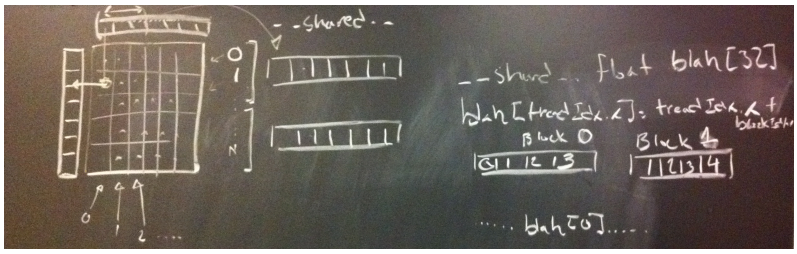
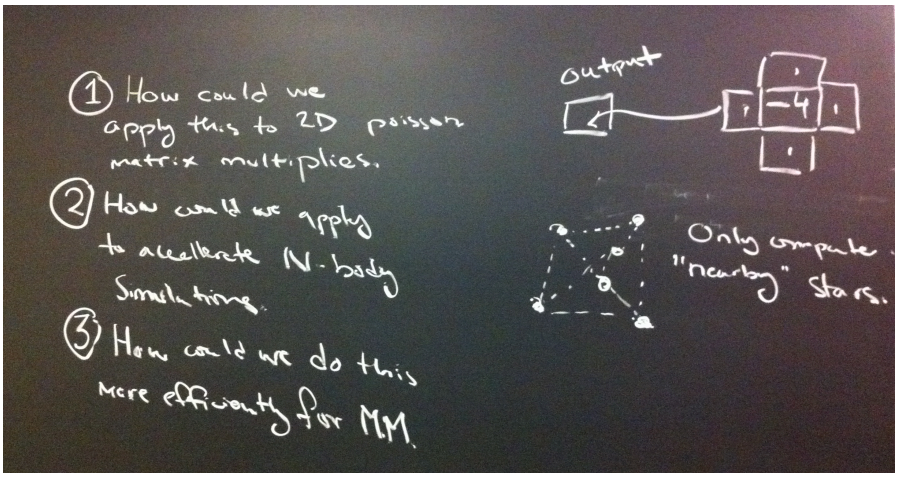
Matrix Multiplication (with caching)




```
#include <stdio.h>
#include <assert.h>
#include "cuda_helpers.hh"

static const int s_block_size = 32;

void init_a_big_vector(float * vec, unsigned int length) {
   for (unsigned int ii = 0 ; ii < length ; ii++) {
         vec[ii] = ii + 1.0f;
   }
}

float * create_a_big_vector(unsigned int length) {
   float * vec = (float *) malloc(length * sizeof(float));
   assert(vec);
   return vec;
}

void print_vec(float * const vec, unsigned int length) {
   printf("%p -> [ ", vec);
   for (unsigned int ii = 0 ; ii < length ; ii++) {
         printf("%f ", vec[ii]);
   }
   printf("]\n");
}

void print_mat(float * const mat, unsigned int width, unsigned int
height) {
   printf("%p ->\n", mat);
   for (unsigned int ii = 0 ; ii < height ; ii++) {
         printf("[ ");
         for (unsigned int jj = 0 ; jj < width ; jj++) {
               printf("%f ", mat[ii * width + jj]);
         }
         printf("]\n");
   }
}

void mat_vec_mult(float * const mat, float * const xx, float * bb,
   unsigned int width, unsigned int height) {
   for (unsigned int ii = 0 ; ii < height ; ii++) {
         bb[ii] = 0.0;
         for (unsigned int jj = 0 ; jj < width ; jj++) {
               bb[ii] += (mat[ii * width + jj]) * xx[jj];
         }
   }
}

void cpu_version(float * const mm, float * const xx, float * bb, unsigned
int length) {
   // // print the input
   printf("CPU Matrix Multiply\n");
   // print_vec(xx, length);
   // print_mat(mm, length, length);

   // do the multiply
   for (int times = 0 ; times < 10000 ; times++)
         mat_vec_mult(mm, xx, bb, length, length);

   // // print the output
   // printf("\n");
   // print_vec(bb, length);
}

__global__
void mat_vec_mult_kernel(float * mat, float * xx, float * bb, unsigned
int width, unsigned int height)
{
   const unsigned int ii = blockIdx.x * blockDim.x + threadIdx.x;
   if (ii > height)
         return;

   __shared__ float xx_cache[s_block_size];

   float bb_local = 0.0;
   for (unsigned int jj = 0 ; true ; jj += blockDim.x) {
         unsigned int my_jj = jj + threadIdx.x;
         if (my_jj >= width)
               break;
         xx_cache[threadIdx.x] = xx[my_jj];
         __syncthreads();
         for (unsigned int jj2 = jj ; jj2 < jj + blockDim.x ; jj2 += 1) {
               if (jj2 >= width)
                     break;
               bb_local += (mat[ii * width + jj2]) * xx_cache[jj2 - jj];
         }
   }
   bb[ii] = bb_local;

   // float bb_local = 0.0;
   // for (unsigned int jj = 0 ; jj < width ; jj++) {
   //    bb_local += (mat[ii * width + jj]) * xx[jj];
   // }
   // bb[ii] = bb_local;
}

__global__
void mat_vec_mult_kernel_2(float * mat, float * xx, float * bb, unsigned
int width, unsigned int height)
{
   const unsigned int jj = blockIdx.x * blockDim.x + threadIdx.x;
   if (jj > width)
         return;

   float xx_cache = xx[jj];

   for (unsigned int ii = 0 ; ii < height ; ii++) {
         // bb[ii] += (mat[ii * width + jj]) * xx_cache[threadIdx.x];
         bb[ii] += (mat[ii * width + jj]) * xx_cache; // [threadIdx.x];
   }
}

void gpu_mat_vec_mult(float * const mm, float * const xx, float * bb,
unsigned int length) {
   // // print the input
   printf("GPU Matrix Multiply\n");
   // print_vec(xx, length);
   // print_mat(mm, length, length);

   // allocate data
   float * xx_gpu = 0;
   float * bb_gpu = 0;
   float * mm_gpu = 0;
   cudaMalloc((void **) &xx_gpu, length * sizeof(float));
   cudaMalloc((void **) &bb_gpu, length * sizeof(float));
   cudaMalloc((void **) &mm_gpu, length * length * sizeof(float));
   assert(xx_gpu && bb_gpu && mm_gpu);

   CHECK_FOR_CUDA_ERROR();

   // copy the data
   cudaMemcpy(xx_gpu, xx, length * sizeof(float),
cudaMemcpyHostToDevice);
   cudaMemcpy(mm_gpu, mm, length * length * sizeof(float),
cudaMemcpyHostToDevice);
   cudaMemset(bb_gpu, 0, length * sizeof(float));
   CHECK_FOR_CUDA_ERROR();

   // perform the matrix multiply
   int n_blocks = length / s_block_size;
   if ((length % s_block_size) != 0)
         n_blocks += 1;
   for (int times = 0 ; times < 5000 ; times++)
         mat_vec_mult_kernel<<<n_blocks, s_block_size>>>(mm_gpu, xx_gpu,
bb_gpu, length, length);

   CHECK_FOR_CUDA_ERROR();

   // copy the data back
   cudaMemcpy(bb, bb_gpu, length * sizeof(float),
cudaMemcpyDeviceToHost);

   // print the output
   // print_vec(bb, length);

   // free the data
   cudaFree(xx_gpu);
   cudaFree(bb_gpu);
   cudaFree(mm_gpu);

}

int main (int argc, char ** argv) {
   const unsigned int length = 512;

   // allocate memory
   float * xx = create_a_big_vector(length);
   float * bb = create_a_big_vector(length);
   float * mm = create_a_big_vector(length * length);

   // init memory
   init_a_big_vector(xx, length);
   init_a_big_vector(mm, length * length);

   // do stuff
   gpu_mat_vec_mult(mm, xx, bb, length);
   // cpu_version(mm, xx, bb, length);

   // free memory
   free(xx);
   free(bb);
   free(mm);

   return 0;
}
```

Student Answers to Challenges



(1) Accelerating Poisson Multiplication

- use a space-filling curve to store images (spatial locality)
- divide the image into 2D blocks
  - each thread (2D) is a pixel in the image

- first do within each block
  - for each thread
    - cache the pixel value at the center
    - BUT SOME PIXELS ARE LEFT OUTSIDE OF THE CACHE
      - the pixels on the border of the blocks
        - what is the solution to this?!?
      - maybe this isn't such a big deal
        - even corner pixels have 3 cache "hits" (2 "misses")
  - lay out the memory so that the blocks fall into one contiguous block of memory
- then "do it" between blocks

(2) Accelerating N-Body Systems

- difficulty: the particles are moving
  - which is near to which is not constant!
- spatial datastructures
  - octree?
  - "grid it up"
    - particles might be able to go into two grid cells at once
      - there will be relatively few particles in each grid cell
      - have there be one block for each grid cell
      - thread for each particle in that grid cell
      - cache particles within that block
  - "dynamic grid sizes?"
    - gives you a MAXIMUN # of bodies in each grid cell
  - use a mesh
    - the connection between each pair of particles is a therad
- each thread is a particle
  - simultaneous load into memory

(3) Accelerating (General) Matrix Multiplies

- cache parts of the input matrix? (matrix-matrix multiplies)
  - split it into sub-matrices