

# Template Matching

## [Aula 3 parte 2. Início.]

**Resumo:** Nesta aula, desenvolveremos as fases 3 e 4 do projeto. Na fase 3, faremos um programa que detecta uma certa placa nos quadros de um vídeo. Usaremos processamento paralelo para acelerar o processamento. Na fase 4, faremos um sistema cliente-servidor que faz o carrinho seguir a placa.

**Cuidado:** Em todos os demais componentes do nosso projeto, o componente desenvolvido ou funciona ou não funciona – é fácil perceber quando tem algum problema. No casamento de modelos, existem diferentes “graus de funcionamento”: pode não funcionar, pode funcionar mal, pode funcionar mais ou menos, pode funcionar bem a maior parte do tempo, até funcionar muito bem. Se template matching não estiver funcionando perfeitamente, o carrinho não conseguirá seguir a placa dependendo da iluminação, da “poluição visual” do ambiente, etc.

**Nota 2024 e 2025:** Nos anos anteriores, temos utilizado template matching para detectar a placa na câmera, que tem funcionado bastante bem. Porém, quase certamente, é possível detectar a placa de forma ainda mais robusta usando redes neurais convolucionais. Só que, para isso, é necessário dispor de uma quantidade grande de imagens de treino. Assim, peço que vocês gravem vídeos que possam ser usados para treinar rede neural convolucional, a ser usado nos próximos anos. Cada grupo deve gravar três vídeos da visão da câmera (sem nenhuma anotação), de aproximadamente 1 minuto cada um, nas fases 4 (carrinho seguindo placa), 6 (controle automática do carrinho) e 7 (controle automática/manual). Os vídeos devem ter placas visíveis. Coloquem os vídeos no formato AVI no Google Drive:

<https://drive.google.com/drive/folders/1Sc8kmIPNH4HIiz-6PfyhXVcr3RkaXqyug>

com nome, por exemplo:

FulanoDeTal\_CiclanoDeQual\_Fase4.avi

Exemplos desse tipo de vídeo são *capturado2.avi* e *capturado3.avi* do [site](#).

## 1 Introdução

A parte teórica do template matching já foi vista na disciplina teórica PSI-3471 Fundam. Sist. Eletrônicos Inteligentes ([tmatch-ead.pdf](#) ou [tmatch-ead.odt](#) no site [www.lps.usp.br/hae/apostila](http://www.lps.usp.br/hae/apostila)). Em especial, preste atenção em:

- 1) Como obter template matching invariante somente ao brilho (correlação cruzada ou **CC - CV\_TM\_CCORR**) e invariante por brilho e contraste (coeficiente de correlação normalizada ou **NCC - CV\_TM\_CCOEFF\_NORMED**).
- 2) O que se deve fazer para que alguns pixels sejam considerados “don’t care”.

*Nota:* OpenCV2 denomina os modos de template matching de:

CV\_TM\_CCORR e CV\_TM\_CCOEFF\_NORMED.

OpenCV4 os chama de:

TM\_CCORR e TM\_CCOEFF\_NORMED.

OpenCV3 aceita as duas denominações.

## Resumo de casamento de modelos

Para facilitar, escrevo abaixo a “receita de bolo” de como fazer *template matching*, sem explicar a teoria:

1) Em OpenCV, há a função *matchTemplate*, cuja sintaxe é:

```
C++: void matchTemplate(InputArray image, InputArray templ,
                       OutputArray result, int method)
```

```
Ex: matchTemplate(I, T, R, CV_TM_CCORR)
```

```
Ex: matchTemplate(I, T, R, CV_TM_CCOEFF_NORMED)
```

onde  $I$  é a imagem onde será feita a busca,  $T$  é o modelo a ser procurado e  $R$  é o resultado. Essa função trabalha no modo “valid”, onde o resultado  $R$  é menor do que a imagem de busca  $I$ , e o resultado do casamento é colocado no canto superior esquerdo da ocorrência do modelo em  $R$ . Veja a figura 1 e leia o manual de [OpenCV](#) para maiores detalhes.

Vamos supor que as imagens  $I$  e  $T$  são em ponto flutuante com valores de 0 (preto) a 1 (branco). Neste caso, os valores de saída em  $R$  irão de -1 a +1 e o local de ocorrência será marcado com um pico (valor alto).

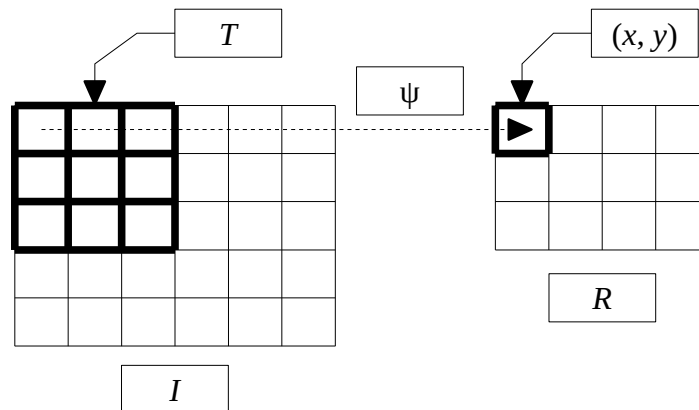


Figura 1: Template matching no modo “valid”.

2) Normalmente, queremos que o resultado da correlação fique no centro do modelo em  $R$ , em vez de canto superior esquerdo, e que a imagem de saída  $R$  tenha o mesmo tamanho que a entrada  $I$  (modo “same”). A função `matchTemplateSame` de [raspberry.hpp](#) faz isso (figura 2).

```
Mat_<FLT> matchTemplateSame(Mat_<FLT> I, Mat_<FLT> T, int method,
                           FLT backg=0.0);
Ex: R=matchTemplateSame(I, T, CV_TM_CCOEFF_NORMED, 0.0)
```

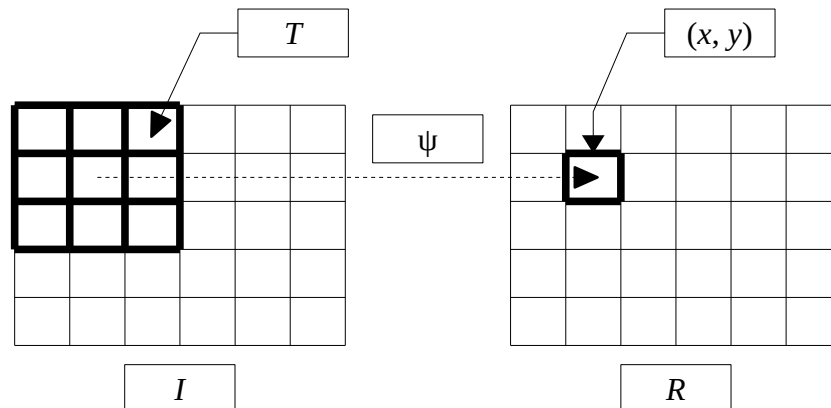


Figura 2: Template matching no modo “same”.

onde `backg` é o valor com que a função irá preencher os pixels fora do domínio da imagem de saída da função `matchTemplate` original, para aumentar o tamanho da imagem de saída. Note que `matchTemplateSame` só aceita imagens em ponto flutuante 32 bits (FLT).

3) Para aplicar o casamento de modelos invariante somente por brilho (correlação cruzada ou **CC**), você deve usar `method=CV_TM_CCORR`. Este método detecta preferencialmente um objeto de alto contraste, mesmo que o seu formato difira um pouco do formato do modelo. Além disso, deve pré-processar o modelo:

```
T = somaAbsDois( dcReject(T) )
```

As funções `dcReject` e `somaAbsDois` estão em [raspberry.hpp](#). Veja a apostila *tmatch-ead* para maiores detalhes.

4) Para aplicar o casamento de modelos invariante por brilho e contraste (coeficiente de correlação normalizada ou **NCC**) você deve usar `method=CV_TM_CCOEFF_NORMED`. Este método detecta objetos com o formato parecido ao modelo buscado, mesmo que sejam de baixo contraste. Neste caso, não precisa pré-processar o modelo, se não quiser usar pixels “don’t care”. Para usar pixels “don’t care” é necessário pré-processar o modelo com `dcReject` com dois parâmetros (veja item 6 abaixo).

5) Estou usando ambos os métodos (CC e NCC) para detectar a placa. Primeiro, faço a detecção usando modo CC. Depois, verifico se a detecção é verdadeira usando modo NCC. O importante é levar em consideração de alguma forma os dois métodos (CC e NCC). Uma detecção só será verdadeira se os dois métodos resultarem em correlações altas na mesma posição ( $l, c$ ) e escala  $s$ .

6) Para especificar alguns pixels como “don’t care”, você deve fornecer o nível de cinza a ser considerado “don’t care” como o segundo parâmetro da função `dcReject`. Na figura 3, queremos considerar a terceira letra como “don’t care”. Para isso, todos os pixels em torno da terceira letra do modelo `letramore3.pgm` foram pintados em cinza com valor 128. Para dizer que pixels com valor 128 são “don’t care”, forneça o segundo parâmetro à função `dcReject`:

```
T = somaAbsDois( dcReject(T, 128.0/255.0) )
```

O valor float 128.0/255.0 é o nível de cinza 128 convertido para ponto flutuante entre 0 e 1. O modelo  $T$  assim pré-processado irá considerar os pixels originalmente com valor 128 como “don’t care” tanto usando método CC como NCC. Veja na apostila *tmatch-ead* para mais detalhes.

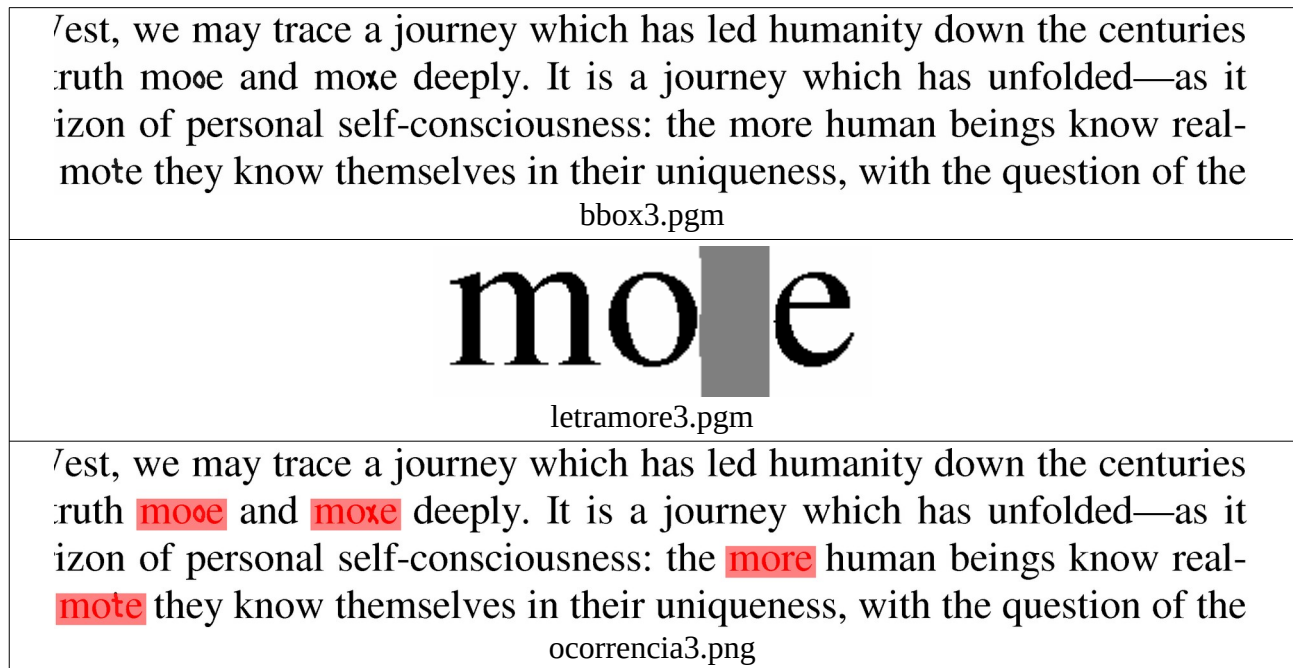


Figura 3: Template matching com pixels “don’t care”.

*Nota:* Versões recentes da função *matchTemplate* de OpenCV possuem o conceito “máscara”, que é uma outra forma de especificar pixels “don’t care”.

## 2 Fase 3 do projeto

**[Lição de casa 2 da aula 3 – 5 pontos]** Faça um programa *fase3.cpp* que lê um vídeo 240×320 pixels (*capturado.avi*, *capturado2.avi* ou *capturado3.avi*), um modelo de placa (*quadrado.png*) e localiza a placa-modelo nos seus quadros, toda vez que a placa estiver entre aproximadamente 30 e 150 cm da câmera, gerando o vídeo de saída com a localização da placa (figura 4). Imprima também quantos quadros/segundo o seu programa conseguiu processar. Quadros/segundo deve ser medida sem mostrar o vídeo na janela do computador (imshow) pois mostrando vídeo em 30 fps, o processamento nunca irá passar dos 30 fps.

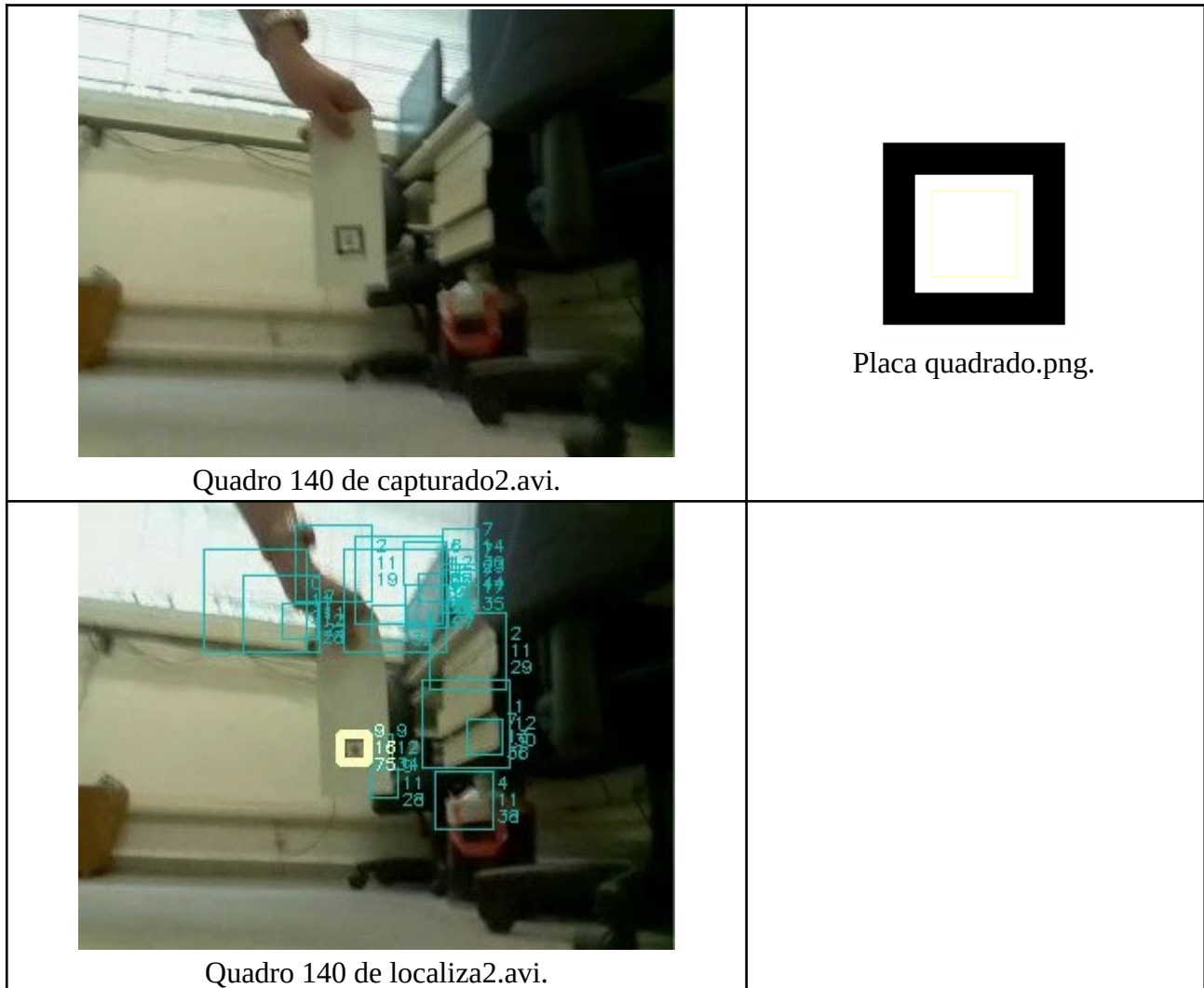


Figura 4: Localização da placa. Os 20 retângulos azuis são os 20 picos mais altos das correlações CC, separados por distância de pelo menos 10 pixels. Entre eles, foi escolhido o pico (amarelo) que tinha alto coeficiente NCC. O número 9 amarelo indica a escala (tamanho) da placa. Os números 16 e 75 indicam que os resultados dos template matchings CC e NCC deram 0,16 e 0,75 respectivamente.



Figura 5: Falso negativo - o programa não conseguiu localizar a placa, pois o quadro está muito borrado.

Executando o seu programa como abaixo:

```
$ fase3 capturado.avi quadrado.png seu_localiza.avi
```

deverá ler o vídeo *capturado.avi*, a imagem *quadrado.png* e gerar o vídeo *seu\_localiza.avi*.

Exemplos de saída estão em:

<http://www.lps.usp.br/hae/apostilaraspi/localiza.avi>

<http://www.lps.usp.br/hae/apostilaraspi/localiza2.avi>

<http://www.lps.usp.br/hae/apostilaraspi/localiza3.avi>

Nota: Em Linux, pode-se usar o programa *avidemux* para visualizar o vídeo quadro a quadro: <https://www.fosshub.com/Avidemux.html>.

### ***Don't care***

No interior da placa *quadrado.png*, há um quadrado menor amarelo pouco visível. Dentro do quadrado amarelo, será escrito à mão um dígito entre “0” e “9”. Durante a localização da placa, você deve considerar o conteúdo no interior do quadrado amarelo como “don't care”, para que o dígito manuscrito não atrapalhe a busca da placa.

Todos os pixels brancos dentro do quadrado amarelo possuem valores (255, 255, 255), enquanto que os pixels brancos fora do quadrado amarelo possuem valores (255, 255, 254) ou (255, 254, 255) - em ordem (b, g, r). Convertendo para float (FLT), apenas os pixels exatamente iguais a 1.0 (que tinham valor (255, 255, 255) antes da conversão) devem ser considerados “don't care”:

```
Mat_<FLT> T=somaAbsDois(dcReject(T,1.0));
```

Veja na figura 6 a diferença entre usar ou não “don't care”. Nas duas imagens, o pixel com o maior valor foi mapeado no branco e o pixel com o menor valor foi mapeado no preto. Sem usar “don't care”, há vários pixels com alto brilho dificultando localizar a placa, enquanto que usando “don't care”, praticamente somente a localização verdadeira da placa fica altamente brilhante.

Conversão de imagem colorida para float pode ser feita com o comando *converte* do [raspberrypi.hpp](http://raspberrypi.org):

```
Mar_<COR> C; Mat_<FLT> T; (...)
converte(C,T);
```

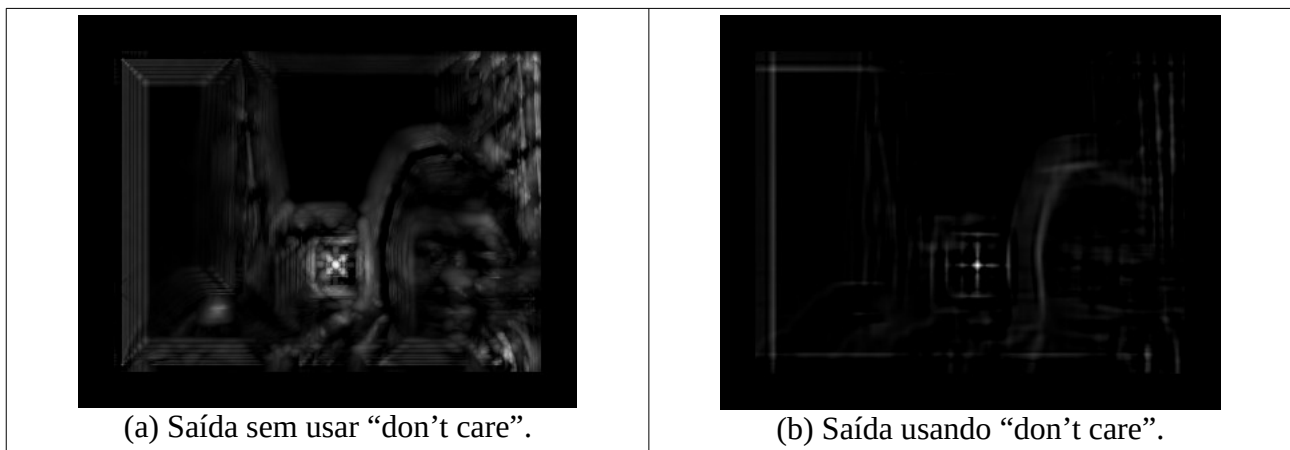


Figura 6: Diferença entre usar ou não pixels “don’t care”. Nas duas imagens, o pixel com o maior valor foi mapeado no branco e o pixel com o menor valor foi mapeado no preto. Sem usar “don’t care”, há vários pixels com alto brilho.

O seguinte código pré-processa o modelo para que alguns pixels sejam considerados “don’t care”.

```
//Le quadrado.png como imagem em ponto flutuante (0=preto, 1=branco)
int nl=240; int nc=320;
Mat_<COR> temp=imread("../quadrado.png",1);
if (temp.total()==0) erro("Erro leitura quadrado.png");
Mat_<FLT> T; converte(temp,T); //401x401
//Cuidado: Voce deve ler video como Mat_<COR>
//e converter para Mat_<FLT> usando converte(a,b)
//VideoCapture vi; ... Mat_<COR> a; Mat_<FLT> b; vi >> a; converte(a,b); ...
//Redimensione aqui T para a escala desejada, usando interpolação vizinho+próximo.
//Faz o pré-processamento, considerando pixels=1.0 como don't care
Mat_<FLT> T2=somaAbsDois(dcReject(T,1.0));
```

A função *converte* está no [raspberry.hpp](#):

```
void converte(Mat_<COR> ent, Mat_<FLT>& sai) {
    Mat_<Vec3f> temp; ent.convertTo(temp, CV_32F, 1.0/255.0, 0.0);
    cvtColor(temp, sai, CV_BGR2GRAY);
}
```

Você deve repetir os passos acima para cada escala desejada, obtendo um vetor de modelos pré-processados.

**Nota:** Recomenda-se variar escala em progressão geométrica (pense por quê).

**Nota importante:** Para fazer mudança de escala da placa, deve usar a interpolação vizinho mais próximo (modo *INTER\_NEAREST* da função *resize* do OpenCV). Isto assegura que na figura redimensionada não irá aparecer cores que não existiam na figura original (isto é, a cor 1.0 que indica “don’t care” não irá combinar com as cores dos pixels vizinhos).

## CC e NCC

Nota: A quantidade de falsos positivos (dizer que há placa quando não há) do seu programa deve ser muito pequena, caso contrário o carrinho pode começar a andar mesmo na ausência de uma placa. Da mesma forma, a quantidade de erros de localização (localizar placa numa posição incorreta) do seu programa deve ser muito pequena para que o carrinho não ande numa direção errada (atrás de algum objeto que “se parece” com a placa). Por outro lado, o seu programa pode cometer alguns falsos negativos (dizer que não há placa quando há), pois neste caso o carrinho estará dando “solavancos”, o que é menos grave do que os dois erros anteriores. A figura 5 mostra um caso de falso negativo devido ao borrão de movimento.

**Sugestão para levar em conta CC e NCC:** Para ter um casamento verdadeiro, deve ter tanto a saída CC e quanto NCC altas na mesma posição e na mesma escala. Estou usando a seguinte técnica para combinar template matching CC com NCC. Você pode fazer diferente.

A imagem-modelo da placa possui 401×401 pixels. Primeiro, construo 10 modelos em progressão geométrica entre escalas 0,1721 (69×69 pixels) e 0,0473 (19×19 pixels), usando reamostragem vizinho mais próximo. Faço o “pré-processamento” de cada um desses modelos (*dcReject* e *somaAbsDois*). Procuo todos esses 10 modelos na imagem usando casamento de modelos CC. Pego os 20 picos mais altos de correlação CC separados por pelo menos 10 pixels, com suas respectivas escalas. Estes 20 picos são “candidatos” ao casamento com o modelo da placa (quadrados ciano da imagem de saída). Calculo os casamentos NCC nas 20 posições, nas mesmas escalas que deram pico CC (*nota: é mais rápido fazer template matching NCC uma única vez e pegar as correlações nas 20 posições*). Entre as 20 correlações NCC, procuro a de maior valor. Se esta correlação for maior que 0,55, considero que encontrei a placa na imagem. Caso contrário, considero que não há placa na imagem.

**Ajuda:** Como ler e gravar vídeo.

```
(...)  
int nl=240; int nc=320;  
VideoCapture w(argv[1]); // abre vídeo de entrada  
if (!w.isOpened()) erro("Erro: Abertura de vídeo");  
VideoWriter vo(argv[3], CV_FOURCC('X','V','I','D'), 20, Size(nc,nl));  
Mat_<COR> a,d; Mat_<FLT> f;  
while (true) {  
    // w.grab(); // se quiser limpar buffer da camera  
    w >> a; // pega um quadro do vídeo  
    if (!a.data) break;  
    converte(a,f);  
    (... processa a/f e gera imagem de saída d ...)  
    vo << d;
```

**Ajuda:** Como calcular intervalo de tempo:

```
double t1=timeSinceEpoch();  
(...)  
double t2=timeSinceEpoch();  
double t=t2-t1; // t contem segundos decorridos entre t1 e t2
```

**[Aula 3 parte 2. Fim.]**

## 3 Processamento Paralelo:

Atualmente, é possível fazer processamento paralelo de várias formas:

**1) Multi-núcleo:** A programação multi-núcleo é crucial para aproveitar o poder dos processadores modernos com múltiplos núcleos. Se você não especificar que se deseja fazer processamento paralelo, o seu programa C/C++ irá utilizar só um núcleo, o que é um desperdício.

Existem várias formas de utilizar vários núcleos de CPU:

1.1) *Funções do sistema operacional:* Os sistemas operacionais Linux e Windows possuem funções que permitem escrever programas que utilizam vários núcleos. Linux utiliza o modelo de execução “posix threads” ou pthreads. Windows possui outras funções próprias. É melhor usar as bibliotecas independentes do sistema operacional (em vez destas funções específicas dos sistemas operacionais), pois isso aumentará a portabilidade do seu programa.

1.2) *Thread de C++:* A linguagem C++ (dialetos 2011 em diante) oferece suporte a paralelismo:

Bjarne Stroustrup, *The C++ Programming Language* (4th Edition), 2013.

O “manual de referência” da linguagem C++ está disponível no site:

<http://www.cplusplus.com> e <https://cplusplus.com/reference/thread/thread/>

1.3) *OpenMP:* É provavelmente a forma mais simples de escrever programas paralelos. Nos casos simples, basta inserir comandos “#pragma” nos lugares adequados do código C/C++. Alguns slides didáticos sobre OpenMP:

<https://www.openmp.org>

<http://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf>

[https://sc.tamu.edu/files/training/Introduction\\_OpenMP\\_Spring2017.pdf](https://sc.tamu.edu/files/training/Introduction_OpenMP_Spring2017.pdf)

1.4) *Outras:* Há muitas outras bibliotecas para fazer programas paralelos multi-core:

- Boost.Thread
- Dlib
- Qt QThread
- oneTBB [<https://link.springer.com/content/pdf/10.1007/978-1-4842-4398-5.pdf>]
- IPP

**2) Instruções vetoriais:** Os processadores atuais possuem instruções vetoriais SIMD (single instruction multiple data). Instruções vetoriais são um tipo de instrução de processador que permite executar a mesma operação sobre vários dados ao mesmo tempo.

Por exemplo, considere somar dois vetores  $U=V+W$ , isto é,  $[u_1, u_2, u_3, u_4] = [v_1, v_2, v_3, v_4] + [w_1, w_2, w_3, w_4]$ . Essas quatro somas podem ser executadas em paralelo usando uma única instrução vetorial.

Os processadores x86 possuem MMX (MultiMedia eXtension - 1996), SSE (streaming SIMD extensions - 1999), SSE2 (2001), AVX (advanced vector extensions - 2008), AVX-512, etc. Essas instruções podem ser utilizadas dentro de programas C/C++ como se fossem funções, sem ter que programar em linguagem de máquina ou assembler.

[<http://software.intel.com/sites/landingpage/IntrinsicsGuide/>].

Os processadores ARM possuem instruções vetoriais NEON e SVE/SVE2.

Exemplo: O programa abaixo utiliza variáveis “\_\_m256” para armazenar 8 variáveis *float*, e utiliza `_mm256_add_ps` para somar em paralelo 8 variáveis *float*.

<pre>//vetorial1.cpp //compila vetorial1 -avx #include &lt;iostream&gt; #include &lt;cstdio&gt; #include &lt;immintrin.h&gt; int main() {     __m256 a,b,c;     float *pa, *pb, *pc;      pa=(float*)&amp;a;     pa[0]=1.0; pa[1]=2.0; pa[2]=3.0; pa[3]=4.0;     pa[4]=5.0; pa[5]=6.0; pa[6]=7.0; pa[7]=8.0;      pb=(float*)&amp;b;     pb[0]=10.0; pb[1]=20.0; pb[2]=30.0; pb[3]=40.0;     pb[4]=50.0; pb[5]=60.0; pb[6]=70.0; pb[7]=80.0;      c = _mm256_add_ps(a,b); // Calcula c = a+b      pc=(float*)&amp;c;     for (int i=0; i&lt;8; i++)         printf("i=%d c[i]=%f\n",i,c[i]); }</pre>	<pre>i=0 c[i]=11.000000 i=1 c[i]=22.000000 i=2 c[i]=33.000000 i=3 c[i]=44.000000 i=4 c[i]=55.000000 i=5 c[i]=66.000000 i=6 c[i]=77.000000 i=7 c[i]=88.000000</pre>
--	--

**3) GPU:** Muitos computadores possuem processador gráfico (GPU - graphics processing unit) que podem ser utilizados para efetuar computação paralela genérica. Os principais plataformas que permitem esse tipo de programação são CUDA (Compute Unified Device Architecture) da NVIDIA e OpenCL (open computing language) não associado a nenhum fabricante específico. Nas disciplinas PSI3471 e PSI3472, já utilizamos GPU para acelerar o treino de rede neural.

## Programação multi-núcleo com OpenMP em C++

Uma outra vantagem da linguagem C++ sobre Python (além de ser 10-100 vezes mais rápido) é que é mais simples e eficiente fazer processamento paralelo multi-núcleo.

Multithreading em Python: O Python utiliza o GIL (Global Interpreter Lock) para garantir a segurança da memória. O GIL permite que apenas uma thread nativa do Python execute código bytecode Python por vez, mesmo em sistemas com vários núcleos. Para tarefas que dependem muito da CPU (uso intensivo de cálculos), o multithreading em Python não oferece paralelismo verdadeiro entre núcleos, mas sim concorrência.

Multiprocessing em Python: O Multiprocessing cria processos separados, e não threads. Cada processo tem seu próprio interpretador Python e seu próprio espaço de memória. Como cada processo possui sua própria instância do GIL, eles podem ser executados em paralelo verdadeiro em diferentes núcleos da CPU. Porém, a comunicação e o compartilhamento de dados entre processos são mais complexos (requerem mecanismos como Pipes ou Queues).

Um programa escrito em C++ normalmente irá usar um único núcleo do processador. Isto é um desperdício, pois a maioria dos computadores atuais possui 4, 6, 8, 12 ou 16 núcleos físicos. Se o seu computador possui 4 núcleos físicos, o seu programa (em teoria) poderia rodar 4 vezes mais rápido se usasse todos os núcleos. Na prática, o programa paralelo não chega a ser 4 vezes mais rápido do que o sequencial, devido principalmente ao gargalo de comunicação entre processador e memória, mas fica consideravelmente mais rápido.

Nota: Processadores Intel com *hyper-threading*

[<https://www.intel.com.br/content/www/br/pt/gaming/resources/hyper-threading.html>]

ou AMD com *simultaneous multithreading* (SMT)

[[https://en.wikipedia.org/wiki/Simultaneous\\_multithreading](https://en.wikipedia.org/wiki/Simultaneous_multithreading)]

aparentam ter o dobro de núcleos do que a quantidade física. Por exemplo, um processador com 4 núcleos físicos aparenta possuir 8 núcleos virtuais. Porém, a velocidade de processamento paralelo não irá aumentar além da quantidade real de núcleos.

**Nota sobre VirtualBox:** Se você está usando *VirtualBox*, o ambiente *Linux Mint* fornecido está configurado para usar um único núcleo do processador. Evidentemente, não é possível fazer processamento paralelo usando um único núcleo. Você deve aumentar o número de núcleos do sistema virtual nas configurações de *VirtualBox*.

Entre as muitas bibliotecas para fazer processamento paralelo (listadas na seção 3.1 acima), vamos usar apenas *OpenMP*, pois é provavelmente a forma mais fácil de fazer processamento paralelo. Para maiores detalhes, veja:

- [A “Hands-on” Introduction to OpenMP.](#)
- [Manual de OpenMP.](#)
- [paralelismo.pdf](#), [paralelismo.odt](#).
- <https://www.openmp.org>
- [https://sc.tamu.edu/files/training/Introduction\\_OpenMP\\_Spring2017.pdf](https://sc.tamu.edu/files/training/Introduction_OpenMP_Spring2017.pdf)

## Exemplo simples de processamento paralelo com OpenMP:

Considere o seguinte programa. Compilando e executando esse programa, (evidentemente) imprime “Hello world”.

<pre>//hello1.cpp #include &lt;stdio.h&gt; int main() {     printf("Hello world\n"); }</pre>	Hello world
--	-------------

Agora, vamos fazer a versão paralela desse programa:

<pre>//hello2.cpp //compila hello2 -omp #include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main() {     #pragma omp parallel     {         printf("Hello world(%d)\n", omp_get_thread_num());     } }</pre>	Hello world(0) Hello world(5) Hello world(8) Hello world(2) Hello world(1) Hello world(10) Hello world(9) Hello world(4) Hello world(6) Hello world(7) Hello world(3) Hello world(11)
--	---

Este programa deve ser compilado com o comando:

```
$ compila hello2 -omp (com Cekeikon)
OU
$ g++ hello2.cpp -o hello2 -fopenmp (sem Cekeikon)
```

Executando o programa obtemos 12 impressões. Uma cópia do comando *printf* dentro das chaves (em verde) é executado em cada núcleo. Como o meu computador tem 6 núcleos físicos e 12 núcleos virtuais, esse comando foi executado em paralelo nos 12 núcleos. A função *omp\_get\_thread\_num()* retorna a identificação do thread (núcleo) que está em execução. Repare que os threads são executados em qualquer ordem.

É possível especificar a quantidade de núcleos a serem usados:

<pre>//hello2b.cpp //compila hello2b -omp #include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main() {     omp_set_num_threads(4);     #pragma omp parallel     {         printf("Hello world(%d)\n", omp_get_thread_num());     } }</pre>	Hello world(2) Hello world(0) Hello world(1) Hello world(3)
--	---

Se quisermos que os comandos após “#pragma omp parallel” seja executado uma única vez:

<pre>//hello3.cpp //compila hello3 -omp #include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main() {     #pragma omp parallel     #pragma omp single     {         printf("Hello world(%d)\n", omp_get_thread_num());     } }</pre>	Hello world(4)
---	----------------

Com isso, podemos executar várias tarefas (*tasks*) diferentes em paralelo:

<pre>//hello4.cpp //compila hello4 -omp #include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main() {     #pragma omp parallel     #pragma omp single     {         #pragma omp task         { printf("Hello world1(%d)\n", omp_get_thread_num()); }         #pragma omp task         { printf("Hello world2(%d)\n", omp_get_thread_num()); }         #pragma omp task         { printf("Hello world3(%d)\n", omp_get_thread_num()); }         #pragma omp task         { printf("Hello world4(%d)\n", omp_get_thread_num()); }     } }</pre>	Hello world2(6) Hello world1(11) Hello world4(4) Hello world3(3)
--	---

Os comandos dentro de um *task* são executados sequencialmente:

<pre>//hello5.cpp //compila hello5 -omp #include &lt;stdio.h&gt; #include &lt;omp.h&gt; int main() {     #pragma omp parallel     #pragma omp single     {         #pragma omp task         { printf("Hello world1(%d)\n", omp_get_thread_num());           printf("Hello OMP1(%d)\n", omp_get_thread_num());         }         #pragma omp task         { printf("Hello world2(%d)\n", omp_get_thread_num());           printf("Hello OMP2(%d)\n", omp_get_thread_num());         }         #pragma omp task         { printf("Hello world3(%d)\n", omp_get_thread_num());           printf("Hello OMP3(%d)\n", omp_get_thread_num());         }         #pragma omp task         { printf("Hello world4(%d)\n", omp_get_thread_num());           printf("Hello OMP4(%d)\n", omp_get_thread_num());         }     } }</pre>	Hello world1(11) Hello OMP1(11) Hello world4(6) Hello OMP4(6) Hello world3(8) Hello OMP3(8) Hello world2(5) Hello OMP2(5)
--	--

## Paralelizar laço “for” com OpenMP

Vou dar só um exemplo de como paralelizar o laço “for”, que é o suficiente para o nosso projeto. O exemplo abaixo calcula a imagem negativa, paralelizando o laço “for” externo. Veja como é simples paralelizar: basta colocar uma única linha no código de C++, sem alterar em nada o programa sequencial.

Para compilar o programa que usa biblioteca paralelo *OpenMP* usando *Cekeikon*, escreva:

```
$ compila omp_neg -ocv -v3 -omp
```

Se você não escrever “-omp”, o compilador irá considerar que a linha em amarelo é um comentário e gerará um programa sequencial.

O processamento de cada linha da imagem será enviado ao núcleo que estiver desocupado, em qualquer ordem, até que todas as linhas tenham sido processadas. Assim, o número de linhas da imagem *a* não precisa ser múltiplo do número de núcleos do seu computador.

```
//omp_neg.cpp - grad2024
//compila omp_neg -ocv -v3 (sem OpenMP)
//compila omp_neg -ocv -v3 -omp (com OpenMP)
#include <opencv2/opencv.hpp>
using namespace std;
using namespace cv;
typedef uint8_t GRY;

Mat_<GRY> negative(Mat_<GRY> a) { //uma única variável a.
    Mat_<GRY> b(a.rows,a.cols); //uma única variável b.
    #pragma omp parallel for
    for (int l=0; l<a.rows; l++) //variável l usada na paralelizacao.
        for (int c=0; c<a.cols; c++) //uma variável c para cada thread.
            b(l,c)=255-a(l,c);
    return b;
}

int main(int argc, char** argv) {
    Mat_<GRY> a=imread("mickey_reduz.bmp",0);
    if (a.total()==0) perror("Erro leitura");
    Mat_<GRY> b=negative(a);
    imwrite("omp_neg.pgm",b);
}
```

Você deve tomar alguns cuidados ao paralelizar laço “for”.

1) Paralelizar possui “overhead” (custo extra). Só vale a pena paralelizar “for” se os comandos dentro do “for” gastarem um tempo considerável. No programa acima, a versão paralela irá demorar mais do que a versão sequencial, pois os comandos dentro do laço são muito simples e rápidos. Neste caso, o “overhead” é maior do que o benefício obtido com a paralelização. No nosso projeto, devemos calcular *matchTemplate* em aproximadamente 10 escalas. Neste caso, realmente vale a pena paralelizar o laço “for”, pois o cálculo de cada *matchTemplate* demora um tempo considerável.

2) Só se pode paralelizar laços onde cada iteração não dependa do resultado da iteração anterior. No exemplo acima, calcular a imagem negativa de uma linha não depende das outras linhas, e assim é possível paralelizar. No caso do projeto, você deve assegurar que o cálculo de um *matchTemplate* paralelizado não dependa dos resultados de outros *matchTemplate*'s.

3) Para cada *thread*, haverá uma cópia de cada uma das variáveis locais dentro do bloco paralelo. No exemplo acima, se o seu computador tiver 8 *threads* paralelos, haverá 8 variáveis *c*, para poder percorrer de forma independente cada uma das linhas. Por outro lado, haverá uma única cópia das variáveis fora do bloco paralelo (as imagens *a* e *b*). Todos os *threads* irão compartilhar a mesma imagem de entrada *a* e a mesma imagem de saída *b*. Vários núcleos tentando acessar a memória ao mesmo tempo constitui o gargalo da paralelização. A variável do loop paralelizado *l* será controlado pelo OpenMP, que irá assinalar o valor correto de *l* para cada *thread*.

4) Sempre que possível, paralelize o *loop* mais externo. Cada *loop* externo executa vários *loops* internos. Paralelizar *loop* externo tem mais chance de valer a pena, pois demora mais tempo e pode compensar *overhead*. Uma iteração do *loop* interno gasta pouco tempo e há maior chance do *overhead* ser maior do que a economia de tempo obtida pela paralelização.

Se a paralelização estiver funcionando corretamente, todos os *threads* terão carga de trabalho próximo de 100%. Veja as figuras 8 e 9.

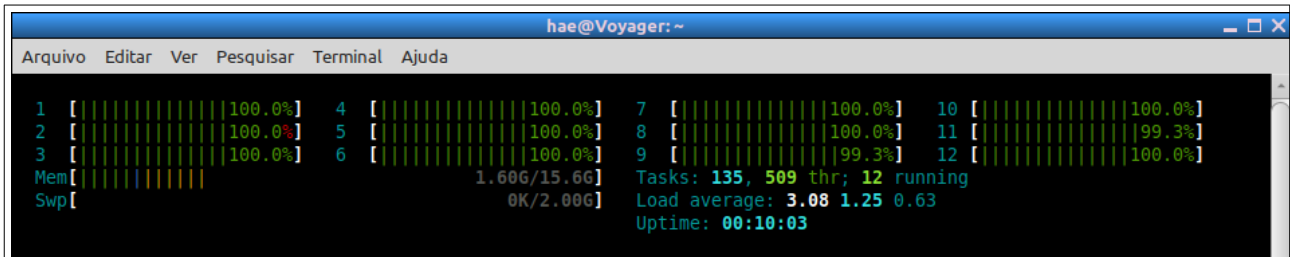
Nota: Pode ser que o comando abaixo:

```
#pragma omp parallel for schedule(static)
```

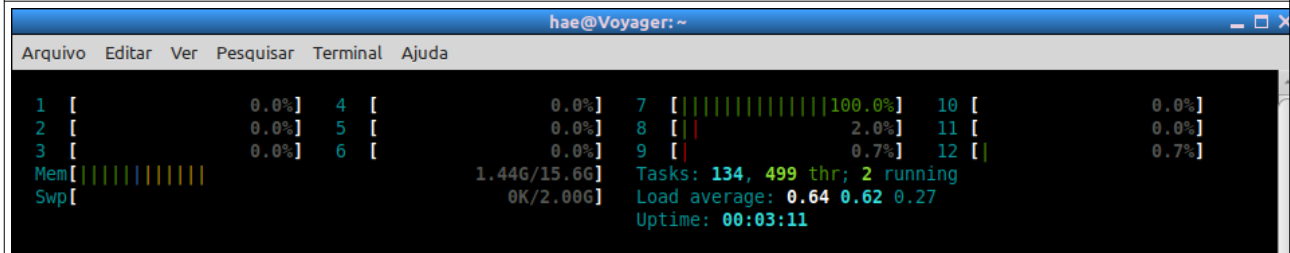
diminua o *overhead* do OpenMP

O programa “htop” de Linux mostra a carga de trabalho de cada núcleo. Para instalá-lo e usá-lo:

```
$ sudo apt install htop  
$ htop
```



(a) As cargas de trabalho de todos os núcleos são 100% durante a execução do programa *fase3* paralelo.

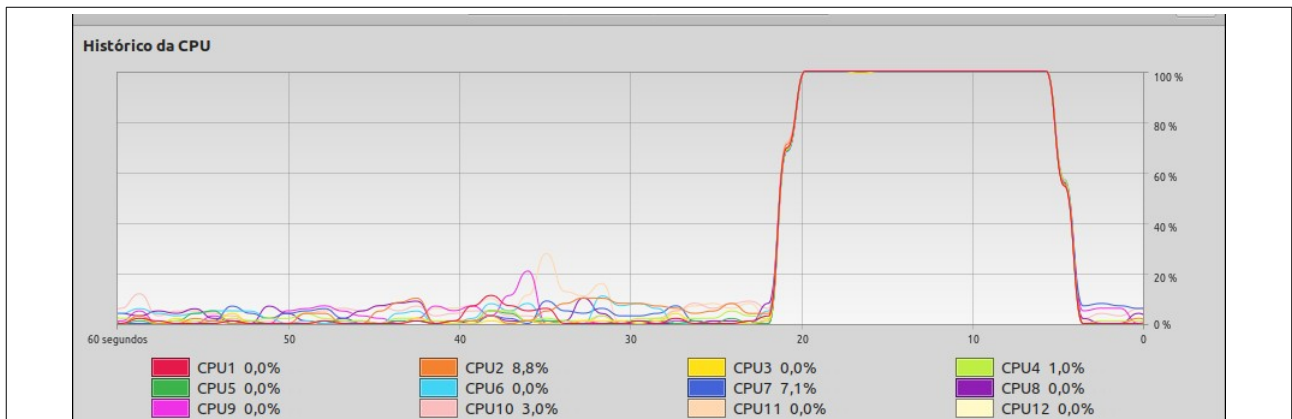


(b) Somente um núcleo trabalha 100% durante a execução do programa *fase3* sequencial.

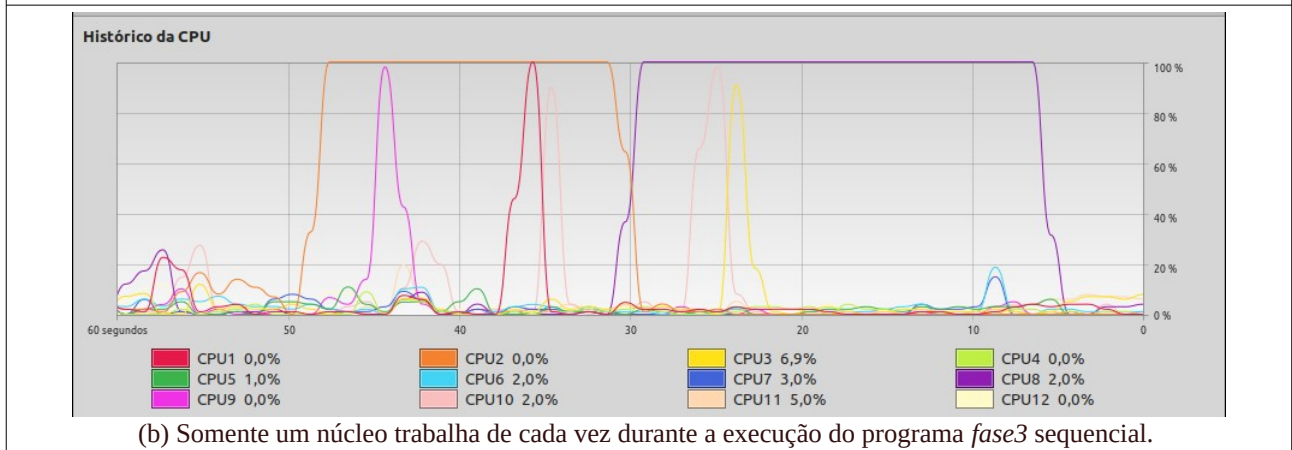
Figura 8: Telas do programa *htop* mostrando a carga de trabalho de cada núcleo durante a execução do programa (a) *fase3p* paralelo, (b) *fase* sequencial. Veja como todos os 12 núcleos trabalham no programa paralelo, enquanto que só um núcleo trabalha no programa sequencial.

Linux Mint (e também várias outras distribuições Linux) possui “monitor do sistema” (system monitor) que também pode ser usado para monitorar o trabalho de cada núcleo:

Iniciar → Administração → Monitor do sistema



(a) Todos os núcleos trabalham durante a execução do programa *fase3p* paralelizado. O “planalto” que se observa no gráfico corresponde à execução do programa *fase3* paralelo.



(b) Somente um núcleo trabalha de cada vez durante a execução do programa *fase3* sequencial.

Figura 9: Telas do programa “monitor de sistema” do Linux Mint durante a execução do programa (a) *fase3p* paralelo, (b) *fase3* sequencial.

**[Lição de casa 1 da aula 4 – 5 pontos]** Acelere o programa *fase3.cpp* usando o processamento paralelo OpenMP e nomeie o novo programa como *fase3p.cpp* (“computador\$ compila fase3p -ocv -v3 -omp”). O importante é paralelizar os casamentos de modelos, pois é a parte mais pesada do programa. Calcule fps (quadros/segundo) que os seus programas paralelo e sequencial conseguem atingir. O programa paralelo está mais rápido do que o programa sequencial? A tabela abaixo mostra fps em diferentes computadores e compilações.

- No meu computador i7 com 6 núcleos (12 threads) usando OpenCV2, obtive 16 fps com processamento sequencial e 60 fps com processamento paralelo.
- A função *matchTemplate* foi melhorada no OpenCV3 e, usando esta versão, obtive 46 fps com processamento sequencial (“compila fase3 -ocv -v3”) e 120 fps com processamento paralelo (“compila fase3p -ocv -v3 -omp”).
- No ambiente VM dentro de Windows 10, obtive 32 fps com processamento sequencial e 71 fps com processamento paralelo (usando OpenCV3 e com VM configurado para 8GB de memória e 4 núcleos).
- Em Raspberry 3 modelo B, obtive 2,6 fps com processamento sequencial e 5,0 fps com processamento paralelo, mostrando que não é possível fazer este processamento em tempo real usando Raspberry.

	Sequencial (fps)	Paralelo (fps)
Raspberry 3 model B, OpenCV3	2,6	5
i7, 6 núcleos, 12 threads, OpenCV2	16	60
i7, 6 núcleos, 12 threads, OpenCV3	46	120
i7, VM em Windows 10, 4 núcleos, OpenCV3	32	71
i9, 16 núcleos, 32 threads, OpenCV3	60	153

*Nota:* As medidas acima foram obtidas sem mostrar os quadros do processamento intermediário na tela do computador. O meu programa que lê o vídeo de entrada e gera o vídeo de saída, sem mostrar o processamento intermediário na tela. Se mostrar os quadros na tela, *fps* cai drasticamente.

*Nota:* Rode o programa “htop” num outro terminal e verifique se há outros processos que estão consumindo memória e/ou processamento. Neste caso, “mate” todos os outros programas que gastam muita memória/processamento (como browser de internet). Note que rodar *Zoom* ou *Google Meet* junto com o seu programa faz *fps* cair consideravelmente.

*Nota:* Um dos principais gargalos da computação é o acesso à memória: vários núcleos não conseguem trabalhar ao mesmo tempo na velocidade máxima se todos eles necessitarem acessar memória. Assim, muitas vezes o processamento paralelo com *n* núcleos não consegue diminuir o tempo de processamento por *n*.

*Nota:* Se você está enfrentando dificuldades para paralelizar o seu programa, sugiro que estruture o seu programa paralelo da seguinte forma:

- 1) Primeiro, calcule em paralelo 10 template matchings CC e outros 10 template matchings NCC, obtendo 20 imagens com correlações resultantes. Esta é a parte mais pesada do programa que precisa ser paralelizada.
- 2) Utilize as 20 imagens obtidas acima para localizar a placa no quadro. Aqui, não precisa paralelizar, pois o processamento não demorará muito.

## 4 Fase 4 do projeto

**[Lição de casa 2 da aula 4 – 5 pontos]** Modifique o programa *fase3p.cpp* para controlar o carrinho de forma que ele siga continuamente a placa *quadrado.png*. Grave os programas resultantes como *servidor4.cpp* e *cliente4.cpp*. O processamento mais pesado (localizar as placas) deve ser feito no computador (*cliente4.cpp*). Para isso, você irá usar somente a posição *x* da localização da placa para girar o carrinho para direita/esquerda, desprezando a posição *y* (não vamos movimentar o carrinho para cima/baixo). O programa deve terminar quando apertar ESC.

```
raspberrypi$ servidor4
computador$ cliente4 192.168.0.110 [videosaida.avi] [t/c]
```

onde:

- 192.168.0.110 é o endereço IP da Raspberry.
- *Videosaida.avi* é o nome do vídeo opcional onde as imagens mostradas na tela ou capturadas pela câmera serão armazenadas.
- O argumento *t* ou *c* indica respectivamente gravar uma cópia da tela (com quadrados e números que ajudam a debugar o programa) ou somente a imagem capturada pela câmera.

Compilar:

```
Computador$ compila cliente4 -ocv -v3 -omp
Raspberrypi$ compila servidor4 -ocv -w
```

**[Lição de casa extra 2025 (vale +1 ponto):]** Entregue um vídeo de aproximadamente 1 minuto gravado com argumento “c” (somente visão da câmera) de carrinho seguindo a placa. Coloque os vídeos no Google Drive:

<https://drive.google.com/drive/folders/1Sc8kmIPNH4HIiz-6PfyhXVcr3RkaXqyug>

com o nome:

FulanoDeTal\_CiclanoDeQual\_Fase4.avi

Este vídeo é para tentar localizar a placa usando deep learning nos próximos anos.

**Como eu fiz:** Fiz com que a diferença das velocidades das rodas esquerda e direita seja diretamente proporcional à distância entre a posição *x* da placa e o centro da imagem. Isto é, se a placa estiver muito à esquerda, o carrinho irá girar rapidamente para esquerda. Se a placa estiver ligeiramente fora do centro, o carrinho irá corrigir a rota suavemente. Isto faz com que o movimento do carrinho fique “suave”.

O carrinho deve parar nas duas situações:

- a) Quando não conseguir localizar a placa no quadro do vídeo capturado.
- b) Quando a câmera do Raspberry estiver muito próximo da placa, para evitar uma colisão.

Os vídeos abaixo mostram o carrinho seguindo a placa:

[https://drive.google.com/file/d/1XIFDBiEnT\\_KrHgLsO-VrGiCZFtXff-pS/view](https://drive.google.com/file/d/1XIFDBiEnT_KrHgLsO-VrGiCZFtXff-pS/view)

<https://drive.google.com/file/d/14NZWYULSyNik5utFIGm9YkUQTZz5N-YS/view>

**[Aula 4. Fim.]**