

Template Matching

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.

1 Introdução

A parte teórica do template matching já foi vista na disciplina teórica PSI-3471 Fundam. Sist. Eletrônicos Inteligentes. Veja a apostila tmatch-ead no site 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: Template matching funciona bastante bem quando implementada corretamente. Porém, pode ser que seja possível chegar a uma solução ainda mais robusta usando rede neural convolucional. Pode ser que, para os próximos anos, seja interessante trocar template matching por alguma técnica de aprendizagem profunda para localizar a placa.

Resumo de casamento de modelos

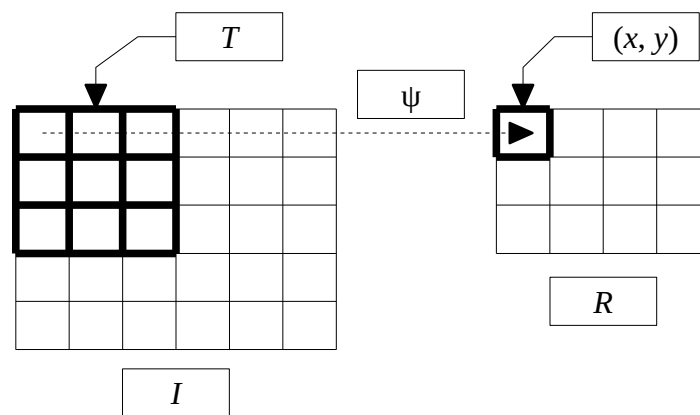
Para facilitar, escrevo abaixo a “receita de bolo” de como fazer casamento de modelos, sem explicar a teoria:

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)
```

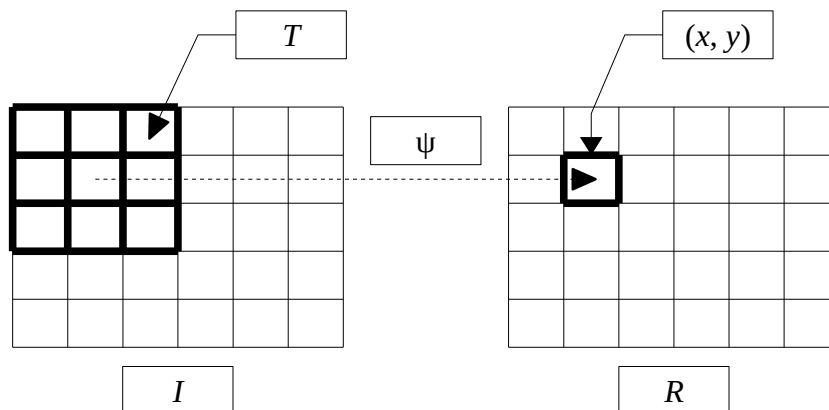
onde *I* é a imagem de entrada onde vai fazer a busca, *T* é o modelo a ser procurado e *R* é o resultado. Essa função trabalha no modo “valid”, onde o resultado *R* é menor do que a imagem de busca *I*, e o resultado do casamento é colocado no canto superior esquerdo da ocorrência do modelo em *R*. Veja a figura abaixo. Veja 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).



2) Normalmente, queremos que o resultado da correlação fique no centro do modelo em *R*, em vez de no canto superior esquerdo, fazendo com que a imagem de saída *R* tenha o mesmo tamanho que a entrada *I* (modo “same”). A função *matchTemplateSame* de Cekeikon faz isso (figura abaixo).

```
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)
```



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 o 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* são de Cekeikon. 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*.

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 abaixo, 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.

est, we may trace a journey which has led humanity down the centuries
ruth more and more deeply. It is a journey which has unfolded—as it
izon of personal self-consciousness: the more human beings know real-
more they know themselves in their uniqueness, with the question of the

bbox3.pgm

more

letramore3.pgm

est, we may trace a journey which has led humanity down the centuries
ruth more and more deeply. It is a journey which has unfolded—as it
izon of personal self-consciousness: the more human beings know real-
more they know themselves in their uniqueness, with the question of the

ocorrencia3.png

2 Terceira fase do projeto

[Lição de casa 1 da aula 5] 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 1). Imprima quantos quadros/segundo o seu programa conseguiu processar.



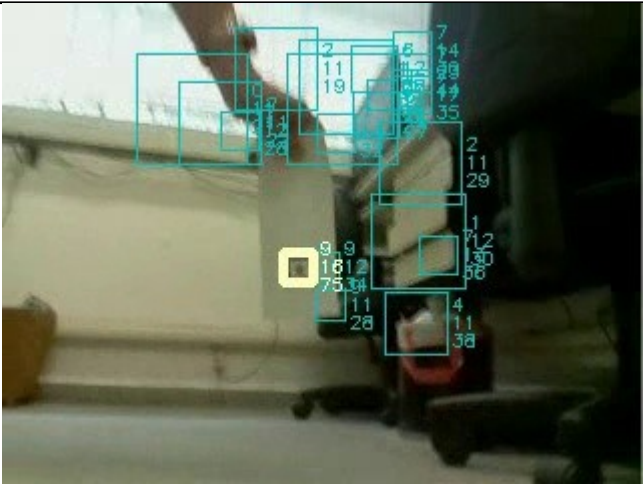
 <p>Quadro 140 de capturado2.avi.</p>	 <p>Placa quadrado.png.</p>
 <p>Quadro 140 de localiza2.avi.</p>	

Figura 1: 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 alta 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.



Figura 2: Falso negativo - o programa não conseguiu localizar a placa muito borrada.

Os vídeos-testes (capturado*.avi) e a placa (quadrado.png) estão em:

<http://www.lps.usp.br/hae/apostilaraspi/capturado.avi>
<http://www.lps.usp.br/hae/apostilaraspi/capturado2.avi>
<http://www.lps.usp.br/hae/apostilaraspi/capturado3.avi>
<http://www.lps.usp.br/hae/apostilaraspi/quadrado.png>

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>

Em Linux, pode-se usar o programa *avidemux* para visualizar o vídeo quadro a quadro.

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. 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 do modelo. 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 3 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. Repare que, sem usar “don't care”, há vários pixels com alto brilho dificultando a localização da placa, enquanto que

usando “don’t care”, praticamente somente a localização verdadeira da placa fica altamente brilhante.

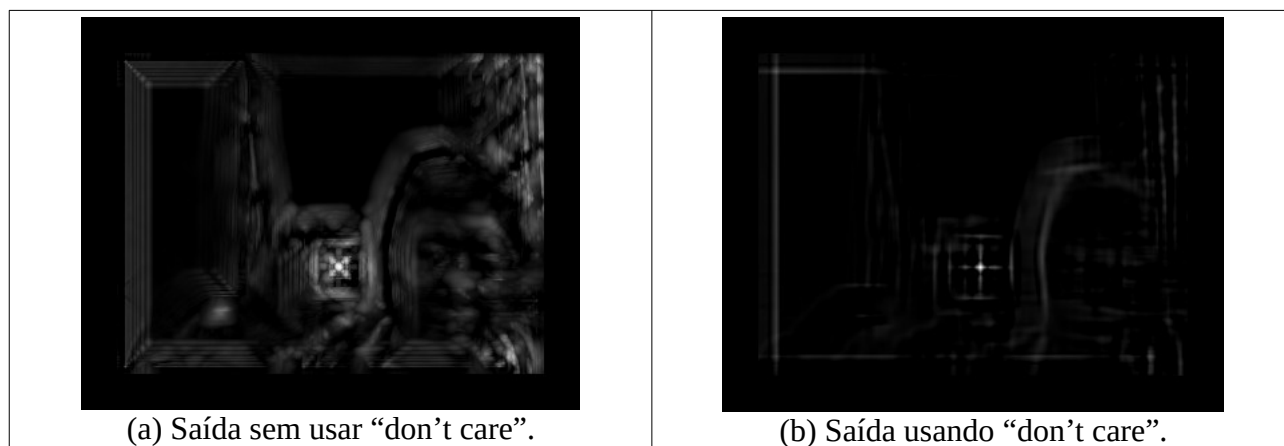


Figura 3: Diferença entre usar ou não pixels “don’t care”. Nas duas imagens, o pixel com o maior valor foi mapeado no branco e o pixel com o menor valor foi mapeado no preto. Sem usar “don’t care”, há vários pixels com alto brilho.

O seguinte código em Cekeikon faz essa tarefa:

```
//Le quadrado.png como imagem em ponto flutuante (0=preto, 1=branco)
Mat_<FLT> T; le(T,"quadrado.png");
//Cuidado: Nao se pode ler diretamente do video para Mat_<FLT>.
//Errado: VideoCapture vi; ... Mat_<FLT> a; vi >> a;
//Voce deve ler vídeo como Mat_<COR> e converter para Mat_<FLT> usando converte(a,b)
//Funcao converte e' de Cekeikon
//Certo: 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));
```

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 não irá combinar com as cores dos pixels vizinhos).

CC e NCC

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 2 mostra um caso de falso negativo.

Sugestão (você pode fazer diferente, o importante é levar em conta CC e NCC conjuntamente): Para diminuir os falsos positivos, estou usando a seguinte técnica, combinando template matching CC com NCC. Template matching CC localiza a instância de alto contraste, mesmo que a forma da instância difira um pouco do modelo procurado. Template matching NCC localiza a instância com o formato semelhante ao modelo, mesmo que a instância seja de baixíssimo contraste.

A 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*). Procuro 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. Calculo os casamentos NCC nas 20 posições, nas mesmas escalas que deram pico CC. Entre as 20 correlações NCC, procuro a maior correlação. Se esta correlação for maior que 0,55, considero que encontrei a placa na imagem. Caso contrário, considero que não há placa na imagem.

3 Processamento Paralelo:

Atualmente, há várias formas de se escrever programas paralelos em computadores "normais":

1) Multi-núcleo: Quase todos os processadores atuais possuem vários núcleos. Os códigos comuns C/C++ utilizam só um desses núcleos, o que é um “desperdício”. Existem várias formas de utilizar mais de um núcleo:

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. Provavelmente, a melhor referência é o livro:

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

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). Por exemplo, os processadores x86 possuem MMX (MultiMedia eXtension - 1996), SSE (streaming SIMD extensions - 1999), SSE2 (2001), AVX (advanced vector extensions - 2008), etc. Os processadores ARM possuem instruções vetoriais NEON. 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/>].

Por 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 <iostream> #include <cstdio> #include <immintrin.h> int main() { __m256 a,b,c; float *pa, *pb, *pc; pa=(float*)&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*)&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*)&c; for (int i=0; i<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: A maioria dos computadores possuem processadores gráficos (GPU - graphics processing unit) que podem ser utilizados para efetuar computação paralelo genérico (GPGPU - general-purpose GPU). 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.

Processamento usando vários núcleos do CPU com OpenMP

Outra vantagem da linguagem C++ sobre Python (além de ser 10-100 vezes mais rápido) é que é mais simples fazer processamento paralelo usando os vários núcleos do CPU em C++. É possível fazer processamento paralelo em Python, só que dá mais trabalho: “Python can only execute one task at a time. This is a consequence of flexible datatypes — Python needs to make sure each variable has only one datatype, and parallel processes could mess that up.”
[<https://towardsdatascience.com/why-python-is-not-the-programming-language-of-the-future-30ddc5339b66>]

Um programa escrito em C++ normalmente irá usar um único núcleo do processador. Isto é um desperdício, pois a maioria dos computadores atuais possui 2, 4, 6 ou 8 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, mas fica consideravelmente mais rápido.

Nota sobre hyper-threading e simultaneous multithreading: 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]

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 aumenta muito além da quantidade real de núcleos (no exemplo, 4 vezes).

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 pode aumentar o número de núcleos do sistema virtual nas configurações de VirtualBox.

Entre as muitas bibliotecas para fazer processamento paralelo, vamos usar apenas *OpenMP*, pois é provavelmente a forma mais fácil de fazer processamento paralelo. Vou dar só exemplos muito simples. Para maiores detalhes, veja:

- [A "Hands-on" Introduction to OpenMP.](#)
- [Manual de OpenMP.](#)
- [paralelismo.pdf](#), [paralelismo.odt](#).

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

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

Agora, vamos fazer a versão paralela desse programa:

<pre>//hello2.cpp //compila hello2 -omp #include <stdio.h> #include <omp.h> 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 <stdio.h> #include <omp.h> 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 <stdio.h> #include <omp.h> int main() { #pragma omp parallel #pragma omp single { printf("Hello world(%d)\n", omp_get_thread_num()); } }</pre>	<p>Hello world(4)</p>
---	-----------------------

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

<pre>//hello4.cpp //compila hello4 -omp #include <stdio.h> #include <omp.h> 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>	<p>Hello world2(6) Hello world1(11) Hello world4(4) Hello world3(3)</p>
--	---

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

<pre>//hello5.cpp //compila hello5 -omp #include <stdio.h> #include <omp.h> 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>	<p>Hello world1(11) Hello OMP1(11) Hello world4(6) Hello OMP4(6) Hello world3(8) Hello OMP3(8) Hello world2(5) Hello OMP2(5)</p>
--	--

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, escreva:

```
$ compila omp_neg -cek -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 (thread) 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 - pos2017
//compila omp_neg -c -omp
#include <cekeikon.h>

Mat_<GRY> negative(Mat_<GRY> a) {
    Mat_<GRY> b(a.rows,a.cols); //uma única variável b
    #pragma omp parallel for
    for (int l=0; l<a.rows; l++) { //uma única variável l
        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;
    le(a, "mickey_reduz.bmp");
    Mat_<GRY> b=negative(a);
    imp(b, "omp_neg.pgm");
}
```

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

1) Paralelizar possui “overhead”. Só vale a pena paralelizar “for” se os comandos dentro do “for” gastar 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” com 10 *matchTemplate* dentro, pois o cálculo de cada *matchTemplate* demora um tempo considerável.

2) Só se pode paralelizar laços onde cada iteração não depende do resultado da iteração anterior. No exemplo acima, calcular a imagem negativa de uma linha não depende do cálculo 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 *matchTemplates*.

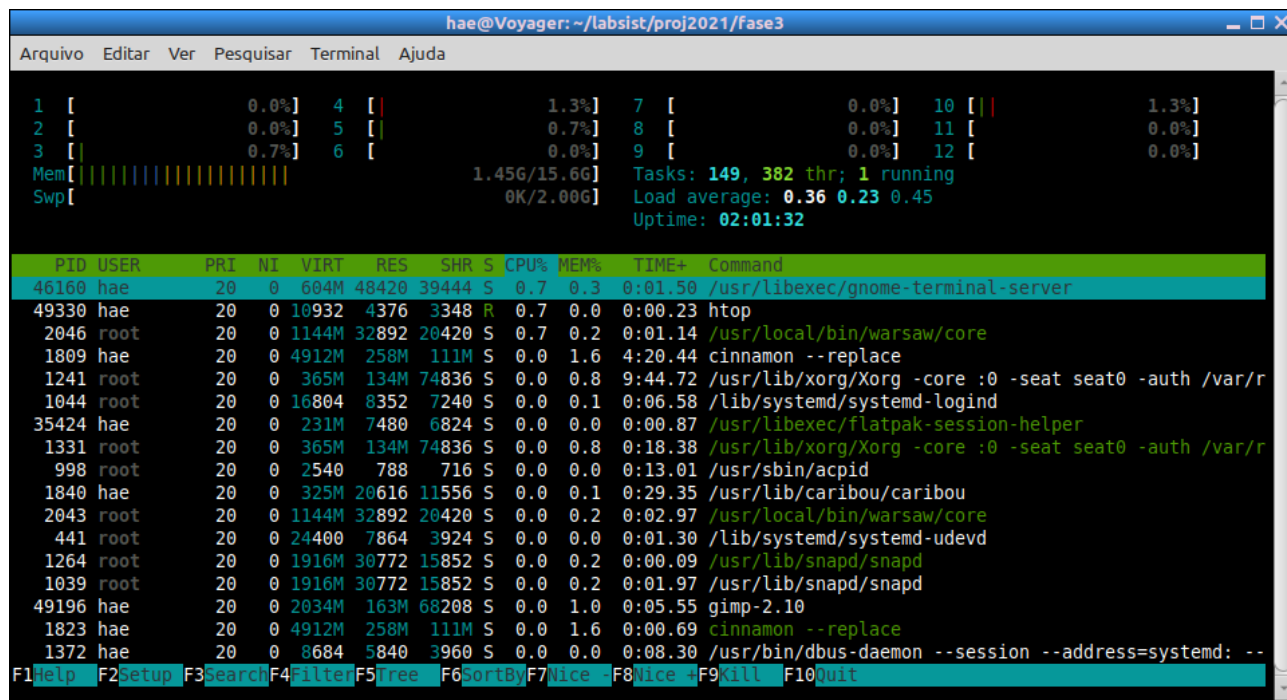
3) Para cada *thread*, haverá uma cópia de cada uma das variáveis locais dentro do bloco paralelo. No exemplo acima, se o seu computador tiver 8 *threads* paralelos, haverá 8 variáveis *c*, para poder percorrer de forma independente as colunas de cada uma das linhas. Por outro lado, haverá uma única cópia das variáveis fora do bloco paralelo (as imagens *a* e *b* e a variável inteira *l*). Todos os *threads* irão compartilhar a mesma imagem de entrada *a* e a mesma imagem de saída *b*.

4) Sempre que possível, paralelize o *loop* mais externo. Cada *loop* externo executa vários *loops* internos. Assim, cada iteração do *loop* externo equivale a várias iterações do *loop* interno em tempo gasto. Paralelizar *loop* externo tem mais chance de valer a pena (economia de tempo ser maior do que *overhead*). Uma iteração do *loop* interno gasta pouco tempo e há maior chance do *overhead* ser maior do que a economia de tempo da paralelização.

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

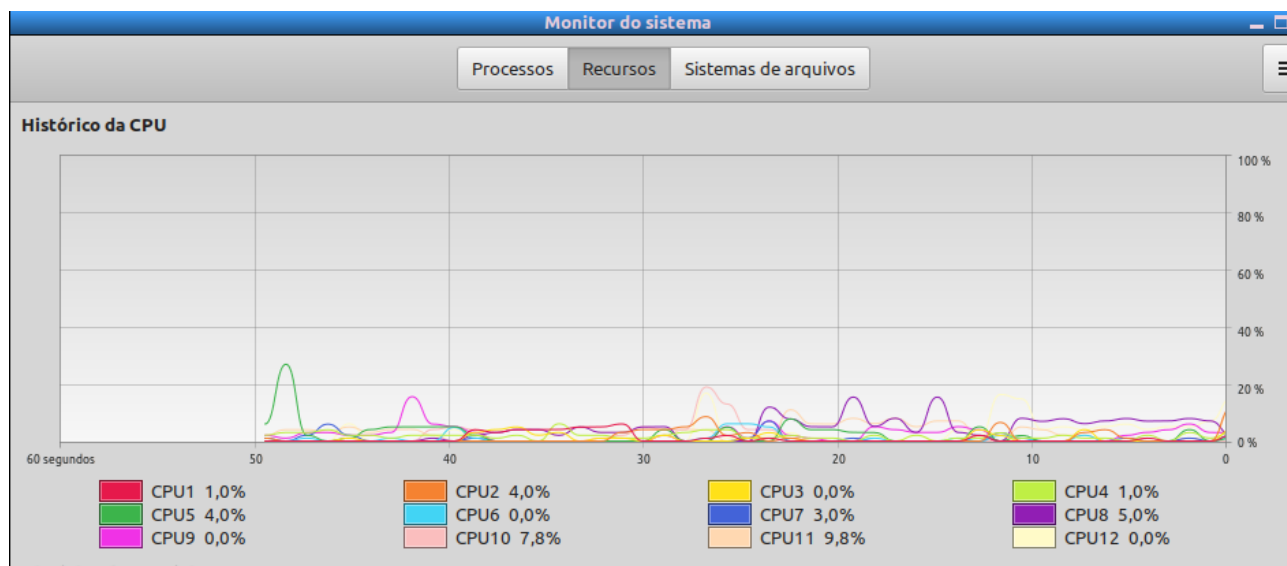
```
$ sudo apt install htop
```

```
$ htop
```



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

Iniciar → Administração → Monitor do sistema



[Lição de casa 2 da aula 5] Acelere o programa fase3 usando o processamento paralelo OpenMP (“compila fase3 -c -v3 -omp”). O importante é paralelizar os casamentos de modelos, pois é a parte mais “pesada” do programa. Calcule quadros/segundo do seu programa paralelo. O programa paralelo está mais rápido do que o programa sequencial?

No meu computador i7 com 6 núcleos (12 núcleos virtuais com hyper-threading), 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 -c -v3”) e 120 fps com processamento paralelo (“compila fase3 -c -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.

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.

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 (como browser de internet) e/ou processamento. Note que rodar Zoom ou Google Meet junto com o seu programa faz cair consideravelmente *fps* (o processamento fica bem mais lento).

4 Quarta fase do projeto

[Lição de casa 3 da aula 5] Modifique o programa da fase 3 para controlar o carrinho de forma que ele siga continuamente a placa *quadrado.png*. Para isso, você irá usar somente a posição x da localização da placa, desprezando a posição y .

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

Como eu fiz: Fiz com que a diferença das velocidades das rodas esquerda e direita dependa do quão distante a posição x da placa está do centro da imagem. Isto é, se a placa estiver muito à esquerda, o carrinho irá girar rapidamente para direita. Se a placa estiver ligeiramente fora do centro, o carrinho irá corrigir a rota suavemente. Isto fez com que o movimento do carrinho ficasse “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.

Este programa deve rodar no modo servidor-cliente (*servidor4.cpp* e *cliente4.cpp*), onde o processamento mais pesado (a localização da placa) deve ser realizado no computador, de preferência usando processamento paralelo.

Os vídeos abaixo mostram o carrinho seguindo a placa:

https://drive.google.com/file/d/1XlFDBiEnT_KrHgLS0-VrGiCZFtXff-pS/view

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