3.2 Instalación de OpenCV 4

3.2.2. Responda las siguientes preguntas relacionadas con la instalación de Python 3 y OpenCV 4:
¿Cuáles son las ventajas de usar un ambiente virtual de Python? (2 pts)
¿Qué es NumPy? (2 pts)
Describa ejemplos de 2 aplicaciones de numpy. (2 pts)
¿Para qué sirve CMake? (2 pts)
¿Qué ventajas/desventajas tiene CMake ante otras herramientas de configuración de Makefiles (Autotools, qmake, Scons, etc) en sistemas embebidos? (2 pts)

3.3 Perfilado de aplicaciones en Raspberry Pi 2

3.3.1. Utilice el ambiente virtual configurado en la sección 3.2 para ejecutar el código motion_detector.py con la Raspberry Pi Camera Module v2.

- Se utilizó el siguiente comando para activar el ambiente virtual configurado en la sección anterior

$ workon SEAD_p2

3.3.2. Utilice cProfile para obtener métricas de perfilado de la aplicación. Ejecute la aplicación con cProfile por al menos 10 segundos y máximo 15 segundos

- Se utlizó el siguiente comando para obtener las métricas.

$ DISPLAY=:0 python3 -m cProfile -o prof_rpi_cam.out motion_detector.py

3.3.3. Utilice pstats para reordenar los resultados obtenidos en el paso anterior y visualizar las 10 funciones con mayor tiempo interno (no tiempo acumulado). Asegúrese de utilizar la función strip para hacer más legible los resultados. Puede basarse en el link:
https://www.stefaanlippens.net/python_profiling_with_pstats_interactive_mode
Escriba el resultado obtenido con "stats 10" en el reporte final. (5 pts)

- Se utilizaron los siguientes comandos para obtener las 10 funciones con mayor tiempo interno

$ python3 -m pstats prof_rpi_cam.out

% strip
% sort time

% stats 10

```
Thu Jul 11 05:21:20 2019    prof_rpi_cam.out

         82741 function calls (79969 primitive calls) in 13.548 seconds

   Ordered by: internal time
   List reduced from 1289 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      152    3.363    0.022    3.363    0.022 {GaussianBlur}
      152    3.032    0.020    3.032    0.020 {resize}
        1    2.002    2.002    2.002    2.002 {built-in method time.sleep}
      150    1.391    0.009    1.391    0.009 {waitKey}
       13    0.716    0.055    0.729    0.056 {built-in method _imp.create_dynamic}
      450    0.625    0.001    0.625    0.001 {imshow}
        1    0.460    0.460    0.460    0.460 {method 'read' of 'cv2.VideoCapture' objects}
      150    0.358    0.002    0.358    0.002 {findContours}
      300    0.219    0.001    0.219    0.001 {putText}
      150    0.159    0.001    0.159    0.001 {dilate}
```

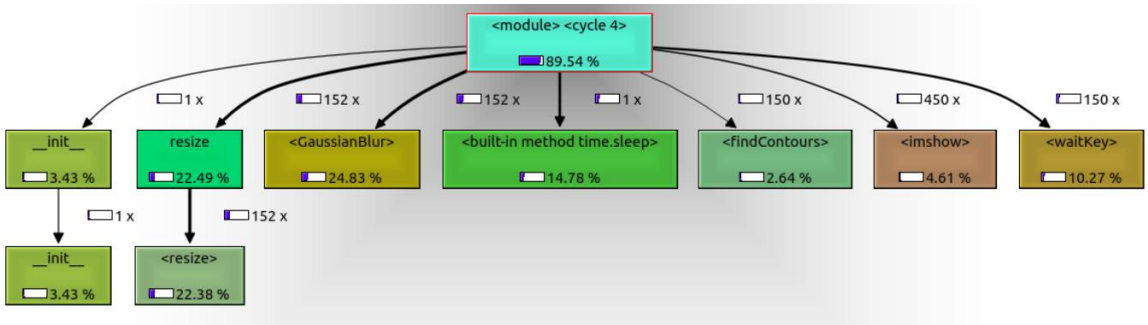
3.3.4. Utilice KCacheGrind como herramienta de visualización de perfilado para obtener el Call Graph de la aplicación. Para utilizar KCacheGrind con Python, se recomienda utilizar el script pyprof2calltree.py.

- Para convertir el archivo de metricas *prof_rpi_cam.out* en un formato soportado por KcacheGrind se utilizó la herramienta pyprof2calltree. Los siguientes comandos fueron necesarios para la instalación y uso de la misma.

$ pip3 install pyprof2calltree
$ pyprof2calltree -i prof_rpi_cam.out -o prof_rpi_cam_callgrind.out

- Para la visualización del perfilado se utilizó KcacheGrind. A continuación el comando de instalación y resultado.

$ sudo apt-get install kcachegrind



3.3.5. Utilice la Raspberry Pi Camera Module v2 para grabar un video corto (mínimo 5 segundos). Puede utilizar herramientas incluidas en la distribución de Raspbian para esto.

- Se utilizaron los siguientes comandos para grabar un video de 10 segundos y resolución 640x480 y su respectiva conversión a formato MP4.

$ raspivid -t 10000 -w 640 -h 480 -o test_video.h264
$ MP4Box -add test_video.h264 test_video.mp4

3.3.7. Similar a los pasos 3.3.2 y 3.3.3, utilice cProfile y pstats para obtener métricas de perfilado con cProfile y visualización de las 10 funciones con mayor tiempo interno sólo que esta vez debe hacerlo con el video corto grabado: python -m cProfile -o out.prof motion_detector.py –video video.mp4

- Se utilizo el siguiente comando para obtener las métricas de la ejecución de motion_detector utilizando el video grabado.

$ DISPLAY=:0 python3 -m cProfile -o prof_rpi_vid1.out motion_detector.py --video test_video.mp4

- Se utilizaron los siguientes comandos para obtener las 10 funciones con mayor tiempo interno

$ python3 -m pstats prof_rpi_vid1.out

% strip
% sort time
% stats 10

```
Sat Jul 13 00:23:34 2019    prof_rpi_vid1.out

         86429 function calls (83652 primitive calls) in 9.272 seconds

   Ordered by: internal time
   List reduced from 1285 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      288    3.660    0.013    3.660    0.013 {resize}
      289    2.006    0.007    2.006    0.007 {method 'read' of 'cv2.VideoCapture' objects}
       13    0.707    0.054    0.721    0.055 {built-in method _imp.create_dynamic}
      288    0.655    0.002    0.655    0.002 {GaussianBlur}
      287    0.485    0.002    0.485    0.002 {waitKey}
      861    0.342    0.000    0.342    0.000 {imshow}
      574    0.248    0.000    0.248    0.000 {putText}
        1    0.168    0.168    9.272    9.272 motion_detector.py:6(<module>)
      166    0.109    0.001    0.109    0.001 {built-in method marshal.loads}
      287    0.070    0.000    0.070    0.000 {findContours}
```

3.3.8. En el código motion_detector.py, busque la línea de código donde se llama a la función resize, para modificar el tamaño de la ventana:

```
# resize the frame, convert it to grayscale, and bl
frame = imutils.resize(frame, width=500)
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
# resize the frame, convert it to grayscale, and
frame = imutils.resize(frame, width=100)
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Vuelva a ejecutar los pasos 3.3.2 y 3.3.3.

- Se utilizó el siguiente comando para obtener las métricas de la ejecución de motion_detector utilizando el video grabado.

$ DISPLAY=:0 python3 -m cProfile -o prof_rpi_vid2.out motion_detector.py --video test_video.mp4

- Se utilizaron los siguientes comandos para obtener las 10 funciones con mayor tiempo interno

$ python3 -m pstats prof_rpi_vid2.out

% strip
% sort time
% stats 10

```
Sat Jul 13 00:23:54 2019    prof_rpi_vid2.out

         86432 function calls (83655 primitive calls) in 9.298 seconds

   Ordered by: internal time
   List reduced from 1285 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      288    3.659    0.013    3.659    0.013 {resize}
      289    2.027    0.007    2.027    0.007 {method 'read' of 'cv2.VideoCapture' objects}
       13    0.698    0.054    0.712    0.055 {built-in method _imp.create_dynamic}
      288    0.642    0.002    0.642    0.002 {GaussianBlur}
      287    0.507    0.002    0.507    0.002 {waitKey}
      861    0.354    0.000    0.354    0.000 {imshow}
      574    0.248    0.000    0.248    0.000 {putText}
        1    0.165    0.165    9.299    9.299 motion_detector.py:6(<module>)
      166    0.109    0.001    0.109    0.001 {built-in method marshal.loads}
      287    0.069    0.000    0.069    0.000 {findContours}
```

Comparacion con 3.3.7 ……

3.3.9. (Opcional)
Optimice la función “resize” para obtener al menos un 20% de mejora en el tiempo de ejecución total de la aplicación. La única restricción es que no puede modificar el tamaño de resizing. Es decir, no puede cambiar el argumento “width=500” ni modificar de ninguna otra forma el tamaño actual de las ventanas

3.4 Perfilado de aplicaciones en Nvidia Jetson Nano

3.4.2 Utilice cProfile para obtener métricas de perfilado de la aplicación con el video corto grabado:

- Se utilizo el siguiente comando para obtener las métricas de la ejecución de motion_detector utilizando el video grabado.

$ python -m cProfile -o prof_jetson_vid1.out motion_detector.py --video test_video.mp4

3.4.3 Utilice pstats para reordenar los resultados obtenidos en el paso anterior y visualizar las 10 funciones con mayor tiempo interno (no tiempo acumulado).

- Se utilizaron los siguientes comandos para obtener las 10 funciones con mayor tiempo interno

$ python -m pstats prof_jetson_vid1.out

% strip
% sort time
% stats 10

```
Thu Jul 11 16:30:18 2019    out.prof

        97133 function calls (94368 primitive calls) in 8.026 seconds

   Ordered by: internal time
   List reduced from 1308 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      288    2.873    0.010    2.873    0.010 {resize}
      288    1.266    0.004    1.266    0.004 {GaussianBlur}
      287    1.207    0.004    1.207    0.004 {waitKey}
      289    0.739    0.003    0.739    0.003 {method 'read' of 'cv2.VideoCapture' objects}
      861    0.335    0.000    0.335    0.000 {imshow}
       13    0.291    0.022    0.303    0.023 {built-in method _imp.create_dynamic}
      287    0.227    0.001    0.227    0.001 {findContours}
      287    0.125    0.000    0.125    0.000 {dilate}
      574    0.116    0.000    0.116    0.000 {putText}
      288    0.116    0.000    0.116    0.000 {cvtColor}
```

3.4.4 Recompile opencv para agregar soporte de CUDA. Una vez finalizada la instalación, utilizando el ambiente virtual de python ya creado, ejecute la aplicación como se muestra en el repositorio de github del proyecto pyhton-opencv-cuda. Utilize el video capturado en el paso 3.3.5.

- Se obtuvieron las siguientes imágenes al ejecutar la aplicación indicada:

$ python cpu-opt_flow.py
$ python gpu-opt_flow.py

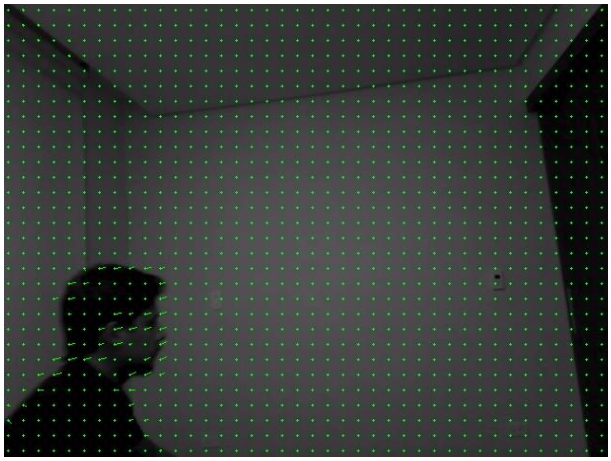


Imagen obtenida de gpu-opt_flow.py (frame 90).

Imagen obtenida de cpu-opt_flow.py (frame 150).

3.4.5 Utilice el profiler de Nvidia “nvprof” para obtener métricas de CUDA al correr la aplicación gpu-opt_flow.py y generar un archivo con los resultados de perfilado.

- Se utilizó el siguiente comando para obtener las métricas de la ejecución gpu-opt_flow.

$ sudo /usr/local/cuda/bin/nvprof –log-file nvprof_jetson_gpu.log python gpu-opt_flow.py

- Según el reporte, las tres funciones que más tiempo consumieron son:

Tabla 3.4.1 Funciones que más tiempo consumieron

Función	Tiempo (%)
cudaMallocPitch	41.83%
cudaFree	34.95%
cudaStreamSynchronize	17.23%

41.83% 23.3283s 11193 2.0842ms 17.239us 12.3457s cudaMallocPitch
34.95% 19.4934s 11193 1.7416ms 13.437us 86.503ms cudaFree
17.23% 9.60743s 16359 587.29us 3.4900us 87.465ms cudaStreamSynchronize

3.4.6 Vuelva a perfilar la aplicación, pero esta vez, investigue las opciones para generar un reporte de las actividades del GPU de manera cronológica. Además, agregue la opción necesaria para limitar la ejecución del perfilador a 10 segundos.

- Se presentaron problemas al tratar de ejecutar nvprof como “sudo”. Para solucionar lo anterior se creó un script (envMount.sh) que inicializa el ambiente virtual y ejecuta el profiler.

```
#!/bin/bash

export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
source /usr/local/bin/virtualenvwrapper.sh
workon cv
python gpu-opt_flow.py
```

Script envMount.sh

- Se utilizó el siguiente comando para obtener las métricas requeridas:

$ sudo /usr/local/cuda/bin/nvprof --log-file vprof_jetson_gputrace_10sec.log --profile-child-processes ./envMount.sh

3.4.7 Utilice nvprof para intentar obtener métricas con la aplicación cpu-out_flow.py. Porque no es posible obtener resultados?

En este caso no es posible obtener resultados porque no se está ejecutando ningún cuda. En este caso solamente se está utilizando el cpu para ejecutar el programa.