# cuFAST: 基于 GPU 的 FAST 算法加速及其在 ORB-SLAM2 中的应用

December 20, 2020

## 1 背景介绍

### 1.1 ORB-SLAM

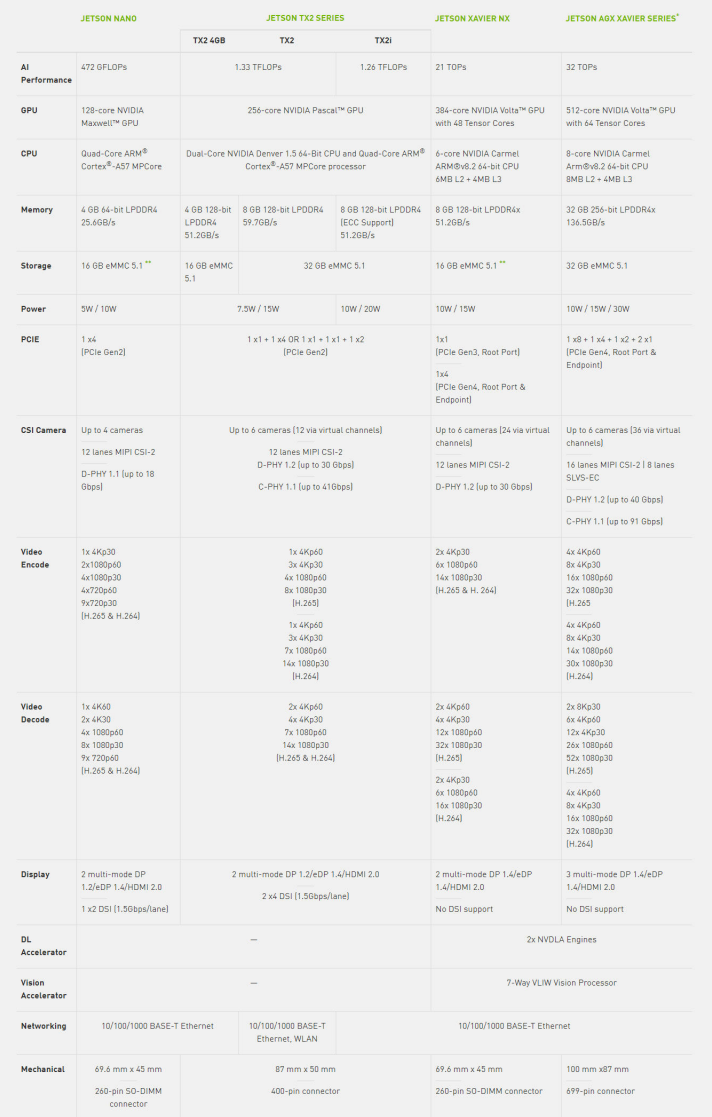
SLAM(Simultaneous Localization and Mapping) 指同步定位与建图，是自主移动机器人进行环境感知和自主导航的重要算法。ORB-SLAM [1] 是一个实时视觉 SLAM 库, 支持单目相机，可以进行相机位姿计算和稀疏三维地图重建，之后推出的 ORB-SLAM2 增加了对双目相机和 RGB-D 深度相机的支持。在不混淆算法的描述的情况下，下文对 ORB-SLAM 和 ORB-SLAM2 混用。

### 1.2 Nvidia Jeson 家族

Nvidia Jeson 家族显卡是英伟达公司推出的适用于新一代自主机器的嵌入式系统的显卡系列。图 1.1 是 Jeson 家族显卡的横向对比图[2]。

[1] https://github.com/raulmur/ORB_SLAM2

[2] https://developer.nvidia.com/embedded/jetson-modules



| | JETSON NANO | JETSON TX2 SERIES | | | JETSON XAVIER NX | JETSON AGX XAVIER SERIES* |
|---|---|---|---|---|---|---|
| | | TX2 4GB | TX2 | TX2i | | |
| AI Performance | 472 GFLOPs | 1.33 TFLOPs | | 1.26 TFLOPs | 21 TOPs | 32 TOPs |
| GPU | 128-core NVIDIA Maxwell™ GPU | 256-core NVIDIA Pascal™ GPU | | | 384-core NVIDIA Volta™ GPU with 48 Tensor Cores | 512-core NVIDIA Volta™ GPU with 64 Tensor Cores |
| CPU | Quad-Core ARM® Cortex®-A57 MPCore | Dual-Core NVIDIA Denver 1.5 64-Bit CPU and Quad-Core ARM® Cortex®-A57 MPCore processor | | | 6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6MB L2 + 4MB L3 | 8-core NVIDIA Carmel Arm®v8.2 64-bit CPU 8MB L2 + 4MB L3 |
| Memory | 4 GB 64-bit LPDDR4 25.6GB/s | 4 GB 128-bit LPDDR4 51.2GB/s | 8 GB 128-bit LPDDR4 59.7GB/s | 8 GB 128-bit LPDDR4 (ECC Support) 51.2GB/s | 8 GB 128-bit LPDDR4x 51.2GB/s | 32 GB 256-bit LPDDR4x 136.5GB/s |
| Storage | 16 GB eMMC 5.1 ** | 16 GB eMMC 5.1 | 32 GB eMMC 5.1 | | 16 GB eMMC 5.1 ** | 32 GB eMMC 5.1 |
| Power | 5W / 10W | 7.5W / 15W | | 10W / 20W | 10W / 15W | 10W / 15W / 30W |
| PCIE | 1 x4 (PCIe Gen2) | 1 x1 + 1 x4 OR 1 x1 + 1 x1 + 1 x2 (PCIe Gen2) | | | 1x1 (PCIe Gen3, Root Port)<br>1x4 (PCIe Gen4, Root Port & Endpoint) | 1 x8 + 1 x4 + 1 x2 + 2 x1 (PCIe Gen4, Root Port & Endpoint) |
| CSI Camera | Up to 4 cameras<br>12 lanes MIPI CSI-2<br>D-PHY 1.1 (up to 18 Gbps) | Up to 6 cameras (12 via virtual channels)<br>12 lanes MIPI CSI-2<br>D-PHY 1.2 (up to 30 Gbps)<br>C-PHY 1.1 (up to 41Gbps) | | | Up to 6 cameras (24 via virtual channels)<br>12 lanes MIPI CSI-2<br>D-PHY 1.2 (up to 30 Gbps) | Up to 6 cameras (36 via virtual channels)<br>16 lanes MIPI CSI-2 \| 8 lanes SLVS-EC<br>D-PHY 1.2 (up to 40 Gbps)<br>C-PHY 1.1 (up to 91 Gbps) |
| Video Encode | 1x 4Kp30<br>2x1080p60<br>4x1080p30<br>4x720p60<br>9x720p30<br>(H.265 & H.264) | 1x 4Kp60<br>3x 4Kp30<br>4x 1080p60<br>8x 1080p30<br>(H.265)<br>1x 4Kp60<br>3x 4Kp30<br>7x 1080p60<br>14x 1080p30<br>(H.264) | | | 2x 4Kp30<br>6x 1080p60<br>14x 1080p30<br>(H.265 & H. 264) | 4x 4Kp60<br>8x 4Kp30<br>16x 1080p60<br>32x 1080p30<br>(H.265<br>4x 4Kp60<br>8x 4Kp30<br>14x 1080p60<br>30x 1080p30<br>(H.264) |
| Video Decode | 1x 4K60<br>2x 4K30<br>4x 1080p60<br>8x 1080p30<br>9x 720p60<br>(H.265 & H.264) | 2x 4Kp60<br>4x 4Kp30<br>7x 1080p60<br>14x 1080p30<br>(H.265 & H.264) | | | 2x 4Kp60<br>4x 4Kp30<br>12x 1080p60<br>32x 1080p30<br>(H.265)<br>2x 4Kp30<br>6x 1080p60<br>16x 1080p30<br>(H.264) | 2x 8Kp30<br>6x 4Kp60<br>12x 4Kp30<br>26x 1080p60<br>52x 1080p30<br>(H.265)<br>4x 4Kp60<br>8x 4Kp30<br>16x 1080p60<br>32x 1080p30<br>(H.264) |
| Display | 2 multi-mode DP 1.2/eDP 1.4/HDMI 2.0<br>1 x2 DSI (1.5Gbps/lane) | 2 multi-mode DP 1.2/eDP 1.4/HDMI 2.0<br>2 x4 DSI (1.5Gbps/lane) | | | 2 multi-mode DP 1.4/eDP 1.4/HDMI 2.0<br>No DSI support | 3 multi-mode DP 1.4/eDP 1.4/HDMI 2.0<br>No DSI support |
| DL Accelerator | — | | | | 2x NVDLA Engines | |
| Vision Accelerator | — | | | | 7-Way VLIW Vision Processor | |
| Networking | 10/100/1000 BASE-T Ethernet | 10/100/1000 BASE-T Ethernet, WLAN | | 10/100/1000 BASE-T Ethernet | | |
| Mechanical | 69.6 mm x 45 mm<br>260-pin SO-DIMM connector | 87 mm x 50 mm<br>400-pin connector | | | 69.6 mm x 45 mm<br>260-pin SO-DIMM connector | 100 mm x87 mm<br>699-pin connector |

Figure 1.1: Jeson 家族显卡的横向对比图

Jeson 系列显卡可以显著加速自主移动机器人的 SLAM 过程，从而在保证建图和导航的精度的同时，较好地提高了 SLAM 的实时性，目前比较流行的使用神经网络的 SLAM 框架中，大多通过 Jeson 系列显卡，如 TX2，进行并行加速，大大提高了框架的实时性。因此，利用 Jeson 显卡对传统的 ORB-SLAM2 框架中的部分算法进行加速，将提高 ORB-SLAM2 在嵌入式设备的运行速度。

## 1.3 课题简介

本课题旨在通过对 ORB-SLAM2 进行问题拆解，利用 GPU 并行计算对 ORB-SLAM2 中的部分算法进行加速，从而提高 ORB-SLAM2 框架的实时性。本课题着眼于 ORB-SLAM2 的 Tracking 线程中的特征提取算法，将特征提取算法分为计算图像金字塔、FAST 角点提取、四叉树角点均匀化、计算旋转不变性、添加描述子五部分。

本课题完成了基于 GPU 的 FAST 角点提取的并行化版本 cuFAST，cuFAST 相对于 opencv 的 FAST 角点提取算法加速比最高达到了 1.62，相对于 ORB-SLAM2 中的 FAST 算法的加速比达到了 14.84。

由于时间关系，本课题仅完成了 FAST 算法的并行化，而在 ORB-SLAM2 的特征提取算法中 FAST 角点提取与四叉树角点均匀化算法有着一定的耦合关系，在 ORB-SLAM2 中使用 cuFAST 算法而没有对角点均匀化并行，只能提取到线特征而不是角点（将在实验部分详述），但是，这证明了对 ORB-SLAM2 的并行化加速的可行性，之后可以继续现在的工作进行完善。

# 2 问题拆解

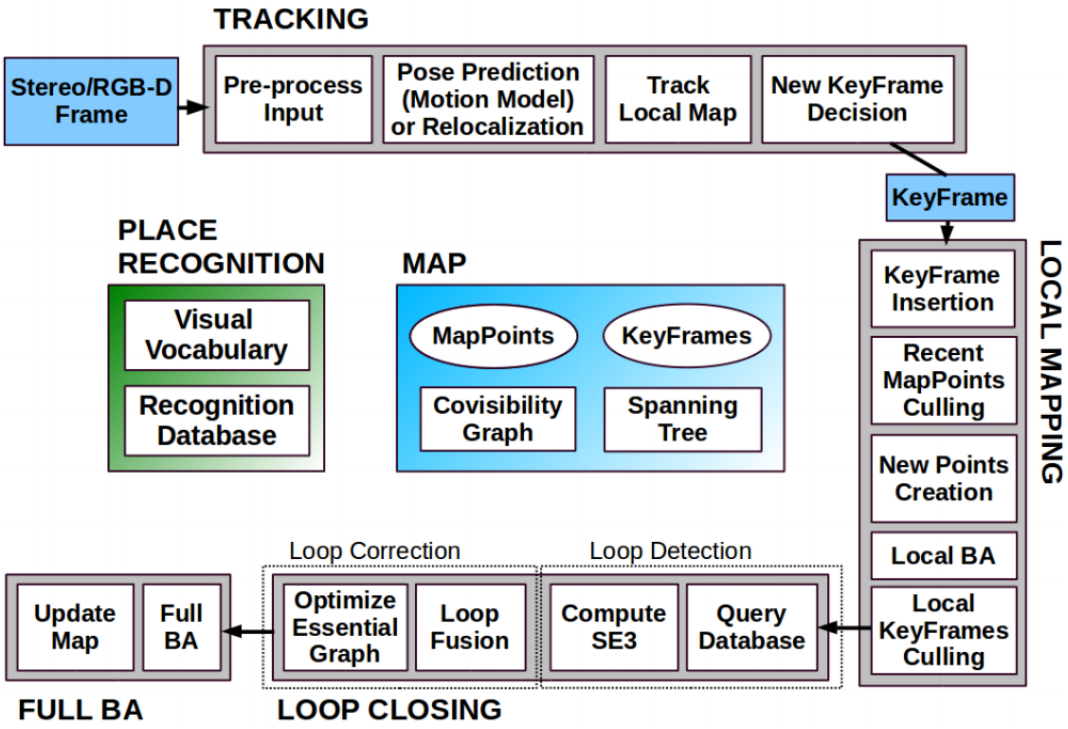
## 2.1 ORB-SLAM 框架拆分

ORB-SLAM2 的框架图如图 2.1 所示。ORB-SLAM 通过 Tracking、Local Mapping、Loop Closing 三个线程并行运行。

在 Tracking 线程中，先对输入的图像数据进行预处理，然后进行特征提取并构造图像关键帧。

在 Local Mapping 线程中，对从 Tracking 线程传输来的关键帧进行三角化，求解关键帧之间的相对位姿并进行局部 BA 优化。

ORB-SLAM2 维护了一个一关键帧为顶点相对位姿为边的关键帧共视图，在 Loop Closing 线程中，当该共视图构成闭环的时候，会通过计算重合的两个关键帧的相对位姿误差并进行误差传播从而进行全局的位姿优化。




(a) System Threads and Modules.

Figure 2.1: ORB-SLAM2 框架图

ORB-SLAM 在图像特征提取、相对位姿矩阵计算、共视图 (Covisibility Graph) 维护过程中存在大量的可并行成分，如果能够利用 GPU 对这些部分进行加速，将大大提高 ORB-SLAM 的实时性。在本次课题中，我以图像特征提取为例进行了 GPU 并行加速，下面对 ORB 的图像特征提取算法进行分析。

## 2.2 ORB-SLAM 中的图像特征提取

ORB-SLAM 的图像特征提取流程如图 2.2 所示。

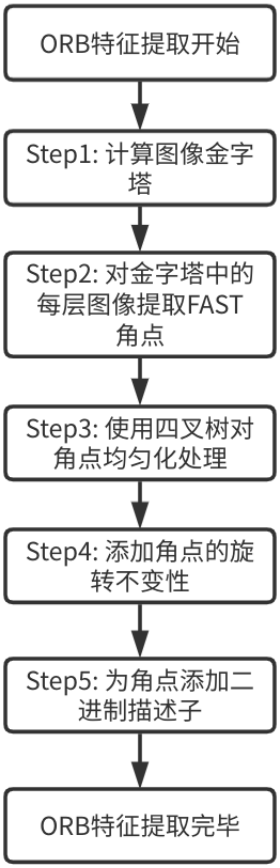



Figure 2.2: ORB-SLAM 图像特征提取流程

这其中，第 2-5 步是以图像为单位、对图像金字塔中的每一层图像分别进行处理，而对图像中的每个像素点的处理是相对独立的，因此均可以通过并行化加速。在本课题中，由于时间关系，只对第二步进行了并行加速，即 FAST 角点提取算法，提出了 cuFAST。下面分别介绍这一步的算法原理、基于 CPU 的朴素实现、ORB-SLAM 中的实现（基于 OpenCV）、使用 GPU 加速的朴素实现、进行优化后的实现。

# 3 FAST 角点提取及并行化

## 3.1 FAST 角点提取算法

FAST（Features from Accelerated Segment Test）是由 Edward Rosten 和 Tom Drummond 在 2006 年提出的角点提取算法。因为 FAST 算法数学原理很简洁、提取速度很快、提取精度很高，所以在众多角点提取算法中，FAST 脱引而出，并在 SLAM、模式识别、机

器学习等诸多邻域中得到广泛的应用。算法 1 对 FAST 角点提取进行了描述。

Algorithm 1 FAST 角点提取

Input:

灰度图像矩阵（单通道），I ;

角点提取阈值，threshold;

Output:

角点集合，corners;

1: 选取图像中的一个像素 p，设其灰度值为 $I_p$，考虑以 p 为圆心、半径为 r 的 Bresenham 圈，设在该 Bresenham 圈上的像素个数为 k;

2: 对从步骤 1 中选取的 k 个像素点，如果其中有连续 l(l < k) 个像素点的灰度值都比 $I_p + threshold$ 大，即比 p 更亮，或者其中有连续 l(l < k) 个像素点的灰度值都比 $I_p - threshold$ 小，即比 p 更暗，则将 p 加入角点集合，并计算 p 的得分，即对 p 与周围 k 个像素点的灰度值的差值的绝对值求和;

对图像的所有像素点重复步骤 1-2。

3: 非极大值抑制：对 corners 中的所有角点，进行两两对比，如果两个角点相邻，则删除得分较小的角点;

4: return corners;

在算法的第 2 步中，对 r 和 k 及 l 的选择有多种，如当 $r = 3$ 时，以 p 为圆心的该 Bresenham 圈上的像素个数为 16，设 $l = 9$，即连续 9 个像素点满足条件时，则将 p 加入角点集合，这种模式称为 16-9 角点提取。如图 3.1 所示。

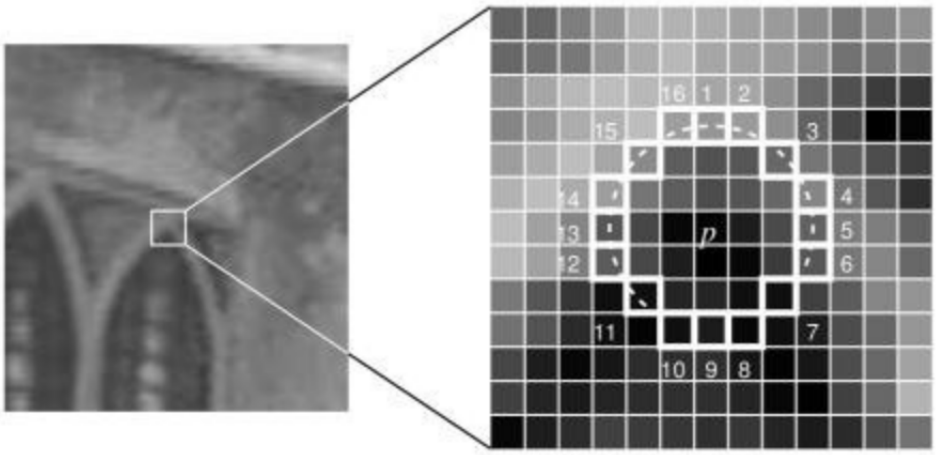



Figure 3.1: 16-9 角点提取

在算法的第三步中，可以通过两种方法判断角点是否相邻。一种方法是遍历角点集合，通过判断两个角点在图像中坐标的欧氏距离，如果小于某一既定阈值，则称这两个角点相邻。

另一种是构造角点在图像上的控制域，即考虑角点 A 的一个正方形邻域，判断角点 B 是否出现在角点 A 的邻域中，如果在邻域内，则称这两个角点相邻。如图 3.2 所示，其中 B 在 A 的 $3\times3$ 邻域中，而 C 不在，故 A 与 B 相邻，与 C 不相邻。

使用第二种方法基于 GPU 实现可以有效并行非极大值抑制部分，结合算法中第 1-2 部分，将大大提高角点提取速度。

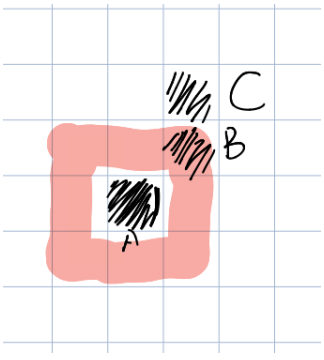



Figure 3.2: 角点相邻判断。其中 B 在 A 的 $3\times3$ 邻域中，而 C 不在，故 A 与 B 相邻，与 C 不相邻

## 3.2 朴素的 CPU 版本 FAST 算法实现

该版本的实现仅仅对算法进行了复现，采用了 16-9 角点提取模式，为了提高算法第 2 步的速度，首先对 16 个点中的 1,5,9,13 进行了初步检测，然后对所有点进行进一步检测。除此外，并没有进行任何优化。

## 3.3 OpenCV 中的 FAST 算法实现

ORB-SLAM 采用了 Opencv 库中的 FAST 算法实现，OpenCV 中对 FAST 算法进行了多个提取模式的实现，并通过部分步骤的优化（如 CPU_SSE 指令集 [3] 优化）提高了 FAST 算法的检测速度。

但是 SSE 的指令集是 X86 架构 CPU 特有的，对于 ARM 架构、MIPS 架构等 CPU 是不支持的，所以使用了 SSE 指令集的程序，是不具备可移植标准的。而移动机器人平台使用的 CPU 大多为了保证低功耗采用了 ARM 架构，所以这样的加速在移动机器人平台上会失效，角点提取的速度会大大降低。相反利用 GPU 进行并行加速，可以有效解决这个问题。

```
#if CV_SSE2
    if( patternSize == 16 )
    {
    for(; j < img.cols - 16 - 3; j += 16, ptr += 16)
    {
      __m128i m0, m1;
      __m128i v0 = _mm_loadu_si128((const __m128i*)ptr);
      __m128i v1 = _mm_xor_si128(_mm_subs_epu8(v0, t), delta);
      v0 = _mm_xor_si128(_mm_adds_epu8(v0, t), delta);

      __m128i x0 = _mm_sub_epi8(_mm_loadu_si128((const __m128i*)(ptr + pi
```

Figure 3.3: openCV 的 FAST 算法实现中使用的 SSE 指令集加速

英伟达公司的 Jeson 家族显卡是在移动机器人平台上的 SLAM 算法加速一个有效的解决方案，如 Jetson TX2 是基于 NVIDIA Pascal™ 架构的 AI 单模块超级计算机，性能强大（1 TFLOPS），外形小巧，节能高效（7.5W），非常适合机器人、无人机、智能摄像机和便携医疗设备等智能终端设备。

[3] SSE 的全称是 Sreaming SIMD Extensions，它是一组 Intel CPU 指令，用于像信号处理、科学计算或者 3D 图形计算一样的应用。其优势包括：更高分辨率的图像浏览和处理、高质量音频、MPEG2 视频、同时 MPEG2 加解密；语音识别占用更少 CPU 资源；更高精度和更快响应速度。使用 SSE 指令集，主要是通过 8 个 128-bit 的寄存器：xmm0 到 xmm7 来完成的。

## 3.4 ORB-SLAM 分网格提取角点的弊端

ORB-SLAM2 对图像金字塔中的每一层图像进行角点提取的时候，为了方便随后将提取的角点使用四叉树结构进行均匀化处理，首先将图像划分成若干网格，然后对每个网格分别提取角点，这样破坏了数据访问的空间局部性，使得整幅图像的角点速度受到很大影响。虽然之后进行角点均匀化处理的时候只需要从网格中筛选节点即可，简化了算法复杂度，但是这也是数据无法并行处理的权衡办法。而当采用 GPU 加速后使角点提取和角点均匀化都可以并行进行的时候，这样的问题就可以迎刃而解，并且大大提高算法的速度。

## 3.5 GPU 加速的 FAST 算法实现及其优化

本课题实现了基于 GPU 加速的 FAST 算法 cuFAST，在朴素实现中，只在遍历图像判断每个像素点是否满足角点的条件并计算得分的时候进行了数据并行处理，除此外和 CPU 的朴素实现一致，只不过将计算过程转移到了 GPU 中。优化版本中使用的优化技术主要有：

1) constant memory。

   由于图像中每一像素点的坐标与其 Bresenham 圈中 16 个邻像素点的相对位置是相同的，而且在非极大值抑制中像素点坐标与其邻域像素点的坐标的相对位置是相同的，所以可以分别将其储存在 GPU 的 Constant Memory 中，加速访问。

2) shared memory。

   cuFAST 在两方面使用 shared memory 进行优化。

   首先，在判断图像中每个像素点是否是角点并计算得分的时候，对每一个 block 中的像素，首先进行 padding，设 block 的大小为 $K \times K$，构造一个 $(K+padding) \times K+padding$ 的区域，其中 padding 是 Bresenham 圈的半径大小，该范围作为当前 block 中所有像素与自己的 Bresenham 圈的像素值对比时候需要的所有可能的像素值。如图 3.4 所示。虽然在 block 边界的像素值不存在 Bresenham 圈（无法构成完整的圈），但是通过地址映射可以在 shared memory 忽略其不完整的圈中的不存在的像素点。

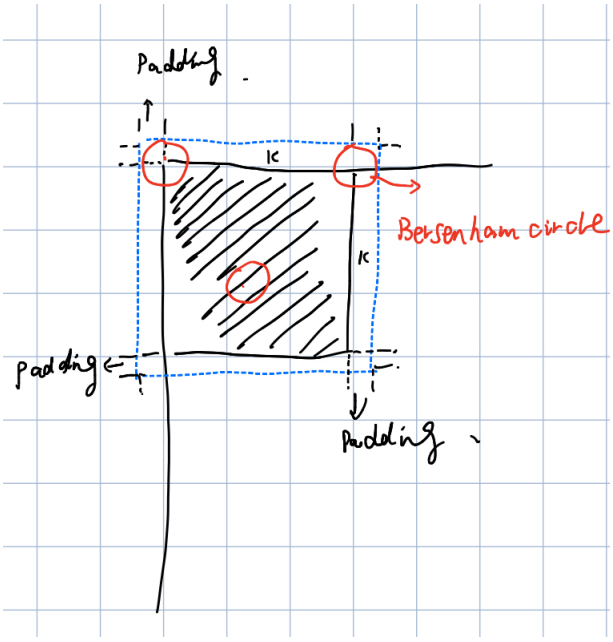



Figure 3.4: 对 block 进行 padding 示意图

由于只需要使用 block 中的线程索引确定扩展后的范围中的所有像素在 shared memory 中的位置，首先将 block 中的像素重新映射到扩展后的范围中（从头依次），可以证明，在 block 大小远大于 padding 的时候（比如 padding 为 3，blocksize 为 32），block 中的像素个数大于扩展后的像素的一半，故通过只需要取重映射到扩展范围的前一半像素点然后将其在 shared memory 中的索引复制到后一半即可。这样大大降低了时间复杂度。地址映射代码如图 3.5 所示。

```
extern __shared__ unsigned char shared_data[];
int idx_x = threadIdx.x + blockIdx.x * blockDim.x;
int idx_y = threadIdx.y + blockIdx.y * blockDim.y;
int g_coord1d = cuGet1dcoords(idx_x, idx_y, height, width, true);
// fiil date in shared memory
int idx_first =
    cuGet1dcoords(threadIdx.x, threadIdx.y, BLOCKSIZE, BLOCKSIZE, false);
int shared_x1 = (idx_first % width_shared) - PADDING;
int shared_y1 = (idx_first / width_shared) - PADDING;
// map to shared memory
int extern_global_x1 = shared_x1 + blockDim.x * blockIdx.x;
int extern_global_y1 = shared_y1 + blockDim.y * blockIdx.y;
int shared_x2 = ((idx_first % width_shared) - PADDING);
int shared_y2 = ((idx_first / width_shared) - PADDING) + width_shared / 2;
int extern_global_x2 = extern_global_x1;
int extern_global_y2 = extern_global_y1 + width_shared / 2;
int extern_global_idx1 =
    cuGet1dcoords(extern_global_x1, extern_global_y1, height, width, false);
int extern_global_idx2 =
    cuGet1dcoords(extern_global_x2, extern_global_y2, height, width, false);
if (idx_first < (size_sharedmem / 2) && extern_global_x1 >= 0 &&
    extern_global_x1 < width && extern_global_y1 >= 0 &&
    extern_global_y1 < height && extern_global_x2 >= 0 &&
    extern_global_x2 < width && extern_global_y2 >= 0 &&
    extern_global_y2 < height) {
  shared_data[(shared_y1 + PADDING) * width_shared + (shared_x1 + PADDING)] =
      d_input[extern_global_idx1];
  shared_data[(shared_y2 + PADDING) * width_shared + (shared_x2 + PADDING)] =
      d_input[extern_global_idx2];
}
__syncthreads();
```

Figure 3.5: shared memory 与 block 之间的地址映射

另一方面，在非极大值抑制中，构造了一个得分矩阵，其大小与图像大小一致，然后用 shared memory 存储该得分矩阵，做类似的地址映射后，再使用 shared memory 进行非极大值抑制。

3) stream。

在 cuda 编程中，stream 可以用来隐藏数据传输的过程，从而节省时间。但是，stream 的并行度受到诸多因素的影响，如数据处理之间如果存在较大延迟的 IO 的话，或者数据传输的同步操作，会影响 stream 的并行度，甚至导致并行无法起作用。在序列图像处理中，cuFAST 采用了 stream 进行了加速。具体实现中，为每一张图像的角点提取创建一条流。由于图像读取操作延迟很大，所以我将待处理图像先预存起来，然后分别处理。使用 stream 后的效果如图 3.6 所示。

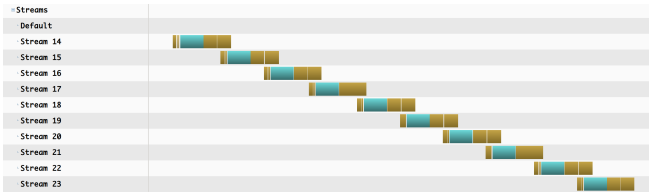



Figure 3.6: stream 化在 Nvidia Visual Profiler 中的效果

可以看到，数据的传输被较好地隐藏了起来。但是在实际的工程中使用流是一件比较棘手的事情。首先流的创建需要耗费一定的时间，其次，流的并行性很脆弱，而且和数据同步之间是互相矛盾的，需要精巧处理。在一些较大计算量的任务中，流带来的性能提升其实很小。

# 4 实验测试

实验分别在两台主机完成。

机器 1：

CPU：Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz

GPU：GeForce MX250

机器 2：

CPU：Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz

GPU：Tesla K40m

详细配置如图 4.1-图 4.4 所示。

```
架构：              x86_64
CPU 运行模式：      32-bit, 64-bit
字节序：            Little Endian
CPU：               12
在线 CPU 列表：     0-11
每个核的线程数：    2
每个座的核数：      6
座：                1
NUMA 节点：         1
厂商 ID：           GenuineIntel
CPU 系列：          6
型号：              166
型号名称：          Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz
步进：              0
CPU MHz：           884.635
CPU 最大 MHz：      1100.0000
CPU 最小 MHz：      400.0000
BogoMIPS：          3199.92
虚拟化：            VT-x
L1d 缓存：          32K
L1i 缓存：          32K
L2 缓存：           256K
L3 缓存：           12288K
NUMA 节点0 CPU：    0-11
```

Figure 4.1: 机器 1CPU

```
Detected 1 CUDA Capable device(s)

Device 0: "GeForce MX250"
  CUDA Driver Version / Runtime Version          11.0 / 10.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 2003 MBytes (2099904512 bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1582 MHz (1.58 GHz)
  Memory Clock rate:                             3504 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.0, CUDA Runtime Version = 10.2, NumDevs = 1, Device0 = GeForce MX250
Result = PASS
```

Figure 4.2: 机器 1GPU

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
座：                   2
NUMA 节点：            2
厂商 ID:               GenuineIntel
CPU 系列：             6
型号：                 79
型号名称：             Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
步进：                 1
CPU MHz:               1200.347
CPU max MHz:           3000.0000
CPU min MHz:           1200.0000
BogoMIPS:              4190.29
虚拟化：               VT-x
L1d 缓存：             32K
L1i 缓存：             32K
L2 缓存：              256K
L3 缓存：              20480K
NUMA 节点0 CPU:        0-7,16-23
NUMA 节点1 CPU:        8-15,24-31
```

Figure 4.3: 机器 2CPU

```
Detected 1 CUDA Capable device(s)

Device 0: "Tesla K40m"
  CUDA Driver Version / Runtime Version          10.2 / 8.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11441 MBytes (11996954624 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA Runtime Version = 8.0, NumDevs = 1, Device0 = Tesla K40m
Result = PASS
```

Figure 4.4: 机器 2GPU

1. 不同版本的运行时间对比。本实验对五个版本的 fast 算法进行了六次测试对比，分别为朴素 CPU 实现（i7-10710U）、ORB-SLAM 实现（i7-10710U）、朴素 GPU 实现（GeForce MX250）、优化 GPU 实现（GeForce MX250）、opencv 实现（i7-10710U）、优化 GPU 实现（Tesla K40m）。运行时间如图 4.5 所示。

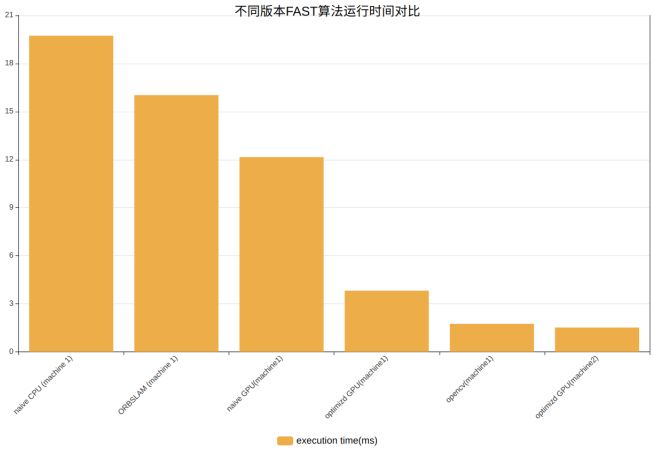



Figure 4.5: 不同版本 FAST 算法运行时间对比

可以看到在 Tesla K40m 上的优化 GPU 版本要比在 i7-10710U 的 opencv 版本快，加速比为

$$\frac{1.75}{1.51901} = 1.15$$

而其相对于在 i7-10710U 的 ORB-SLAM 版本加速比达到了

$$\frac{16.027}{1.51901} = 10.55$$

在自主移动机器人上，由于其 CPU 为 arm 架构，opencv 的指令集加速将失效，GPU 相对 opencv 版本的加速比会进一步提高。

2. Block Size 对运行时间的影响。本实验在 Tesla K40m 上测试了不同 blocksize 对 GPU 优化版本的运行时间影响。如图 4.6 所示。

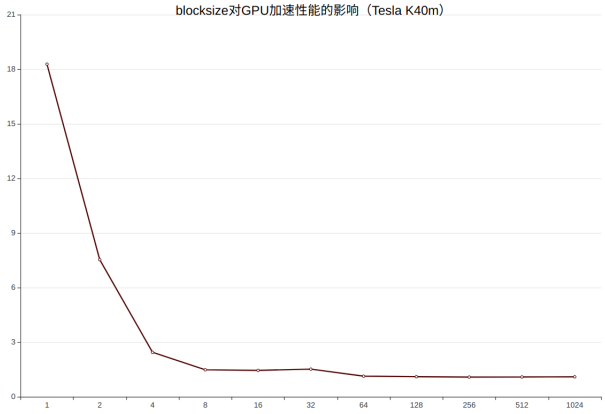



Figure 4.6: blocksize 对 GPU 加速性能的影响（Tesla K40m）

可以看到，随着 blocksize 的提高 GPU 的并行优势逐渐增大，但是在 blocksize 接近每个 block 的 thread 数量上限的时候，运行时间会下降，这是因为每个 block 中 thread 分到的计算资源（如寄存器）会减小，thread 不得不把数据放到 global memory 中，增加了访存延迟。

最佳的 blocksize 是 256，此时运行时间为 1.083 ms。相对在 i7-10710U 的 opencv 版本的加速比为

$$\frac{1.75}{1.083} = 1.62$$

而其相对于在 i7-10710U 的 ORB-SLAM 版本加速比达到了

$$\frac{16.027}{1.083} = 14.84$$

3. 在 ORB-SLAM 中嵌入 GPU 加速的 FAST 角点提取。ORB-SLAM 中特征提取算法的实现，是将图像分成网格，在每个网格进行角点提取和均匀化，而 cuFAST 的并行特性使得可以直接对整幅图像进行角点提取和均匀化。如图 4.7 为使用 cuFAST 对一帧图像构成的图像金字塔进行的角点提取。

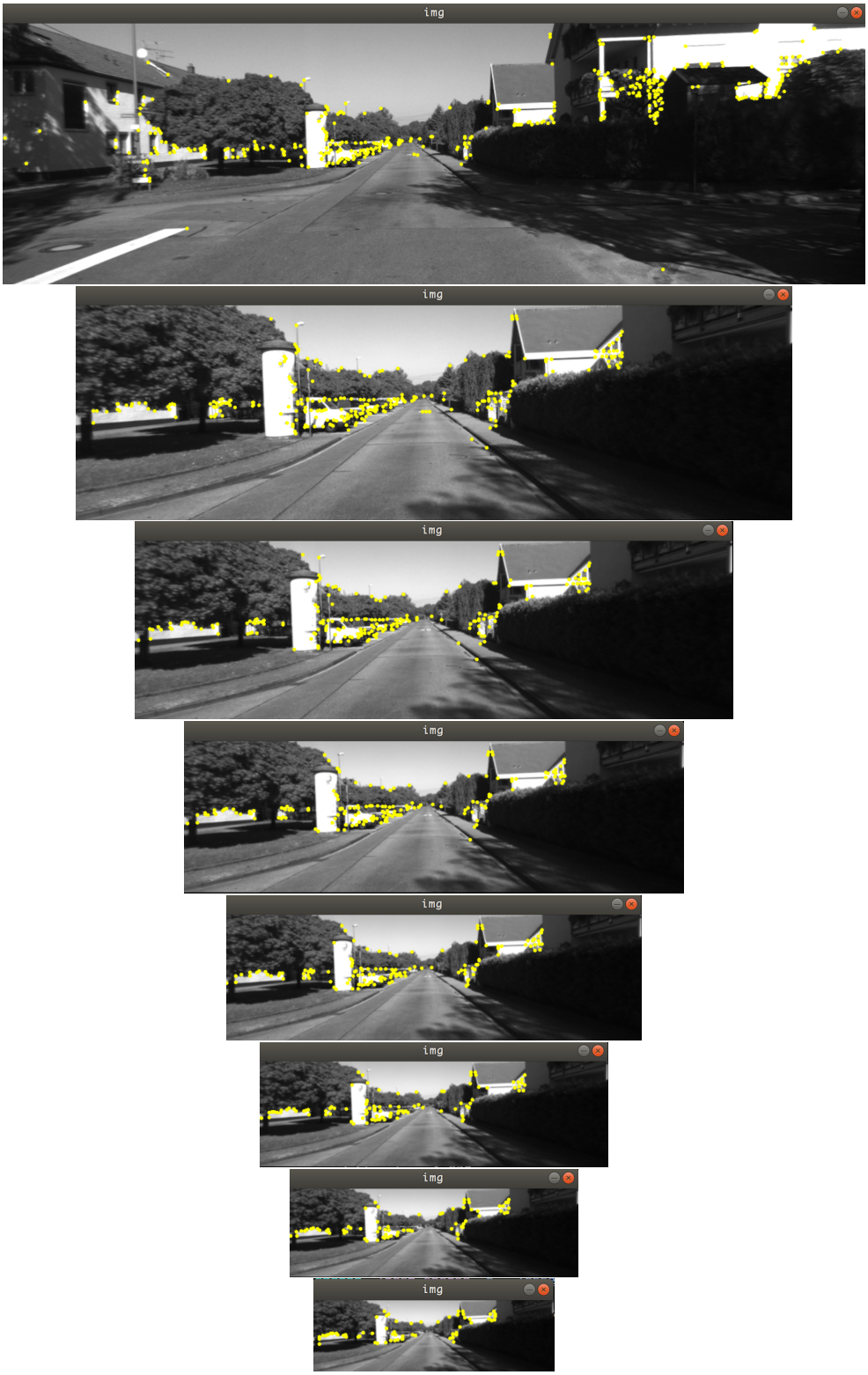



Figure 4.7: cuFAST 对图像金字塔的角点提取

由于没有对四叉树角点均匀算法进行并行化，使得无法提取到点特征，而只能提取线特征，线特征对 SLAM 来说是无益的。如图 4.8 所示。

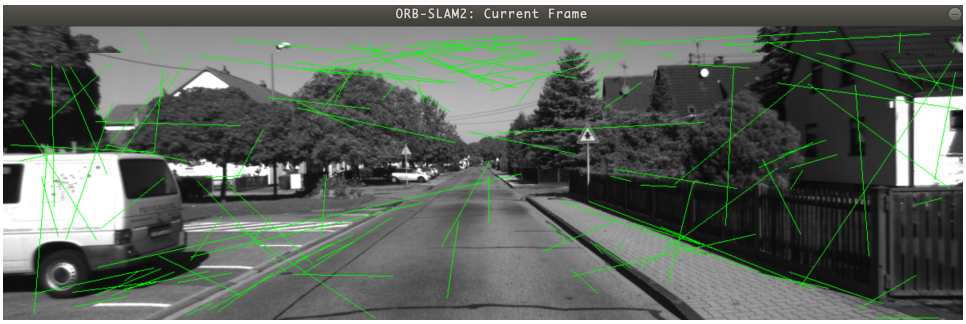



Figure 4.8: cuFAST 提取的线特征

点特征提取及 SLAM 过程如图 4.9、4.10 所示。

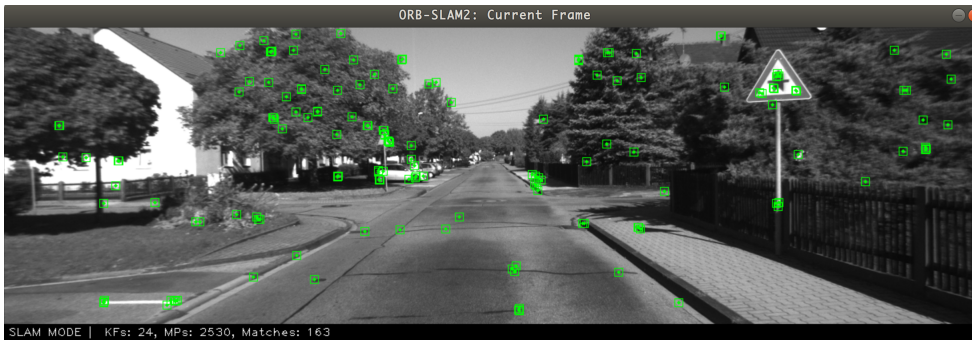



Figure 4.9: ORB-SLAM 提取的角特征

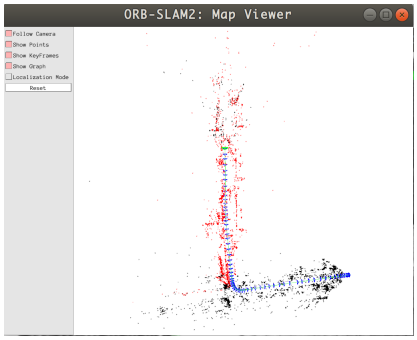



Figure 4.10: ORB-SLAM 的 SLAM 过程示意图

但是 cuFAST 提出了一种将 cuda 和 ORB-SLAM 结合起来，利用 GPU 对 ORB-

SLAM 进行加速的可行方法，在之后进一步的对特征提取中的其他步骤进行并行的算法后，ORB-SLAM 的实时性一定可以得到很大提升。

# 5 讨论和展望

1. 其余算法的加速和对较大工程的 GPU 加速的思考。

   在 ORB-SLAM 中，除了特征提取，在相对位姿转换中存在大量矩阵运算，在共视图中需要维护图结构，这些都可以利用其数据并行性通过 GPU 进行有效加速。但是在对较大工程中的 GPU 加速中，像 cuFAST 这样细粒度地操作内存将会使生产力大大降低，在 cuda 中有很多抽象化的算法库如 thrust[4]、cuBLAS[5]，使用这些库的 API 可以有效提升生产力。虽然在性能上使用库 API 会有一定损失，但相对来说是不错的选择。而且，opencv 中也已经提供了对 cuda 的支持，使用 opencv 的 cudaAPI 也能够大大提高生产力。

2. 使用 openMP 对双目相机 SLAM 的加速。

   双目相机 SLAM 需要同时处理左右目图像，使用 openMP 的多线程并行特性与多个 GPU 建立一对一映射，就可以同时对左右目图像进行基于 GPU 的加速处理了。但是，前提是你有多个 GPU 设备。

3. cuda 编程的一些陷阱。

   初学 cuda 中会遇到很多 cuda 的编程陷阱。我遇到最典型的有两个。

   (1) 不能将 constant memory 变量作为函数参数传递，这样会丧失 constant memory 的特性，而且 constant memory 不需要最终显示 free。

   (2) 不要在 debug 模式中测试性能。在 debug 模式和 release 模式下编译生成的机器代码会非常不同，而且可能会导致出现很多竞争之类的情况，使得并行特性几乎完全损坏。因此，测试性能一定要在 release 模式下进行编译。

4. 从 cuda 编程对体系结构的思考

   现在，很多人在说，摩尔定律正在消逝。的确，在大数据和人工智能飞速发展的今天，功耗墙和内存墙似乎变得越来越显著，想要单纯通过提高制程就提高芯片性能和降低功耗变得越来越难。尤其，对于深度学习训练这种需要大量计算的任务，采用传统的 CPU 架构显然已经不可能。

[4] https://thrust.github.io/

[5] https://docs.nvidia.com/cuda/cublas/index.html

近些年，随着 GPU 的广泛应用和神经网络加速器的发展，从体系结构进行创新似乎是一个不错的解决思路。无论是 14 年起中科寒武纪提出的电脑系列架构，还是后来清华微电子提出了 Thinker 系列架构，亦或是谷歌提出的 TPU，我们可以看到，这种针对深度学习进行的特定架构设计都有的共同点是：一、对卷积等特定计算采用硬件固化（如计算阵列）的方法进行加速；二、利用深度学习任务的数据可重用性、数据相关性提高计算速度；三、设计精巧的内存机制缩短访存的时间。

未来十年是计算机体系结构的黄金十年，不仅仅说的是芯片中计算资源占比越来越大，而更是更巧妙的内存机制的设计。此外，随着新型器件的商业化，访存延迟也会不断降低，芯片性能也必然会突破瓶颈，不断进步。体系结构随应用场景需求而不断创新，人们总可以有新的解决方案，这件事情回顾近几十年芯片行业的发展历史就了然了。

5. 下一步计划

   下一步的计划是利用 opencv 的 cuda 库对特征提取的剩余算法进行加速（以提高生产效率），同时对 ORB-SLAM 中其他两个线程进行算法分解和并行化。

# 6 相关工作

ORB-SLAM2 GPU Optimization[6]基于 openCV 的 cuda 库进行 ORB-SLAM2 加速，得到了很不错的性能。

# 7 总结

本课题完成了基于 GPU 的 FAST 角点提取的并行化 cuFAST，cuFAST 相对于 opencv 的 FAST 角点提取算法加速比最高达到了 1.62，相对于 ORB-SLAM2 中的 FAST 算法的加速比达到了 14.84。同时，本课题将 cuda 代码嵌入到 ORB-SLAM 中提出了使 ORB-SLAM 的基于 GPU 的加速的方案，之后的工作可以利用 opencv 中的 cuda 库实现从而提高生产效率。

# 附录：基于 cmake 的 c++ 和 cuda 工程编译

CMake 简化了跨平台的编译流程，它首先允许开发者编写一种平台无关的 CMakeList.txt 文件来定制整个编译流程，然后再根据目标用户的平台进一步生成所需的本地化 Makefile 和工程文件，如 Unix 的 Makefile 或 Windows 的 Visual Studio 工程。从而做到 “Write once, run everywhere”。CMake 在一些知名开源项目诸如 VTK、ITK、KDE、OpenCV、OSG 等都有应用。基于 cmake 的 c++ 和 cuda 编译是将编译好的 cuda 程序作

[6] https://yunchih.github.io/ORB-SLAM2-GPU2016-final/

为普通的库链接到 c++ 编译的程序中的，如图 6.1 所示。在编译的时候要加上 cuda 的编译选项。

cmake .. -DCMAKE_BUILD_TYPE=Release -DCUDA_ENABLE=cuda

```
IF(CUDA_ENABLE)
    enable_language(CUDA)
    add_library(cudafast src/cuda/cufast.cu)
    target_compile_features(cudafast PUBLIC cxx_std_11)
    # target_link_libraries(stereo_kitti
    # cudafast
    # )
    target_link_libraries(mono_kitti
    cudafast
    )
ENDIF()
```

Figure 7.1: 在 cmake 中加入 cuda

# 参考文献


1. Raúl Mur-Artal, J. M. M. Montiel and Juan D. Tardós. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. IEEE Transactions on Robotics, vol. 31, no. 5, pp. 1147-1163, 2015.

2. Raúl Mur-Artal and Juan D. Tardós. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. IEEE Transactions on Robotics, vol. 33, no. 5, pp. 1255-1262, 2017.

3. Rosten E , Porter R , Drummond T . Faster and better: a machine learning approach to corner detection[J]. IEEE Trans Pattern Anal Mach Intell, 2008, 32(1):105-119.