_x):
            for y in range(j * grid_y, (j + 1) * grid_y):
                if mask[x, y]:
                    tgt_[x, y] = calc(grad[x, y], neighbor(tgt, x, y))
```

4.3.3 Synchronization vs Converge Speed

Since Jacobi Method is an iterative method for solving matrix equations, there is a trade-off between the quality of solution and the frequency of synchronization.

Share Memory Programming Model

The naive approach is to create another matrix to store the solution. Once all pixels are computed, the algorithm refreshes the original array with the new values:

```
for _ in range(n_iter):
    tmp = np.zeros_like(x)
    parallel for i in range(1, N):
        tmp[i] = calc(b[i], neighbor(x, i))
    x = tmp
```

It's quite similar to the "gradient decent" approach in machine learning where only one step of optimization is performed using all data samples each iteration. Interestingly, "stochastic gradient decent"-style Jacobi Method works quite well:

```
for _ in range(n_iter):
    parallel for i in range(1, N):
        x[i] = calc(b[i], neighbor(x, i))
```

It's because Jacobi Method guarantees its convergence, and w/o such a barrier, the error per pixel will always become smaller. Comparing with the original approach, it also has a faster converge speed.

Non-shared Memory Programming Model

The above approach works for shared memory programming models such as OpenMP and CUDA. However, for non-shared memory programming models such as MPI, the above approach cannot work well. The solution will be discussed in Section Parallelization Strategy - MPI.

4.3.4 Parallelization Strategy

This section will cover the implementation detail with three different backend (OpenMP/MPI/CUDA) and two different solvers (EquSolver/GridSolver).

OpenMP

As mentioned before, OpenMP EquSolver first divides the pixels into two folds by `(x + y) % 2`, then parallelizes per-pixel iteration inside a group in each step.

This strategy can utilize the thread-local assessment as the position of four neighborhood become closer. However, it requires the entire array to be processed twice because of the division. In some cases, such as CUDA, this approach introduces an overhead that exceeds the original computational cost. However, in OpenMP, it has a significant runtime improvement.

OpenMP GridSolver assigns equal amount of blocks to each thread, with size `(grid_x, grid_y)` per block. Each thread simply iterates all pixels in each block independently.

We use static assignment for both solvers to minimize the overhead of runtime task allocation, since the workload is uniform per pixel/grid.

MPI

MPI cannot use the shared memory program model. We need to reduce the amount of data communicated, while maintaining the quality of the solution.

Each MPI process is responsible for only a portion of the computation and synchronizes with other process per `mpi_sync_interval` steps, denoted as S in this section. When S is too small, the synchronization overhead dominates the computation; when S is too large, each process computes the solution independently without global information, therefore the quality of the solution gradually deteriorates.

For MPI EquSolver, it's hard to say which part of the data should be exchanged to other process, since it relabels all pixels at pre-process stage. We assign an equal number of equations to each process and use `MPI_Bcast` to force synchronization of all data per S iteration.

MPI GridSolver uses line partition: process `i` exchanges its first and last line data with process `i-1` and `i+1` separately per S iterations. This strategy has a continuous memory layout, thus has less overhead compared to block-level partition.

The workload per pixel is small and fixed. In fact, this type of workload is not suitable for MPI.

CUDA

The strategy used on the CUDA backend is quite similar to OpenMP.

CUDA EquSolver performs equation-level parallelization. It has sequential labels per pixel instead of dividing into two folds as OpenMP does. Each block is assigned with an equal number of equations to execute Jacobi Method independently. The threads in a block iterate over only a single equation. We also tested the shared memory kernel, but it's much slower than non-shared memory version.

For GridSolver, each grid with size `(grid_x, grid_y)` will be in the same block. The threads in a block iterates over a single pixel only.

There are no barriers during the iteration of both solvers. The reason has been discussed in Section *Share Memory Programming Model*.

4.4 Experiments

4.4.1 Experiment Setting

Hardware and Software

We use GHC83 to run all of the following experiments. Here is the hardware and software configuration:

- OS: Red Hat Enterprise Linux Workstation 7.9 (Maipo)
- CPU: 8x Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
- GPU: GeForce RTX 2080 8G
- Python: 3.6.8
- Python package version:
  - numpy==1.19.5
  - opencv-python==4.5.5.64
  - mpi4py==3.1.3
  - numba==0.53.1
  - taichi==1.0.0

Data

We generate 10 images for benchmarking performance, 5 square and 5 circle. The script is tests/data.py. You can find the detail information in this table:

ID	Size	# pixels	# unmasked pixels	Image
square6	66x66	4356	4356	
square7	130x130	16900	16900	
square8	258x258	66564	66564	
square9	514x514	264196	264196	

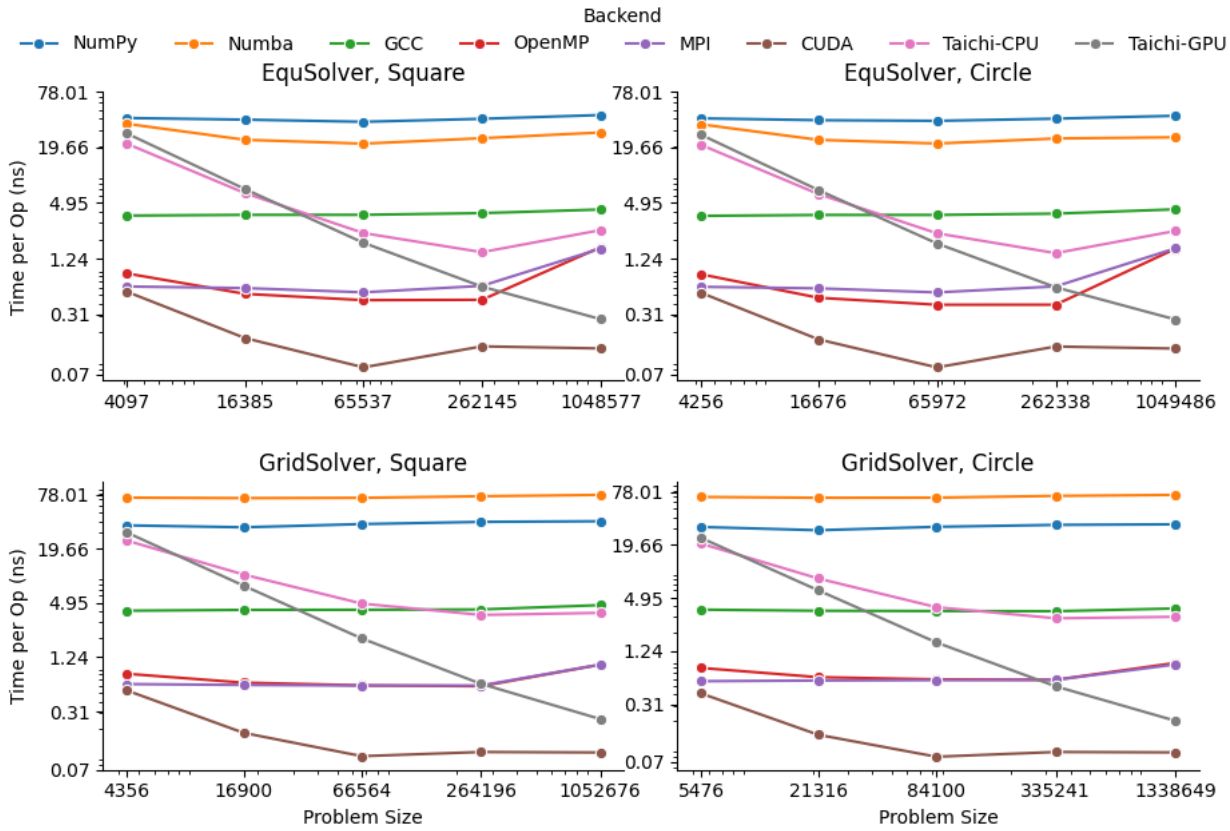
We try to keep the number of unmasked pixels of circleX and squareX to be the same level. For EquSolver there's no difference, but for GridSolver it cannot be ignored, since it needs to process all pixels no matter it is masked.

Metric

We measure the performance by "Time per Operation" (TpO for short) and "Cache Miss per Operation" (CMpO for short). TpO is derived by `total time / total number of iteration / number of pixel`. The smaller the TpO, the more efficient the parallel algorithm will be. CMpO is derived by `total cache miss / total number of iteration / number of pixel`.

4.4.2 Result and Analysis

We use all seven backend to run benchmark experiments. `GCC` (single-threaded C++ implementation) is the baseline. Details of the following experiment (commands and tables) can be found on Benchmark page. For the sake of simplicity, we only demonstrate the plot in the following sections. Most plots are in logarithmic scale.



EquSolver vs GridSolver

If GridSolver's parameters `grid_x` and `grid_y` are carefully tuned, in most cases it can perform better than EquSolver in a handwritten backend configuration (OpenMP/MPI/CUDA). The analysis will be performed in the following sections. However, it is difficult to say which one is better using other third-party backends. This may be due to the internal design of these libraries.

Analysis for 3rd-party Backend

NumPy

NumPy is 10~11x slower than GCC with EquSolver, and 8~9x slower than GCC with GridSolver. This result shows that the overhead of the NumPy solver is non-negligible. Each iteration requires repeated data transfers between C and Python and the creation of some temporary arrays to compute the results. Even if we have used vector operations in all the computations, it cannot take advantage of the memory layout.

Numba

Numba is a just-in-time compiler for numerical functions in Python. For EquSolver, Numba is about twice as fast as NumPy; however, for GridSolver, Numba is about twice as slow as NumPy. This result suggests that Numba does not provide a general speedup for any NumPy operations, not to mention that it is still slower than GCC.

Taichi

Taichi is an open-source, imperative, parallel programming language for high-performance numerical computation. If we use Taichi with small size input images, it does not get much benefit. However, when increasing the problem size to a very large scale, the advantage becomes much clear. We think it is because the pre-processing step in Taichi is a non-negligible overhead.

On the CPU backend, EquSolver is faster than GCC, while GridSolver performs almost as well as GCC. This shows the access pattern largely affects the actual performance.

On the GPU backend, although the TpO is twice as slow as CUDA with extremely large-scale input, it is still faster than other backends. We are quite interested in other 3rd-party GPU solution's performance, and leave it as future work.

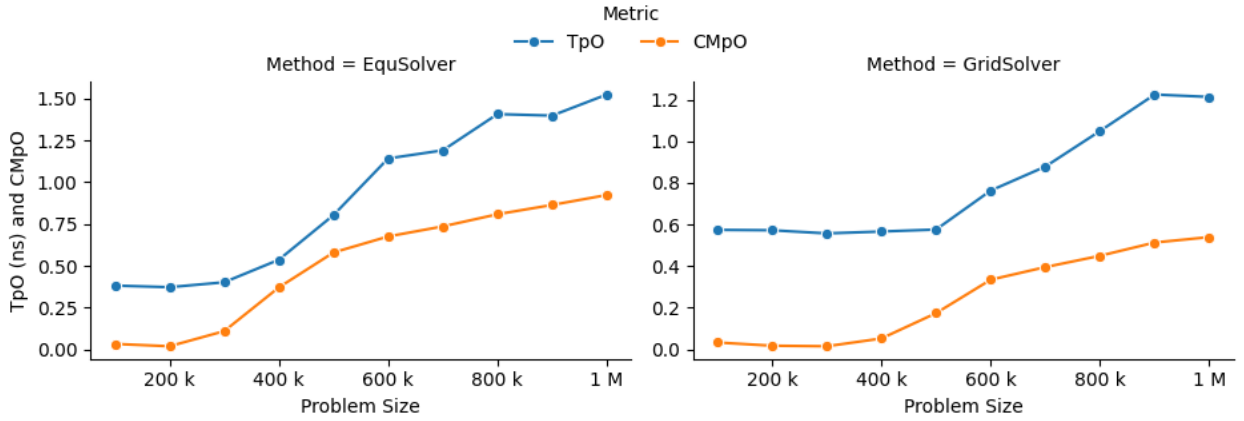
Analysis for Non 3rd-party Backend

OpenMP and MPI can achieve almost the same speed, but MPI's converge speed is slower because of the synchronization trade-off. CUDA is the fastest in all conditions.

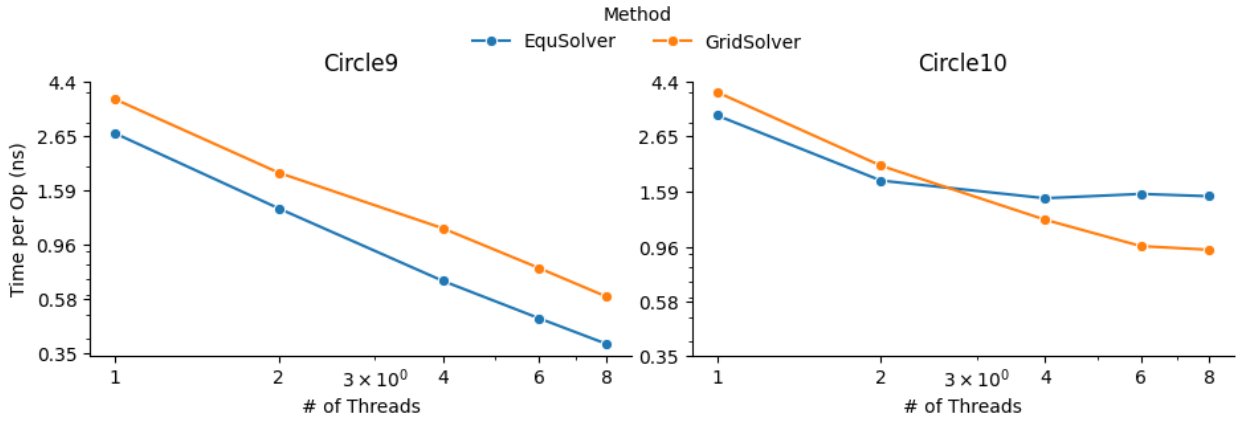
OpenMP

EquSolver is 8~9x faster than GCC; GridSolver is 6~7x faster than GCC. However, there is a huge performance drop when the problem size is greater than 1M for both two solvers. The threshold is 300k ~ 400k for EquSolver and 500k ~ 600k for GridSolver. We suspect that is because of cache-miss, confirmed by the following numerical result:

OpenMP	# of pixels	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
Equ-Solver	Time (s)	0.1912	0.3728	0.6033	1.073	2.0081	3.4242	4.1646	5.6254	6.2875	7.6159
Equ-Solver	TpO (ns)	0.3824	0.3728	0.4022	0.5365	0.8032	1.1414	1.1899	1.4063	1.3972	1.5232
Equ-Solver	CMpO	0.0341	0.0201	0.1104	0.3713	0.5799	0.6757	0.7356	0.8083	0.8639	0.9232
Grid-Solver	Time (s)	0.2870	0.5722	0.8356	1.1321	1.4391	2.2886	3.0738	4.1967	5.5097	6.0635
Grid-Solver	TpO (ns)	0.5740	0.5722	0.5571	0.5661	0.5756	0.7629	0.8782	1.0492	1.2244	1.2127
Grid-Solver	CMpO	0.0330	0.0174	0.0148	0.0522	0.1739	0.3346	0.3952	0.4495	0.5132	0.5394



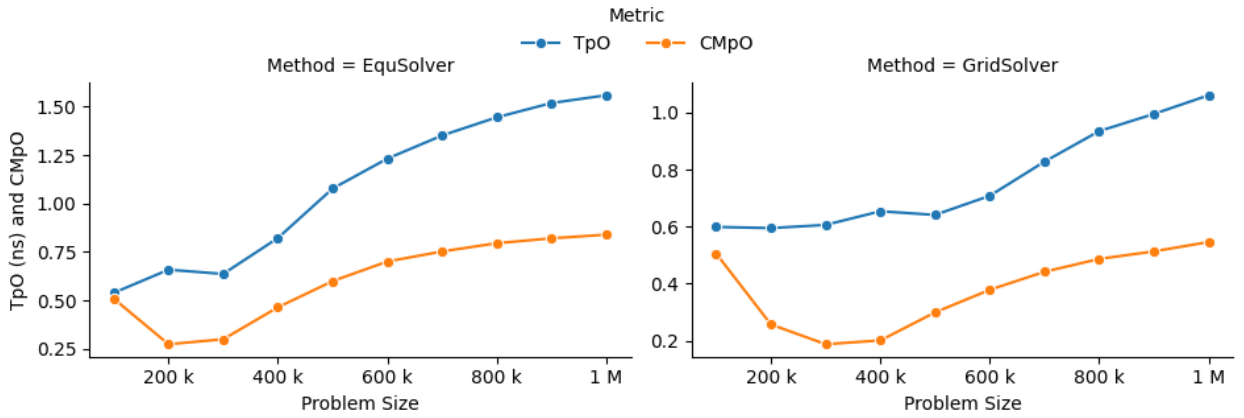
We also investigated the impact of the number of threads on the performance of the OpenMP backend. There is a linear speedup when the aforementioned cache-miss problem does not occur; when the cache-miss problem is encountered, its performance rapidly saturates, especially for EquSolver. We believe the reason behind is that GridSolver can take better advantage of locality compared to EquSolver, since it has no relabeling pixel process and keep all of the 2D information.



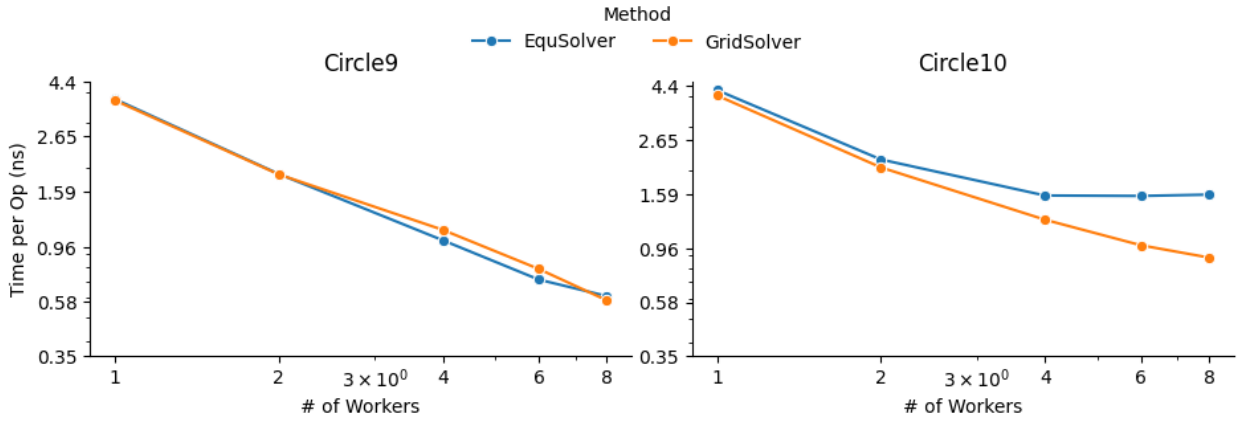
MPI

EquSolver and GridSolver is 6~7x faster than GCC. Like OpenMP, there is a huge performance drop. The threshold is 300k ~ 400k for EquSolver and 400k ~ 500k for GridSolver. Fortunately, the following table and plot confirms our assumption of cache-miss:

MPI	# of pixels	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
Equ-Solver	Time (s)	0.2696	0.6584	0.9549	1.6435	2.6920	3.6933	4.7265	5.7762	6.8305	7.7894
Equ-Solver	TpO (ns)	0.5392	0.6584	0.6366	0.8218	1.0768	1.2311	1.3504	1.4441	1.5179	1.5579
Equ-Solver	CMpO	0.5090	0.2743	0.2998	0.4646	0.5995	0.7006	0.7525	0.7951	0.8204	0.8391
Grid-Solver	Time (s)	0.2994	0.5948	0.9088	1.3075	1.6024	2.1239	2.8969	3.7388	4.4776	5.3026
Grid-Solver	TpO (ns)	0.5988	0.5948	0.6059	0.6538	0.6410	0.7080	0.8277	0.9347	0.9950	1.0605
Grid-Solver	CMpO	0.5054	0.2570	0.1876	0.2008	0.2991	0.3783	0.4415	0.4866	0.5131	0.5459



A similar phenomenon occurs on the MPI backend when the number of processes changes:

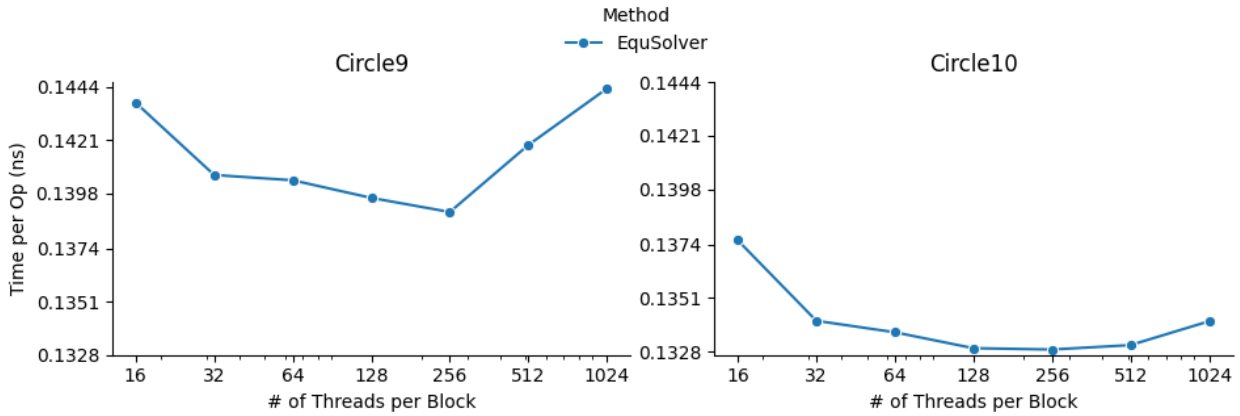


CUDA

EquSolver is 27~44x faster than GCC; GridSolver is 38~42x faster than GCC. The performance is consistent across different input sizes.

We investigated the impact of different block size on CUDA EquSolver. For a better demonstration, we didn't use GridSolver because it requires tuning two parameters `grid_x` and `grid_y`. By increasing the block size, the performance improves first, reaches a peak, and finally drops. The best configuration is block size = 256.

When the block size is too small, it will use more grids for computation and therefore the overhead of communication across grids will increase. When the block size is too large, the cache invalidation problem dominates, even though fewer grids are used – since we are not using shared memory in this CUDA kernel and there are no barriers to calling this kernel, we suspect that the program will often read values that cannot be cached and will also often write values to invalidate the cache.



4.5 Contribution

Each group member's contributions are on GitHub.

4.6 REFERENCE

[1] Pérez, Patrick, Michel Gangnet, and Andrew Blake. "Poisson image editing." *ACM SIGGRAPH 2003 Papers*. 2003. 313-318.

[2] Jacobi Method, https://en.wikipedia.org/wiki/Jacobi_method

[3] Harris, Charles R., et al. "Array programming with NumPy." *Nature* 585.7825 (2020): 357-362.

[4] Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler." *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015.

[5] Hu, Yuanming, et al. "Taichi: a language for high-performance computation on spatially sparse data structures." *ACM Transactions on Graphics (TOG)* 38.6 (2019): 1-16.