Processamento Paralelo:

Os algoritmos de processamento de imagens são especialmente adequados para serem acelerados utilizando processamento paralelo, pois normalmente uma mesma sequência de instruções se repete cada região da imagem. 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. 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 funções próprias. Sugiro que, em vez de usar estas funções específicas dos sistemas operacionais, usem as funções padrões de C++ ou OpenMP, pois isso aumentará a portabilidade do seu programa.

1.2) Thread de C++: A linguagem C++ (padrões 2011 e 2014) 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/

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:

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 outras bibliotecas para fazer programas paralelos multi-core. Por exemplo, TBB da Intel (Threading Building Blocks (TBB) is a C++ template library developed by Intel for parallel programming on multi-core processors).

2) Instruções vetoriais: Os processadores atuais possuem instruções SIMD (single instruction multiple data) que efetuam, por exemplo, a soma vetorial onde todos os elementos são somados ao mesmo tempo. 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 NEON. Essas instruções podem ser utilizadas dentro de programas C/C++ como se fossem funções, sem ter que programar em linguagem de máquina.

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. Utilizaremos somente OpenCL.

No projeto do carrinho que segue placa, precisa efetuar template matching em várias escalas. Efetuar template matchings em 8 escalas diferentes num computador com 4 núcleos pode ser feita em dois passos sequenciais (4 escalas + 4 escalas) em vez de 8 passos sequenciais.

Definição de thread e core (núcleo):

[http://www.tomshardware.com/forum/306079-28-what-core-thread]
A thread is a single line of commands that are getting processed, each application has at least one thread, most have multiples. A core is the physical hardware that works on the thread. In general a processor can only work on one thread per core, CPUs with hyper threading can work on up to two threads per core.

For processors with hyper threading, there are extra registers and execution units in the core so it can store the state of two threads and work on them both, normally to change threads you have to empty the registers into the cache, write that back to the main memory, then load up the cache with the new values and load up the registers, context switches hurt performance significantly.

Nos processadores Intel com hyperthread, normalmente há *n* núcleos e 2*n* threads (por exemplo 4 núcleos e 8 threads).

Importante:
Para poder executar *n* trechos de um programa em paralelo, não pode haver dependência entre eles. Isto é, um trecho não pode depender do resultado de cálculo de outro trecho.

Importante:
Existem 3 tipos de variáveis em C/C++: automática, estática e dinâmica. Existe uma única cópia de cada variável (automática ou estática) declarada fora do loop paralelo. Por outro lado, cada thread trabalhará com uma instância independente das variáveis automáticas declaradas dentro do loop paralelo. Assim, você tem que decidir se cada variável deve ficar fora ou dentro do loop paralelo.

Exemplo muito simples de C++ thread (C++11 em diante)

(retirado de www.cplusplus.com)

```
// thread example
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo()
{
  // do stuff...
}

void bar(int x)
{
  // do stuff...
}

int main()
{
  std::thread first (foo);     // spawn new thread that calls foo()
  std::thread second (bar,0);  // spawn new thread that calls bar(0)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

Cada thread que corre em paralelo é uma função.

Exemplo muito simples de OpenMP

(retirado de www.openmp.org)

```
void simple(int n, float *a, float *b) {
  int i;
  #pragma omp parallel for
  for (i=1; i<n; i++) /* i is private by default */
    b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Os threads que correm em paralelo são diferentes passos de um loop. O número de threads default depende da variável ambiente OMP_NUM_THREADS, se tiver sido definida. Se essa variável não tiver sido definida, OpenMP escolhe um valor default (provavelmente o número de núcleos do processador).

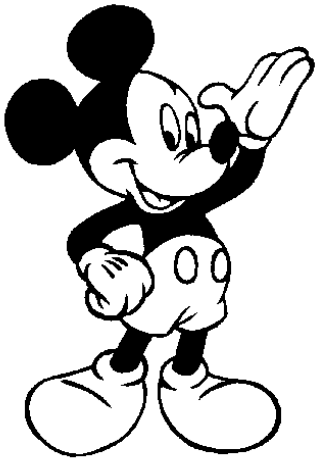
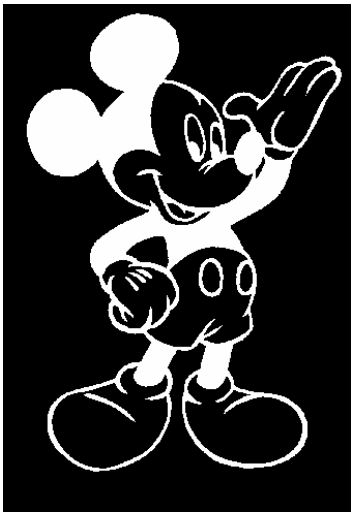
Paralelizar programa que calcula imagem negativa:

Calcula imagem negativa serialmente

```
//ser_neg.cpp - pos2017
//compila ser_neg -c
#include <cekeikon.h>

Mat_<GRY> negative(Mat_<GRY> a) {
  Mat_<GRY> b(a.rows,a.cols);
  for (int l=0; l<a.rows; l++)
    for (int c=0; c<a.cols; c++)
      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,"ser_neg.pgm");
}
```

	
mickey_reduz.bmp	ser_neg.pgm

Calcula imagem negativa usando OpenMP

```
//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");
}
```

Calcula imagem negativa usando thread do C++.

```
//thr_neg.cpp - pos2017
//compila thr_neg -pt
#include <cekeikon.h>
#include <thread>

Mat_<GRY> ap;
Mat_<GRY>  b;
int nthreads=8;

void _negative(int t) {
  int m=teto(int(b.rows),nthreads);
  unsigned inic=t*m;
  unsigned fim=min( (t+1)*m, b.rows );
  for (int l=inic; l<fim; l++)
    for (int c=0; c<b.cols; c++)
      b(l,c)=255-ap(l,c);
}

Mat_<GRY> negative(Mat_<GRY> _a) {
  ap=_a;
  b.create(_a.rows,_a.cols);
  vector<thread> threads(nthreads);
  for (int t=0; t<nthreads; t++)
    threads[t] = thread{_negative,t};
  for (int t=0; t<nthreads; t++) threads[t].join();
  return b;
}

int main(int argc, char** argv) {
  Mat_<GRY> a;
  le(a,"mickey_reduz.bmp");
  Mat_<GRY> b=negative(a);
  imp(b,"thr_neg.pgm");
}
```

Obs 1: Para programa muito simples (como o acima), as versões paralelas demoram mais do que a sua versão serial - tente explicar por quê.
Obs 2: Usando thread do C++, é preciso alterar muito a versão serial para obter a versão paralela.
Obs 3: OpenMP é mais simples - quase nenhuma alteração é necessária para escrever a versão paralela a partir da versão serial. No caso acima, se ignorar "#pragma", a versão paralela torna-se a versão serial.
Obs 4: É mais eficiente paralelizar o loop mais externo. Paralelizar o loop interno implica overhead grande.

Segmentação de feijão pela cor usando vizinho +px serial:

```
//feijao_ser.cpp
//pos2017 segmentacao de feijao pela forca-bruta
#include <cekeikon.h>

int main() {
  Mat_<COR> ax; le(ax,"ax.ppm");
  Mat_<GRY> ay; le(ay,"ay.ppm");
  Mat_<COR> qx; le(qx,"qx.jpg");
  Mat_<GRY> qp(qx.size());
  TimePoint t1=timePoint();
  for (int l=0; l<qx.rows; l++) {
    printf("%9d ",l);
    for (int c=0; c<qx.cols; c++) {
      COR busca=qx(l,c);
      double mindist=1e10; int minl, minc;
      for (int l2=0; l2<ax.rows; l2++)
        for (int c2=0; c2<ax.cols; c2++) {
          double d=distancia(busca,ax(l2,c2));
          if (d<mindist) { mindist=d; minl=l2; minc=c2; }
        }
      qp(l,c)=ay(minl,minc);
    }
  }
  printf("\n");
  impTempo(t1);
  imp(qp,"feijao_ser.png");
}
```

Saída:

```
      ...
       336        337        338        339        340        341        342
343
      344       345       346       347
48 s
```

 ax.ppm	 ay.ppm
 qx.jpg	 feijao_ser.png

Segmentação pela cor usando vizinho +px usando OpenMP:

```
//feijao_par.cpp
//pos2017 segmentacao de feijao pela forca-bruta usando paralelismo openmp
//>compila feijao_par -c -omp
#include <cekeikon.h>

int main() {
  Mat_<COR> ax; le(ax,"ax.tga"); //uma única variável ax
  Mat_<GRY> ay; le(ay,"ay.tga"); //uma única variável ay
  Mat_<COR> qx; le(qx,"qx.jpg"); //uma única variável qx
  Mat_<GRY> qp(qx.size()); //uma única variável qp
  TimePoint t1=timePoint();
  #pragma omp parallel for
  for (int l=0; l<qx.rows; l++) { //uma única variável l
    //#pragma omp critical
    printf("%9d ",l);
    for (int c=0; c<qx.cols; c++) { //uma variável c para cada thread
      COR busca=qx(l,c); //uma variável busca para cada thread
      double mindist=1e10; int minl, minc; //uma variável para cada thread
      for (int l2=0; l2<ax.rows; l2++) //uma variável l2 para cada thread
        for (int c2=0; c2<ax.cols; c2++) { //uma variável para cada thread
          double d=distancia(busca,ax(l2,c2));
          if (d<mindist) { mindist=d; minl=l2; minc=c2; }
        }
      qp(l,c)=ay(minl,minc);
    }
  }
  printf("\n");
  impTempo(t1);
  imp(qp,"feijao_ser.png");
}
```

Saída:

```
      260       217        85       346       303       129      173
41
      261       218        86       347       304             130 174
42
       87       175       131        43
    8 s
```

Obs1: O tempo de processamento diminuiu de 48 para 8.
Obs2: As linhas não são processadas em sequência.
Obs3: Para evitar printf quebrado, escreva "#pragma omp critical" antes do printf.
Obs4: É sempre mais vantajoso paralelizar o loop externo do que o loop interno. No programa acima, é mais vantajoso paralelizar o loop verde (externo) do que o loop interno (amarelo). Pense por quê.

Segmentação pela cor usando vizinho +px usando C++ thread:

```
//feijao_thr.cpp
//pos2017 segmentacao de feijao pela forca-bruta usando C++ thread
//>compila feijao_thr -c -pt
#include <cekeikon.h>
#include <thread>

Mat_<COR> ax;
Mat_<GRY> ay;
Mat_<COR> qx;
Mat_<GRY> qp;
int nthreads=8;

void knn(int t) {
  int m=teto(int(qx.rows),nthreads);
  unsigned inic=t*m;
  unsigned fim=min( (t+1)*m, qx.rows );
  for (int l=inic; l<fim; l++) {
    printf("%9d ",l);
    for (int c=0; c<qx.cols; c++) {
      COR busca=qx(l,c);
      double mindist=1e10; int minl, minc;
      for (int l2=0; l2<ax.rows; l2++)
        for (int c2=0; c2<ax.cols; c2++) {
          double d=distancia(busca,ax(l2,c2));
          if (d<mindist) { mindist=d; minl=l2; minc=c2; }
        }
      qp(l,c)=ay(minl,minc);
    }
  }
}

int main() {
  Mat_<COR> _ax; le(_ax,"ax.ppm");
  Mat_<GRY> _ay; le(_ay,"ay.ppm");
  Mat_<COR> _qx; le(_qx,"qx.jpg");
  Mat_<GRY> _qp(_qx.size());

  TimePoint t1=timePoint();

  ax=_ax;
  ay=_ay;
  qx=_qx;
  qp=_qp;
  vector<thread> threads(nthreads);
  for (int t=0; t<nthreads; t++)
    threads[t]=thread{knn,t};
  for (int t=0; t<nthreads; t++) threads[t].join();
  printf("\n");

  impTempo(t1);
  imp(qp,"feijao_ser.png");
}
```

```
(...)
      305       41      261       85      306      130      174      218       42      262       86
      307      131      175      219       43      263       87
        8 s
```

I. Multiplicação matricial.

Nesta seção, tentaremos acelerar a multiplicação de duas matrizes usando diferentes técnicas de paralelização.

1) Multiplicação matricial convencional, usando 3 loops:

```
// Convencional.cpp
// Para compilar: Compila convencional.cpp -cek
#include <cekeikon.h>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
{ A.create(n,n);
  B.create(n,n);
  srand(seed);
  for (int l=0; l<n; l++)
    for (int c=0; c<n; c++)
      A(l,c)=rand()%n;
  for (int l=0; l<n; l++)
    for (int c=0; c<n; c++)
      B(l,c)=rand()%n;
}

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B,
Mat_<float>& D)
{ D.create(A.rows,B.cols);
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<B.cols; c++) {
      float d=0.0;
      for (int i=0; i<A.cols; i++)
        d=d+A(l,i)*B(i,c);
      D(l,c)=d;
    }
}

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("Convencional: Multiplica duas matrizes nxn\n");
    printf("Convencional n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D;
  multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);
}
```

Vou fazer testes em dois computadores:
C1: Intel i7-2670QM 2.2GHz 4núcleos 6GBytes com NVidia GT555M (2GBytes)
C2: Intel i7-4500U 1.80GHz 2 núcleos 8GBytes com Intel HD Graphics 4400

No meu computador, este programa demora (usando seed=7):

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C2-Convencional (s)	0.31	28.76	?

O algoritmo é $O(n^3)$, onde n é número de linhas e colunas.
O algoritmo é $O(m^{1,5})$, onde $m=n^2$ é o número de elementos das matrizes ou tamanho do dado.

2) Algoritmo do OpenCV. Não sei exatamente o que este algoritmo faz.

```
// OpenCV.cpp
// Usando algoritmo de multiplicacao de opencv
// Para compilar: Compila opencv -cek
#include <cekeikon.h>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B,
Mat_<float>& D)
...

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
{ if (A.size()!=B.size()) erro("Erro diferenca");
  int conta=0;
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<A.cols; c++)
      if (abs(A(l,c)-B(l,c))>abs(A(l,c)/100000.0)) conta++;
  return conta;
}

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("OpenCV: Multiplica duas matrizes nxn\n");
    printf("  Usa algoritmo OpenCV e compara com alg. convencional\n");
    printf("OpenCV n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D;
  multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> E;
  E=A*B;
  int t4=centseg()-t3;
  printf("Tempo gasto OpenCV: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

No meu computador usando GCC 32 bits, este programa demora (usando seed=7):

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C2-OpenCV (s)	0.22	2.61	20.92

A função “diferenca” conta o número de elementos diferentes. Ela deve devolver zero se a implementação estiver correta.

Algoritmo “OpenCV” é bem mais rápido do que o algoritmo convencional (cerca de 10 vezes mais rápido). Será que vamos conseguir uma implementação melhor do que o do OpenCV?

3a) Multiplicação matricial multi-thread, disparando um thread por elemento da matriz-resultado:

```
// MThread1.cpp
// Um thread por elemento da matriz resultado
// Para compilar: Compila mthread1 -cek
#include <cekeikon.h>
#include <thread>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

namespace MTHREAD1 {
  Mat_<float> A,B,D;

  void mult(int l, int c) {
    float d=0.0;
    for (int i=0; i<A.cols; i++)
      d=d+A(l,i)*B(i,c);
    D(l,c)=d;
  }
}

void multMatMThread1(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ D.create(A.rows,B.cols);
  MTHREAD1::A=A; MTHREAD1::B=B; MTHREAD1::D=D;
  vector<thread> threads(D.total());
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<B.cols; c++)
      threads[l*B.cols+c] = thread(MTHREAD1::mult,l,c);
  for (unsigned i=0; i<D.total(); i++) threads[i].join();
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("MThread1: Multiplica duas matrizes nxn\n");
    printf("  Gera um thread por elemento da matriz resultado\n");
    printf("MThread1 n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D;
  multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> E;
  multMatMThread1(A,B,E);
  int t4=centseg()-t3;
  printf("Tempo gasto mthread1: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

No meu computador, este programa demora (usando seed=7):

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C1-MThread1 (s)	116.64s	?	?
C2-MThread1 (s)	61.01s	?	?

Algoritmo “multi-thread com um thread por elemento” demora bem mais do que o algoritmo convencional ou OpenCV! Parece que a quantidade excessiva de threads faz o programa demorar. No próximo exemplo, vamos diminuir o número de threads, gerando um thread por linha da matriz-resultado.

Parece que o compilador 32bits GCC/G++ 4.8.1 ainda não implementou inteiramente o processamento pararelo definido em C++11. Ele é incapaz de passar parâmetros por referência para threads, usando operador &. Assim, declarei as matrizes A, B e D como variáveis globais dentro do namespace MTHREAD1. Isto foi um artifício para contornar defeito do compilador.

Deve-se incluir:

```
#include <thread>
```

O comando:

```
thread(MTHREAD1::mult,l,c);
```

cria um novo thread chamando função **`MTHREAD1::mult`** com parâmetros l e c.

O comando:

```
threads[i].join();
```

espera **`threads[i]`** terminar.

3b) Multiplicação matricial multi-thread, disparando um thread por linha da matriz-resultado:

```
// MThread2.cpp
// Um thread por linha da matriz-resultado
// Para compilar: Compila mthread2 -cek
#include <cekeikon.h>
#include <thread>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

namespace MTHREAD2 {
  Mat_<float> A,B,D;

  void mult(int l) {
    for (int c=0; c<B.cols; c++) {
      float d=0.0;
      for (int i=0; i<A.cols; i++)
        d=d+A(l,i)*B(i,c);
      D(l,c)=d;
    }
  }
}

void multMatMThread2(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ D.create(A.rows,B.cols);
  MTHREAD2::A=A; MTHREAD2::B=B; MTHREAD2::D=D;
  vector<thread> threads(A.rows);
  for (int l=0; l<A.rows; l++)
    threads[l] = thread(MTHREAD2::mult,l);
  for (int i=0; i<A.rows; i++) threads[i].join();
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("MThread2: Multiplica duas matrizes nxn\n");
    printf("  Gera um thread por linha da matriz-resultado\n");
    printf("MThread2 n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D;
  multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> E;
  multMatMThread2(A,B,E);
  int t4=centseg()-t3;
  printf("Tempo gasto mthread2: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

No meu computador, este programa demora (usando seed=7):

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C1-MThread1 (s)	116.64	?	?
C1-MThread2 (s)	0.20	7.76	77.49
C2-MThread2 (s)	0.20	15.85	147.07

Algoritmo “multi-thread com um thread por linha” é aproximadamente 3,5 vezes mais rápido do que o algoritmo com um thread. Isto faz sentido, considerando que o meu computador C1 possui 4 núcleos. Computador C2 demora aproximadamente duas vezes mais que C1.

Porém, “multi-thread com um thread por linha” é aproximadamente 3 vezes mais lento do que o algoritmo do OpenCV.

O tempo de processamento diminuiu sensivelmente quando diminuimos o número de threads. Será que se diminuir o número de threads para 4 (número de núcleos do meu processador) o tempo de processamento melhora? Veremos na próxima implementação.

3c) Multiplicação matricial multi-thread, disparando 8 threads (4 núcleos com hyperthreading dá 8 threads):

```
// MThread3.cpp
// Quatro threads
// Para compilar: Compila mthread3.cpp -cek
#include <cekeikon.h>
#include <thread>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

namespace MTHREAD3 {
  //gcc 4.8.1 nao aceita passagem por ref no thread
  Mat_<float> a,b,d;
  int nthreads;

  void mult(int t) {
    int m=teto(a.rows,nthreads);
    // calcula A.rows/nthreads arredondado para cima
    int inic=t*m;
    int fim=min( (t+1)*m, a.rows );
    for (int l=inic; l<fim; l++) {
      for (int c=0; c<b.cols; c++) {
        float s=0.0;
        for (int i=0; i<a.cols; i++)
          s=s+a(l,i)*b(i,c);
        d(l,c)=s;
      }
    }
  }
}

void multMatMThread3(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ using namespace MTHREAD3;
  D.create(A.rows,B.cols);
  a=A; b=B; d=D;

  nthreads=8;
  vector<thread> threads(nthreads);
  for (int t=0; t<nthreads; t++)
    threads[t] = thread(mult,t);
  for (int t=0; t<nthreads; t++) threads[t].join();
  // Espera todos threads terminarem
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("MThread3: Multiplica duas matrizes nxn\n");
    printf("  Gera 8 threads\n");
    printf("MThread3 n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D(n,n,0.0f);
  //multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
```

```
  Mat_<float> E;
  multMatMThread3(A,B,E);
  int t4=centseg()-t3;
  printf("Tempo gasto mthread3: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C1-MThread1 (s)	116.64	?	?
C1-MThread2 (s)	0.20	7.76	77.49
C1-MThread3 (s)	0.14	7.79	73.90
C2-MThread3 (s)	0.16	15.57	147.02

Parece que não houve melhora no tempo ao diminuir o número de threads para 8. Porém disparar número de threads igual ao número de linhas *n* pode dar erro se *n* for muito grande. É mais seguro disparar um número fixo (e pequeno) de threads.

O comando:

```
int d=teto(A.rows,nthreads);
```

Arredonda para cima o resultado da divisão A.rows/nthreads.

Com isso, chegamos à máxima aceleração possível de ser obtida usando o paralelismo oferecido pelos 4 núcleos. Cada processador tem quantidade de núcleos diferente. Por exemplo, i3 e i5 possuem 2 núcleos, i7 possui 4 núcleos e Xeon pode possuir até 10 núcleos.

4a) Nesta seção, utilizaremos instruções SIMD para acelerar a multiplicação matricial. Utilizando instruções SSE2, é possível efetuar quatro operações em ponto flutuante ao mesmo tempo.

Em vez de usar linguagem de máquina, é possível usar instruções SIMD em C/C++ usando "intrinsics". A lista dessas instruções com descrição está em:

http://software.intel.com/sites/landingpage/IntrinsicsGuide/

Para verificar quais conjuntos de instruções SIMD o seu processador suporta, instale e rode:

"CPUID CPU-Z"

Tome certo cuidado ao instalar este programa. Há certos programas falsos (com vírus) que parecem ser "CPUID CPU-Z".

```
// http://www.codeproject.com/Articles/4522/Introduction-to-SSE-Programming
// compila sse.cpp -cek -sse

// <mmintrin.h>  MMX
// <xmmintrin.h> SSE
// <emmintrin.h> SSE2
// <pmmintrin.h> SSE3
// <tmmintrin.h> SSSE3
// <smmintrin.h> SSE4.1
// <nmmintrin.h> SSE4.2
// <ammintrin.h> SSE4A
// <wmmintrin.h> AES
// <immintrin.h> AVX

#include <cekeikon.h>
#include <emmintrin.h>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

void multMatSSE(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ int n=B.cols;
  int nextN=teto(n,4)*4;
  //teto(n,4) calcula n/4 arredondado para cima
  int nLoop=nextN/4;

  // n linhas, nextN colunas
  float* a = (float*)_aligned_malloc(n*nextN * sizeof(float), 16);
  if (a==NULL) erro("Erro a");
  float* b = (float*)_aligned_malloc(n*nextN * sizeof(float), 16);
  if (b==NULL) erro("Erro b");

  for (int l=0; l<A.rows; l++) {
    for (int c=0; c<n; c++) a[l*nextN+c]=A(l,c);
    for (int c=n; c<nextN; c++) a[l*nextN+c]=0.0f;
  }
  for (int l=0; l<A.rows; l++) {
    for (int c=0; c<n; c++) b[l*nextN+c]=B(c,l);
    for (int c=n; c<nextN; c++) b[l*nextN+c]=0.0f;
  }
  D.create(A.rows,B.cols);

  __m128 e;
  float* pe=(float*)&e;
  __m128 m;
  __m128* pa;
  __m128* pb;
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<n; c++) {
      e=_mm_setzero_ps(); // e[0, 1, 2, 3] = 0.0
      pa = (__m128*) (&a[l*nextN]);
      pb = (__m128*) (&b[c*nextN]);
      for (int i=0; i<nLoop; i++) {
        //e=e+a[l][i]*b[c][i];
```

```
        m = _mm_mul_ps(*pa, *pb);  // m = *pa * *pb
        e = _mm_add_ps(e, m);      // e = e + m
        pa++;
        pb++;
      }
      D(l,c)=pe[0]+pe[1]+pe[2]+pe[3];
    }
  _aligned_free(a);
  _aligned_free(b);
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("SSE: Multiplica duas matrizes nxn\n");
    printf("  Utiliza instrucoes SSE2\n");
    printf("SSE n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D(n,n,0.0f);
  multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> E;
  multMatSSE(A,B,E);
  int t4=centseg()-t3;
  printf("Tempo gasto SSE: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C1-MThread1 (s)	116.64	?	?
C1-MThread2 (s)	0.20	7.76	77.49
C1-MThread3 (s)	0.14	7.79	73.90
C1-SSE (s)	0.09	1.45	10.85
C2-SSE (s)	0.07	1.08	8.58

Usando instruções SSE2, conseguimos um programa duas vezes mais rápido que a implementação de OpenCV! E 20 vezes mais rápido que a implementação convencional!

Deve-se incluir

```
#include <emmintrin.h>
```

Para usar instruções SSE2. Deve-se compilar com opções:

```
>compila sse.cpp -cek -sse
```

As instruções SSE2 funcionam em “pacotes” de 128 bits (ou 16 bytes). Para poder usá-las, é preciso alinhar as estruturas dados no início de 16 bytes. Assim, criamos vetores alinhados com função _aligned_malloc. As matrizes A e B originais são copiados em vetores a e b ali-

nhados. A transposta do vetor B é copiada em b, para acessar uma coluna original de B como uma linha em b.

As variáveis usadas por instruções SSE2 são do tipo **__m128** (128 bits ou 16 bytes). Armazenaremos 4 variáveis float em cada **__m128**. A instrução:

```
m = _mm_mul_ps(*pa, *pb);  // m = (*pa) * (*pb)
```

Multiplica 4 float ao mesmo tempo. De modo semelhante:

```
e = _mm_add_ps(e, m);      // e = e + m
```

Soma 4 float ao mesmo tempo.

4b) Nesta seção, utilizaremos instruções AVX, que trabalha com registradores de 256 bits (ou 32 bytes, ou 8 variáveis float). As instruções AVX foram criadas em 2011 e só estão presentes em processadores modernos.

```
// avx.cpp
// compila avx.cpp -cek -avx

// <mmintrin.h>  MMX
// <xmmintrin.h> SSE
// <emmintrin.h> SSE2
// <pmmintrin.h> SSE3
// <tmmintrin.h> SSSE3
// <smmintrin.h> SSE4.1
// <nmmintrin.h> SSE4.2
// <ammintrin.h> SSE4A
// <wmmintrin.h> AES
// <immintrin.h> AVX

#include <cekeikon.h>
#include <immintrin.h>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

void multMatAVX(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ int n=B.cols;
  int nextN=teto(n,8)*8;
  int nLoop=nextN/8;

  float* a = (float*)_aligned_malloc(n*nextN * sizeof(float), 32);
  if (a==NULL) erro("Erro a");
  float* b = (float*)_aligned_malloc(n*nextN * sizeof(float), 32);
  if (b==NULL) erro("Erro b");

  for (int l=0; l<A.rows; l++) {
    for (int c=0; c<n; c++) a[l*nextN+c]=A(l,c);
    for (int c=n; c<nextN; c++) a[l*nextN+c]=0.0f;
  }
  for (int l=0; l<A.rows; l++) {
    for (int c=0; c<n; c++) b[l*nextN+c]=B(c,l);
    for (int c=n; c<nextN; c++) b[l*nextN+c]=0.0f;
  }
  D.create(A.rows,B.cols);

  __m256 e;
  float* pe=(float*)&e;
  __m256 m;
  __m256* pa;
  __m256* pb;
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<n; c++) {
      e=_mm256_setzero_ps(); // e[0, 1, 2, 3] = 0.0
      pa = (__m256*) (&a[l*nextN]);
      pb = (__m256*) (&b[c*nextN]);
      for (int i=0; i<nLoop; i++) {
        //e=e+a[l][i]*b[c][i];
        m = _mm256_mul_ps(*pa, *pb);  // m = *pa * *pb
        e = _mm256_add_ps(e, m);      // e = e + m
        pa++;
        pb++;
      }
      D(l,c)=pe[0]+pe[1]+pe[2]+pe[3]+pe[4]+pe[5]+pe[6]+pe[7];
    }
  _aligned_free(a);
  _aligned_free(b);
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=3) {
```

```
    printf("AVX: Multiplica duas matrizes nxn\n");
    printf("  Utiliza instrucoes AVX 256 bits\n");
    printf("AVX n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D(n,n,0.0f);
  //multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> E;
  multMatAVX(A,B,E);
  int t4=centseg()-t3;
  printf("Tempo gasto AVX: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C1-MThread1 (s)	116.64	?	?
C1-MThread2 (s)	0.20	7.76	77.49
C1-MThread3 (s)	0.14	7.79	73.90
C1-SSE (s)	0.09	1.45	10.85
C1-AVX (s)	0.06	1.25	9.70
C2-AVX (s)	0.06	0.89	7.35

O tempo de processamento melhorou um pouco em relação a instruções SSE2, mas não substancialmente.

4c) Nesta seção, utilizaremos instruções AVX sem criar matrizes alinhadas em 32 bytes. Muitas vezes, não é possível utilizar estruturas alinhadas. Testaremos quanto o processamento fica mais lento.

```
// avx2.cpp
#include <cekeikon.h>
#include <immintrin.h>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

void multMatAVX2(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ assert(A.cols==B.rows);
  Mat_<float> BT=B.t();
  D.create(A.rows,B.cols);
  __m256 a,b,m,e;
  int limite=8*(A.cols/8);
  float* pe=(float*)(&e);
  float s;
  for (int l=0; l<A.rows; l++) {
    for (int c=0; c<B.cols; c++) {
      e=_mm256_setzero_ps(); // e[0,1,2,3,4,5,6,7] = 0.0
      for (int i=0; i<limite; i+=8) {
        //e=e+A[l][i]*BT[c][i];
        a=_mm256_loadu_ps(&A(l,i));
        b=_mm256_loadu_ps(&BT(c,i));
        m=_mm256_mul_ps(a, b);  // m = a * b
        e=_mm256_add_ps(e, m);  // e = e + m
      }
      s=0;
      for (int i=limite; i<A.cols; i++) {
        s=s+A[l][i]*BT[c][i];
      }
      D(l,c)=pe[0]+pe[1]+pe[2]+pe[3]+pe[4]+pe[5]+pe[6]+pe[7]+s;
    }
  }
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("AVX2: Multiplica duas matrizes nxn\n");
    printf("  Sem criar matriz alinhada\n");
    printf("  Utiliza instrucoes AVX 256 bits\n");
    printf("AVX2 n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }
  ...
}
```

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C1-MThread1 (s)	116.64	?	?
C1-MThread2 (s)	0.20	7.76	77.49
C1-MThread3 (s)	0.14	7.79	73.90
C1-SSE (s)	0.09	1.45	10.85
C1-AVX (s)	0.06	1.25	9.70
C1-AVX2 (s)	0.12	1.72	13.65
C2-AVX2 (s)	0.10	1.39	11.28

O tempo de processamento aumenta, mas ainda é bom.

5) Nesta seção, utilizaremos instruções AVX (para fazer 8 operações float ao mesmo tempo) junto com multithread (para utilizar todos os 4 cores do meu computador C1). Com isso, efetuaremos 32 operações float ao mesmo tempo. Dispararemos um thread por linha da matriz-resultado.

```
// avx-th.cpp
// compila avx-th.cpp -cek -avx

#include <cekeikon.h>
#include <immintrin.h>
#include <thread>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

namespace AVXTH {

float* AN; float* BN; Mat_<float> DN;
int rows, cols, nextN, nLoop;
int nthreads;

void mult(int t)
{
  __m256 e;
  float* pe=(float*)&e;
  __m256 m;
  __m256* pa;
  __m256* pb;

  int d=teto(rows,nthreads);
  int inic=t*d;
  int fim=min( (t+1)*d, rows );

  for (int l=inic; l<fim; l++) {
    for (int c=0; c<cols; c++) {
      e=_mm256_setzero_ps(); // e[0, 1, 2, 3] = 0.0
      pa = (__m256*) (&AN[l*nextN]);
      pb = (__m256*) (&BN[c*nextN]);
      for (int i=0; i<nLoop; i++) {
        //e=e+AN[l][i]*BN[c][i];
        m = _mm256_mul_ps(*pa, *pb);  // m = *pa * *pb
        e = _mm256_add_ps(e, m);      // e = e + m
        pa++;
        pb++;
      }
      DN(l,c)=pe[0]+pe[1]+pe[2]+pe[3]+pe[4]+pe[5]+pe[6]+pe[7];
    }
  }
}

} // namespace

void multMatAvxTh(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ using namespace AVXTH;
  cols=B.cols;
  rows=A.rows;
  nextN=teto(cols,8)*8;
  nLoop=nextN/8;
  //printf("cols=%d nextN=%d nLoop=%d\cols",cols,nextN,nLoop);

  AN = (float*)_aligned_malloc(cols*nextN * sizeof(float), 32);
  if (AN==NULL) erro("Erro AN");
  BN = (float*)_aligned_malloc(cols*nextN * sizeof(float), 32);
  if (BN==NULL) erro("Erro BN");

  for (int l=0; l<cols; l++) {
    for (int c=0; c<cols; c++) AN[l*nextN+c]=A(l,c);
    for (int c=cols; c<nextN; c++) AN[l*nextN+c]=0.0f;
  }
  for (int l=0; l<cols; l++) {
    for (int c=0; c<cols; c++) BN[l*nextN+c]=B(c,l);
```

```
    for (int c=cols; c<nextN; c++) BN[l*nextN+c]=0.0f;
  }
  D.create(cols,cols); DN=D;

  nthreads=4;
  vector<thread> threads(nthreads);
  for (int t=0; t<nthreads; t++) threads[t]=thread(mult,t);
  for (int t=0; t<nthreads; t++) threads[t].join();

  _aligned_free(AN);
  _aligned_free(BN);
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("AVX: Multiplica duas matrizes nxn\n");
    printf("  Utiliza instrucoes AVX 256 bits\n");
    printf("AVX n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);

  int t1=centseg();
  Mat_<float> D(n,n,0.0f);
  //multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> E;
  multMatAvxTh(A,B,E);
  int t4=centseg()-t3;
  printf("Tempo gasto AVX: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

n	650	1500	3000
C1-Convencional (s)	0.38	28.44	251.07
C1-OpenCV (s)	0.23	3.02	23.83
C1-MThread1 (s)	116.64	?	?
C1-MThread2 (s)	0.20	7.76	77.49
C1-MThread3 (s)	0.14	7.79	73.90
C1-SSE (s)	0.09	1.45	10.85
C1-AVX (s)	0.06	1.25	9.70
C1-AVX2 (s)	0.12	1.72	13.65
C1-AVX-TH (s)	0.01	0.34	3.58
C2-AVX-TH (s)	0.03	0.64	5.74

Com isso, conseguimos uma implementação 70 vezes mais rápido que a convencional e 6 vezes mais rápido que a do OpenCV. Note que:
1) Nem todos os problemas podem ser paralelizados.
2) Para poder usar todos os recursos do pararelismo, a implementação paralela torna-se bem mais complexa do que a serial convencional.

6a) Nesta seção, utilizaremos OpenCL para usar o processamento do GPU.

O meu computador C1 possui uma placa nVidia GT555M com 2GBytes e 144 núcleos. Com isso, é possível efetuar 144 operações simultaneamnte. Além disso, trabalha com

ente, mais do que 8x4=32 operações simultâneas

Como veremos, OpenCL não consegue uma aceleração substancial (pelo menos usando o meu GPU). Além disso, há vários problemas de software (como compilador que não dá mensagem de erro e gera programa que trava) e o programa instável (o programa que multiplica matrizes funciona corretamente para matrizes de até aproximadamente 650x650 mas trava para matrizes grandes sem explicação plausível; outro programa roda corretamente 4 vezes e trava 1 vez; etc). Não parece ser defeito de hardware, pois essa placa funciona perfeitamente para computação gráfica (usando OpenGL). Mas dá problemas em OpenCL e CUDA.

Há duas formas de programar GPU:
1) CUDA: Só funciona para placas nVidia com compilador VisualC.
2) OpenCL: Funciona com placas nVidia e AMD. Funciona com compiladores VisualC e GCC. Porém, há várias funções que só funcionam com placa AMD (ex: álgebra linear e FFT).

Apresento abaixo multiplicação matricial que funciona até aproximadamente 650x650. Dispara um processo para cada elemento da matriz-resultado.

```
// ocl1.cpp
// compila ocl1.cpp -cek -ocl
// Um thread por elemento

#include <cekeikon.h>
#include <CL/cl.hpp>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

void inicializaOCL(string nome, const string& code, cl::Context& context,
                cl::CommandQueue& queue, cl::Kernel& kernel)
{ //get all platforms (drivers)
  vector<cl::Platform> all_platforms;
  cl::Platform::get(&all_platforms);
  if(all_platforms.size()==0) {
    cout << " No platforms found. Check OpenCL installation!\n"; exit(1);
  }
  cl::Platform default_platform=all_platforms[0];
  cout << "Using platform: " << default_platform.getInfo<CL_PLATFORM_NAME>() << "\n";

  //get default device of the default platform
  vector<cl::Device> all_devices;
  default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
  if(all_devices.size()==0){
    cout <<" No devices found. Check OpenCL installation!\n"; exit(1);
  }
  cl::Device device=all_devices[0];
  cout << "Using device: " << device.getInfo<CL_DEVICE_NAME>() << "\n";

  cl::Context contextTemp({device}); context=contextTemp;
  cl::Program::Sources sources;
  sources.push_back({code.c_str(),code.length()});

  cl::Program program(context,sources);
  if(program.build({device})!=CL_SUCCESS){
    cout << " Error building: " <<
      program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device) <<"\n";
    exit(1);
```

```
  }
  cl::CommandQueue queueTemp(context,device); queue=queueTemp;
  cl::Kernel kernelTemp(program,nome.c_str()); kernel=kernelTemp;
}

void multMatOCL(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ assert(A.rows==A.cols && A.rows==B.rows && B.rows==B.cols);

  string code=
    "__kernel void processa(__global float* A,    \n"
    "    __global float* B, __global float* D,    \n"
    "    int n) {                                 \n"
    "  float d=0.0;                              \n"
    "  int l=get_global_id(0);                    \n"
    "  int c=get_global_id(1);                    \n"
    "  int i;                                     \n"
    "  int a=l*n;                                 \n"
    "  int b=c;                                   \n"
    "  for (i=0; i<n; i++) {                      \n"
    "    d=d+A[a]*B[b];                           \n"
    "    a++;                                     \n"
    "    b+=n;                                    \n"
    "  }                                          \n"
    "  D[l*n+c]=d;                                \n"
    "}                                            \n";
  cl::Context context;
  cl::CommandQueue queue;
  cl::Kernel kernel;
  inicializaOCL("processa",code,context,queue,kernel);

  int n=A.rows;
  // create buffers on the device
  cl::Buffer buffer_A(context,CL_MEM_READ_ONLY, sizeof(float)*n*n);
  cl::Buffer buffer_B(context,CL_MEM_READ_ONLY, sizeof(float)*n*n);
  cl::Buffer buffer_D(context,CL_MEM_WRITE_ONLY,sizeof(float)*n*n);

  //write matrices A and B to the device
  queue.enqueueWriteBuffer(buffer_A,CL_TRUE,0,sizeof(float)*n*n,A.data);
  queue.enqueueWriteBuffer(buffer_B,CL_TRUE,0,sizeof(float)*n*n,B.data);

  //run the kernel
  cl::KernelFunctor processa(kernel, queue,
                       cl::NullRange, cl::NDRange(n,n), cl::NullRange);
  processa(buffer_A, buffer_B, buffer_D, n);

  //read result D from the device to array D
  D.create(n,n);
  queue.enqueueReadBuffer(buffer_D,CL_TRUE,0,sizeof(float)*n*n,D.data);
}

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
{ assert(A.size()==B.size());
  int conta=0;
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<A.cols; c++)
      if (abs(A(l,c)-B(l,c))>abs(A(l,c)/100000.0)) conta++;
  return conta;
}

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("OCL: Multiplica duas matrizes nxn\n");
    printf("  Utiliza instrucoes OpenCL\n");
    printf("OCL n seed\n");
    printf("  As matrizes sao preenchidas com numeros de 0 a n-1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);
  Mat_<float> A,B;
  inicializa(A,B,n,seed);
```

```
  int t1=centseg();
  Mat_<float> D(n,n,0.0f);
  multMatConvencional(A,B,D);
  int t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> E;
  multMatOCL(A,B,E);
  int t4=centseg()-t3;
  printf("Tempo gasto OCL: %d.%02ds\n",t4/100,t4%100);

  int conta=diferenca(D,E);
  printf("Numero de elementos diferentes: %d\n",conta);
}
```

n	650	1500	3000
Convencional (s)	0.38	28.44	251.07
OpenCV (s)	0.23	3.02	23.83
MThread1 (s)	116.64	?	?
MThread2 (s)	0.20	7.76	77.49
MThread3 (s)	0.14	7.79	73.90
SSE (s)	0.09	1.45	10.85
AVX (s)	0.06	1.25	9.70
AVX2 (s)	0.12	1.72	13.65
AVX-TH (s)	0.01	0.34	3.58
OCL1 (s)	1.38	erro	erro

Por motivo desconhecido, o programa OpenCL só multiplica matrizes de até 650×650.

O tempo de processamento 1.38s do OCL inclui o tempo para compilar e linkar kernel.

6b) OpenCL disparando um thread por linha.

```
// ocl2.cpp
// compila ocl2.cpp -cek -ocl
// Um thread por linha

#include <cekeikon.h>
#include <CL/cl.hpp>

void inicializa(Mat_<float>& A, Mat_<float>& B, int n, int seed)
...

void multMatConvencional(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
...

void inicializaOCL(string nome, const string& code, cl::Context& context,
                cl::CommandQueue& queue, cl::Kernel& kernel)
...

void multMatOCL(const Mat_<float>& A, const Mat_<float>& B, Mat_<float>& D)
{ assert(A.rows==A.cols && A.rows==B.rows && B.rows==B.cols);
  string code=
    "__kernel void processa(__global float* A,   \n"
    "     __global float* B, __global float* D,   \n"
    "     int n) {                                \n"
    "  float d;;                                  \n"
    "  int l=get_global_id(0);                    \n"
    "  int c,i;                                   \n"
    "  for (c=0; c<n; c++) {                      \n"
    "    d=0.0;                                   \n"
    "    for (i=0; i<n; i++) {                    \n"
    "      d += A[l*n+i]*B[i*n+c];                \n"
    "    }                                        \n"
    "    D[l*n+c]=d;                              \n"
    "  }                                          \n"
    "}                                            \n";
  cl::Context context;
  cl::CommandQueue queue;
  cl::Kernel kernel;
  inicializaOCL("processa",code,context,queue,kernel);

  int n=A.rows;
  // create buffers on the device
  cl::Buffer buffer_A(context,CL_MEM_READ_ONLY, sizeof(float)*n*n);
  cl::Buffer buffer_B(context,CL_MEM_READ_ONLY, sizeof(float)*n*n);
  cl::Buffer buffer_D(context,CL_MEM_WRITE_ONLY,sizeof(float)*n*n);

  //write matrices A and B to the device
  queue.enqueueWriteBuffer(buffer_A,CL_TRUE,0,sizeof(float)*n*n,A.data);
  queue.enqueueWriteBuffer(buffer_B,CL_TRUE,0,sizeof(float)*n*n,B.data);

  //run the kernel
  cl::KernelFunctor processa(kernel, queue,
                       cl::NullRange, cl::NDRange(n), cl::NullRange);
  processa(buffer_A, buffer_B, buffer_D, n);

  //read result D from the device to array D
  D.create(n,n);
  queue.enqueueReadBuffer(buffer_D,CL_TRUE,0,sizeof(float)*n*n,D.data);
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
...
```

n	650	1500	3000
Convencional (s)	0.38	28.44	251.07
OpenCV (s)	0.23	3.02	23.83
MThread1 (s)	116.64	?	?
MThread2 (s)	0.20	7.76	77.49
MThread3 (s)	0.14	7.79	73.90
SSE (s)	0.09	1.45	10.85
AVX (s)	0.06	1.25	9.70
AVX2 (s)	0.12	1.72	13.65
AVX-TH (s)	0.01	0.34	3.58
OCL1 (s)	1.38	erro	erro
OCL2 (s)	2.30	erro	erro

Disparando um thread por linha, o programa fica mais lento do que disparar um thread por elemento.

Moral da história: OpenCL é uma tecnologia nova e ainda não é suficientemente boa para ser usada na prática.

A tabela de tempos usando compilador GCC 64 bits é:

n	650	1500	3000
Convencional 64 (s)	0.36	27.94	247.94
OpenCV 64 (s)	0.23	3.05	23.35
MThread1 64 (s)	146.06	?	?
MThread2 64 (s)	0.20	7.71	76.75
MThread3 64 (s)	0.14	7.88	79.45
SSE 64 (s)	0.06	1.28	10.38
AVX 64 (s)	0.04	1.17	9.70
AVX-TH 64 (s)	0.01	0.38	3.68

Por motivo desconhecido, não dá para usar OpenCL usando GCC 64 bits.

Conclusão: Os tempos de processamento de programa compilado em GCC 32 bits e 64 bits são praticamente iguais.

II. Usar funções prontas de OpenCL do OpenCV.

Nesta seção, vamos rodar funções prontas do OpenCV que usa OpenCL. É muito mais simples do que programar diretamente OpenCL. O programa abaixo aplica 100 vezes média móvel 9x9.

```
//filter2d.cpp
#include <cekeikon.h>
#include "opencv2/ocl/ocl.hpp"
//#include "opencv2/nonfree/ocl.hpp"
//#include "opencv2/nonfree/nonfree.hpp"
using namespace cv::ocl;

int main(int argc, char** argv)
{
  if (argc!=2) {
    printf("Filter2d: Aplica 100 vezes media movel 9x9 usando OCL e CPU\n");
    printf("Filter2d ent.jpg\n");
    erro("Erro: Numero de argumentos invalido");
  }

  Mat_<GRY> a; le(a,argv[1]);

  cout
    << "Device name:"
    << cv::ocl::Context::getContext()->getDeviceInfo().deviceName
    << endl << endl;

  int t1=centseg();
  oclMat A; A=a;
  oclMat B;
  for (int i=0; i<100; i++) {
    ocl::filter2D(A,B,-1,(1.0/(9*9))*Mat_<FLT>(9,9,1.0f));
    B.copyTo(A);
  }
  Mat_<GRY> b; b=B;
  int t2=centseg()-t1;
  printf("OCL=%d.%02ds\n",t2/100,t2%100);
  mostra(b,"OCL");

  int t3=centseg();
  for (int i=0; i<100; i++) {
    filter2D(a,b,-1,(1.0/(9*9))*Mat_<FLT>(9,9,1.0f));
    b.copyTo(a);
  }
  int t4=centseg()-t3;
  printf("CPU=%d.%02ds\n",t4/100,t4%100);
  mostra(b,"CPU");
}
```

oclMat é a estrutura para armazenar imagem na memória do GPU.

Os tempos de processamento foram:

```
Device name:GeForce GT 555M
OCL=0.68s
CPU=0.78s
```

Nota: ocl::gemm que deveria efetuar multiplicação matricial em OpenCV-OCL, não funciona em nVidia (só em AMD).

III. Filtro Linear

1) Vamos fazer primeiro filtro linear 8x8:

```
// filtro1.cpp
// Implementa filtro linear 8x8 ou menor usando instrucoes AVX.
// Este programa pode nao rodar nos processadores um pouco antigos.
// Neste caso, reescreva usando instrucoes SSE
// compila filtro1.cpp -cek -avx
#include <cekeikon.h>
#include <immintrin.h>

void inicializa(Mat_<float>& A, int n, int seed)
{ A.create(n,n);
  mysrand(seed);
  for (int l=0; l<n; l++)
    for (int c=0; c<n; c++)
      A(l,c)=myuniform();
}

Mat_<float> filtro88convencional(const Mat_<float>& A, const Mat_<float>& F)
{ assert(F.rows==8 && F.cols==8);
  Mat_<float> AE(A.rows+7,A.cols+7,0.0f);
  Mat_<float> roi(AE,Rect(4,4,A.rows,A.cols));
  A.copyTo(roi);

  Mat_<float> D(A.rows,A.cols);

  for (int l=0; l<A.rows; l++)
    for (int c=0; c<A.cols; c++) {
      float s=0.0f;
      for (int l2=0; l2<8; l2++)
        for (int c2=0; c2<8; c2++)
          s += AE(l+l2,c+c2)*F(l2,c2);
      D(l,c)=s;
    }
  return D;
}

Mat_<float> filtro88avx(const Mat_<float>& A, const Mat_<float>& F)
{ assert(F.rows==8 && F.cols==8);
  Mat_<float> AE(A.rows+7,A.cols+7,0.0f);
  Mat_<float> roi(AE,Rect(4,4,A.rows,A.cols));
  A.copyTo(roi);

  float* FA = (float*)_aligned_malloc(8*8 * sizeof(float), 32);
  assert(FA!=NULL);
  for (int i=0; i<64; i++) FA[i]=F(i);

  Mat_<float> D(A.rows,A.cols);

  __m256 a,f,s,m;
  float* ps=(float*)(&s);
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<A.cols; c++) {
      s=_mm256_setzero_ps(); // s[0,1,2,3,4,5,6,7] = 0.0
      for (int i=0; i<8; i++) {
        //s=s+AE[l+i][c]*FA[i];
        a=_mm256_loadu_ps(&AE(l+i,c)); // carga nao-alinhada
        f=_mm256_load_ps(&FA[8*i]); // carga alinhada
        m=_mm256_mul_ps(a, f);  // m = a * f
        s=_mm256_add_ps(s, m);  // s = s + m
      }
      D(l,c)=ps[0]+ps[1]+ps[2]+ps[3]+ps[4]+ps[5]+ps[6]+ps[7];
    }
  _aligned_free(FA);
  return D;
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
{ if (A.size()!=B.size()) erro("Erro diferenca");
  int conta=0;
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<A.cols; c++)
```

```
      if (abs(A(l,c)-B(l,c))>abs(A(l,c)/100000.0)) conta++;
  return conta;
}

int main(int argc, char** argv) {
  if (argc!=3) {
    printf("Filtro1: Aplica filtro 8x8\n");
    printf("  Utiliza instrucoes AVX 256 bits\n");
    printf("Filtro1 n seed\n");
    printf("  A imagem e' preenchida com numeros de 0 a 1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n;
  convArg(n,argv[1]);
  int seed;
  convArg(seed,argv[2]);

  Mat_<float> A;
  inicializa(A,n,seed);

  Mat_<float> F =
    ( Mat_<float>(8,8) <<
        0,1,1,1,1,1,1,0,
        1,1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,1,
        1,1,1,3,9,1,1,1,
        1,1,1,3,3,1,1,1,
        1,1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,1,
        0,1,1,1,1,1,1,0 );
  F = somatoriaUm(F);

  int t1=centseg();
  Mat_<float> B=filtro88avx(A,F);
  int t2=centseg()-t1;
  printf("Tempo gasto AVX: %d.%02ds\n",t2/100,t2%100);

  int t3=centseg();
  Mat_<float> D;
  filter2D(A,D,-1,F,Point(-1,-1),0,BORDER_CONSTANT);
  int t4=centseg()-t3;
  printf("Tempo gasto filter2D: %d.%02ds\n",t4/100,t4%100);

  int t5=centseg();
  Mat_<float> E;
  E=filtro88convencional(A,F);
  int t6=centseg()-t5;
  printf("Tempo gasto convencional: %d.%02ds\n",t6/100,t6%100);

  int conta=diferenca(B,D);
  printf("Numero de elementos diferentes BD: %d\n",conta);

  conta=diferenca(B,E);
  printf("Numero de elementos diferentes BE: %d\n",conta);
}
```

No meu computador, os tempos de processamento do Filtro1 são (usando seed=7 e janela 8x8):

n	2000	4000	6000
AVX (s)	0.10	0.36	0.78
Filter2D (s)	0.04	0.16	0.34
Convencional (s)	0.19	0.78	1.70

2) Vamos fazer filtro linear de dimensão arbitrária, usando instruções AVX e um thread:

```
// filtro2.cpp
// Filtro linear com nucleo arbitrario usando AVX
// compila filtro2.cpp -cek -avx
#include <cekeikon.h>
#include <immintrin.h>

void inicializa(Mat_<float>& A, int n, int seed)
...

Mat_<float> filtroConvencional(const Mat_<float>& A, const Mat_<float>& F)
{ Mat_<float> AE(A.rows+F.rows-1, A.cols+F.cols-1, 0.0f);
  int lc=F.rows/2;
  int cc=F.cols/2;
  Mat_<float> roi(AE, Rect(lc,cc,A.rows,A.cols) );
  A.copyTo(roi);

  Mat_<float> D(A.rows,A.cols);
  float s;
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<A.cols; c++) {
      s=0.0f;
      for (int l2=0; l2<F.rows; l2++)
        for (int c2=0; c2<F.cols; c2++)
          s += AE(l+l2,c+c2)*F(l2,c2);
      D(l,c)=s;
    }
  return D;
}

Mat_<float> filtroAvx(const Mat_<float>& A, const Mat_<float>& F)
{ Mat_<float> AE(A.rows+F.rows-1,A.cols+F.cols-1,0.0f);
  int lc=F.rows/2;
  int cc=F.cols/2;
  Mat_<float> roi(AE, Rect(lc,cc,A.rows,A.cols) );
  A.copyTo(roi);

  int nextCols=teto(F.cols,8)*8;
  float* FA = (float*)_aligned_malloc( F.rows*nextCols*sizeof(float), 32 );
  assert(FA!=NULL);
  for (int i=0; i<F.rows*nextCols; i++) FA[i]=0.0f;
  for (int l=0; l<F.rows; l++)
    for (int c=0; c<F.cols; c++)
      FA[l*nextCols+c]=F(l,c);

  Mat_<float> D(A.rows,A.cols);
  __m256 a,f,s,m;
  float* ps=(float*)(&s);
  for (int l=0; l<A.rows; l++)
    for (int c=0; c<A.cols; c++) {
      s=_mm256_setzero_ps(); // s[0,1,2,3,4,5,6,7] = 0.0
      for (int l2=0; l2<F.rows; l2++)
        for (int c2=0; c2<nextCols; c2+=8) {
          //s=s+AE[l+l2][c+c2]*FA[l2][c2];
          a=_mm256_loadu_ps(&AE(l+l2,c+c2)); // carga nao-alinhada
          f=_mm256_load_ps(&FA[nextCols*l2+c2]); // carga alinhada
          m=_mm256_mul_ps(a, f);  // m = a * f
          s=_mm256_add_ps(s, m);  // s = s + m
        }
      D(l,c)=ps[0]+ps[1]+ps[2]+ps[3]+ps[4]+ps[5]+ps[6]+ps[7];
    }
  _aligned_free(FA);
  return D;
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=4) {
    printf("Filtro2: Aplica filtro mxm na imagem nxn\n");
    printf("  Utiliza instrucoes AVX 256 bits e um thread\n");
    printf("Filtro2 n m seed\n");
    printf("  A imagem e' preenchida com numeros de 0 a 1\n");
    erro("Erro: Numero de argumentos invalido");
```

```
  }

  int n; convArg(n,argv[1]);
  int m; convArg(m,argv[2]);
  int seed; convArg(seed,argv[3]);

  Mat_<float> A,F;
  inicializa(A,n,seed);
  inicializa(F,m,3*seed);
  F = somatoriaUm(F);

  int t1=centseg();
  Mat_<float> D1=filtroAvx(A,F);
  int t2=centseg()-t1;
  printf("Tempo gasto AVX: %d.%02ds\n",t2/100,t2%100);

  t1=centseg();
  Mat_<float> D2;
  filter2D(A,D2,-1,F,Point(-1,-1),0,BORDER_CONSTANT);
  t2=centseg()-t1;
  printf("Tempo gasto filter2D: %d.%02ds\n",t2/100,t2%100);

  t1=centseg();
  Mat_<float> D3=filtroConvencional(A,F);
  t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int conta=diferenca(D1,D2);
  printf("Numero de elementos diferentes D1 D2: %d\n",conta);

  conta=diferenca(D1,D3);
  printf("Numero de elementos diferentes D1 D3: %d\n",conta);
}
```

No meu computador, os tempos de processamento do Filtro2 são (usando seed=7):

	m×*m*	*n*=2000	*n*=4000	*n*=6000
AVX (s)	8×8	0.19	0.77	1.73
Filter2D (s)		0.05	0.15	0.36
Convencional (s)		0.28	1.14	2.61
AVX (s)	12×12	0.36	1.43	3.20
Filter2D (s)		0.19	0.72	1.58
Convencional (s)		0.61	2.39	5.43
AVX (s)	16×16	0.45	1.85	4.09
Filter2D (s)		0.21	0.72	1.55
Convencional (s)		1.07	4.18	9.39

Nota: A rotina filter2D do OpenCV começa usar FFT para calcular filtro a partir do *m*×*m*=11×11.

3) Vamos fazer filtro linear de dimensão arbitrária, usando instruções AVX e multi-thread:

```
// filtro3.cpp
// Filtro linear usando AVX e multi-thread
// compila filtro3.cpp -cek -avx
#include <cekeikon.h>
#include <immintrin.h>
#include <thread>

void inicializa(Mat_<float>& A, int n, int seed)
...

Mat_<float> filtroConvencional(const Mat_<float>& A, const Mat_<float>& F)
...

namespace AVXTH {

Mat_<float> AE,D; float* FA;
int Arows, Acols, Frows, nextCols, nthreads;

void filt(int t)
{
  int d=teto(Arows,nthreads);
  int inic=t*d;
  int fim=min( (t+1)*d, Arows );

  __m256 a,f,s,m;
  float* ps=(float*)(&s);
  for (int l=inic; l<fim; l++) {
    for (int c=0; c<Acols; c++) {
      s=_mm256_setzero_ps(); // s[0,1,2,3,4,5,6,7] = 0.0
      for (int l2=0; l2<Frows; l2++)
        for (int c2=0; c2<nextCols; c2+=8) {
          //s=s+AE[l+l2][c+c2]*FA[l2][c2];
          a=_mm256_loadu_ps(&AE(l+l2,c+c2)); // carga nao-alinhada
          f=_mm256_load_ps(&FA[nextCols*l2+c2]); // carga alinhada
          m=_mm256_mul_ps(a, f);  // m = a * f
          s=_mm256_add_ps(s, m);  // s = s + m
        }
      D(l,c)=ps[0]+ps[1]+ps[2]+ps[3]+ps[4]+ps[5]+ps[6]+ps[7];
    }
  }
}

} // namespace

Mat_<float> filtroAvx2(const Mat_<float>& A, const Mat_<float>& F)
{ using namespace AVXTH;
  Arows=A.rows; Acols=A.cols;
  Frows=F.rows;
  nextCols=teto(F.cols,8)*8;

  AE.create(A.rows+F.rows-1,A.cols+F.cols-1); AE.setTo(0.0f);
  int lc=F.rows/2;
  int cc=F.cols/2;
  Mat_<float> roi(AE, Rect(lc,cc,A.rows,A.cols) );
  A.copyTo(roi);

  int nextRows=teto(F.rows,8)*8;
  int nextCols=teto(F.cols,8)*8;
  FA = (float*)_aligned_malloc( nextRows*nextCols*sizeof(float), 32 );
  assert(FA!=NULL);
  for (int i=0; i<nextRows*nextCols; i++) FA[i]=0.0f;
  for (int l=0; l<F.rows; l++)
    for (int c=0; c<F.cols; c++)
      FA[l*nextCols+c]=F(l,c);

  D.create(A.rows,A.cols);

  nthreads=8;
  vector<thread> threads(nthreads);
  for (int t=0; t<nthreads; t++) threads[t]=thread(filt,t);
  for (int t=0; t<nthreads; t++) threads[t].join();

  _aligned_free(FA);
  return D;
```

```
}

int diferenca(const Mat_<float>& A, const Mat_<float>& B)
...

int main(int argc, char** argv) {
  if (argc!=4) {
    printf("Filtro3: Aplica filtro mxm na imagem nxn\n");
    printf("  Utiliza instrucoes AVX 256 bits e multi-thread\n");
    printf("Filtro3 n m seed\n");
    printf("  A imagem e' preenchida com numeros de 0 a 1\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int n; convArg(n,argv[1]);
  int m; convArg(m,argv[2]);
  int seed; convArg(seed,argv[3]);

  Mat_<float> A,F;
  inicializa(A,n,seed);
  inicializa(F,m,3*seed);
  F = somatoriaUm(F);

  int t1=centseg();
  Mat_<float> D1=filtroAvx2(A,F);
  int t2=centseg()-t1;
  printf("Tempo gasto AVX: %d.%02ds\n",t2/100,t2%100);

  t1=centseg();
  Mat_<float> D2;
  filter2D(A,D2,-1,F,Point(-1,-1),0,BORDER_CONSTANT);
  t2=centseg()-t1;
  printf("Tempo gasto filter2D: %d.%02ds\n",t2/100,t2%100);

  t1=centseg();
  Mat_<float> D3=filtroConvencional(A,F);
  t2=centseg()-t1;
  printf("Tempo gasto convencional: %d.%02ds\n",t2/100,t2%100);

  int conta=diferenca(D1,D2);
  printf("Numero de elementos diferentes D1 D2: %d\n",conta);

  conta=diferenca(D1,D3);
  printf("Numero de elementos diferentes D1 D3: %d\n",conta);
}
```

No meu computador, os tempos de processamento do Filtro3 são (usando seed=7):

	m×*m*	*n*=2000	*n*=4000	*n*=6000
AVX (s)	8×8	0.08	0.26	0.58
Filter2D (s)		0.05	0.17	0.36
Convencional (s)		0.28	1.16	2.56
AVX (s)	12×12	0.14	0.46	1.04
Filter2D (s)		0.19	0.74	1.56
Convencional (s)		0.59	2.42	5.41
AVX (s)	16×16	0.15	0.61	1.31
Filter2D (s)		0.21	0.73	1.58
Convencional (s)		1.07	4.21	9.42

Nota: A rotina filter2D do OpenCV começa usar FFT para calcular filtro a partir do *m*×*m*=11×11.