Optimization of vSLAM Applications

CS61064
High Performance Parallel Programming

CUDA Term Project: **Group B**

Bipin Kumar Mandal
Mayur Manik Jiwatode
Utkarsh Patel
Kamalesh Garnayak
Girish Kumar
Rajas Bhatt
Lanke Leela Manohar

Contents

Introduction

In this project, we explore the usage of GPUs to parallelize the ORB-SLAM2 algorithm. ORB-SLAM [1] is a state-of-the-art visual SLAM (Simultaneous Localization and Mapping) algorithm ([2], [3]), which uses monocular, stereo or RGB-D cameras to track the position and orientation of a camera in an unknown environment and create a map of the environment at the same time.

The name "ORB" stands for Oriented FAST and Rotated BRIEF [4], which are two key components of the algorithm. FAST is a corner detection algorithm, which is used to detect features in the images, and BRIEF is a binary descriptor, which is used to describe the features. ORB-SLAM also incorporates loop closing and map optimization techniques to improve the accuracy and robustness of the system.

ORB-SLAM has been widely used in robotics, augmented reality, and virtual reality applications, and has been benchmarked against other state-of-the-art SLAM algorithms, showing superior performance in many cases. ORB-SLAM2 is an improved version of the original ORB-SLAM algorithm that incorporates several advancements and new features. It was developed by the same research group that created the original algorithm and was published in 2017.

Some of the key improvements in ORB-SLAM2 include:

1. Increased robustness and accuracy: ORB-SLAM2 uses more advanced techniques for feature detection, feature tracking, and loop closing, resulting in improved robustness and accuracy.
2. Support for multiple cameras: ORB-SLAM2 can handle different camera configurations, including monocular, stereo, and RGB-D cameras.
3. Real-time performance: ORB-SLAM2 is designed to run in real-time on a standard CPU, allowing it to be used in applications that require fast processing speeds.
4. Map reuse: ORB-SLAM2 can reuse previously created maps, allowing it to continue tracking a camera's position even if it loses tracking temporarily.
5. Support for large-scale environments: ORB-SLAM2 can create and maintain maps of large-scale environments, making it suitable for applications such as autonomous driving and robotics.
6. ORB-SLAM2 has been widely adopted in both academia and industry and has become one of the most popular visual SLAM algorithms available. It has been benchmarked extensively and has demonstrated superior performance compared to other state-of-the-art SLAM algorithms.

Motivation

Simultaneous Localization and Mapping (SLAM) is a fundamental problem in computer vision and robotics. The goal of SLAM is to construct a map of an environment while simultaneously estimating the pose of a sensor within that environment. SLAM is a challenging problem due to the high computational requirements, limited sensor information, and the complexity of the environment. ORB-SLAM2 is a popular SLAM algorithm that uses monocular, stereo, and RGB-D cameras to construct a map and estimate the pose of a camera.

In recent years, Graphics Processing Units (GPUs) have become an essential tool for accelerating computer vision and machine learning applications. GPUs can perform thousands of operations in parallel, making them ideal for SLAM algorithms that require intensive computations. In this essay, we will discuss the motivation for using GPUs in ORB-SLAM2 and the benefits of GPU acceleration.

Motivation for using GPUs in ORB-SLAM2

ORB-SLAM2 is a feature-based SLAM algorithm that uses the Oriented FAST and Rotated BRIEF (ORB) feature detector and descriptor. The ORB feature detector is fast and robust to changes in lighting and viewpoint, making it suitable for real-time applications. However, the ORB feature descriptor is computationally expensive, as it involves computing the binary Hamming distance between two feature descriptors.

The ORB feature descriptor is used in the keypoint matching stage of the SLAM algorithm, where it matches features in two consecutive frames to estimate the motion of the camera. This process involves searching for the best match between two sets of features, which is a computationally intensive task. Moreover, the ORB-SLAM2 algorithm also performs bundle adjustment, which involves optimizing the camera poses and 3D points in the map to reduce the reprojection error.

The extraction of ORB features involves the detection of corners, computation of orientations and generation of BRIEF descriptors for each feature point. These tasks can be speed up by running them in parallel on a GPU. The corner detection and orientation assignment can be done on separate threads and therefore, allow us to take advantage of GPU architectures.

The ORB-SLAM2 algorithm is computationally intensive, and as the size of the map increases, the computational requirements grow exponentially. The computational bottleneck of the ORB-SLAM2 algorithm is the keypoint matching stage, which involves matching thousands of features in real-time. GPUs can provide significant speedups in this stage, as they can perform the computationally intensive tasks in parallel.

Benefits of GPU acceleration for ORB-SLAM2

There are several benefits to using GPUs for accelerating ORB-SLAM2:

1. **Speedup**: GPUs can perform thousands of operations in parallel, making them ideal for computationally intensive tasks such as keypoint matching and bundle adjustment. By using GPUs, the computational time of ORB-SLAM2 can be significantly reduced, leading to real-time performance.

2. **Scalability**: ORB-SLAM2 is a scalable algorithm that can build large-scale maps of complex environments. However, as the size of the map increases, the computational requirements grow exponentially. GPUs can provide the necessary computational power to handle the increasing complexity of the map.
3. **Accuracy**: ORB-SLAM2 relies on accurate feature matching and bundle adjustment to construct an accurate map of the environment. By using GPUs, the computational time of these tasks can be significantly reduced, leading to more accurate results.
4. **Flexibility**: GPUs can be easily integrated into existing hardware platforms, making them a flexible solution for ORB-SLAM2 applications. Moreover, GPUs can also be used for other computer vision and machine learning applications, making them a versatile tool for research and development.
5. **Cost-Effective**: GPUs are cost-effective compared to other hardware accelerators such as FPGAs and ASICs. Moreover, GPUs are widely available and can be easily integrated into existing hardware platforms, making them a cost-effective solution for ORB-SLAM2 applications.

Problem Description

ORB-SLAM2 is a state-of-the-art vSLAM system that uses feature-based methods for real-time camera tracking and 3D reconstruction. It is based on the ORB (Oriented FAST and Rotated BRIEF) feature detector and descriptor, which is a fast and efficient method for detecting and matching feature points in images. It uses a combination of geometric and photometric information to estimate the camera pose and create a 3D map of the environment.

ORB-SLAM2 is a computationally intensive task and can lead to significant performance improvements if the parallel processing power of the GPU can be harnessed and optimized. The implementation of ORB-SLAM2 in CUDA involves modifying the existing CPU-based code to use CUDA for parallel processing on the GPU. This involves identifying the computationally intensive parts of the code and converting them into GPU kernels, which can be executed in parallel on the GPU.

Methodology

Matrix Normalization Kernel

```
namespace ORB_SLAM2 { namespace cuda {

__device__ uint8_t d_val;

MatNormGPU::MatNormGPU() {
    // GpuMat should be already stored in memory
    //cudaMallocManaged(&subtract_val, sizeof(uint8_t));
}

MatNormGPU::~MatNormGPU() {
}

__global__
void kernel_get_mat_pixel (uint8_t * src, int w, int step) //uint8_t * pSub)
{
    // assuming that we resized it to CV_32F so the channel number is 1
    //*d_subtract_val = src[ (w*step) + (w)];
    //*pSub = src[ (w * step) + (w)];
    d_val = src[ (w * step) + w];
}

void MatNormGPU::setSubtractValue(const cv::cuda::GpuMat _img, int w)
{
    uint8_t subtract_val;
    //cudaMalloc(&d_subtract_val, sizeof(int));
    //uint8_t * imgPtr;
    //cudaMalloc((void **)&imgPtr, _img.rows*_img.step);
    //cudaMemcpy(imgPtr, _img.ptr<uint8_t>(), _img.rows*_img.step, cudaMemcpyDeviceToDevice);
    //std::cout << "start subtract value" << std::endl;
    //cudaMallocManaged(&subtract_val, sizeof(uint8_t));

    //SET_CLOCK(t0);
    kernel_get_mat_pixel<<<1, 1>>>(_img.data, w, _img.step);
    //SET_CLOCK(t1);
    //cout << TIME_DIFF(t1, t0) << endl;
    //std::cout << "finish kernel command" << std::endl;

    cudaMemcpyFromSymbol(&subtract_val, d_val, sizeof(uint8_t), 0, cudaMemcpyDeviceToHost);
    subMat = cv::cuda::GpuMat(_img.rows, _img.cols, _img.type(), subtract_val, cuda::gpu_mat_allocator);
    //cudaMemcpy(subtract_val, d_subtract_val, sizeof(int), cudaMemcpyDeviceToHost);
    //cudaFree(d_subtract_val);
    //std::cout << "gpu subtract value " << subtract_val << " here" << std::endl;
}

__global__
void kernel_subtract_pixel_from_mat (uint8_t * src, int MaxRows, int MaxCols, int step, int sub)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x; //Row number
    int rowStride = blockDim.x * gridDim.x;
    int col = blockIdx.y * blockDim.y + threadIdx.y; //Column number
    int colStride = blockDim.y * gridDim.y;

    //unsigned int ch = blockIdx.z * blockDim.z + threadIdx.z; //Channel 0
    for (int i = row; i < MaxRows; i += rowStride) {
  for (int j = col; j < MaxCols; j += colStride) {

          if (row<MaxRows && col<MaxCols) {
          int idx = i * step + j; // maxChannels is 1 and ch is 0
          src[idx] = src[idx] - sub;
          }
      }
    }
}

void MatNormGPU::subtract_pixel_from_mat (cv::cuda::GpuMat _img)
{
    //std::cout << "start subtract pixel " << int(subtract_val) << std::endl;

    //const dim3 block(16, 16);
    //const dim3 grid(((_img.cols + block.x - 1) / block.x), ((_img.rows + block.y - 1)/ block.y));

    //kernel_subtract_pixel_from_mat<<<grid, block>>> (_img.data,_img.rows, _img.cols, _img.step,
    subtract_val);
    //cv::cuda::subtract(_img, Scalar::all(subtract_val), _img);
    cv::cuda::subtract(_img, subMat, _img);
    //std::cout << "finish subtract pixel" << std::endl;
}

} }
```

Listing 1: Matrix Normalization Kernel

This code defines a class MatNormGPU that performs GPU-based image normalization by subtracting a scalar value from each pixel in an image. The class has two main functions, setSubtractValue and subtract_pixel_from_mat.

setSubtractValue takes in a cv::cuda::GpuMat object and an integer w. The function uses a kernel (kernel_get_mat_pixel) to read a single pixel from the image at position (w, w) and save its value as subtract_val (a member variable of type uint8_t). It then creates a cv::cuda::GpuMat object subMat with the same size as the input image, but with each pixel set to the value subtract_val.

subtract_pixel_from_mat takes in another cv::cuda::GpuMat object _img. It subtracts subMat (created in the setSubtractValue function) from _img using the cv::cuda::subtract function.

The code uses CUDA to execute the kernel_get_mat_pixel and kernel_subtract_pixel_from_mat kernels on the GPU. The cv::cuda library is used to perform element-wise subtraction between two cv::cuda::GpuMat objects.

Overall, this code allows for efficient image normalization on a CUDA-enabled GPU.

1. **Usage of Shared Memory** This code does not make use of shared memory. Shared memory in CUDA is a fast on-chip memory that can be shared between threads within a block. However, in this code, the kernel functions only use global memory, which is slower than shared memory. Therefore, there is no benefit to using shared memory in this code.

2. **Divergence, Reduction, and Transpose** Regarding divergence, the current implementation checks if the current thread is within the limits of the matrix before executing the subtraction operation. However, this introduces divergence, as some threads will be executing the conditional statement while others skip it. But, the conditions used inside the `if` statements are not related to block and thread indices, but rather on the property of pixels. Hence, there is no scope for reducing divergence. Since the code does not use any reduction and transpose operations, the code also has no scope for optimization with respect to these techniques.

Keypoint Calculation Kernel

```
__global__ void tileCalcKeypoints_kernel(const PtrStepSzb img, short2 * kpLoc, float * kpScore, const
    unsigned int maxKeypoints, const int highThreshold, const int lowThreshold, PtrStepi scoreMat,
    unsigned int * counter_ptr)
{
    // Each thread, in a warp, will be processing 1 element from four consecutive rows. Hence, for a
    warp, the global memory accesses are coalesced.
    const int j = threadIdx.x + blockIdx.x * blockDim.x + 3;
    const int i = (threadIdx.y + blockIdx.y * blockDim.y) * 4 + 3;
    const int tid = threadIdx.y * blockDim.x + threadIdx.x;

    // Is any keypoint found?
    __shared__ bool hasKp;
    if (tid == 0) {
      hasKp = false;
    }

    // Store whether the pixels the thread is processing is a keypoints wrt. High Threashold and Low
    Threshold
    bool isKp[4] = {0};

    // Finding keypoints wrt. High Threshold
    for (int t = 0; t < 4; ++t) {
      if (i+t < img.rows - 3 && j < img.cols - 3) {
        isKp[t] = isKeyPoint2(img, i+t, j, highThreshold, scoreMat);
      }
    }

    __syncthreads();

    for (int t = 0; t < 4; ++t) {
      if (isKp[t]) {
        isKp[t] = false;
        short2 loc = make_short2(j, i+t);
        if (isMax(loc, scoreMat)) {
          int score = scoreMat(loc.y, loc.x);
          hasKp = true;
          const unsigned int ind = atomicInc(counter_ptr, (unsigned int)(-1));
          if (ind < maxKeypoints) {
            kpLoc[ind] = loc;
            kpScore[ind] = static_cast<float>(score);
          }
        }
      }
    }

    __syncthreads();

    // If any keypoint wrt. High Threshold found, return
    if (hasKp) return ;

    // No keypoint wrt. High Threshold found. Look wrt. Low Threshold.
    for (int t = 0; t < 4; ++t) {
      if (i+t < img.rows - 3 && j < img.cols - 3) {
        isKp[t] = isKeyPoint2(img, i+t, j, lowThreshold, scoreMat);
      }
    }

    __syncthreads();

    for (int t = 0; t < 4; ++t) {
      if (isKp[t]) {
        short2 loc = make_short2(j, i+t);
        if (isMax(loc, scoreMat)) {
          int score = scoreMat(loc.y, loc.x);
          const unsigned int ind = atomicInc(counter_ptr, (unsigned int)(-1));
          if (ind < maxKeypoints) {
            kpLoc[ind] = loc;
            kpScore[ind] = static_cast<float>(score);
          }
        }
      }
    }
  }
```

Listing 2: Keypoint calculation kernel

The above code is a CUDA kernel function called **tileCalcKeypointskernel** that performs keypoint detection on an input image using the ORB-SLAM2 algorithm. Key points are locations in an image where there are significant changes in intensity or texture. These key points are important for computer vision tasks such as feature matching, object recognition, and tracking.

The kernel function takes several arguments, including the input image **img**, the maximum number of keypoints **maxKeypoints**, and two threshold values **highThreshold** and **lowThreshold**. The **scoreMat** argument is a matrix that stores the scores for each pixel in the image, and **counterptr** is a pointer to an atomic integer that keeps track of the number of keypoints detected so far.

The kernel function is designed to process a subset of the image, referred to as a "tile", in parallel by multiple threads running on the GPU. The tile size is determined by the block and thread dimensions specified when the kernel is launched. Each thread is responsible for detecting keypoints in a 4x4 patch of pixels within the tile.

The first part of the kernel function uses a loop to check each 4x4 patch for keypoints. For each patch, the function calls the **isKeyPoint2** function to determine if it contains a keypoint. If a keypoint is detected and it has a higher score than any other keypoint in its local neighborhood, it is added to the list of detected keypoints. The **atomicInc** function is used to increment the **counterptr** pointer in a thread-safe manner.

If no keypoints are detected in the first pass, the function lowers the threshold value and tries again. This process is repeated until either the maximum number of keypoints is reached or no more keypoints can be detected.

The code makes use of shared memory to improve performance. The **hasKp** variable is stored in shared memory and is used to determine if any keypoints have been detected by any of the threads in the block. If a keypoint has been detected, the function returns immediately to avoid unnecessary computation.

Overall, the code efficiently utilizes the parallel processing power of the GPU to detect keypoints in an input image using the ORB-SLAM2 algorithm.

Keypoint Calculation Kernel Optimization

We try to apply various optimization techniques taught in the course on the kernel above and comment on their usability in the Keypoint Calculation Kernel.

1. **Usage of Shared Memory** As each thread operates on different pixel location, i.e. no sharing of input, the shared memory is not used to store the `img` matrix.

2. **Divergence, Reduction and Transpose** The code is divergent. But, the conditions used inside the `if` statements are not related to block and thread indices, rather on the property of pixels. Hence, there is no scope for reducing divergence. Since the code does not use any reduction and transpose operations, the code also has no scope for optimization with respect to these techniques.

3. **Memory Access Coalescing** We perform global memory transactions on image, the `img` matrix. Each thread, in a warp, will be processing 1 element from 4 consecutive rows in the `img` matrix. Hence, for a warp, the global memory accesses are all coalesced.

4. **Coarsening** Coarsening of the threads can be tried, however it is unlikely to yield positive results. Since the number of keypoints remains a small number in the hundreds from empirical observations, the kernel launch overhead is minimal.

Keypoint Gradient Angle Calculation Kernel

```
__global__ void IC_Angle_kernel(const PtrStepb image, KeyPoint * keypoints, const int npoints, const
   int half_k)
 {
   __shared__ int smem0[8 * 32];
   __shared__ int smem1[8 * 32];

   int* srow0 = smem0 + threadIdx.y * blockDim.x;
   int* srow1 = smem1 + threadIdx.y * blockDim.x;

   cv::cuda::device::plus<int> op;

   const int ptidx = blockIdx.x * blockDim.y + threadIdx.y;

   if (ptidx < npoints)
   {
     int m_01 = 0, m_10 = 0;

     const short2 loc = make_short2(keypoints[ptidx].pt.x, keypoints[ptidx].pt.y);

     // Treat the center line differently, v=0
     for (int u = threadIdx.x - half_k; u <= half_k; u += blockDim.x)
       m_10 += u * image(loc.y, loc.x + u);

     reduce<32>(srow0, m_10, threadIdx.x, op);

     for (int v = 1; v <= half_k; ++v)
     {
       // Proceed over the two lines
       int v_sum = 0;
       int m_sum = 0;
       const int d = c_u_max[v];

       for (int u = threadIdx.x - d; u <= d; u += blockDim.x)
       {
         int val_plus = image(loc.y + v, loc.x + u);
         int val_minus = image(loc.y - v, loc.x + u);

         v_sum += (val_plus - val_minus);
         m_sum += u * (val_plus + val_minus);
       }

       reduce<32>(smem_tuple(srow0, srow1), thrust::tie(v_sum, m_sum), threadIdx.x, thrust::make_tuple(
   op, op));

       m_10 += m_sum;
       m_01 += v * v_sum;
     }

     if (threadIdx.x == 0)
     {
       //                vv  what is this ?
       //float kp_dir = ::atan2f((float)m_01, (float)m_10);
       float kp_dir = atan2f((float)m_01, (float)m_10);
       kp_dir += (kp_dir < 0) * (2.0f * CV_PI_F);
       kp_dir *= 180.0f / CV_PI_F;

       keypoints[ptidx].angle = kp_dir;
     }
   }
 }
```

Listing 3: Keypoint Gradient Angle calculation kernel

Above function is a global CUDA kernel that takes in an image (image), an array of KeyPoint structures (keypoints), the number of keypoints (npoints), and a kernel radius (half_k) as input arguments.

The kernel uses shared memory to reduce memory accesses and improve performance. It first declares two shared memory arrays smem0 and smem1, each of size 8x32. It then initializes two pointers srow0 and srow1 to point to the start of the shared memory arrays.

Each thread in the kernel is identified by its unique threadIdx.x and threadIdx.y values, which specify its position within the block. The total number of threads is given by the product of the thread block size and the number of blocks launched.

For each keypoint, the kernel calculates the angle of the gradient at the point. It does this by computing the x- and y-components of the gradient (`m_10` and `m_01`, respectively) using the Sobel operator.

The kernel then proceeds to calculate the orientation of the gradient. It calculates the orientation using the formula `atan2(m_01, m_10)`, which computes the arctangent of `m_01/m_10` in radians. It then converts the result to degrees and stores it in the angle member of the `KeyPoint` structure.

Finally, the kernel uses the reduce function to perform parallel reductions on the shared memory arrays `smem0` and `smem1`. The reduce function computes the sum of the elements in each row of the arrays and stores the result in `srow0`. The results are used in the subsequent calculations to compute the angle of the gradient.

Keypoint Gradient Angle Calculation Kernel Optimization

We try to apply various optimization techniques taught in the course on the kernel above and comment on their usability in the Keypoint Gradient Angle Calculation Kernel

1. **Usage of Shared Memory** The IC_Angle_kernel uses shared memory to speed up the computation of the orientation angle of a set of key points. The shared memory is used to reduce the partial sums of two terms, m_10 and m_01. By using shared memory to store the partial sums, the reduction operation avoids the need to access global memory repeatedly, which can be slow due to its high latency.
2. **Divergence, Reduction, and Transpose** The code is divergent. There is a conditional statement that checks whether the current thread's index is less than the total number of key points. But, there is no scope for reducing divergence as the conditions are related to the property of keypoints. The kernel uses the reduce function to perform a parallel reduction operation to sum the values of m_10 and m_sum computed by the threads. There is no transpose operation. Hence, there is no scope for optimization for this technique.
3. **Memory Access Coalescing** In this particular kernel, memory access coalescing is achieved by using the threads in each warp to access adjacent memory locations in global memory, which helps to minimize the number of memory transactions required to fetch the data needed by the threads.
4. **Coarsening** There is no scope for coarsening in the above kernel as it is already optimized for parallelism at the thread level by launching one thread per keypoint. Each thread is responsible for computing the direction of a single keypoint, and coarsening would result in losing the granularity needed to compute the correct result for each keypoint.

ORB Feature Extraction Kernel

```
//  __constant__ unsigned char c_pattern[sizeof(Point) * 2 * 256];

#define GET_VALUE(idx) \
    image(loc.y + __float2int_rn(pattern[idx].x * b + pattern[idx].y * a), \
        loc.x + __float2int_rn(pattern[idx].x * a - pattern[idx].y * b))

__global__ void calcOrb_kernel(const PtrStepb image, KeyPoint * keypoints, const int npoints, PtrStepb
    descriptors) {
    int id = blockIdx.x;
    int tid = threadIdx.x;
    if (id >= npoints) return;

    const KeyPoint &kpt = keypoints[id];
    short2 loc = make_short2(kpt.pt.x, kpt.pt.y);
    const Point * pattern = ((Point *)c_pattern) + 16 * tid;

    uchar * desc = descriptors.ptr(id);
    const float factorPI = (float)(CV_PI/180.f);
    float angle = (float)kpt.angle * factorPI;
    float a, b;
    sincosf(angle, &b, &a);

    int t0, t1, val;
    t0 = GET_VALUE(0); t1 = GET_VALUE(1);
    val = t0 < t1;
    t0 = GET_VALUE(2); t1 = GET_VALUE(3);
    val |= (t0 < t1) << 1;
    t0 = GET_VALUE(4); t1 = GET_VALUE(5);
    val |= (t0 < t1) << 2;
    t0 = GET_VALUE(6); t1 = GET_VALUE(7);
    val |= (t0 < t1) << 3;
    t0 = GET_VALUE(8); t1 = GET_VALUE(9);
    val |= (t0 < t1) << 4;
    t0 = GET_VALUE(10); t1 = GET_VALUE(11);
    val |= (t0 < t1) << 5;
    t0 = GET_VALUE(12); t1 = GET_VALUE(13);
    val |= (t0 < t1) << 6;
    t0 = GET_VALUE(14); t1 = GET_VALUE(15);
    val |= (t0 < t1) << 7;

    desc[tid] = (uchar)val;
}

#undef GET_VALUE

/* LAUNCH PARAMETERS OF THE KERNEL

    dim3 dimBlock(32);        // Each block contains 32 threads

    dim3 dimGrid(npoints);    // The number of keypoints for which we need the descriptors

    calcOrb_kernel<<<dimGrid, dimBlock, 0, stream>>>(image, keypoints, npoints, desc);

*/
```

Listing 4: ORB Feature Extraction Kernel

The CUDA kernel above calculates the descriptors of keypoints in an image using the ORB feature detection algorithm. The kernel takes as input an image (in the form of a `PtrStepb` object, which is basically another format of a `cv::GpuMat`), a list of keypoints (`KeyPoint *keypoints`) , the number of keypoints (`const int npoints`, and an output buffer for the computed descriptors (`PtrStepb descriptors`).

The ORB descriptor for an image is a 256-bit binary string which stores the local image information in the region around the keypoint. Each bit in the binary string is computed by comparing the intensity of two pixels which are defined relative to the keypoint and chosen from the pre-defined set of positions (stored in the `c_pattern` array in constant memory). This constitutes one test. 256 such tests are done.

(`c_pattern[0].x, c_pattern[0].y`) and (`c_pattern[1].x, c_pattern[1].y`) are the points constituting the first test. Since we need to check the pixel values at these points relative to the keypoint, we apply translation (using the co-ordinates of the keypoint) and rotation (using the angle of the keypoint) to map the image coordinates to the test point coordinates. This is done by the `GET_VALUE` macro above. It can be seen that the test pattern index (in array `c_pattern`) accessed by each thread has `+ 16 * threadIdx.x` added to it. This is where the division of work between threads takes place.

Each thread computes 8 tests (each of which return a 0 or 1) and this bit-string is stored as an `unsigned char`, **`val`** in the index accessed by the local thread ID (`threadIdx.x`). Since there are 32 threads in a block, we cover a totatl of 256 tests as specified earlier.

It is evident from the launch code that each block computes the descriptor of one keypoint and the number of blocks is equal to the number of keypoints (`npoints`).

ORB Feature Extraction Kernel Optimization

We try to apply various optimization techniques taught in the course on the kernel above and comment on their usability in the ORB Feature Extraction Kernel.

1. **Usage of Shared Memory** We perform global memory transactions on two arrays, namely `image` and `descriptors`. For the `image` array, the pixel that can be accessed ranges across a wide range of values and depends on the location of the keypoint. As such storing the image as a whole in shared memory is not feasible as it will take a tremendous amount of time for an image to be copied from the global memory to the shared memory. Moreover, no computations are shared between the threads inside a block.

2. **Divergence, Reduction and Transpose** The code is not divergent because it has no if statements which divide a warp. Hence, there is no scope for reducing divergence. Since the code does not use any reduction and transpose operations, the code also has no scope for optimization with respect to these techniques.

3. **Memory Access Coalescing** Usually NVidia GPUs have a warp size of 32. Assuming a warp size of 32, it can be seen that one entire block constitutes one warp. Threads in a warp (here a warp constitutes a block) always access continuous memory locations in the array `c_pattern`. For the global array `image`, threads will **NOT** access continuous memory locations. However, this is not under our control and no matter how much we change the access patterns, memory access will depend on the keypoints and will be random. For the `descriptors` array, it can be seen that threads in a warp access contiguous memory locations.

4. **Coarsening** Coarsening of the threads can be tried, however it is unlikely to yield positive results because there are no barrier synchronizations and only a small number of threads per block are being launched and since the number of keypoints (`npoints`) remains a small number in the hundreds from empirical observations, the kernel launch overhead is minimal.

Results

We profile the code using `nvprof` and report the profiler output.

```
==21811== Profiling application: ./build/mono_tum Vocabulary/ORBvoc.txt Examples/Monocular/TUM1.yaml
    rgbd_dataset_freiburg1_xyz
==21811== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   52.13%  8.74918s      6384  1.3705ms  193.70us  29.532ms  ORB_SLAM2::cuda::
    tileCalcKeypoints_kernel
                   12.31%  2.06546s      6384  323.54us  75.001us  6.2314ms  void cv::cudev::
    grid_copy_detail::copy<cv::cudev::RemapPtr1<cv::cudev::BrdBase<cv::cudev::BrdReflect101, cv::cudev::
    GlobPtr<unsigned char>>, _GLOBAL__N__51_tmpxft_000039cf_00000000_6_copy_make_border_cpp1_ii_71482d89
    ::ShiftMap>, unsigned char, cv::cudev::WithOutMask>(cv::cudev::BrdReflect101, cv::cudev::GlobPtr<
    unsigned char>, cv::cudev::GlobPtr<unsigned char>, int, int)
                    8.83%  1.48139s      6384  232.05us  79.585us  2.2729ms  ORB_SLAM2::cuda::
    calcOrb_kernel(cv::cuda::PtrStep<unsigned char>, cv::KeyPoint*, int, cv::cuda::PtrStep<unsigned char
    >)
                    7.09%  1.18968s      6384  186.35us  44.011us  1.6670ms  void column_filter::
    linearColumnFilter<int=7, float, unsigned char, cv::cuda::device::BrdColReflect101<float>>(cv::cuda
    ::PtrStepSz<float>, cv::cuda::PtrStep<unsigned char>, float const *, int, float)
                    5.71%  959.01ms      6384  150.22us  37.135us  1.3142ms  void row_filter::
    linearRowFilter<int=7, unsigned char, float, cv::cuda::device::BrdRowReflect101<unsigned char>>(cv::
    cuda::PtrStepSz<unsigned char>, cv::cuda::PtrStep<float>, float const *, int, unsigned char)
                    5.41%  907.18ms      5586  162.40us  39.741us  2.9133ms  void cv::cuda::device::
    resize_linear<unsigned char>(cv::cuda::PtrStepSz<unsigned char>, unsigned char, float, float)
                    4.29%  720.01ms      6384  112.78us  40.886us  908.87us  ORB_SLAM2::cuda::
    IC_Angle_kernel(cv::cuda::PtrStep<unsigned char>, cv::KeyPoint*, int, int)
                    3.63%  609.92ms     19152  31.846us     572ns  678.56us  [CUDA memset]
                    0.28%  47.248ms     13574  3.4800us     261ns  309.90us  [CUDA memcpy HtoD]
                    0.16%  27.478ms     19152  1.4340us     572ns  14.896us  [CUDA memcpy DtoH]
                    0.16%  26.029ms      6384  4.0770us  1.8230us  39.115us  ORB_SLAM2::cuda::
    addBorder_kernel(cv::KeyPoint*, int, int, int, int, int)
      API calls:   37.22%  10.0938s     50274  200.78us  33.543us  2.82521s  cudaLaunchKernel
                   33.29%  9.02798s     25536  353.54us  35.938us  30.855ms  cudaMemcpyAsync
                   11.47%  3.11046s     19950  155.91us  12.761us  23.616ms  cudaStreamSynchronize
                    6.06%  1.64298s      6384  257.36us  66.200us  6.6212ms  cudaMemcpy2DAsync
                    4.26%  1.15420s     12768  90.397us  49.376us  910.75us  cudaMemset2DAsync
                    2.78%  754.07ms       798  944.95us  689.34us  8.1857ms  cudaFree
                    2.13%  578.79ms        14  41.342ms  6.0420us  561.30ms  cudaStreamCreate
                    1.23%  332.97ms      6384  52.157us  34.271us  910.18us  cudaMemsetAsync
                    0.89%  240.79ms       802  300.24us  46.043us  6.7988ms  cudaMemcpy2D
                    0.32%  86.984ms       804  108.19us  30.365us  3.2462ms  cudaMallocPitch
                    0.23%  62.873ms     50274  1.2500us     677ns  1.0133ms  cudaGetLastError
                    0.09%  24.180ms      6384  3.7870us  2.3440us  829.50us  cudaGetDevice
                    0.02%  4.6674ms        20  233.37us  133.81us  601.68us  cudaMallocManaged
                    0.01%  1.7569ms        10  175.69us  18.490us  700.12us  cudaMalloc
                    0.00%  671.89us       473  1.4200us     677ns  45.365us  cuDeviceGetAttribute
                    0.00%  459.49us         4  114.87us  55.938us  178.23us  cudaMemcpyToSymbol
                    0.00%  70.574us         5  14.114us  9.7390us  17.344us  cuDeviceTotalMem
                    0.00%  60.054us         1  60.054us  60.054us  60.054us  cudaGetDeviceProperties
                    0.00%  53.284us         6  8.8800us  6.9280us  13.177us  cudaStreamDestroy
                    0.00%  25.833us         4  6.4580us  4.1150us  9.6870us  cudaStreamAttachMemAsync
                    0.00%  22.137us         4  5.5340us  4.2710us  7.2920us  cuInit
                    0.00%  14.583us         7  2.0830us  1.3020us  3.4900us  cuDeviceGetCount
                    0.00%  10.572us         4  2.6430us  2.3960us  3.0200us  cuDriverGetVersion
                    0.00%  10.000us         5  2.0000us  1.6150us  2.5000us  cuDeviceGetName
                    0.00%  9.2180us         6  1.5360us     989ns  2.1870us  cuDeviceGet
                    0.00%  5.5710us         5  1.1140us     937ns  1.5630us  cuDeviceGetUuid
                    0.00%  1.1460us         1  1.1460us  1.1460us  1.1460us  cudaGetDeviceCount
```

Listing 5: `nvprof` Results

It can be seen that maximum time among all GPU activities is taken in the calculation of keypoints. This is, however still less than the overhead to launch the kernels (`cudaLaunchKernel`).

Conclusion

In conclusion, ORB-SLAM2 is a popular feature-based visual SLAM algorithm that has become one of the most popular algorithms in both academia and industry due to its accuracy, robustness, and real-time performance. It uses the Oriented FAST and Rotated BRIEF (ORB) feature detector and descriptor for simultaneous localization and mapping. However, as the size of the map increases, the computational requirements grow exponentially, leading to a computational bottleneck.

This bottleneck is especially apparent in the ORB feature descriptor as it is computationally expensive, and the keypoint matching stage of the algorithm involves matching thousands of features in real-time, leading to high computational requirements. To address this issue, GPUs can be used to parallelize the algorithm and provide significant speedups in this stage. GPUs can perform thousands of operations in parallel, making them ideal for computationally intensive tasks such as keypoint matching and bundle adjustment, and can significantly reduce the computational time of ORB-SLAM2, leading to real-time performance.

The use of GPUs in ORB-SLAM2 can significantly reduce the computational time and provide speedup, scalability, accuracy, flexibility, and cost-effectiveness. The implementation of ORB-SLAM2 in CUDA involves modifying the existing CPU-based code to use CUDA for parallel processing on the GPU.

The `tileCalcKeypoints_kernel` CUDA kernel function performs keypoint detection on an input image, and its arguments include the input image, maximum number of keypoints, threshold values, score matrix, and counter pointer. The scalability of ORB-SLAM2 is crucial, as it can handle different camera configurations, including monocular, stereo, and RGB-D cameras, and can create and maintain maps of large-scale environments, making it suitable for applications such as autonomous driving and robotics.

Overall, the use of GPUs in ORB-SLAM2 has promising potential for real-time SLAM applications and can be extended to other computer vision and machine learning tasks.

Contribution List

Roll Number	**Name**	**Contribution**
18CS10031	Lanke Leela Manohar	No contribution except attending 1-2 meeting
18EC32006	Mayur Jiwatode	ORB slam2 code analysis,Understanding and optimising the code for some files, report preparation and presentation
18EC32001	Bipin Kumar Mandal	ORB slam2 code analysis, optimization of GPU kernel code for calculating the angle of gradients of each keypoint, setting up dependencies and libraries, performance analysis, preparation of report and presentation.
18EC35034	Utkarsh Patel	Setting up dependencies and libraries, entire code analysis (with focus on module for calculating keypoints), GPU code analysis for calculating keypoints, optimizing GPU code for calculating keypoints, profiling, performance analysis, preparation of report and presentation
19CS30019	Girish Kumar	ORB slam2 code analysis, optimization of GPU kernel code for calculating keypoints, setting up dependencies and libraries, performance analysis, preparation of report and presentation.
19CS10074	Kamalesh Garnayak	ORB slam2 code analysis, GPU code analysis for ORB extraction, optimizing GPU code for ORB extraction, performance analysis, preparation of presentation
19CS30037	Rajas Bhatt	Setting up dependencies and libraries, analysis of code for optimizations (with focus on the ORB Extractor module), GPU codea analysis for extracting ORB descriptions, profiling of GPU code, preparation of report and presentation

Table 1: List of Contributions of Group Members to the Project

References

[1] Raul Mur-Artal and Juan D. Tardós. "ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras". In: *CoRR* abs/1610.06475 (2016). arXiv: `1610.06475`. URL: `http://arxiv.org/abs/1610.06475`.

[2] Hugh F. Durrant-Whyte and Tim Bailey. "Simultaneous Localisation and Mapping (SLAM) : Part I The Essential Algorithms". In: 2006.

[3] T. Bailey and H. Durrant-Whyte. "Simultaneous localization and mapping (SLAM): part II". In: *IEEE Robotics Automation Magazine* 13.3 (2006), pp. 108–117. DOI: `10.1109/MRA.2006.1678144`.

[4] Ethan Rublee et al. "ORB: an efficient alternative to SIFT or SURF". In: Nov. 2011, pp. 2564–2571. DOI: `10.1109/ICCV.2011.6126544`.