Final

Cache

Index	Valid	Tag	Data	
0	1	0	0xCA	0xFE
1	1	1	0xDE	0xAD
2	1	0	0xBE	0xFF
3	0	1	0xFE	0xED

a

1010

✓ **Answer**

Index: 1

Tag: 1

Offset: 0

Valid, $1 = 1$, tag matches => Cache hit

Data read: `0×DE`

b

1110

✓ **Answer**

Index: 3

Tag: 1

Offset: 0

Not valid => Cache miss

c

0001

✓ **Answer**
Index: 0
Tag: 0
Offset: 1

Valid, $0 = 0$, tag matches => Cache hit

Data read: `0×FE`

d

1101

✓ **Answer**
Index: 2
Tag: 1
Offset: 1

Valid, $0 \neq 1$, tag does not match => Cache miss

Benchmarking

C code

```c
#include <stdio.h>
#include <stdlib.h>

#define N 10000000

int main() {
    float *a, *b, *c;

    // Allocate memory for the arrays
    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    c = (float*)malloc(sizeof(float) * N);

    // Ensure memory was allocated
    if (a == NULL || b == NULL || c == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Initialize arrays a and b to 1 and N - 1
    for (int i = 0; i < N; i++) {
```

```
        a[i] = i * 1.0f;
        b[i] = (N - i) * 1.0f;
    }

    // c = a + b ⇒ 1 + N - 1 = N
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }

    // Free allocated memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Python code

```
import numpy as np

N = 10000000

a = np.repeat(1, N)
b = np.repeat(N - 1, N)

c = a + b
```

Comparison

Then, running the C code (locally or in a slurm job)

```
gcc add.c -o add
time ./add
```

We obtain the output of

```
./add  0.02s user 0.08s system 98% cpu 0.102 total
```

Then, running the python code (locally or in a slurm job)

```
time python add.py
```

We obtain the output of

```
python add.py  0.09s user 0.06s system 98% cpu 0.154 total
```

We notice that the C code is marginally faster than the python code, at a total time of 0.102 vs 0.154 for python. This difference really doesn't matter for such a toy example, especially considering that the python code was significantly faster to write.

When utilizing a stack-based approach:

```c
#include <stdio.h>
#include <stdlib.h>

#define N 10000000

int main() {
    float a[N], b[N], c[N];

    // Initialize arrays a and b to 1 and N - 1
    for (int i = 0; i < N; i++) {
        a[i] = i * 1.0f;
        b[i] = (N - i) * 1.0f;
    }

    // c = a + b ⇒ 1 + N - 1 = N
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

We get the following output:

```
[2]    163493 segmentation fault (core dumped)  ./add
./add  0.00s user 0.00s system 0% cpu 0.228 total
```

As you cannot allocate such large amounts of memory on the stack, causing a stack overflow.

Numba

With the naiive numba code of:

```python
from numba import jit

N = 10000000
```

```
a = [1] * N
b = [N - 1] * N

@jit
def add(A, B):
    return [a + b for a, b in zip(A, B)]

c = add(a, b)
```

And executing with the following command:

```
time python add.py
```

We obtain the output of

```
real  0m22.576s
user  0m21.838s
sys   0m0.595s
```

Which is significantly worse than the C or python implementations, as it was naiively implemented. We should be utilizing lower-level libraries to speed this code up instead of applying band-aids to the pre-existing code.

OpenACC

```
#include <stdio.h>
#include <stdlib.h>

#define N 10000000

int main() {
    float *a, *b, *c;

    // Allocate memory for the arrays
    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    c = (float*)malloc(sizeof(float) * N);

    // Ensure memory was allocated
    if (a == NULL || b == NULL || c == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Initialize arrays a and b to 1 and N - 1
    for (int i = 0; i < N; i++) {
```

```
        a[i] = i * 1.0f;
        b[i] = (N - i) * 1.0f;
    }

    // Utilize openacc pragmas
    #pragma acc data copyin(a[0:N], b[0:N]) copyout(c[0:N])
    {
        #pragma acc parallel loop
        for (int i = 0; i < N; i++) {
            c[i] = a[i] + b[i];
        }
    }

    // Free allocated memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

By using openACC pragmas, we can tell the compiler that certain structures can be parallellized.

Compiling and running with:

```
pgcc -acc -ta=tesla:cc75 -Minfo=accel add.c -o add
time ./add
```

We get the output of:

```
./add  0.02s user 0.06s system 98% cpu 0.43 total
```

Which is faster than the plain C code.

CUDA

With the following CUDA code:

```
#include <stdio.h>
#include <cuda.h>

#define N 10000000

__global__ void vector_add_single(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
```

```
    }
}

int main() {
    float *a, *b, *c;
    float *d_a, *d_b, *d_c;

    size_t bytes = N * sizeof(float);
    cudaMallocHost(&a, bytes);
    cudaMallocHost(&b, bytes);
    cudaMallocHost(&c, bytes);

    for (int i = 0; i < N; i++) {
        a[i] = i * 1.0f;
        b[i] = (N - i) * 1.0f;
    }

    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    cudaMemcpy(d_a, a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, bytes, cudaMemcpyHostToDevice);

    vector_add_single<<<1, 1>>>(d_a, d_b, d_c, N);

    cudaMemcpy(c, d_c, bytes, cudaMemcpyDeviceToHost);

    cudaFreeHost(a); cudaFreeHost(b); cudaFreeHost(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

    return 0;
}
```

Compiling this with:

```
nvcc -arch=sm_75 add.cu -o add
```

I recieve an error compiling due to some missing libraries. We expect this to be faster than OpenACC however.

As for the multiple implementation, swap the `<<<1, 1>>>` with however many blocks and cores to be used.

LAMMPS

```
export WORKIR=/scratch/markov2/users/tln32/LAMMPS
mkdir -p $WORKIR
cd $WORKIR
cp /usr/local/doc/LAMMPS/*.lj $WORKIR

tee mpi-lammps.slurm <<EOF
#!/bin/bash
#SBATCH -A csds312
#SBATCH -p markov_gpu
#SBATCH --gres=gpu:1
#SBATCH -N 2 -n 2 -c 2
#SBATCH --time=01:00:00

module load LAMMPS

srun lmp -sf omp -pk omp 4 -in ./in.lj # 2 MPI task x 2 OMP
cp log.lammps ./mpi-lammps.log
EOF

sbatch mpi-lammps.slurm

tee lammps.slurm <<EOF
#!/bin/bash
#SBATCH -A csds312
#SBATCH -p markov_gpu
#SBATCH --gres=gpu:1
#SBATCH -N 1 -n 1
#SBATCH --time=01:00:00

module load LAMMPS

lmp < ./in.lj
cp log.lammps ./lammps.log
EOF

sbatch lammps.slurm

tee lammps-gpu.slurm <<EOF
#!/bin/bash
#SBATCH -A csds312
#SBATCH -p markov_gpu
#SBATCH --gres=gpu:1
#SBATCH -N 1 -n 1
#SBATCH --mem=5gb
#SBATCH --time=01:00:00

module load LAMMPS

lmp -k on  g 1 -sf kk -in  ./in-gpu.lj
```

```
cp log.lammps ./lammps-gpu.log
EOF

sbatch lammps-gpu.slurm

rm -rf $WORKIR
```

For LAMMPS-MPI, we get this output:

```
Loop time of 70.3225 on 8 procs for 20000 steps with 32000 atoms

Performance: 122862.469 tau/day, 284.404 timesteps/s
174.9% CPU use with 2 MPI tasks x 4 OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 40.727     | 46.52      | 52.314     |  84.9 | 66.15
Neigh   | 8.3883     | 9.2729     | 10.158     |  29.1 | 13.19
Comm    | 4.8284     | 11.652     | 18.475     | 199.9 | 16.57
Output  | 0.018626   | 0.018923   | 0.019219   |   0.2 |  0.03
Modify  | 2.6517     | 2.7955     | 2.9394     |   8.6 |  3.98
Other   |            | 0.0631     |            |       |  0.09

Nlocal:          16000 ave       16016 max       15984 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Nghost:          13055 ave       13071 max       13039 min
Histogram: 1 0 0 0 0 0 0 0 0 1
Neighs:         600034 ave      605891 max      594177 min
Histogram: 1 0 0 0 0 0 0 0 0 1

Total # of neighbors = 1200068
Ave neighs/atom = 37.502125
Neighbor list builds = 1000
Dangerous builds not checked
Total wall time: 0:01:10
```

LAMMPS gives us the output of:

```
Loop time of 353.575 on 1 procs for 20000 steps with 32000 atoms

Performance: 24436.085 tau/day, 56.565 timesteps/s
```

```
99.8% CPU use with 1 MPI tasks x 1 OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 296.56     | 296.56     | 296.56     |   0.0 | 83.87
Neigh   | 47.503     | 47.503     | 47.503     |   0.0 | 13.44
Comm    | 3.2547     | 3.2547     | 3.2547     |   0.0 |  0.92
Output  | 0.026462   | 0.026462   | 0.026462   |   0.0 |  0.01
Modify  | 5.191      | 5.191      | 5.191      |   0.0 |  1.47
Other   |            | 1.043      |            |       |  0.29

Nlocal:          32000 ave       32000 max       32000 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Nghost:          18742 ave       18742 max       18742 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Neighs:    1.20024e+06 ave 1.20024e+06 max 1.20024e+06 min
Histogram: 1 0 0 0 0 0 0 0 0 0

Total # of neighbors = 1200245
Ave neighs/atom = 37.507656
Neighbor list builds = 1000
Dangerous builds not checked
Total wall time: 0:05:53
```

LAMMPS-GPU gives us the output of:

```
Loop time of 9.01642 on 1 procs for 20000 steps with 32000 atoms

Performance: 958251.925 tau/day, 2218.176 timesteps/s
99.6% CPU use with 1 MPI tasks x 1 OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0.45189    | 0.45189    | 0.45189    |   0.0 |  5.01
Neigh   | 1.8816     | 1.8816     | 1.8816     |   0.0 | 20.87
Comm    | 0.7645     | 0.7645     | 0.7645     |   0.0 |  8.48
Output  | 0.013337   | 0.013337   | 0.013337   |   0.0 |  0.15
Modify  | 5.669      | 5.669      | 5.669      |   0.0 | 62.87
Other   |            | 0.2361     |            |       |  2.62

Nlocal:          32000 ave       32000 max       32000 min
```

```
Histogram: 1 0 0 0 0 0 0 0 0 0
Nghost:          18798 ave       18798 max       18798 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Neighs:              0 ave           0 max           0 min
Histogram: 1 0 0 0 0 0 0 0 0 0
FullNghs:  2.39959e+06 ave 2.39959e+06 max 2.39959e+06 min
Histogram: 1 0 0 0 0 0 0 0 0 0

Total # of neighbors = 2399588
Ave neighs/atom = 74.987125
Neighbor list builds = 1000
Dangerous builds not checked
Total wall time: 0:00:09
```

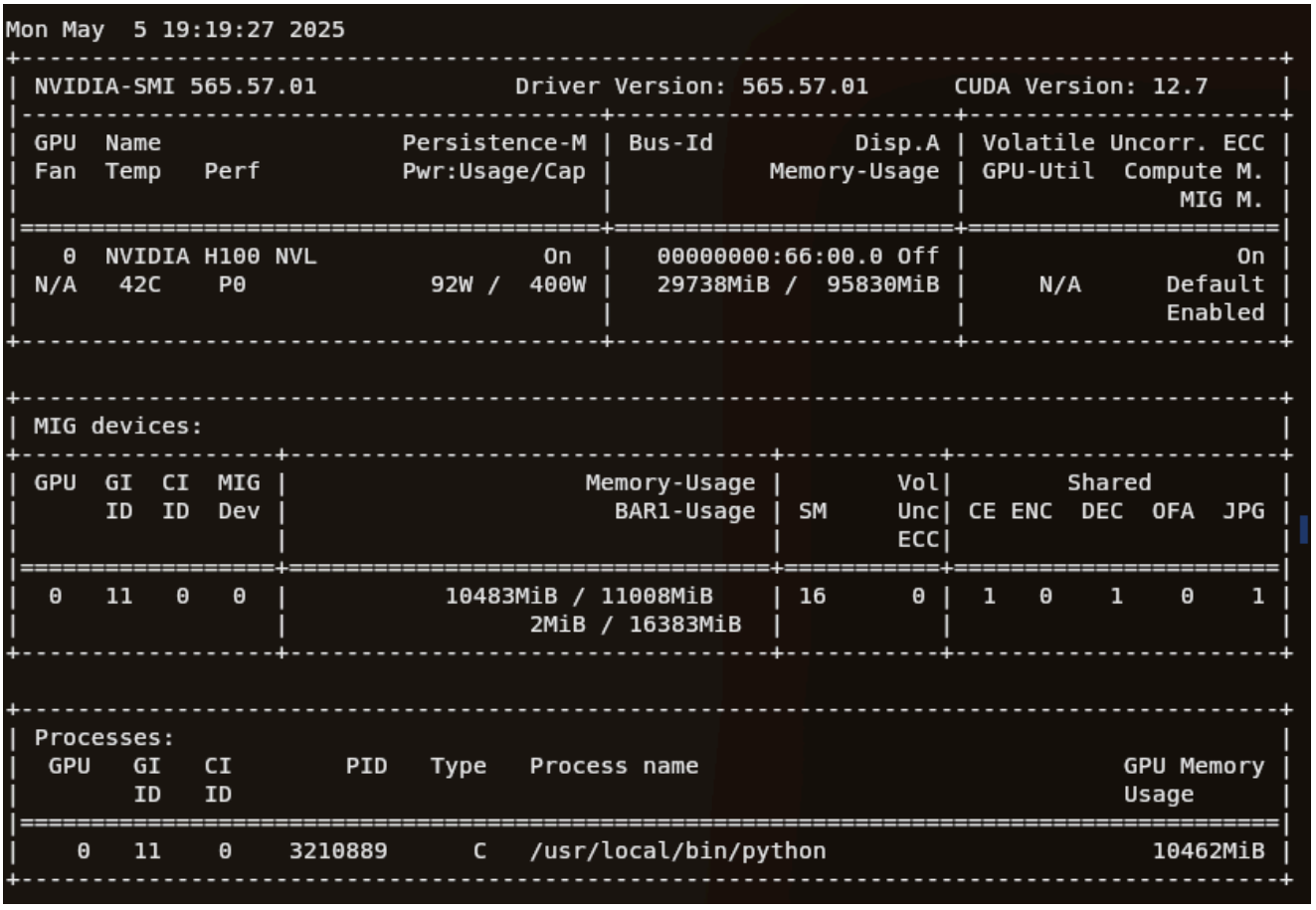
As seen, the single threaded job is ~6 mins, parallel ~1 minute, and the GPU job is ~10 seconds. The GPU was much faster as expected, and the single threaded was much slower, as expected.

Singularity

```
export SINGULARITY_CACHEDIR=/scratch/markov2/users/tln32/singularity
mkdir -p $SINGULARITY_CACHEDIR
cd $SINGULARITY_CACHEDIR
module load singularity
singularity pull docker://tensorflow/tensorflow:latest-gpu-py3
cp ~/csds312/singularity/tensorflow/classifier.py $SINGULARITY_CACHEDIR

tee tensor-classifier.slurm <<EOF
#!/bin/bash
#SBATCH -J tensor-classifier
#SBATCH -A csds312
#SBATCH -p markov_gpu
#SBATCH --gres=gpu:1
#SBATCH --mem=4gb

module unload
module load singularity

nvidia-smi --loop=5 > gpu.txt &
MONITOR_PID=\$!

export CUDA_VISIBLE_DEVICES=0
singularity exec -B /scratch --nv ./tensorflow_latest-gpu-py3.sif python
classifier.py

kill \$MONITOR_PID
```

```
cat gpu.txt

cp -ru * \$SLURM_SUBMIT_DIR
EOF

sbatch $SINGULARITY_CACHEDIR/tensor-classifier.slurm
rm -rf $SINGULARITY_CACHEDIR
quota -s
```

Running the code above, we get a few grabs of `nvidia-smi` in the `gpu.txt` file.



```
Mon May  5 19:20:48 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 565.57.01              Driver Version: 565.57.01      CUDA Version: 12.7     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA H100 NVL                On  |   00000000:66:00.0 Off |                   On |
| N/A   42C    P0             93W /  400W |   10723MiB /  95830MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| MIG devices:                                                                            |
+------------------+--------------------------------+-----------+-----------------------+
| GPU  GI  CI  MIG |                     Memory-Usage |        Vol|        Shared         |
|      ID  ID  Dev |                       BAR1-Usage | SM     Unc| CE ENC  DEC  OFA  JPG |
|                  |                                |        ECC|                       |
|==================+================================+===========+=======================|
|  0   11   0   0  |           10521MiB / 11008MiB  | 16      0 |  1   0    1    0    1 |
|                  |                 2MiB / 16383MiB|           |                       |
+------------------+--------------------------------+-----------+-----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   11    0    3210889      C   /usr/local/bin/python                       10502MiB |
+-----------------------------------------------------------------------------------------+
```

```
Mon May  5 19:23:43 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 565.57.01              Driver Version: 565.57.01      CUDA Version: 12.7     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA H100 NVL                On  |   00000000:66:00.0 Off |                   On |
| N/A   42C    P0             94W /  400W |   10931MiB /  95830MiB |     N/A      Default |
|                                         |                        |              Enabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| MIG devices:                                                                            |
+------------------+--------------------------------+-----------+-----------------------+
| GPU  GI  CI  MIG |                     Memory-Usage |        Vol|        Shared         |
|      ID  ID  Dev |                       BAR1-Usage | SM     Unc| CE ENC  DEC  OFA  JPG |
|                  |                                |        ECC|                       |
|==================+================================+===========+=======================|
|  0   11   0   0  |           10639MiB / 11008MiB  | 16      0 |  1   0    1    0    1 |
|                  |                 2MiB / 16383MiB|           |                       |
+------------------+--------------------------------+-----------+-----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   11    0    3210889      C   /usr/local/bin/python                       10618MiB |
+-----------------------------------------------------------------------------------------+
```

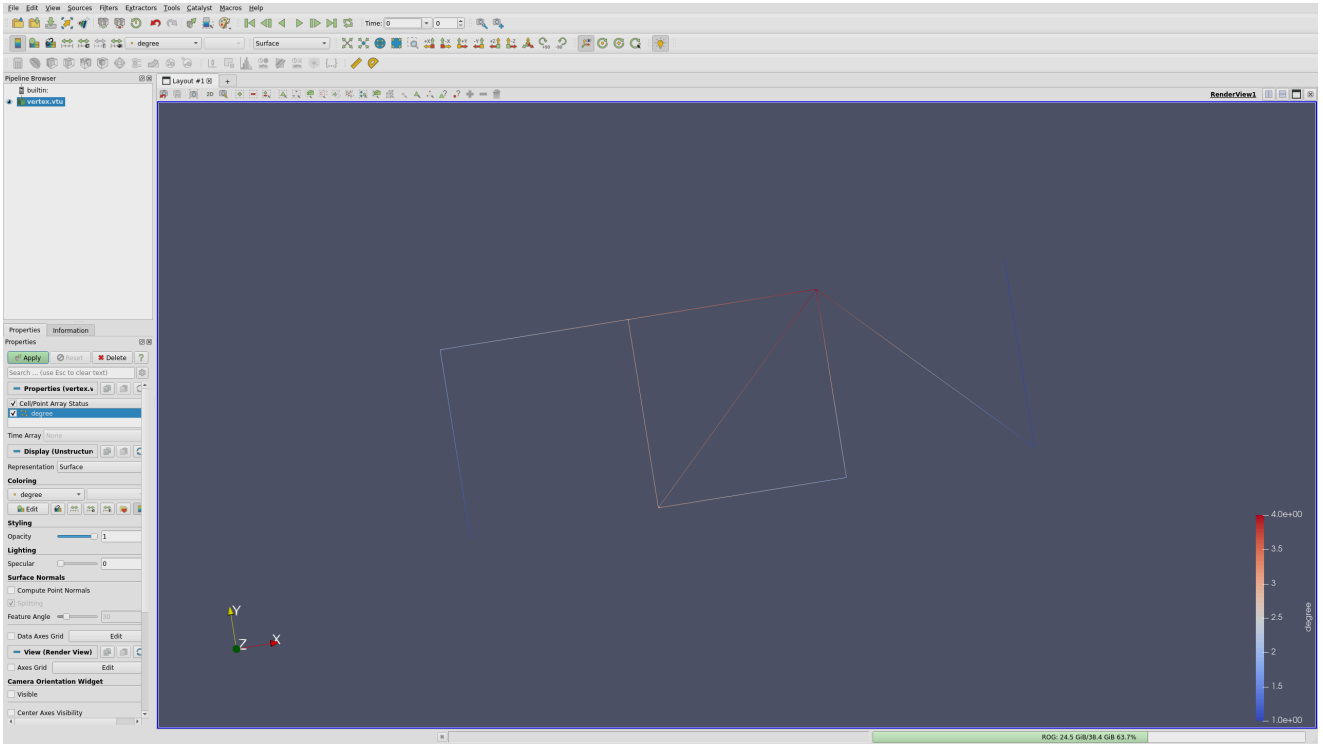
Additionally, after checking the quota, I norice I was in the right directory.

Paraview

```
#!/usr/bin/env python
import vtk

if __name__ == "__main__":
    print ("vtkGraph: Building a graph using Unstructured Grid, dump it
in a vtk file, vertex.vtu, to be visualized using ParaView")

    # Create a user specified number of points
    pointSource = vtk.vtkPointSource()
    pointSource.Update()

    # Create an integer array to store vertex id data and link it with
its degree value as a scalar.
    degree  = vtk.vtkIntArray()
    degree.SetNumberOfComponents(1)
    degree.SetName("degree")
    degree.SetNumberOfTuples(8)
    degree.SetValue(0,2)
    degree.SetValue(1,1)
    degree.SetValue(2,3)
    degree.SetValue(3,3)
    degree.SetValue(4,4)
    degree.SetValue(5,2)
    degree.SetValue(6,1)
    degree.SetValue(7,1) # Make it have one connection

    pointSource.GetOutput().GetPointData().AddArray(degree)

    # Assaign co-ordinates for vertices. vtkPoints represents 3D points
    Points = vtk.vtkPoints()

    Points.InsertNextPoint(0,1,0)
    Points.InsertNextPoint(0,0,0)
    Points.InsertNextPoint(1,1,0)
    Points.InsertNextPoint(1,0,0)
    Points.InsertNextPoint(2,1,0)
    Points.InsertNextPoint(2,0,0)
    Points.InsertNextPoint(3,0,0)
    Points.InsertNextPoint(3,1,0) # Add vertex 7

    # Establish the specified edges using CellArray. It represents cell
connectivity
    line = vtk.vtkCellArray()
    line.Allocate(9) # Increase number of edges
    line.InsertNextCell(2)
    line.InsertCellPoint(0)
    line.InsertCellPoint(1)
    line.InsertNextCell(2)
    line.InsertCellPoint(0)
```

```
    line.InsertCellPoint(2)
    line.InsertNextCell(2)
    line.InsertCellPoint(2)
    line.InsertCellPoint(3)
    line.InsertNextCell(2)
    line.InsertCellPoint(2)
    line.InsertCellPoint(4)
    line.InsertNextCell(2)
    line.InsertCellPoint(3)
    line.InsertCellPoint(4)
    line.InsertNextCell(2)
    line.InsertCellPoint(3)
    line.InsertCellPoint(5)
    line.InsertNextCell(2)
    line.InsertCellPoint(4)
    line.InsertCellPoint(5)
    line.InsertNextCell(2)
    line.InsertCellPoint(4)
    line.InsertCellPoint(6)
    line.InsertNextCell(2)
    line.InsertCellPoint(6)
    line.InsertCellPoint(7) # Add connection from vertex 6 to 7

    G = vtk.vtkUnstructuredGrid()
    G.GetPointData().SetScalars(degree)
    G.SetPoints(Points)
    G.SetCells(vtk.VTK_LINE, line)

    # Dump the graph in VTK unstructured format (.vtu)
    gw = vtk.vtkXMLUnstructuredGridWriter()
    gw.SetFileName("vertex.vtu")
    gw.SetInputData(G)
    gw.Write()
    print ('⟶ ',)

    print ("Feed the vertex.vtu file in ParaView.")
```

After running the python file, and opening it with paraview, we get this output:


File Edit View Sources Filters Extractors Tools Catalyst Macros Help
Pipeline Browser
builtin:
vertex.vtu
Properties
Information
Apply
Reset
Delete
Cell/Point Array Status
degree
Time Array
Display (Unstructun
Representation
Surface
Coloring
Styling
Opacity
Lighting
Specular
Surface Normals
Compute Point Normals
Splitting
Feature Angle
Data Axes Grid
Edit
View (Render View)
Axes Grid
Camera Orientation Widget
Visible
Center Axes Visibility
Layout #1
RenderView1
4.0e+00
3.5
3
2.5
2
1.5
1.0e+00
degree
Y
Z
X