Visão computacional

Visão computacional para todos

Aprenda como obter desempenho e precisão através do uso de todos os recursos disponíveis em sua infraestrutura.

por Alessandro de Oliveira Faria (Cabelo)

Com o avanço da tecnologia de hardware, os sistemas conseguem evoluir com uma velocidade inacreditável. Da mesma forma isso acontece com a tecnologia de visão computacional de código aberto, onde podemos citar a biblioteca openCV [1] (*Open Source Computer Vision*), que na sua versão atual proporciona processamento de imagens utilizando todos os recursos disponíveis à nossa volta, ou seja: CPU, processamento paralelo, GPU e processadores ARM com a badalada plataforma Android, presente nos celulares e tablets. Agora, projetos de robótica, biometria, realidade aumentada, reconhecimento de padrão e/ou processamento de imagens contam com todo esse artefato tecnológico para desenvolver soluções.

A openCV evoluiu muito, embora o processamento de video em tempo real e o processamento de imagens, sempre serão tarefas pesadas, uma vez que as câmeras estão chegando ao mercado com resoluções e definições cada vez mais altas.

Figura 1 Reconhecimento de rostos com programação paralela.

Neste contexto, somente será possível obter desempenho e precisão se utilizarmos todos os recursos disponíveis em nosso hardware, pois o processamento matemático exige grande consumo computacional da CPU, mas agora, em pleno século XXI, contamos com algumas técnicas que tornarão o resultado menos propenso à sobrecarga de sistemas como um todo.

Programação multinúcleo

A programação paralela é necessária para obter o máximo desempenho no que tange à computação de alto desempenho. Para isto existem algumas alternativas como openMP ou TBB (*Threading Building Blocks*) [2] da Intel, que se encarregam da "adaptação" do software ao ambiente, ou seja, determina o número ideal de threads (processos), tornando uma realidade a programação paralela e aproveitando os recursos de hardware de forma mais inteligente.

Assim, utilizando uma dessas alternativas, podemos explorar o po-

tencial do processamento multinuclear sem a necessidade de mágica. Outra grande vantagem é a compatibilidade entre as threads POSIX e Windows. A instalação é bem simples: basta acessar o item download no site oficial e baixar a versão 3 ou superior. Em seguida, descompacte o arquivo e execute a instalação por meio do comando `make`, como no exemplo a seguir:

```
$ wget http://www.
➥ threadingbuildingblocks.org/
➥ uploads/78/154/3.0/
➥ tbb30_018oss_src.tgz
$ tar -zxvf tbb30_018oss_src.tgz
$ make
$ cd examples/
$ make
```

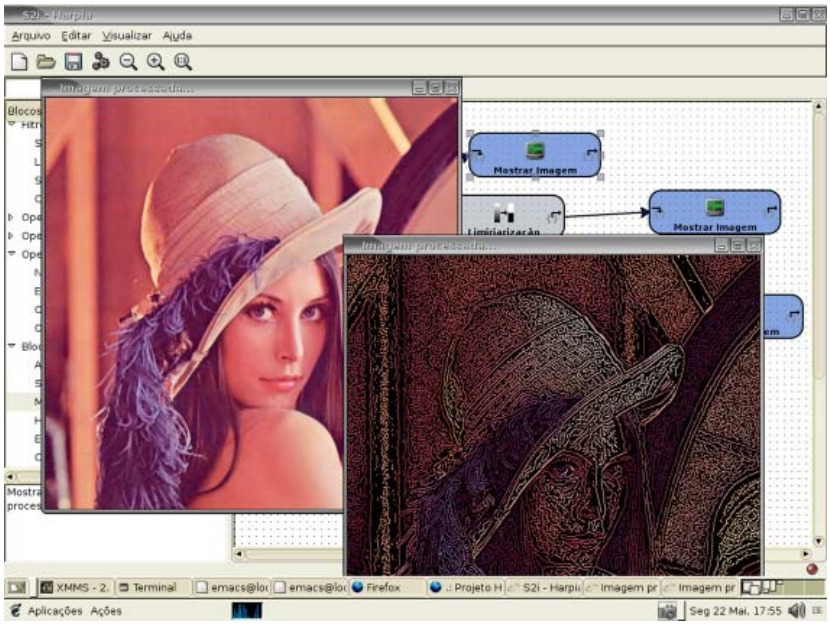

Figura 2 Processamento de imagens com Harpia.

GPU

Para continuar a incansável gincana com o objetivo de obter leveza e suavidade em sistemas de visão computacional, é saudável, juntamente com a programação paralela, fazer uso orquestrado das GPU e CPUs. Resumidamente, 30 fps foi o número máximo obtido com uma rotina convencional escrita em C++ para localizar um rosto diante do video ao vivo (*facefinder*). A mesma rotina compilada para trabalhar com GPU chegou a aproximados 100 fps e com a programação paralela foi possível atingir aproximados 170 fps (**figura 1**) [3].

Como escrevi na revista **Linux Magazine** #64, de março de 2010, no artigo sobre GPU, escrever códigos para processar visão computacional em fluxos de video ao vivo é uma tarefa morosa em função do grande consumo de processamento matemático. Até mesmos os processadores atuais precisam de apoio para não gerar sobrecarga no sistema. Não podemos esquecer que trabalhar com fluxo de video significa analisar 30 quadros por segundo em dimensões mínimas de 640x480. É neste momento que entram as técnicas utilizando a programação multinuclear com a GPU. Neste documento usei a tecnologia NVIDIA® CUDA™ para processar imagens utilizando a GPU.

Mão na massa

A biblioteca openCV, atualmente na versão 2.3, é simples de utilizar e poderosa. Seus recursos e flexibilidade, porém, tornam um pouco trabalhoso o aprendizado. Este software foi desenvolvido pela Intel em 2000, é multiplataforma e livre para fins de pesquisa e comerciais – claro que seguido da licença BSD Intel. Possui módulos de processamento de imagens e videos (leitura e gravação), álgebra linear, interface gráfica básica proporcionando sistema de janelas independentes, manipulação do mouse e teclado e mais de 2500 algoritmos de visão computacional, como filtros de imagem, calibração de câmera, reconhecimento de objetos, análise estrutural e outros. Foi originalmente desenvolvida nas linguagens de programação C/C++, mas atualmente suporta as linguagens Java e Python, entre outras.

O software Harpia [3] é um projeto aprovado dentro do edital CT-INFO 2003 – Software Livre da FINEP. Nele há uma interface RAD sob a forma de diagramas de blocos, que permite a criação rápida em C de programas com visão computacional. Basta arrastar os recursos representados por cada bloco e interligá-los com um fluxo lógico funcional. Assim, o processo de estudo fica mais intuitivo, uma vez que o pacote exporta todo o diagrama em código C. Resumidamente, este projeto torna acessível a tecnologia openCV de maneira muito eficiente (**figura 2**).

Instalação na plataforma x86

Efetue o download dos códigos-fontes na página do projeto hospedada no SourceForge [5] e obtenha o pacote `OpenCV-2.3.1a.tar.bz2` ou superior. As instruções de compilação e configuração a seguir foram testadas na versão 2.3.0 e 2.3.1, as demais versões poderão sofrer pequenas modificações. Após o download, descompacte o pacote:

```
$ wget http://sourceforge.net/
➥ projects/opencvlibrary/files/
➥ opencv-unix/2.3.1/
➥ OpenCV-2.3.1a.tar.bz2
$ tar -jxvf
➥ tars/OpenCV-2.3.1a.tar.bz2
```

Listagem 1: Exemplo de cmake da biblioteca openCV

```
01 General configuration for opencv 2.3.1 =============================
02
03     Built as dynamic libs?:     YES
04     Compiler:                   /usr/bin/c++
05     C++ flags (Release):          -Wall -Wno-long-long -pthread
➥ -ffunction-sections  -O3 -DNDEBUG  -fomit-frame-pointer -msse
➥-msse2 -msse3 -mssse3 -msse4.1 -msse4.2 -DNDEBUG
06     C++ flags (Debug):            -Wall -Wno-long-long -pthread
➥-ffunction-sections  -g  -O0 -DDEBUG -D_DEBUG -ggdb3
07     Linker flags (Release):
08     Linker flags (Debug):
09
10   GUI:
11     QT 4.x:                     YES
12     QT OpenGL support:          YES
13
14   Media I/O:
15     ZLib:                       YES
16     JPEG:                       TRUE
17     PNG:                        TRUE
18     TIFF:                       TRUE
19     JPEG 2000:                  TRUE
20     OpenEXR:                    NO
21     OpenNI:                     NO
22     OpenNI PrimeSensor Modules: NO
23     XIMEA:                      NO
24
25   Video I/O:
26     DC1394 1.x:                 NO
27     DC1394 2.x:                 YES
28     FFMPEG:                     YES
29       codec:                    YES
30       format:                   YES
31       util:                     YES
32       swscale:                  YES
33       gentoo-style:             YES
34     GStreamer:                  NO
35     UniCap:                     NO
36     PvAPI:                      NO
37     V4L/V4L2:                   1/1
38     Xine:                       YES
39
40   Other third-party libraries:
41     Use IPP:                    NO
42     Use TBB:                    YES
43     Use ThreadingFramework:     NO
44     Use Cuda:                   YES
45     Use Eigen:                  NO
46
47   Interfaces:
48     Python:                     YES
49     Python interpreter:         /usr/bin/python2.7 -B (ver 2.7)
50     Python numpy:               YES
51     Java:                       NO
52
53   Documentation:
54     Sphinx:                     NO
55     PdfLaTeX compiler:          /usr/bin/pdflatex
56     Build Documentation:        NO
57
58   Tests and samples:
59     Tests:                      YES
60     Examples:                   YES
61
62   Install path:                 /usr/local
63
64   cvconfig.h is in:             /tmp/OpenCV-2.3.1/opencv.build
65 -----------------------
66 Configuring done
67 Generating done
68 Build files have been written to: /tmp/OpenCV-2.3.1/opencv.build
```

Feito isso, inicie a compilação do código-fonte:

```
$ cd OpenCV-2.3.1/
$ mkdir opencv.build
$ cd opencv.build
```

Parâmetros de compilação

Os parâmetros de compilação ou diretivas de compilação do comando `cmake` determinarão quais recursos estarão disponíveis na biblioteca openCV. A seguir temos o comando que devemos executar (instalando previamente as bibliotecas TBB e CUDA). Há diversos parâmetros, porém os mencionados abaixo são os principais para obtermos sucesso nos teste iniciais.

```
$ cmake  -DBUILD_DOCS=ON \
          -DCMAKE_BUILD_TYPE=
➥ RELEASE -DBUILD_LATEX_DOCS=ON  \
         -DBUILD_OCTAVE_SUPPORT=ON
➥ -DBUILD_PYTHON_SUPPORT=ON \
          -DBUILD_SWIG_PYTHON_
➥ SUPPORT=ONF -DBUILD_TESTS=ON \
          -DENABLE_OPENMP=OFF
➥ -DENABLE_PROFILING=OFF \
         -DBUILD_PYTHON_SUPPORT=ON
➥ -DBUILD_NEW_PYTHON_SUPPORT=ON \
          -DBUILD_EXAMPLES=ON
➥ -DINSTALL_C_EXAMPLES=ON \
          -DINSTALL_OCTAVE_
➥ EXAMPLES=ON  -DINSTALL_PYTHON_
➥ EXAMPLES=ON \
          -DWITH_1394=ON
➥ -DWITH_TBB=ON -DWITH_CUDA=ON
➥ -DWITH_FFMPEG=ON \
          -DWITH_GSTREAMER=OFF
➥ -DWITH_QT=ON -DWITH_GTK=ON \
          -DWITH_JASPER=ON
➥ -DWITH_JPEG=ON -DWITH_PNG=ON
➥ -DWITH_TIFF=ON \
          -DWITH_V4L=ON
➥ -DWITH_XINE=ON -DENABLE_SSE=ON
➥ -DENABLE_SSE2=ON \
          -DENABLE_SSE3=ON
➥ -DENABLE_SSSE3=ON
➥ -DENABLE_SSE41=ON \
          -DENABLE_SSE42=ON
➥ -DWITH_QT=ON -DWITH_QT_OPENGL=ON \
          -DCUDA_NPP_LIBRARY_
➥ ROOT_DIR=/usr/local/cuda/NPP/
➥ SDK/   ..
```

Veremos a seguir o motivo da utilização das principais diretivas

de compilação utilizadas no comando acima:

- `BUILD_DOCS`, `BUILD_LATEX_DOCS`: habilite para disponibilizar toda a documentação presente no pacote openCV. Mesmo aos usuários experientes, sugiro utilizar esta opção para comparação com a versão anterior.
- `BUILD_OCTAVE_SUPPORT`, `BUILD_PYTHON_SUPPORT`: utilize estas opções para habilitar o suporte da biblioteca na linguagem Python e também no Octave.
- `BUILD_TESTS`, `BUILD_EXAMPLES`, `INSTALL_C_EXAMPLES`, `INSTALL_OCTAVE_EXAMPLES`, `INSTALL_PYTHON_EXAMPLES`: estes itens disponibilizam os respectivos exemplos nas linguagens C, Python e Octave.
- `ENABLE_OPENMP`, `WITH_TBB`: é muito racional que, ao habilitar a opção TBB para utilizar os recursos de programação paralela, a utilização da biblioteca OpenMP seja desabilitada.
- `WITH_1394`, `WITH_CUDA`, `WITH_FFMPEG`: como o assunto é processamento de imagem, a interface FireWire (IEEE 1394), as bibliotecas `ffmpeg` e CUDA são imprescindíveis.
- `WITH_JPEG`, `WITH_PNG`, `WITH_TIFF`: sugiro habilitar os principais tipos de imagem para obter total compatibilidade de processamento com arquivos disponíveis no seu disco.
- `WITH_V4L`: este item define os dispositivos de video captura USB ou PCI compatíveis com a API do kernel V4L 1/2 que serão utilizados.
- `ENABLE_SSE`, `ENABLE_SSE2`, `ENABLE_SSE3`, `ENABLE_SSSE3`, `ENABLE_SSE41`, `ENABLE_SSE42`: para processamento de imagens e cálculo de número de ponto flutuante, as instruções SSE são mais do que obrigatórias.
- `CUDA_NPP_LIBRARY_ROOT_DIR=/usr/local/cuda/NPP/SDK/`: esta linha informa a localização do CUDA SDK. Esta é obtida durante a instalação do pacote SDK da NVIDIA.
- `WITH_QT`: exemplos com a biblioteca QT estão disponíveis no pacote, então não hesite em habilitar esta opção.

Se tudo estiver corretamente instalado e configurado, o resultado será semelhante ao exemplo na **listagem 1**. Faça uma comparação e se algo estiver desabilitado, verifique as dependências de pacotes em seu sistema.

Para iniciar na íntegra a compilação, efetue o comando `make`. No término da compilação basta, como super-usuário, utilizar o comando `make install` e `ldconfig`.

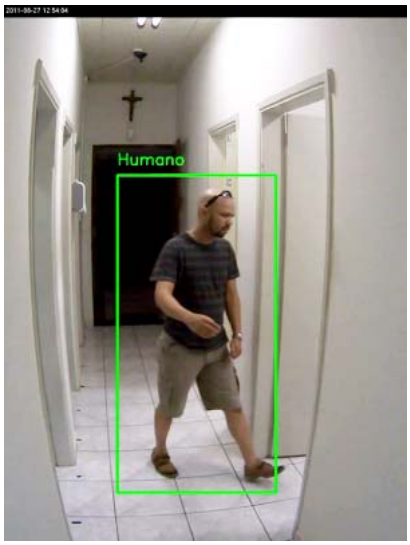

Figura 3 Detecção de um corpo humano em um video em tempo real.

Figura 4 Imagem submetida ao processo de binarização.

Figura 5 Motorola Dext utilizado para processar imagens.

Figura 6 Imagem HelloAndroid.png criada pelo aplicativo.

Existem inúmeros assuntos, funções, recursos e exemplos para relatar sobre a tecnologia openCV. O código fonte em C++ presente na pasta `samples/cpp` (cujo nome é `peopledetect.cpp`) demonstra o potencial para detectar um corpo humano presente em imagens ou video em tempo real com pequenas adaptações (**figura 3**) [6]. Não será possível neste documento detalhar todos os recursos e possibilidades da biblioteca, então irei direto para os assuntos emergentes, ou seja, GPU e Android.

GPU e openCV em ação

Para desmistificar o uso da GPU com a biblioteca openCV, temos abaixo um simples programa para a binarização da imagem, muito utilizado em realidade aumentada e em muitos outros aplicativos que requerem processamento de imagem.

Listagem 2: Binarização de imagem com o openCV

```
01 #include <iostream>
02 #include “opencv2/opencv.hpp”
03 #include “opencv2/gpu/gpu.hpp”
04
05 int main (int argc, char* argv[])
06 {
07     try
08    {
09         cv::Mat src_host = cv::imread(argv[1],
➥ CV_LOAD_IMAGE_GRAYSCALE);
10         cv::gpu::GpuMat dst, src;
11         src.upload(src_host);
12
13        cv::gpu::threshold(src, dst, 128.0, 255.0, CV_THRESH_BINARY);
14
15         cv::Mat result_host = dst;
16         cv::imshow(“Result”, result_host);
17         cv::waitKey();
18     }
19     catch(const cv::Exception& ex)
20     {
21         std::cout << “Error: “ << ex.what() << std::endl;
22     }
23     return 0;
24 }
```

Listagem 3: Conversão de QImage para IplImage

```
01 static IplImage* qImage2IplImage(const QImag&E qImage)
02 {
03     int width = qImage.width();
04     int height = qImage.height();
05     IplImage *img = cvCreateImage(cvSize(width, height),
➥ IPL_DEPTH_8U, 3);
06     char * imgBuffer = img->imageData;
07     int jump = (qImage.hasAlphaChannel()) ? 4 : 3;
08     for (int y=0;y<img->height;y++){
09       QByteArray a((const char*)qImage.scanLine(y),
➥ qImage.bytesPerLine());
10       for (int i=0; i<a.size(); i+=jump){
11           imgBuffer[2] = a[i];
12           imgBuffer[1] = a[i+1];
13           imgBuffer[0] = a[i+2];
14           imgBuffer+=3;
15       }
16     }
17   return img;
18 }
```

Para desmistificar o uso da GPU com a biblioteca openCV, temos abaixo um simples programa para a binarização da imagem, muito utilizado em realidade aumentada e em muitos outros aplicativos que requerem processamento de imagem.

No exemplo a seguir encontramos a imagem passada como parâmetro na linha de comando, que é carregada no objeto `Mat` e posteriormente enviada ao `GpuMat` com o método `upload`.

Com a imagem já presente no objeto `src` (`GpuMat`), a função `threshold` é processada na GPU com a chamada `CV_THRESH_BINARY`. O resultado do processamento é armazenado no objeto `dst` e, por sua vez, copiado no objeto `result_host` (`Mat`) para exibição. Como podemos analisar na **listagem 2**, a biblioteca openCV abstraiu toda a complexidade (não tanta assim) da programação na GPU.

Para compilar o programa apresentado na **listagem 2**, utilize uma das sintaxes a seguir:

```
$ g++  `pkg-config --cflags
➥ opencv` -I/usr/local/cuda/
➥ include -o teste teste.cpp
➥ `pkg-config --libs opencv`
➥ -lopencv_gpu;
```

ou

```
$ g++ -I/usr/include/opencv
➥ -I/usr/local/cuda/include -o
➥ teste teste.cpp -L/usr/lib
➥ -lopencv_core -lopencv_imgproc
➥ -lopencv_highgui -lopencv_ml
➥ -lopencv_video -lopencv_
➥ features2d -lopencv_calib3d
➥ -lopencv_objdetect -lopencv_
➥ contrib -lopencv_legacy
➥ -lopencv_flann -lopencv_gpu;
```

Agora execute o programa recém compilado seguido do nome do arquivo imagem que será submetido ao processamento de binarização. Se tudo estiver funcionando corretamente, teremos como resultado algo similar à **figura 4**.

```
$ ./teste image.png
```

Um pouco de QT

O pacote openCV suporta a biblioteca QT, mas geralmente o principal objetivo ou interesse é a conversão da classe

Figura 7 Aplicativo instalado no dispositivo Android.

Figura 8 Aplicativo em execução.

`QImage` para a estrutura `IplImage` e vice e versa. O real motivo dessa necessidade é o fato de que quase todo processamento de imagem na biblioteca openCV utiliza a estrutura `IplImage`.

A estrutura `IplImage` carrega diversas informações da imagem, como a quantidade de canais de cores, profundidade de bits, dimensão, tamanho da linha alinhada em bits e um vetor com a imagem `Data`. Como a `QImage` é uma classe utilizada para armazenar, manipular e visualizar imagens, nada mais coerente do que obter técnicas para trocar imagens entre esses tipos. As **listagens 3** e **4** apresentam uma receita de bolo muito simples que pode ser encontrada facilmente em fóruns de discussão.

Listagem 4: Conversão de IplImage para QImage

```
01 static QImage IplImage2QImage(const IplImage *iplImage)
02 {
03     int height = iplImage->height;
04     int width = iplImage->width;
05     if  (iplImage->depth == IPL_DEPTH_8U &&
➥ iplImage->nChannels == 3)
06     {
07       const uchar *qImageBuffer =
➥ (const uchar*)iplImage->imageData;
08       QImage img(qImageBuffer, width, height,
➥ QImage::Format_RGB888);
09       return img.rgbSwapped();
10     } else if  (iplImage->depth == IPL_DEPTH_8U &&
➥ iplImage->nChannels == 1){
11         const uchar *qImageBuffer =
➥ (const uchar*)iplImage->imageData;
12         QImage img(qImageBuffer, width, height,
➥ QImage::Format_Indexed8);
13         QVector<QRgb> colorTable;
14         for (int i = 0; i < 256; i++){
15             colorTable.push_back(qRgb(i, i, i));
16         }
17         img.setColorTable(colorTable);
18         return img;
19     }else{
20       qWarning() << “Image cannot be converted.”;
21       return QImage();
22     }
23 }
```

Android e OpenCV

Como se não bastasse tanta aplicabilidade, aparelhos como o Atrix, da Motorola, possuem processadores de 2 núcleos e Tegra 2 [7], ou seja: tudo o que mencionamos até agora neste documento está na palma das nossas mãos. Mas obter desempenho em celulares potentes de última geração é fácil. O problema está em conseguir o mesmo efeito em dispositivos mais antigos. Sendo assim, utilizei o Motorola DEXT, conhecido no exterior como CLIQ, que possui um modesto processador de 528 Mhz (**figura 5**) [8].

Para iniciar a compilação na plataforma Android, defina as variáveis de ambiente `ANDROID_NDK` e `ANDROID_NDK_TOOLCHAIN` conforme o exemplo abaixo:

```
$ export ANDROID_NDK=/home/
➥ cabelo/android-ndk-r5c
$ export ANDROID_NDK_TOOLCHAIN_
➥ ROOT=/home/cabelo/
➥ android-ndk-r5c/toolchains
```

Entre no diretório `android` e execute o script `cmake` com `sh ./scripts/cmake_android_armeabi.sh`.

Caso o seu equipamento Android suporte a tecnologia ARM® NEON™, que permite a aceleração multimídia, video, encode e decode 2D/3D, uti-

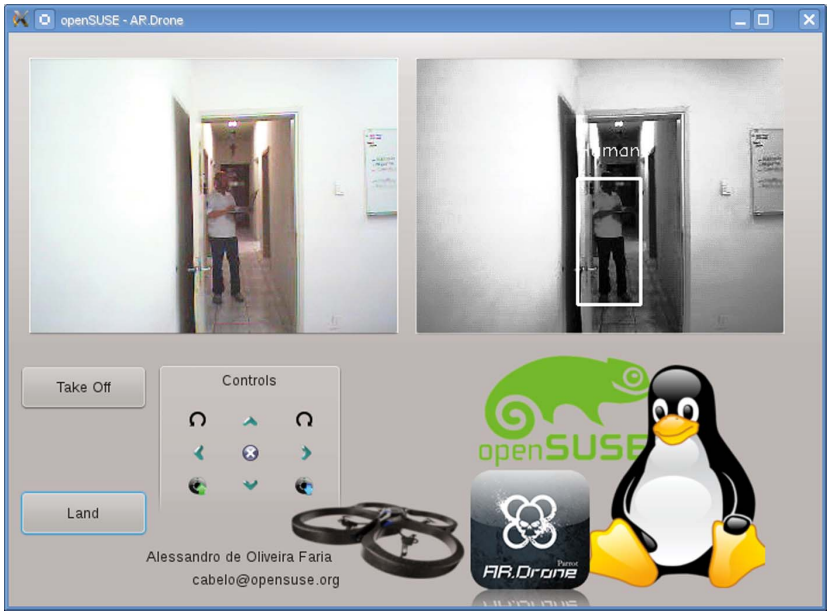

Figura 9 Técnicas de reconhecimento padrão de humanos em videos.

lize o o script `cmake_android_neon.sh` para executar esta tarefa.

Resta apenas executar o tradicional ritual de compilação: `make`, `make install`. Execute os comandos conforme o exemplo seguinte e vá tomar outro café.

```
$ cd build_armeabi
$ make
```

Após alguns minutos utilize o comando `make install` para concluir a instalação. Para testar os exemplos, basta entrar na pasta `bin` e fazer a festa.

Veremos agora como instalar um programa nativo e um pacote `.apk` para Android.

Instalação e execução de programa nativo consistem em cópia, atribuição do direito de escrita e execução na íntegra.

```
$ adb push HelloAndroid /data
$ adb shell chmod 777
➥ /data/HelloAndroid
$ adb shell /data/HelloAndroid
```

Se o programa foi executado corretamente, no cartão microSD teremos a imagem mostrada na **figura 6**.

```
$ adb pull /mnt/sdcard/
➥ HelloAndroid.png
$ xv HelloAndroid.png
```

A instalação do pacote `.apk` é tranquila; basta invocar o comando `adb install` e pronto, o aplicativo aparecerá no seu aparelho restando apenas a tarefa de executá-lo (**figuras 7** e **8**).

```
$ adb install
➥ tutorial-1-addopencv.apk
```

Conclusão

Vou encerrar aqui, pois este documento é apenas um átomo da ponta do iceberg no que tange à visão computacional com openCV. Estou, em minhas poucas horas vagas, escrevendo um aplicativo de visão computacional utilizando openCV e a API do AR.Drone, ou seja, robótica livre com openCV.

Em breve (não tão breve assim) pretendo disponibilizar toda documentação e tutorial de como fazer um quadricóptero AR.Drone acompanhar um corpo humano utilizando técnicas de reconhecimento de padrão (**figura 9**) [9]. ■

Mais informações

[1] openCV: http://opencv.willowgarage.com/wiki/

[2] Threading Building Blocks: http://www.threadingbuildingblocks.org/

[3] NVIDIA CUDA 4, GPU e openCV no openSUSE 11.4: http://www.youtube.com/watch?v=EkuFlpdFkGk

[4] Projeto Harpia: http://s2i.das.ufsc.br/harpia/home.html

[5] Open Computer Vision Library: http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.3.1/

[6] Contando e reconhecendo pessoas no openSUSE Linux com openCV na CPU: http://www.youtube.com/watch?v=aVA5PvlaAp0

[7] Tegra Android Development Pack: http://developer.nvidia.com/tegra-android-development-pack

[8] OpenCV + Android: http://www.youtube.com/watch?v=rFMU8sQTp8U

[9] AR.Drone with openCV in the openSUSE 11.4: http://www.youtube.com/watch?v=Rk9usAaM7po

O autor

Alessandro Faria (Cabelo) é sócio-proprietário da NETi TECNOLOGIA, fundada em Junho de 1996 (*http://www.netitec.com.br*) e especializada em desenvolvimento de software e soluções biométricas. Consultor Biométrico na tecnologia de reconhecimento facial, atuando na área de tecnologia desde 1986. Leva o Linux a sério desde 1998 com desenvolvimento de soluções open-source, é membro colaborador da comunidade Viva O Linux e mantenedor da biblioteca open-source de vídeo captura, entre outros projetos.

Gostou do artigo?

Queremos ouvir sua opinião. Fale conosco em cartas@linuxmagazine.com.br

Este artigo no nosso site: http://lnm.com.br/article/6703