```
#include "mex.h"
#include <pthread.h>
#include <xmmintrin.h>
#include <stdint.h>

#include <iostream>
using namespace std;

/*
 * The original code was developed for Version 4.01 implementation of
 * Felzenszwalb et al 2010 [1] and was adapted by Tom Dean to sparsely
 * compute dot products only at coordinates in a salience map which is
 * generated using the Shi & Tomasi [1994] interest-point operator.
 *
 * [1] P. Felzenszwalb, R. Girshick, D. McAllester, D. Ramanan. Object
 *     Detection with Discriminatively Trained Part Based Models. IEEE
 *     Transactions on Pattern Analysis and Machine Intelligence, 2010.
 */

/* OS X aligns all memory at 16-byte boundaries (and doesn't provide
 * memalign/posix_memalign).  On linux, we use memalign to allocated
 * 16-byte aligned memory.
 */

#if !defined(__APPLE__)
#include <malloc.h>
#define malloc_aligned(a,b) memalign(a,b)
#else
#define malloc_aligned(a,b) malloc(b)
#endif

#define IS_ALIGNED(ptr) ((((uintptr_t)(ptr)) & 0xF) == 0)

#define SSE_DEPTH 32

struct thread_data {
  float *A;
  float *B;
  double *C;
  double *S;
  mxArray *mxC;
  const mwSize *S_dims;
  mwSize A_dims[3];
  mwSize B_dims[3];
  mwSize C_dims[2];
};

// convolve A and B sparsely at S
void *process(void *thread_arg) {
  thread_data *args = (thread_data *)thread_arg;
  float *A = args->A;
  float *B = args->B;
  double *C = args->C;
  double *S = args->S;

  const mwSize *A_dims = args->A_dims;
  const mwSize *B_dims = args->B_dims;
  const mwSize *C_dims = args->C_dims;
const mwSize *S_dims = args->S_dims;

int Bx_off = B_dims[1]/2 + 1;
int By_off = B_dims[0]/2 + 1;

__m128 a,b,c;
double *dst = C;
for (int x = 0; x < C_dims[1]; x++) {
  for (int y = 0; y < C_dims[0]; y++) {
    __m128 v = _mm_setzero_ps();
    const float *A_src = A + y*SSE_DEPTH + x*A_dims[0]*SSE_DEPTH;
    const float *B_src = B;
    if ( S[(x + Bx_off) * S_dims[0] + y + By_off] != 0 ) {
      for (int xp = 0; xp < B_dims[1]; xp++) {
        const float *A_off = A_src;
        const float *B_off = B_src;
        for (int yp = 0; yp < B_dims[0]; yp++) {
          a = _mm_load_ps(A_off+0);
          b = _mm_load_ps(B_off+0);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          a = _mm_load_ps(A_off+4);
          b = _mm_load_ps(B_off+4);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          a = _mm_load_ps(A_off+8);
          b = _mm_load_ps(B_off+8);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          a = _mm_load_ps(A_off+12);
          b = _mm_load_ps(B_off+12);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          a = _mm_load_ps(A_off+16);
          b = _mm_load_ps(B_off+16);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          a = _mm_load_ps(A_off+20);
          b = _mm_load_ps(B_off+20);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          a = _mm_load_ps(A_off+24);
          b = _mm_load_ps(B_off+24);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          a = _mm_load_ps(A_off+28);
          b = _mm_load_ps(B_off+28);
          c = _mm_mul_ps(a, b);
          v = _mm_add_ps(v, c);

          A_off += SSE_DEPTH;
```

```
            B_off += SSE_DEPTH;
          }

          A_src += A_dims[0]*SSE_DEPTH;
          B_src += B_dims[0]*SSE_DEPTH;
        }
      }
      // buf[] must be 16-byte aligned
      float buf[4] __attribute__ ((aligned (16)));
      _mm_store_ps(buf, v);
      _mm_empty();
      *(dst++) = buf[0]+buf[1]+buf[2]+buf[3];
    }
  }
  pthread_exit(NULL);
}

float *prepare(double *in, const int *dims) {
  float *F = (float *)malloc_aligned(16, dims[0]*dims[1]*SSE_DEPTH*sizeof(float));

  if (!IS_ALIGNED(F))
    mexErrMsgTxt("Memory not aligned");

  float *p = F;
  for (int x = 0; x < dims[1]; x++)
    for (int y = 0; y < dims[0]; y++) {
      for (int f = 0; f < dims[2]; f++)
        *(p++) = in[y + f*dims[0]*dims[1] + x*dims[0]];
      for (int f = dims[2]; f < SSE_DEPTH; f++)
        *(p++) = 0;
    }
  return F;
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
  if (nrhs != 5)
    mexErrMsgTxt("Wrong number of inputs");
  if (nlhs != 1)
    mexErrMsgTxt("Wrong number of outputs");

  // get feature map - A
  const mxArray *mxA = prhs[0];
  if (mxGetNumberOfDimensions(mxA) != 3 ||
      mxGetClassID(mxA) != mxDOUBLE_CLASS)
    mexErrMsgTxt("Invalid input: A");

  // get filter bank - B
  const mxArray *cellB = prhs[1];
  mwSize num_bs = mxGetNumberOfElements(cellB);
  int start = (int)mxGetScalar(prhs[2]) - 1;
  int end = (int)mxGetScalar(prhs[3]) - 1;
  if (start < 0 || end >= num_bs || start > end)
    mexErrMsgTxt("Invalid input: start/end");
  int len = end-start+1;

  // get saliency map - S
  const mxArray *mxS = prhs[4];
  if (mxGetNumberOfDimensions(mxS) != 2 ||
      mxGetClassID(mxA) != mxDOUBLE_CLASS)
    mexErrMsgTxt("Invalid input: S");

   // launch all threads
  thread_data *td = (thread_data *)mxCalloc(len, sizeof(thread_data));
  pthread_t *ts = (pthread_t *)mxCalloc(len, sizeof(pthread_t));
  const mwSize *A_dims = mxGetDimensions(mxA);
  float *A = prepare(mxGetPr(mxA), A_dims);
  const mwSize *S_dims = mxGetDimensions(mxS);
  double *S = (double *)mxGetPr(mxS);
  for (int i = 0; i < len; i++) {
    const mxArray *mxB = mxGetCell(cellB, i+start);
    const mwSize *B_dims = mxGetDimensions(mxB);
    float *B = prepare(mxGetPr(mxB), B_dims);
    td[i].A_dims[0] = A_dims[0];
    td[i].A_dims[1] = A_dims[1];
    td[i].A_dims[2] = A_dims[2] + 1;
    td[i].A = A;
    td[i].B_dims[0] = B_dims[0];
    td[i].B_dims[1] = B_dims[1];
    td[i].B_dims[2] = B_dims[2] + 1;
    td[i].B = B;
    td[i].S_dims = S_dims;
    td[i].S = S;
    // check A and B aligned with SSE_DEPTH
    if (mxGetNumberOfDimensions(mxB) != 3 ||
        mxGetClassID(mxB) != mxDOUBLE_CLASS ||
        td[i].A_dims[2] != SSE_DEPTH ||
        td[i].B_dims[2] != SSE_DEPTH)
      mexErrMsgTxt("Invalid input: B");

    int height = td[i].A_dims[0] - td[i].B_dims[0] + 1;
    int width = td[i].A_dims[1] - td[i].B_dims[1] + 1;
    if (height < 1 || width < 1)
      mexErrMsgTxt("Invalid input: B should be smaller than A");
    td[i].C_dims[0] = height;
    td[i].C_dims[1] = width;
    td[i].mxC = mxCreateNumericArray(2, td[i].C_dims, mxDOUBLE_CLASS, mxREAL);
    td[i].C = (double *)mxGetPr(td[i].mxC);

    if (pthread_create(&ts[i], NULL, process, (void *)&td[i]))
      mexErrMsgTxt("Error creating thread");
  }

  // set return values when threads finish
  void *status;
  plhs[0] = mxCreateCellMatrix(1, len);
  for (int i = 0; i < len; i++) {
    pthread_join(ts[i], &status);
    mxSetCell(plhs[0], i, td[i].mxC);
    free(td[i].B);
  }
  mxFree(td);
  mxFree(ts);
  free(A);
}
```