Fast and Accurate Image segmentation

for medical imaging data

Team 17
Xincheng Tan, Yaxin Lei, Hanwen Zhang, Scarlett Gong

00

Objective

Parallel Fast and Accurate
Image Segmentation on Cell Nucleus

Image Segmentation of Cell Nuclei

Data source: 2018 Data Science Bowl challenge

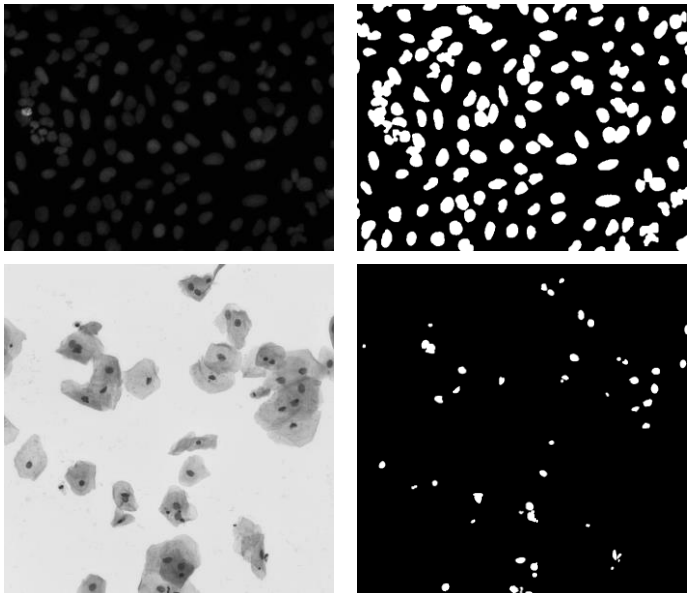
670+ **Training set**

Images of different cell types and tissues

~65 **Test set**

Without ground truth masks

Original image **Ground Truth Masks**



Application Type

An offline, compute intensive toolbox for nuclei segmentation which

- Provides **three** segmentation models:
  - **Otsu's Method** (threshold method implemented in C++)
  - **Boruvka's Algorithm** (tree-based method implemented in C++)
  - **UNet** (Keras + Tensorflow deep learning in Python)
- Allows fast nuclei segmentation for new images
- **Big Compute + Big Data** (more on the former)
- **Cloud infrastructure:** Google Cloud (VM and cluster)

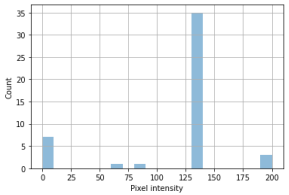
01

Otsu's method

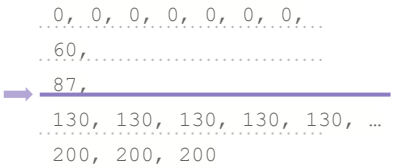
Otsu's Method Algorithm overview

Read image

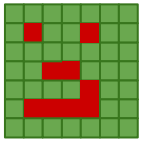
Pixel intensity hist.



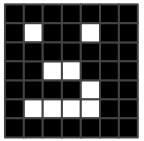
Find threshold



Final segmentation



Write result



Step 0:
Read in image

Step 1:
Build the histogram of pixel intensity.

O(N)

Step 2:
Exhaustive search of threshold to minimizes inter-class variance.
*O(256*256) ~ O(1)*

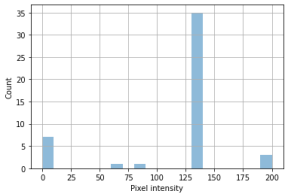
Step 3:
Final image segmentation based on the threshold.
O(N) complexity

Step 4:
Write out image result

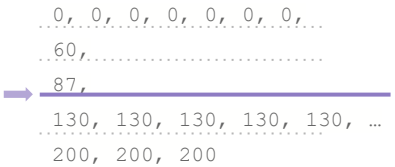
Otsu's Method Algorithm overview

Read image

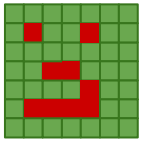
Pixel intensity hist.



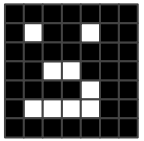
Find threshold



Final segmentation



Write result



Step 0:
Read in image

Step 1:
Build the histogram of pixel intensity.

O(N)

Step 2:
Exhaustive search of threshold to minimizes inter-class variance.
*O(256*256) ~ O(1)*

Step 3:
Final image segmentation based on the threshold.
O(N) complexity

Step 4:
Write out image result

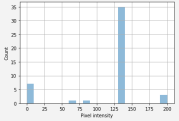
- **Language:** C++ implemented from scratch
- **Optimizations and programming model:**
  - OpenMP (Algorithm parallelism)
  - OpenMP (Data parallelism)
  - MPI+OpenMP+Slurm (Hybrid)
- **Platform:** Google Cloud (Virtual Instance, Cluster with Slurm)

Optimization 1: OpenMP - Algorithm parallelism

```
For cell_img in all images
{
```

Step 0: Input image

Step 1: Pixel intensity histogram



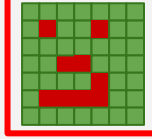
Parallelism using OpenMP

Step 2: Find threshold

```
0, 0, 0, 0, 0, 0, 0,
60,
87,
130, 130, 130, 130, 130, 130…
200, 200, 200
```

Parallelism using OpenMP

Step 3: Final segmentation



Parallelism using OpenMP

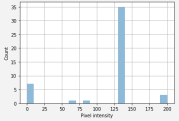
```
}
```

Optimization 1: OpenMP - Algorithm parallelism

```
For cell_img in all images
{
```

Step 0: Input image

Step 1: Pixel intensity histogram



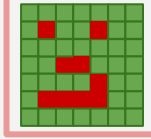
Parallelism using OpenMP

Step 2: Find threshold

```
0, 0, 0, 0, 0, 0, 0,
60,
87,
130, 130, 130, 130, 130, 130…
200, 200, 200
```

Parallelism using OpenMP

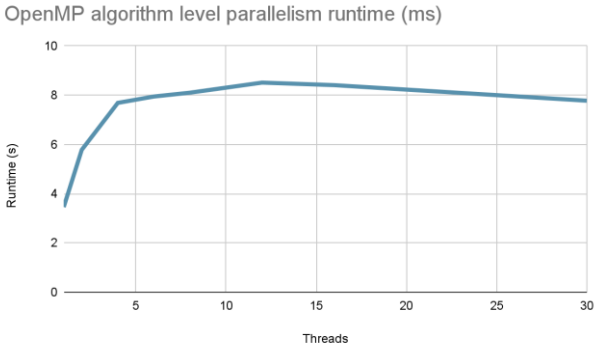
Step 3: Final segmentation



Parallelism using OpenMP

```
}
```

Threads	Runtime (s)	Speedup compared to 1 thread
1	3.456	1
2	5.779	0.59799
4	7.687	0.4496
6	7.945	0.4359
8	8.101	0.4266
12	8.510	0.40614
16	8.407	0.4111
30	7.775	0.4445



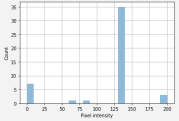
Google Cloud: Compute optimized VM with 30 vCPUs.

Optimization 1: OpenMP - Algorithm parallelism

```
For cell_img in all images
{
```

Step 0: Input image

Step 1: Pixel intensity histogram



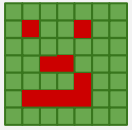
Parallelism using OpenMP

Step 2: Find threshold

```
0, 0, 0, 0, 0, 0, 0,
60,
87,
130, 130, 130, 130, 130, 130…
200, 200, 200
```

Parallelism using OpenMP

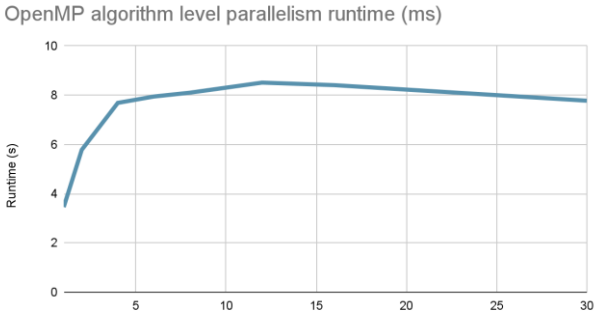
Step 3: Final segmentation

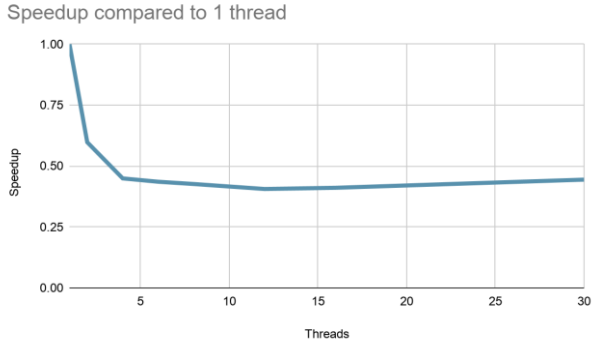


Parallelism using OpenMP

```
}
```

Threads	Runtime (s)	Speedup compared to 1 thread
1	3.456	1
2	5.779	0.59799
4	7.687	0.4496
6	7.945	0.4359
8	8.101	0.4266
12	8.510	0.40614
16	8.407	0.4111
30	7.775	0.4445





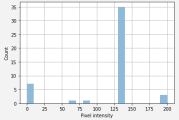
Google Cloud: Compute optimized VM with 30 vCPUs.

Optimization 1: OpenMP - Algorithm parallelism

```
For cell_img in all images
{
```

Step 0: Input image

Step 1: Pixel intensity histogram



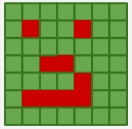
Parallelism using OpenMP

Step 2: Find threshold

```
0, 0, 0, 0, 0, 0, 0,
60,
87,
130, 130, 130, 130, 130, 130…
200, 200, 200
```

Parallelism using OpenMP

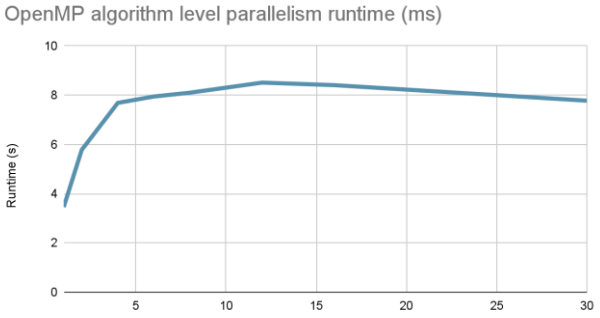
Step 3: Final segmentation

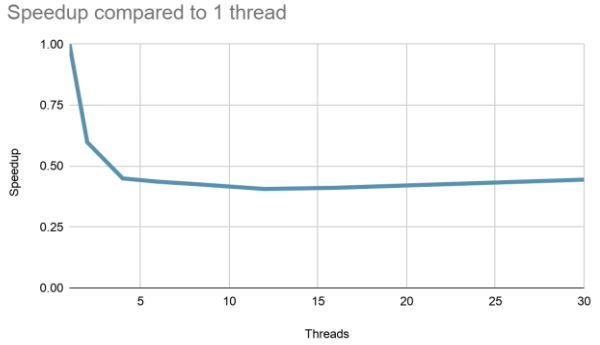


Parallelism using OpenMP

```
}
```

Threads	Runtime (s)	Speedup compared to 1 thread
1	3.456	1
2	5.779	0.59799
4	7.687	0.4496
6	7.945	0.4359
8	8.101	0.4266
12	8.510	0.40614
16	8.407	0.4111
30	7.775	0.4445





Amdahl's law theoretical max speedup 1.58.

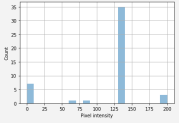
Since each of these parallelized subparts of not compute intensive, the overhead immediately dominates.

Optimization 2: OpenMP - Data parallelism

```
For cell_img in all images
{
```

Step 0: Input image

Step 1: Pixel intensity histogram



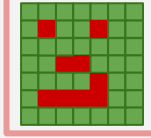
Parallelism using OpenMP

Step 2: Find threshold

```
0, 0, 0, 0, 0, 0, 0,
60,
87,
130, 130, 130, 130, 130, 130…
200, 200, 200
```

Parallelism using OpenMP

Step 3: Final segmentation



Parallelism using OpenMP

```
}
```

 # pragma states before the for loop

Allowing multiple cell images to be simultaneously segmented.

Optimization 2: OpenMP - Data parallelism

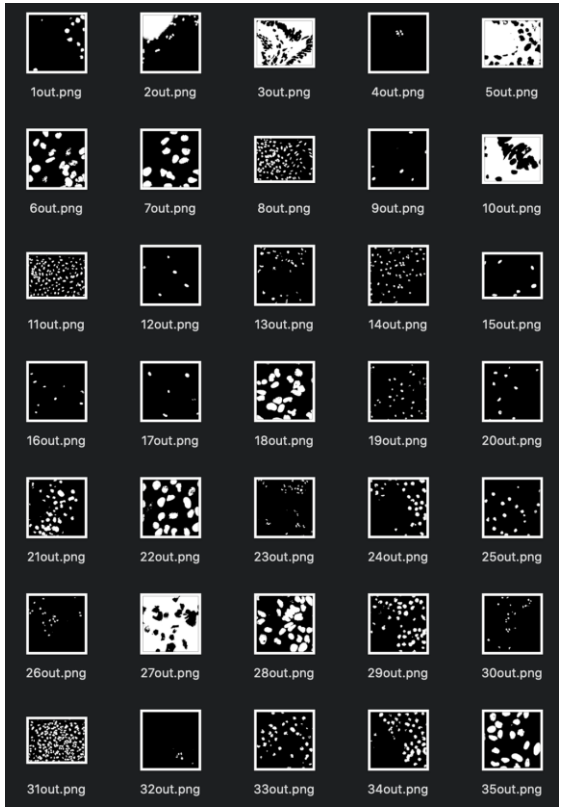
Thread number	Total time(ms)	Speedup of current compared to previous	Speedup of current compared to 1 thread	Step 0: data loading (%)	Step 1: compute histogram (%)	Step 2: finding threshold (%)	Step 3: final segmentation (%)	Step 4: data save (%)
1	3.574s	NA	1	0.46368	0.208419	0.440318	0.19899	0.08334
2	1.794759	1.9915	1.9915	0.471731	0.219885	0.0152618	0.205544	0.0870849
4	0.976506s	1.8379	3.6602	0.473829	0.215001	0.0153819	0.206288	0.0890124
6	0.702265s	1.3905	5.0896	0.471621	0.216026	0.0152663	0.207052	0.089535
8	0.618021s	1.13631	5.7835	0.464095	0.222237	0.0157273	0.208233	0.089284
12	0.539055s	1.14648	6.6307	0.450517	0.218754	0.0158821	0.219383	0.0947447
16	0.442824s	1.2173	8.0716	0.454073	0.218009	0.0157286	0.215558	0.0959675
30	0.387223s	1.1436	**9.2306**	0.536978	0.180146	0.0124914	0.181916	0.0880761
60	0.393248s	0.9847	9.0892	0.760144	0.0795287	0.00555554	0.0804626	0.0733615
90	0.424771s	0.9258	8.4147	0.808752	0.0474346	0.0032952	0.0478846	0.0925408

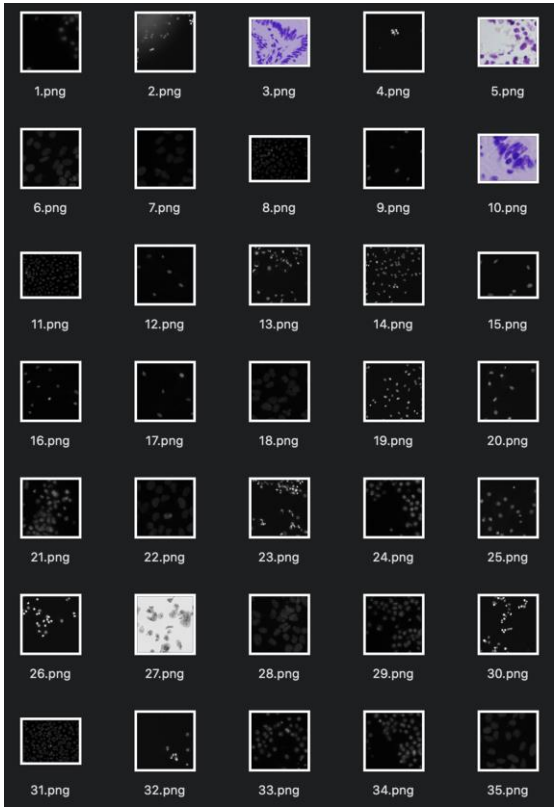
Optimization 3: MPI + OpenMP + Slurm on GCP

- **Google Cloud Cluster:**
  - 8 nodes
  - Each with 4 vCPUs
  - With SLURM
- **Hybrid Model:**
  - MPI
  - OpenMP
- **Parallel Types:**
  - Data parallelism
  - Algorithm parallelism
- **Best speedup:**
  - **5.096**

node\thread	1 threads	2 threads	4 threads	8 threads
1 nodes	**6.570s**	5.490s	4.785s	4.594s
2 nodes	3.292s	2.709s	2.525s	2.557s
4 nodes	2.211s	1.726s	1.880s	1.848s
6 nodes	1.628s	1.435s	1.403s	1.389s
8 nodes	1.432s	1.369s	1.320s	**1.289s**

Results

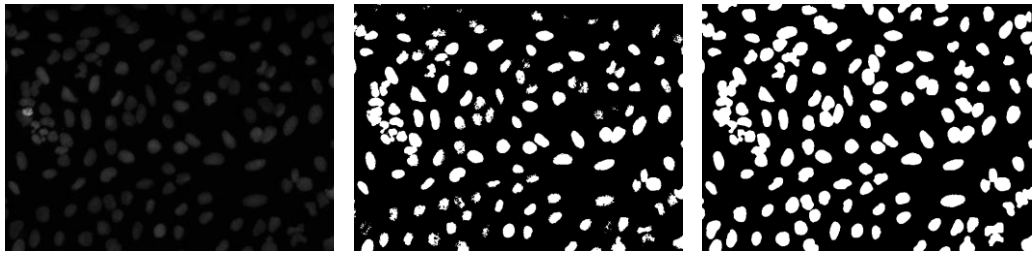

1out.png
2out.png
3out.png
4out.png
5out.png
6out.png
7out.png
8out.png
9out.png
10out.png
11out.png
12out.png
13out.png
14out.png
15out.png
16out.png
17out.png
18out.png
19out.png
20out.png
21out.png
22out.png
23out.png
24out.png
25out.png
26out.png
27out.png
28out.png
29out.png
30out.png
31out.png
32out.png
33out.png
34out.png
35out.png


1.png
2.png
3.png
4.png
5.png
6.png
7.png
8.png
9.png
10.png
11.png
12.png
13.png
14.png
15.png
16.png
17.png
18.png
19.png
20.png
21.png
22.png
23.png
24.png
25.png
26.png
27.png
28.png
29.png
30.png
31.png
32.png
33.png
34.png
35.png

Results and Discussion

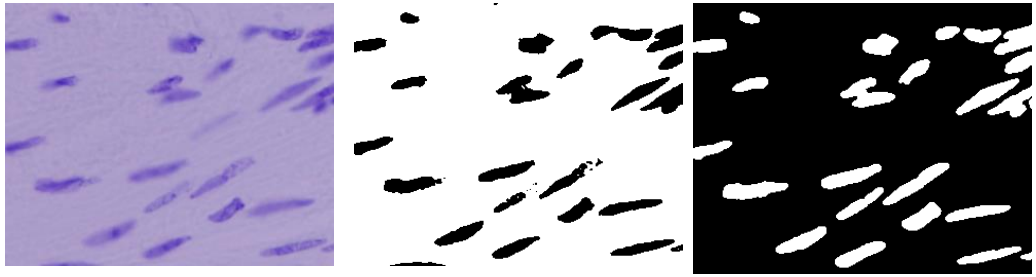
Example 1:

Original image | Otsu's result | Ground truth



Example 2:

Original image | Otsu's result | Ground truth

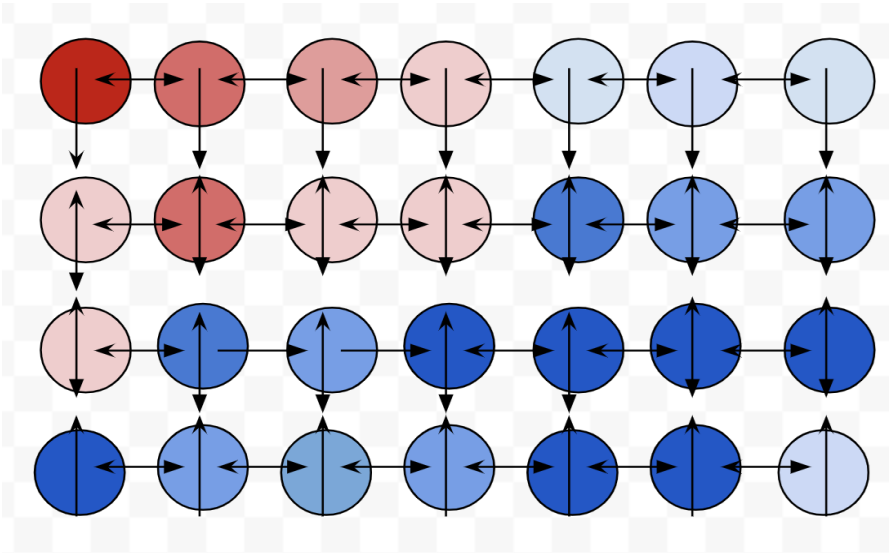


- **Speed:**
  - Extremely fast in speed
  - Great speed up with a maximum of **9.23**
- **Accuracy:**
  - Performs well in most cell segmentations with pixel intensity contrast
- **Pros:**
  - Fast and mostly accurate
  - Little dependency
  - Doesn't need training
- **Cons:**
  - Unable to differentiate foreground and background
- **Not compute intensive:** fast, but not friendly for parallelism

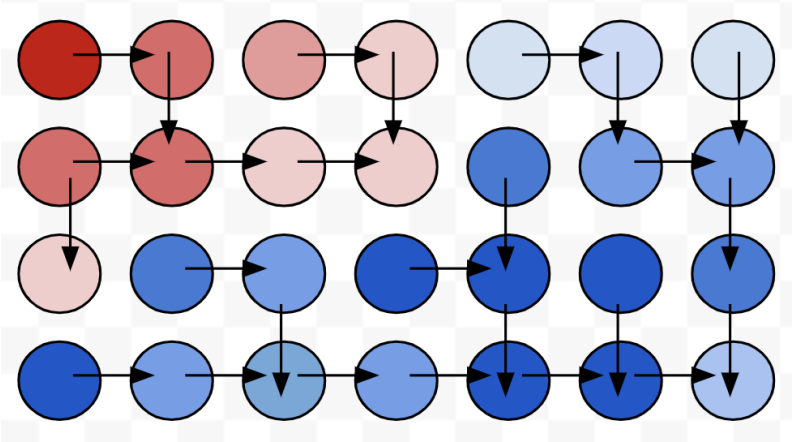
02

Boruvka's method

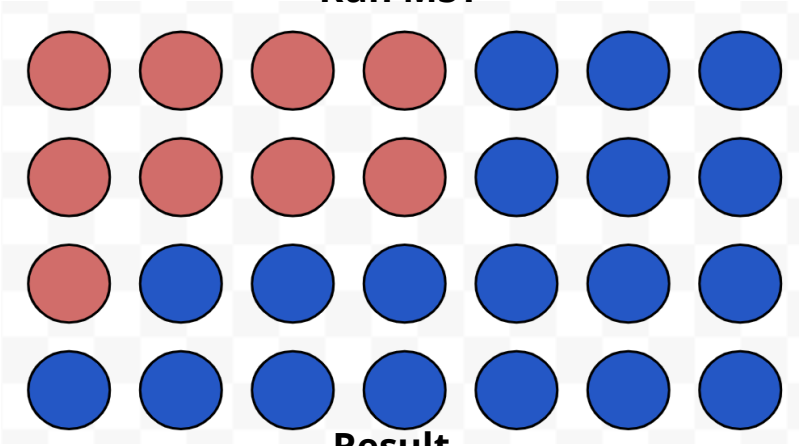
Algorithm



Initialize



Run MST



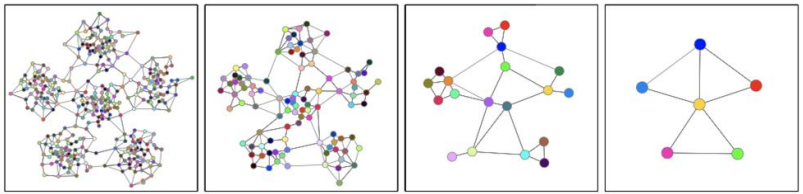
Result

Optimization 1: Memory

Overheads: Storing Graph

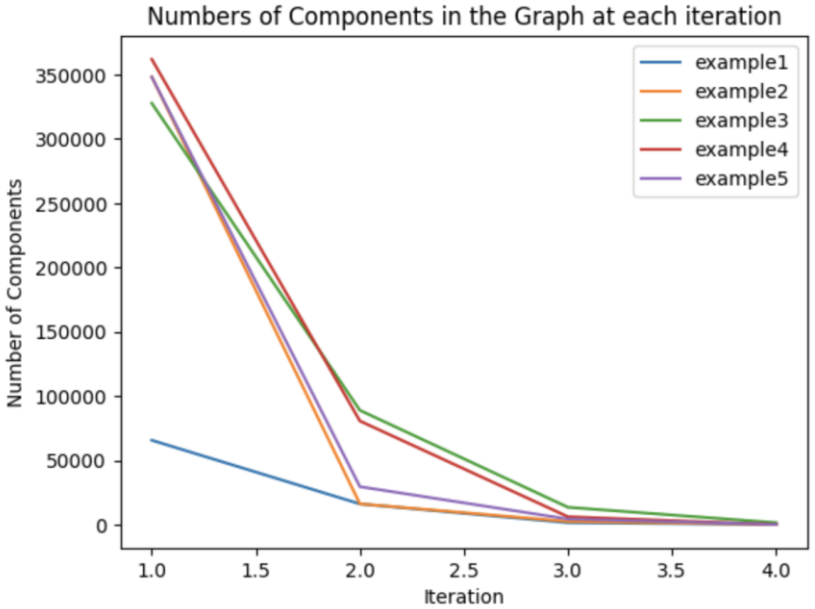
3.5 hours

per 256*256 images



Solution

- Edge compaction



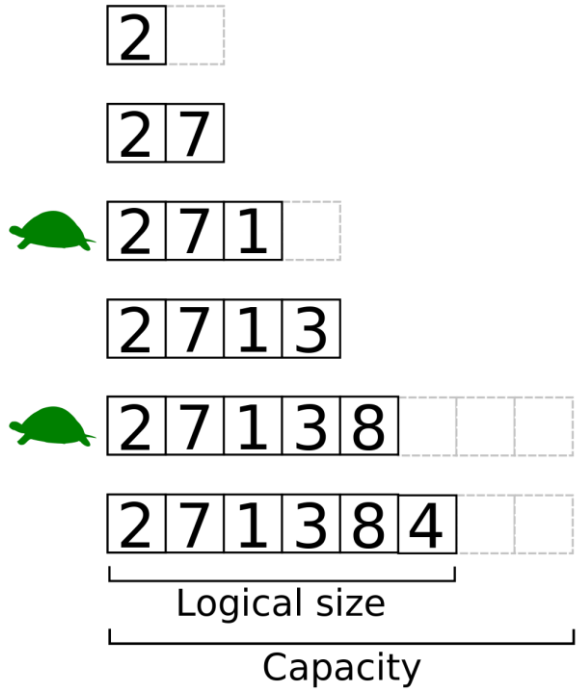
Optimization 1: Memory

Solution (cont.)

- Passed by reference, NOT pass by value!

  (Note: In C++, things are passed by value unless you specify otherwise)

- C++ dynamic memory allocation & deallocation, e.g. malloc(), calloc(), free() delete[] ...



C++'s std::vector: dynamic array

Optimization 1: Memory

Solution (cont.)

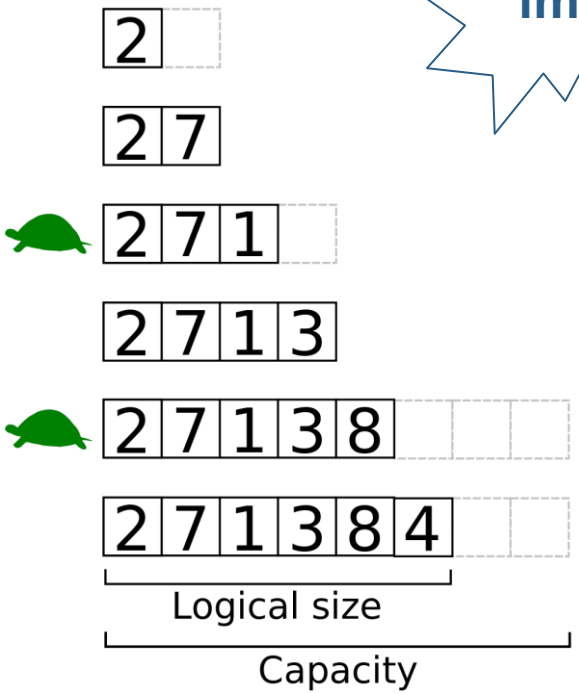
2.96s per image

Don't write code like this!

```
std::vector<int> nums;
for(int i=0; i<n; i++){
    nums.push_back(1);
}
```

Instead,

```
std::vector<int> nums(n,1);
```



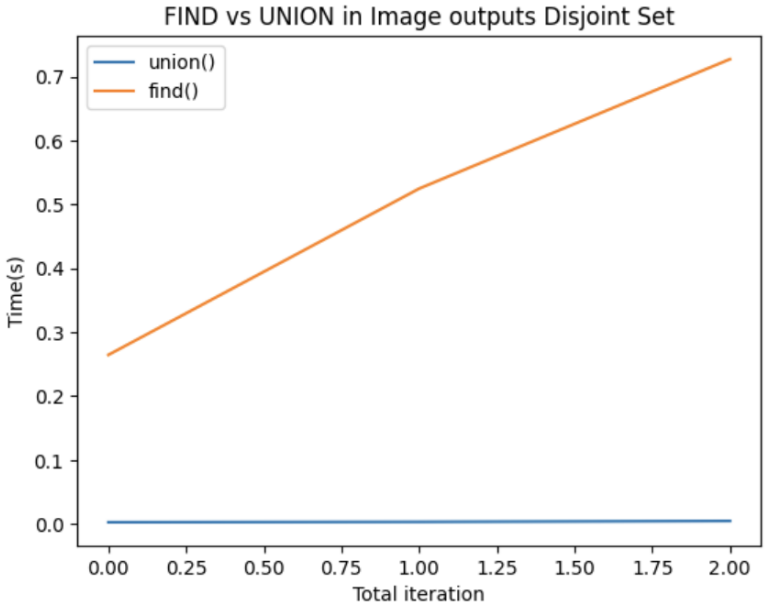
C++'s std::vector: dynamic array

Optimization 2: OpenMP

Total

9m 30s

Now, it takes 2.96s per image! But processing all the images still takes 33m7.4s!



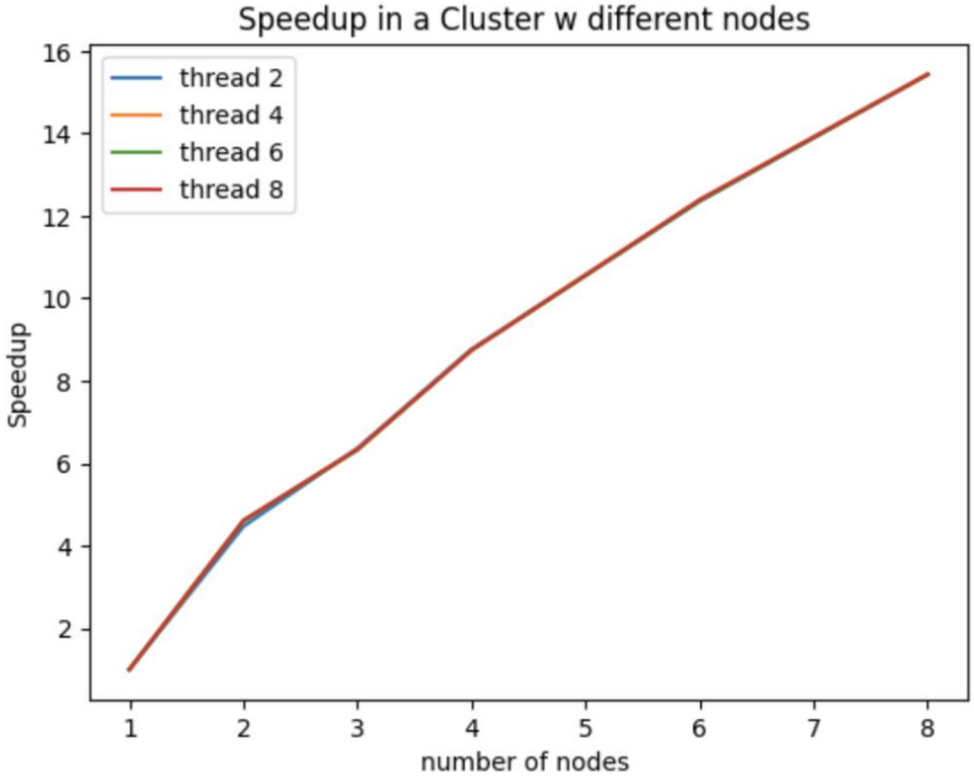
Parallelize find() operation

- OpenMP, #pragma omp parallel for schedule(dynamic)

Overhead: Data Dependency

- C++ built-in function for atomic memory access: __sync_bool_compare_and_swap

Optimization 3: MPI & OpenMP



1 node 8 thread/node, 30vCPU

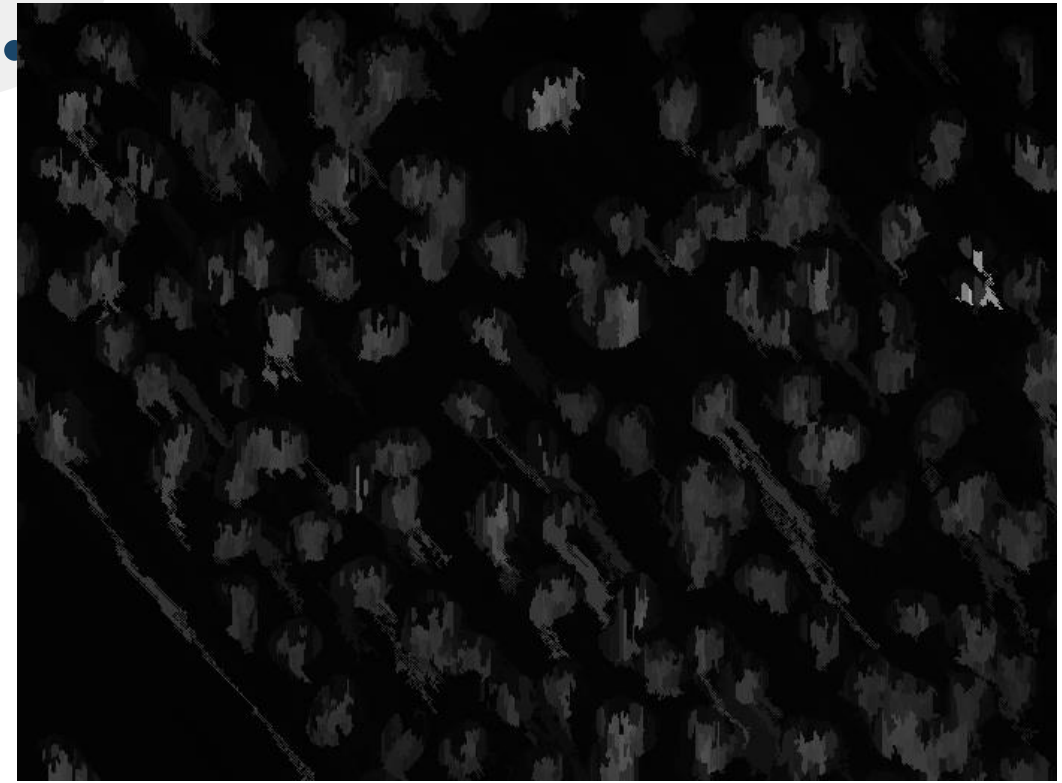
9m 30s

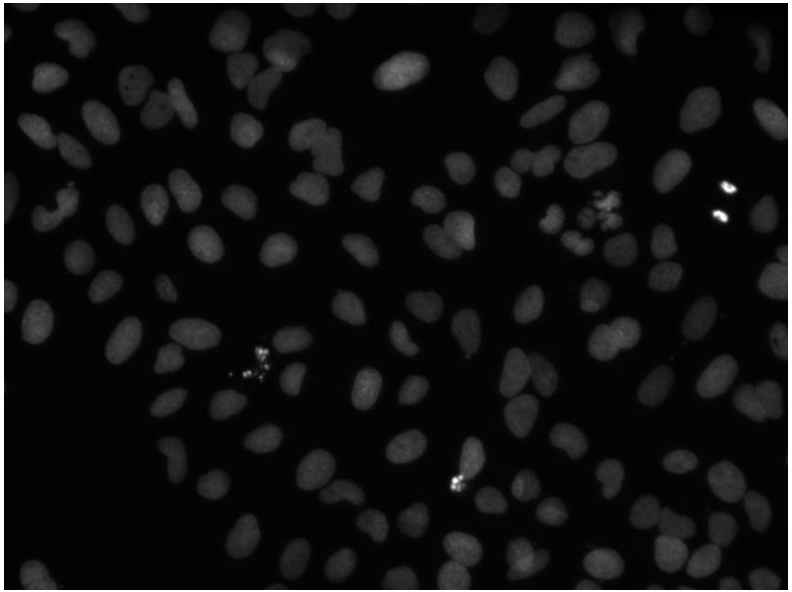
Data Parallelism

8 nodes 4 threads/node, 4vCPU

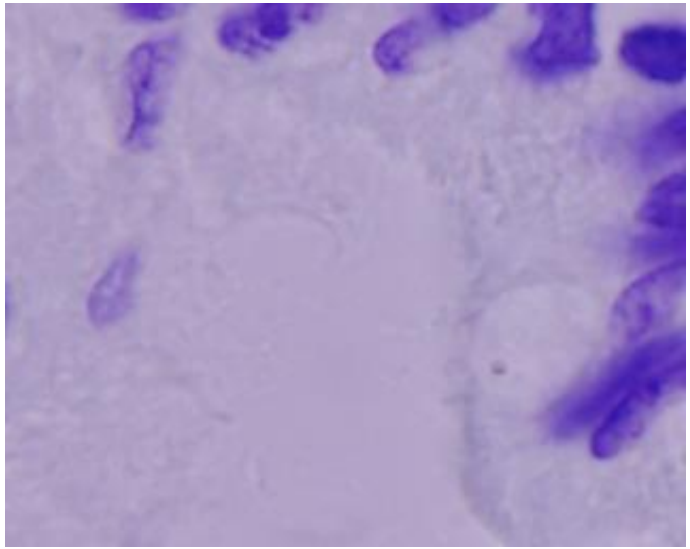
2m 8.79s

Good Results





Bad

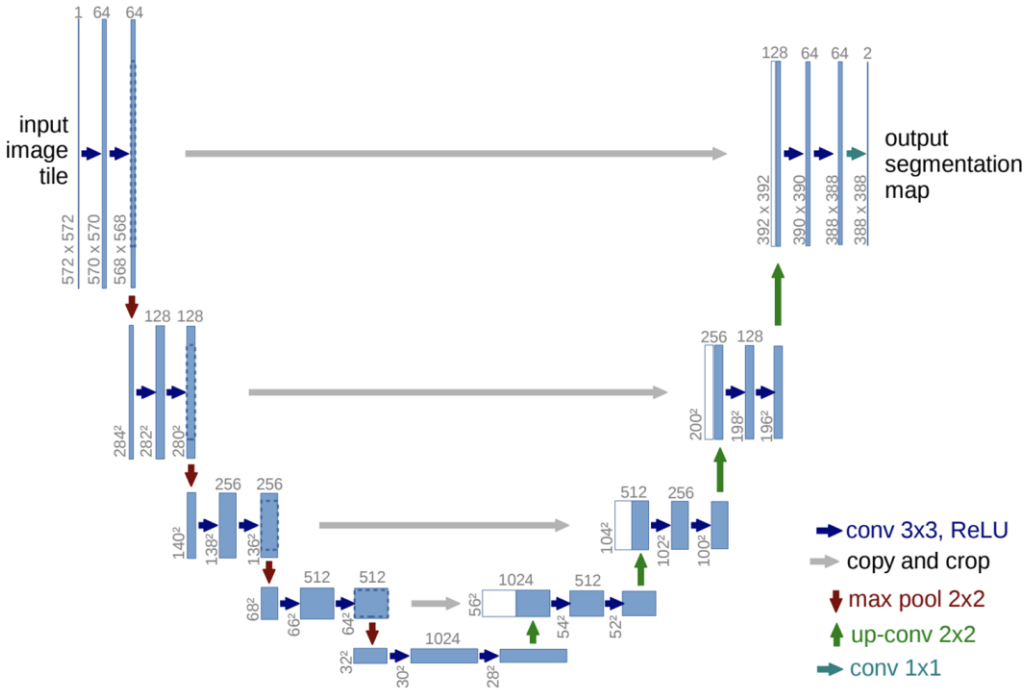


Boruvka: When there is a big color difference, it can do great!

03

UNet

UNet: Model Overview



Convolutional Neural Network

- Highly non-linear function mapping each pixel to 0 or 1
- Huge amount of model parameters

```
Total params: 1,941,105
Trainable params: 1,941,105
Non-trainable params: 0
```

- Learns fast with data augmentation

UNet

Programming Model and Platform

GPU accelerated computing

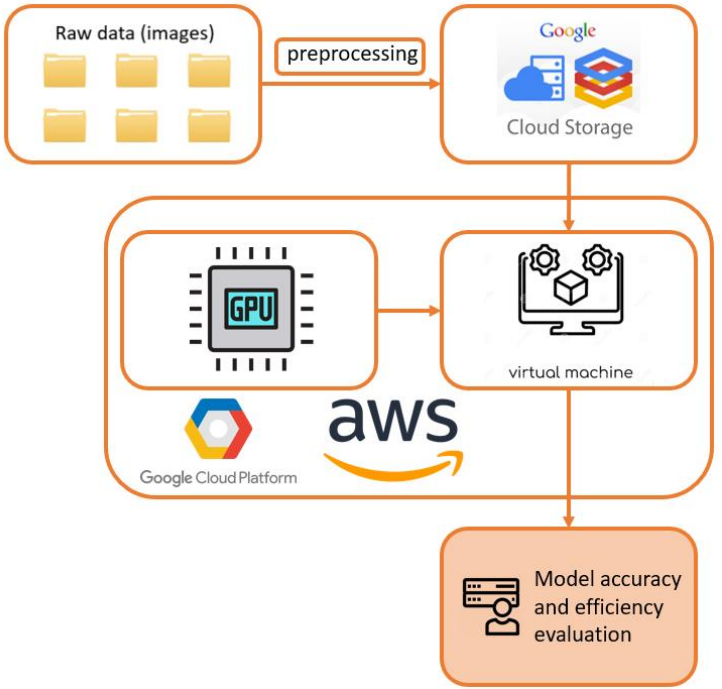
- Single GPU of different types
- Parallel data augmentation and training

Software:

- Python3
- Keras + Tensorflow

Platform:

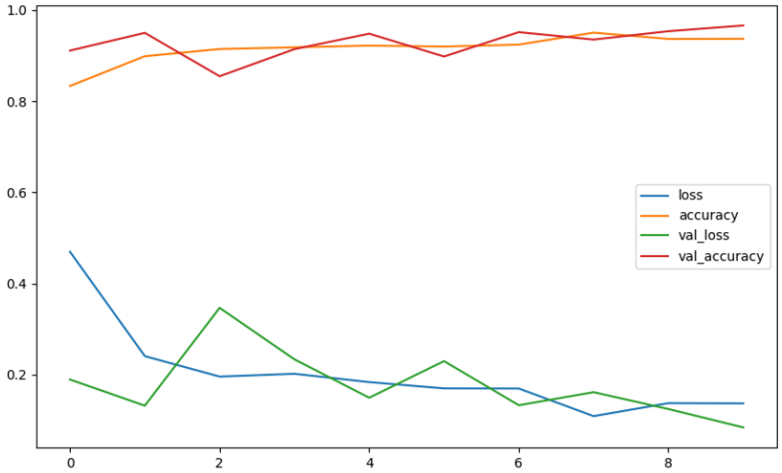
- Google Cloud Platform (GCP)
- AWS



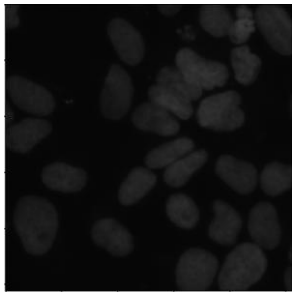
Segmentation Results

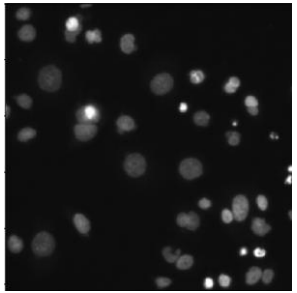
Final pixel classification accuracy: 95%

Learning curve (batch size = 1)

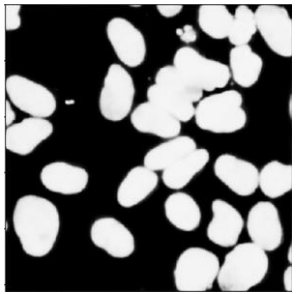


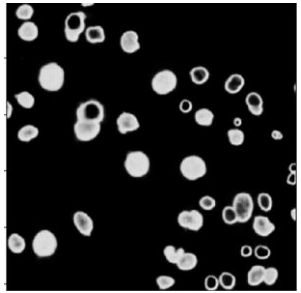
Input Image





UNet Output





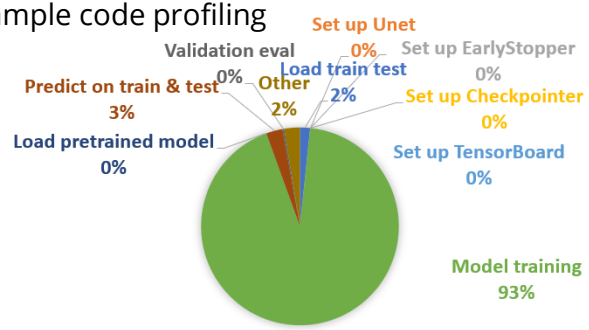
GPU Acceleration

Six VMs (including the baseline)

Instance		GCP c2-standard-30	AWS g3.4xlarge	AWS p2.xlarge	AWS p3.2xlarge	AWS g4dn.4xlarge	GCP n1-standard-16
GPU info	Type	N/A	Tesla M60	Tesla K80	Tesla V100 SXM2	Tesla T4	Tesla P4
	Memory	N/A	7618MiB	11441MiB	16160MiB	15109MiB	7611MiB
	Count	0	1	1	1	1	1
CPU info	Type	Intel(R) Xeon(R) CPU	Intel(R) Xeon(R) Platinum 8259CL @2.50GHz	Intel(R) Xeon(R) E5-2686 v4 @2.30GHz	Intel(R) Xeon(R) E5-2686 v4 @2.30GHz	Intel(R) Xeon(R) Platinum 8259CL @ 2.50GHz	Intel(R) Xeon(R) CPU @2.30GHz
	Count	30	16	4	8	16	16
	Thread(s)/core	2	2	2	2	2	2
	Core(s)/socket	15	8	2	4	8	8
	Socket(s)	1	1	1	1	1	1
	Memory	120 GiB	122 GiB	61 GiB	61 GiB	64 GiB	64 GiB
Price (instance only)		$1.5660/hr	$1.14/hr	$0.90/hr	$3.06/hr	$1.204/hr	$1.376944/hr
Batchsize = 1 Training speed		138s/epoch 696ms/step	14s/epoch 68ms/step	30s/epoch 152ms/step	14s/epoch 70ms/step	12s/epoch 60ms/step	13s/epoch 67ms/step

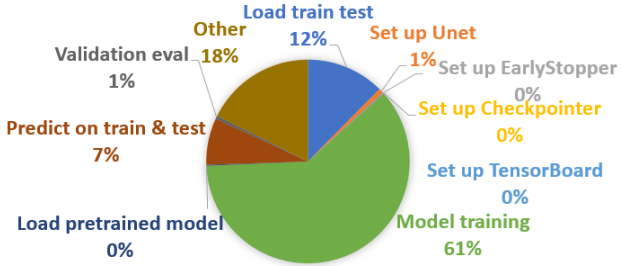
* Much more detailed report can be found in the write-up.

Sample code profiling



BASELINE RUNTIME BREAKDOWN

Script total time: 1521.629 sec

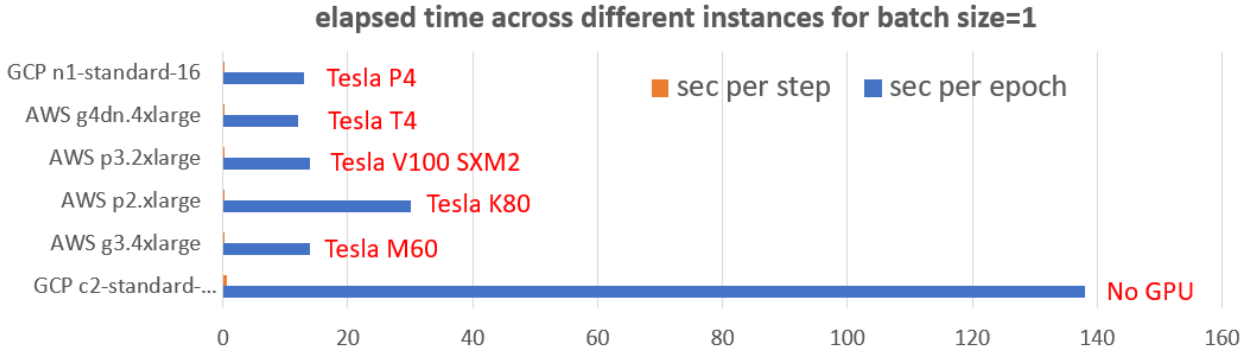


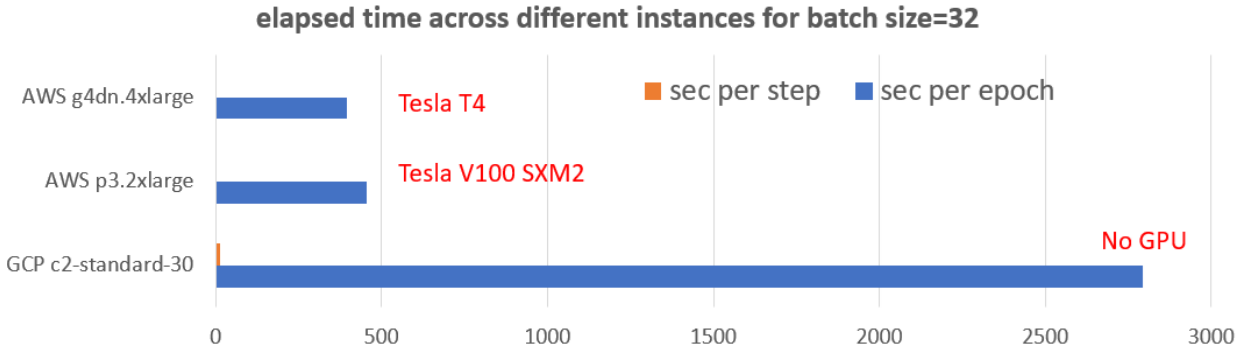
GCP GPU RUNTIME BREAKDOWN

Script total time: 245.146 sec

* Also used cProfile, TensorBoard. Please refer to the write-up.

Speedup and Scalability



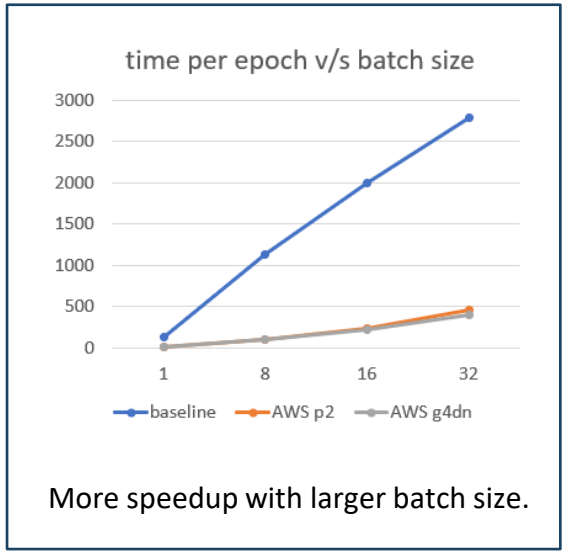


Best speedup: AWS g4dn.4xlarge

~ 11 for batch size = 1

~ 7 for batch size = 32

* compared to baseline

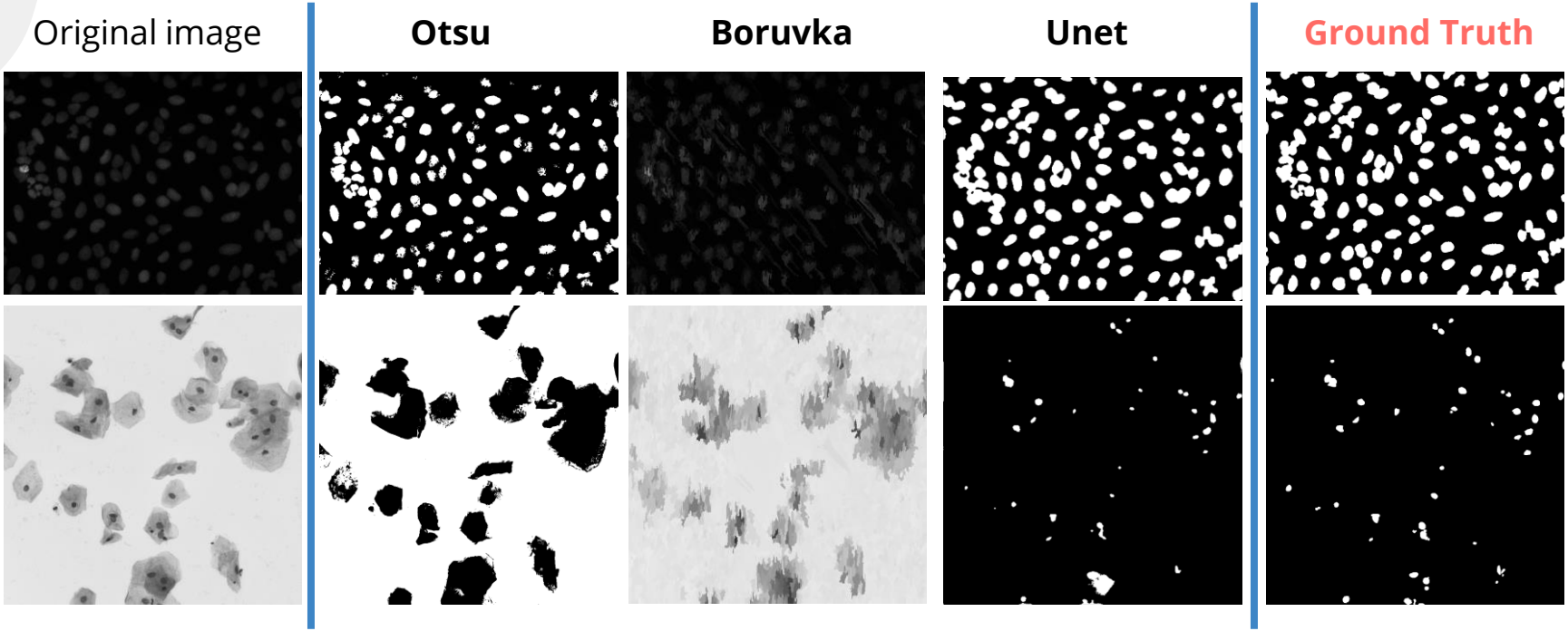


More speedup with larger batch size.

04

Final Recap

Key Results Recap



Levels of Parallelism/Cloud Infra Used

Otsu's / Boruvka's Algorithm

Optimization:
- MPI
- OpenMP

Cloud infrastructure:
- Google Cloud VM
- Google Cloud Cluster

Service:
- Slurm

Levels of Parallelism:
- Task level
- Procedure level
- Loop level

Types of Parallelism:
- Inter node: Data
- Intra node: Function

Language: C++

UNet

Optimization:
- GPU
- Spark

Cloud infrastructure:
- Google Cloud VM + GPU
- AWS VM + GPU

Service:
- Google Cloud Dataproc Cluster

Levels of Parallelism:
- Task level

Types of Parallelism:
- Data parallelism

Language: Python3

Future Work

In addition to the significant speedup of our optimized implementations, we can further leverage

- Multi-GPU instances
- Split huge images for parallel processing

Thank you for watching!

Special thanks to **Haipeng** for discussion and suggestions