SGBM 的 C++仿真与 FPGA 硬件设计

作者： EscapeTHU (github.com)

1. SGBM 引言

在后文的介绍之中，第 2 部分将介绍双目视觉匹配 SGBM 算法的基本原理；第 3 部分将介绍 SGBM 算法的 ubuntu-C++仿真代码编写、算法仿真结果以及针对硬件 FPGA 实现进行的改进与优化；第 4 部分将介绍在硬件 FPGA 上实现 SGBM 的系统设计架构；第 5 部分将介绍 census 变换的调试与改进；第 6 部分将介绍原始代价计算的代码编写、仿真、调试与改进；第 7 部分将介绍第二版代价聚合代码编写、仿真、调试与改进；第 8 部分将介绍后处理代码编写、仿真、调试与改进；第 9 部分将介绍 PL 系统整体设计、编写、仿真与调试；第 10 部分将介绍 PS 系统设计、编写、仿真与调试；第 11 部分将介绍 PL-PS-上位机的联合调试以及展示最终效果；第 12 部分是高度并行化图像处理展望。

2. 双目视觉匹配 SGBM 算法综述

在本次结构化集成电路设计课程设计之中，三维坐标定位部分我选择使用 SGBM 双目视觉匹配算法进行实现。我考虑的主要原因有两个方面：首先，由于本次项目的应用场景是脑机接口的高速高精度定位，需要在室内小范围场景进行高精度小目标定位；其次，本次项目的定位目标是小鼠，不能在小鼠身上部署额外的定位设备。

如下图所示是文献[1][1]给出的不同传感器的定位方式比较，其中最后的视觉定位方式具备小范围内高精度的能力，并且属于主动感知方式、无需向小鼠身上安装其他定位设备，完全符合我们的要求。在视觉定位方式之中，包括单目定位和多目定位，单目定位方式主要依赖内参外参矩阵进行距离的估计，并不能够保证距离估计的准确性，并且需要提前获得较好的内参外参矩阵标定，标定的精度严重限制单目相机的定位精度；相比之下，多目定位使用相对位置固定的 2 台、多台相机，采用双目匹配算法进行三维坐标估计，能够保证精度。因此我最终选择使用双目视觉匹配的方式进行定位。

下图 2 所示展示的是文献[2][2]的 SGBM 算法定位误差，SGBM 算法作为当今最常用、精度相对较高、计算复杂度相对较低的双目匹配算法，被用于多种

[1] F. Zafari, A. Gkelias and K. K. Leung, "A Survey of Indoor Localization Systems and Technologies," in IEEE Communications Surveys & Tutorials, vol. 21, no. 3, pp. 2568-2599, thirdquarter 2019, doi: 10.1109/COMST.2019.2911558.

[2] H. Hirschmuller, "Accurate and efficient stereo processing by semi-global matching and mutual information," 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2005, pp. 807-814 vol. 2, doi: 10.1109/CVPR.2005.56.

领域。下图 2 中课件，SGBM 的定位误差低于其他双目匹配算法，并且最大定位误差仅 2~3%，符合我们对于定位精度的要求。我最终选择使用双目视觉匹配的 SGBM 算法进行实现，作为本项目的三维坐标定位模块。

Technology	Advantages	Disadvantages
IEEE 802.11 n [67] 802.11 ac 802.11 ad	Widely available, high accuracy, does not require complex extra hardware	Prone to noise, requires complex processing algorithms
UWB [68]	Immune to interference, provides high accuracy,	Shorter range, requires extra hardware on different user devices, high cost
Acoustics	Can be used for proprietary applications, can provide high accuracy	Affected by sound pollution, requires extra anchor points or hardware
RFID [69]	Consumes low power, has wide range	Localization accuracy is low
Bluetooth [70]	High throughput, reception range, low energy consumption	Low localization accuracy, prone to noise
Ultrasound [71]	Comparatively less absorption	High dependence on sensor placement
Visible Light [72]	Wide-scale availability, potential to provide high accuracy, multipath-free	Comparatively higher power consumption, range is affected by obstacles, primarily requires LoS

图 1（左）：视觉定位优势

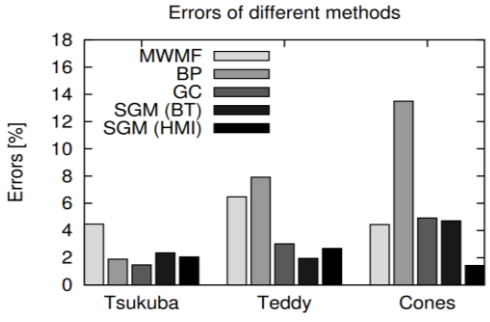


图 2（右）：SGBM 算法相对其他算法误差

3. SGBM 算法 ubuntu-C++代码编写、算法仿真与改进

3.1. SGBM 算法预处理（灰度化、Sobel 算子、图映射）

在标准的 SGBM 算法预处理步骤之中，首先将图像进行灰度化。灰度化的公式是：

$$Grey[i,j] = 0.299 \times R[i,j] + 0.587 \times G[i,j] + 0.114 \times B[i,j]$$

之后为了突出左右眼图像之中的边缘，因为边缘往往是整个图像之中变化最为明显的地方，进行左右眼后续的视差匹配必须优先突出图像之中最有特征的区域，因此标准 SGBM 算法对于灰度图像进一步使用 Sobel 算子进行处理，具体公式如下：

$$Q = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \otimes P$$

Q是经过 Sobel 处理后的图像，P是灰度图像，$\otimes$为卷积。在得到上述 Sobel 图像之后，进行图映射使得图像归一化在$[0,2\alpha]$区间，公式如下：

$$Q'[i,j] = \begin{cases} 0, & Q[i,j] < -\alpha \\ Q[i,j] + \alpha, & -\alpha \le Q[i,j] \le \alpha \\ 2\alpha, & Q[i,j] > \alpha \end{cases}$$

3.2. SGBM 算法预处理改进（Census 变换）

事实上从 3.1 的论述之中可以发现，这样的算法并不适合进行 C++代码复现、也更不适合进行 FPGA 实现，因为其中环节复杂、控制量众多。为了对于代码进行简化，我找到了 Census 算法能够对于 SGBM 的预处理达到更好的效果，因此最终采用 Census 算法进行实现。

Census 算法的基本原理是：对于每一个像素，查找其周围$N \times M$范围内的各个像素，是否比该像素灰度值更大。周围$N \times M$范围内的各个像素按照其位置 S 形顺序分别对应于输出结果的各个二进制位。如果某像素

小于中心像素，则对应位置置为 1，否则置为 0。

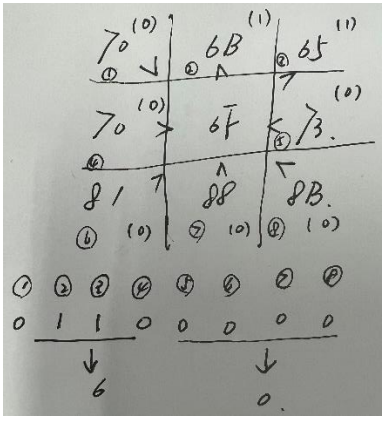


图 3：SGBM 算法预处理 Census 算法原理说明

如上图 3 绘制的是对于$N = M = 3$的情况，计算中心像素的 Census 输出值为 60_{16}。

在 Ubuntu-C++仿真代码之中，实现 Census 算法的代码在 awesome-sgbm/src/sgm_util.cpp 文件的 census_transform_3x3/5x5/9x7 函数之中。三个函数分别对应于N和M分别取三种值的情况。在 FPGA 之中我实现的 Census 尺寸是$N = M = 3$的情况。

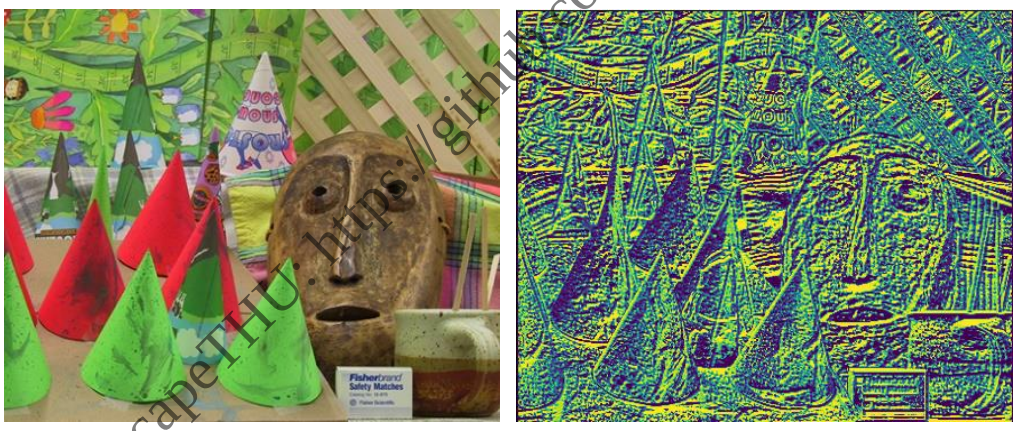


图 4：Ubuntu-C++程序进行 census 变换的可视化结果

上图 4 展示的是使用一张图片进行 census 变换的效果可视化结果，由本人编写的 Ubuntu-C++程序完成。可以看到右图的变换结果之中，所有的边缘地区十分突出，充分具备特征。

3.3. SGBM 算法原始代价计算

原始代价的计算本质上就是生成一个三维的“像素-视差”立方阵。我以左眼为“主视眼”，原始代价的计算主要就是将左眼之中的某一个像素，对应到右眼之中的各个视差的代价求出。具体地说，就是对于左眼像素$Q_l[i,j]$，分别与右眼之中的$\{Q_r[i,j-d_{max}], Q_r[i,j-d_{max}+1], Q_r[i,j-d_{max}+2], \dots, Q_r[i,j-d_{min}]\}$这$d_{max}-d_{min}$个像素分别求取汉明距离，作为原始代价。因此原始代价是一个三维矩阵，表达式如下：

$$L_r[i,j,d] = Hamming\{Q_l[i,j], Q_r[i,j-d]\} \in \mathcal{R}^{H\times W\times(d_{max}-d_{min})}$$

在上面的论述之中，d_{max}是我自己设置的最大深度、d_{min}是最小深度，Q_l是左眼的 census 变换后的结果，Q_r是右眼的 census 变换后的结果，$L_r[i,j,d]$就是最终计算出的原始代价。

在本次项目之中，我设置$d_{max}=128$，$d_{min}=20$，因此$L_r[i,j,d]$的大小就是$H\times W\times 108$。设置$d_{max}=128$，$d_{min}=20$这样的视差是足够本次项目的使用的，因为本次项目使用的双目相机双目距离为 12cm，设置上述视差意味着可以看到的距离为：

$$z=\frac{f\cdot b}{d}\in[0.9375m, 6m]$$

其中$f=1000$为焦距、$b=0.12m$为双目距离基线、d为视差。

本部分的代码在 awesome-sgbm/src/SemiGlobalMatching.cpp 文件的 ComputeCost 函数之中。由于本部分生成的是三维矩阵，难以可视化展示，这里放上一张存入 txt 文件的原始代价如下图 5。

图 5：Ubuntu-C++程序进行原始代价计算结果

上图之中，每一行都包括 216 个十六进制数字，这是因为每一行对应于一个像素位置的 108 个深度上的代价，每一个深度上的代价都是一个 8 位二进制数值（hamming 距离的计算结果是 8 位）。因此一共就是 216 个十六进制数字。

3.4. SGBM 算法代价聚合计算

代价聚合部分的基本目的是对于原始代价进行处理，从而消除视差突变处产生的错误匹配问题，进而将错误传达到后面的像素之中。由于我实现的 SGBM 算法本质上是需要利用水平方向上的信息来消除错误的干扰（本质上左右眼水平排列的时候图像只在左右方向上有差异），因此匹配也主要针对左右方向上进行。

SGBM 算法的代价聚合本质上是按照下式进行计算：

$$L_a[i,j,d] = L_r[i,j,d] + \min\begin{Bmatrix} l1: & L_a[i,j-1,d] \\ l2: & L_a[i,j-1,d-1]+P_1 \\ l3: & L_a[i,j-1,d+1]+P_1 \\ l4: & \min\limits_{\delta} L_a[i,j-1,\delta]+P_2 \end{Bmatrix} - \min\limits_{\delta} L_a[i,j-1,\delta]$$

从上式之中可以看到，在计算$[i,j,d]$位置的$L_a[i,j,d]$时会使用到$L_a[i,j-1,d]$，也就是路径上一个像素的计算结果；此外，还需要使用$\min\limits_{\delta} L_a[i,j-1,\delta]$，也就是路径上一个像素位置处所有视差之中，代价最小的数值。输出结果$L_a[i,j,d] \in \mathcal{R}^{H\times W\times(d_{max}-d_{min})}$，与原始代价一致。

本部分的代码在 awesome-sgbm/src/sgm_util.cpp 文件的 CostAggregateLeftRight 函数之中，事实上我在 Ubuntu-C++还实现了在上下方向、主对角线方向和词次角线方向上的代价聚合，分别对应于该文件之中剩余有 CostAggregate 的函数。下图 6 展示了存入 txt 文件的聚合代价图片。

图 6：Ubuntu-C++程序进行代价聚合计算结果

上图之中，每一行都包括 216 个十六进制数字，这是因为每一行对应于一个像素位置的 108 个深度上的代价，每一个深度上的代价都是一个 8 位二进制数值（代价聚合的计算结果同样是 8 位）。因此一共就是 216 个十六进制数字。

3.5. SGBM 算法后处理视差计算

上面经过原始代价计算和代价聚合，我们求出了左眼之中每个像素、每个深度上的匹配代价，接下来我们要根据这个匹配代价选择左眼之中各

个像素处最好的视差。

后处理过程就是使用各个视差处的代价，找到最优视差。其基本上由三个过程组成：首先是找出代价最小值、以及代价最小值对应的视差；第二是找到代价次小值、以及代价次小值对应的视差；第三是根据代价最小值$L_a^{(\min)}[i,j,d^{(min)}]$、次小值$L_a^{(secmin)}[i,j,d^{(secmin)}]$、代价最小值对应视差$d^{(min)}$、代价次小值对应视差$d^{(secmin)}$来计算最佳视差（以及判定无效视差）。

前两步都是十分直观的，不再赘述。最后一步寻找最优视差分为两个子步骤：视差有效性判定、以及二次插值超分辨。

视差有效性判定是根据下式进行的判定：

$$\frac{L_a^{(\min)}[i,j,d^{(min)}] - L_a^{(secmin)}[i,j,d^{(secmin)}]}{L_a^{(\min)}[i,j,d^{(min)}]} < 1-\eta\%$$

也就是代价最小值与代价次小值的相对误差应当小于$1-\eta\%$阈值，否则这个点将会被淘汰，视差设置为“无效视差”。这样的作法是为了防止出现剧烈变化的灰度的情况导致深度估计出错。

如下示意图 7，二次超分辨则是指根据最佳深度周围的代价和最佳深度的代价进行视差超分辨估计，从而获得小数视差。

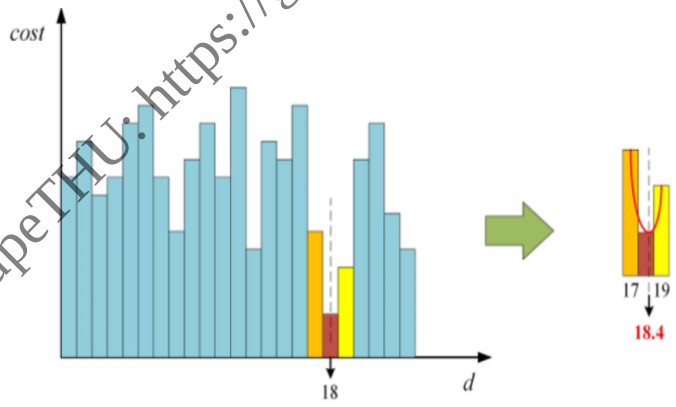


图 7：视差二次插值超分辨算法示意图

于是公式如下：

$$d[i,j] = d^{(min)} + \frac{L_a[i,j,d^{(min)}-1] - L_a[i,j,d^{(min)}+1]}{2\cdot denom[i,j]}$$

$$denom[i,j] = \frac{1}{16}\max\{L_a[i,j,d^{(min)}-1] + L_a[i,j,d^{(min)}+1] - 2\cdot L_a^{(\min)}[i,j,d^{(min)}], 1\}$$

本部分的代码在 awesome-sgbm/src/SemiGlobalMatching.cpp 文件的 ComputeDisparity 函数之中。下图 8 展示的是计算出的各个像素位置的视差图。

图 8：Ubuntu-C++程序进行后处理视差计算的可视化结果

从上图之中已经可以明显看出“远近效应”了，上图之中纯黑的部分是失效视差，说明视差有效性判定过滤掉了边缘较多的点。

3.6. SGBM 算法视差-点云转换

由双目计算模型图 9 所示，是双目立体相机模型，我们给出下式计算像素之中$[i,j]$点在真实世界之中沿着相机的 z 轴坐标为：

$$z[i,j] = \frac{f \cdot b}{d[i,j]}$$

实际上结合内参矩阵我们就可以进一步求出x和y坐标，具体是：

$$x[i,j] = \frac{(i - c_x)}{f_x \cdot d[i,j]}$$

$$y[i,j] = \frac{(i - c_y)}{f_y \cdot d[i,j]}$$

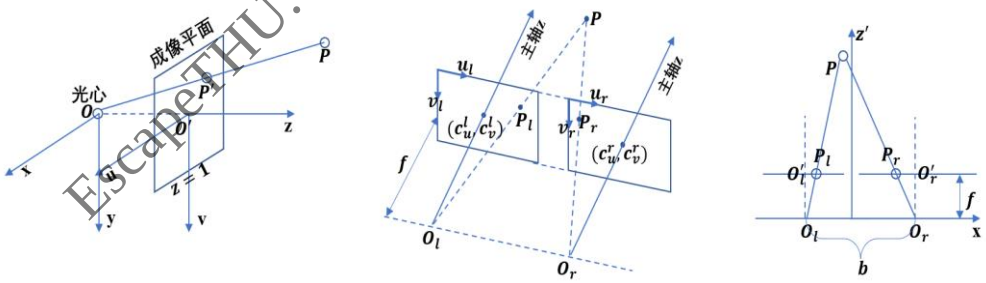


图 9：基于双目视差计算点云

根据上面的三个公式，实际上我们就能够把图像之中的每一个点转化为点云了。如下图所示是我们转化的点云效果：

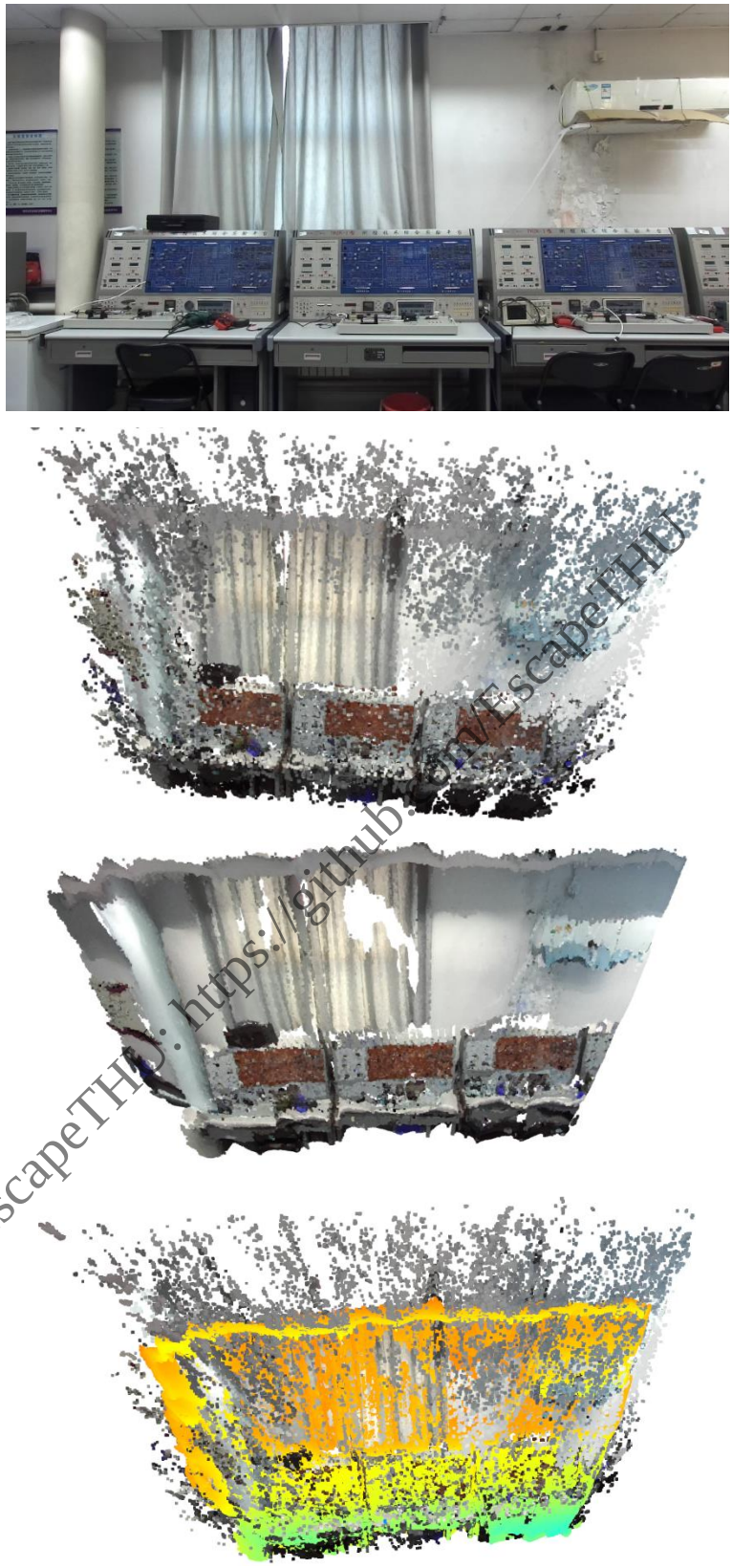


图 10：第一幅图是原图（实验室场景）；第二幅图是 Ubuntu-C++的 SGBM 结合 python 绘制的点云；第三幅图是实验室基线深度学习算法给出的点云；第四幅图是将第三幅图变为黄色点与第二幅图叠在一起进行的比较

从图 10 比较的结果来看，除了个别杂散点之外，大部分的点是与深度学习标准值一致的，说明我的仿真是准确的。

Python 绘制点云的代码在 generate_ptcld.py 之中，需要安装 open3d 点云库、opencv-python 图像库以及 numpy-python 数值计算库。

4. 系统设计综述

从第 4 部分（本部分）开始一直到第 11 部分都是在详细介绍 FPGA 端系统设计工作。首先本部分介绍系统设计的工作，主要是我自主设计的“PL-PS-上位机”系统架构。后面的 5~9 部分是我完成的 PL 端设计、编写、仿真、调试工作；10 是 PS 端设计、编写、调试工作；11 是 PS-PL 联调工作；12 是我设计但是没有完成的 FPGA 高度并行化图像处理系统。

本部分介绍我自主设计的“PL-PS-上位机”系统架构。系统架构图如下图 11 所示。

【PL 部分介绍】

首先是 PL 部分。PL 部分之中，首先由我自主编写的 BRAM 读写控制单元来控制依次读取存储在 BRAM 之中的左右眼灰度图；左右眼灰度像素接着分别被送入左眼 census 变换模块以及右眼 census 变换模块，进行 census 计

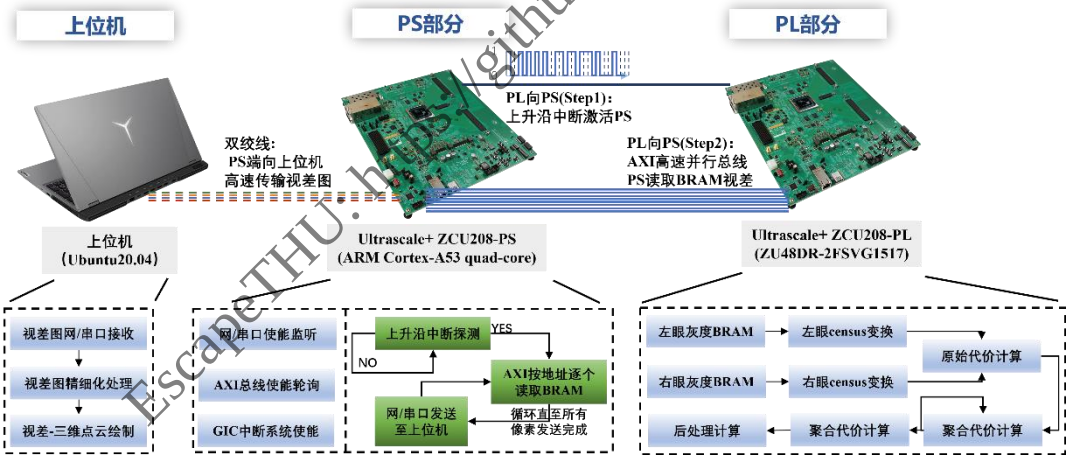


图 11：PL-PS-上位机系统架构

算；左右眼 census 变换结果同步送给原始代价计算模块计算原始代价；原始代价的结果紧接着被送入代价聚合模块计算代价聚合，这里需要注意代价聚合上具有一个“回环”，在后面的“7.第二版聚合代价计算代码编写、仿真、调试与改进”部分将会进行更加详细的介绍；聚合代价被接着送入后处理计算，求取最优视差。

上述的 PL 求取过程完全使用流水线设计，各个环节都能够形成节拍数据流，没有时钟资源浪费。除了聚合代价模块理论上无法实现 ii=1 的流水之外，其他模块的时钟利用率 ii 都是 100%，也就是理论上都能够每一个时钟

周期保证一个有效数据输出。（但是由于代价聚合的瓶颈，导致最终是每 13 个时钟周期输出一次有效数据）

PL 求取最优视差之后，需要将视差进行输出。由于 PL 的时钟预设为 100MHz，而 PL 模块之中，每 13 个时钟周期产生一个有效输出，因此在最快情况下约每$0.13\mu s$产生一次有效数据（我在最终实验时采用的时钟频率为 25MHz，因此每$0.52\mu s$产生一次有效数据）。这样的速率是 PS 直接读取难以承受的，PS 的 AXI 总线约数个μs，难以企及。因此必须在 PL 端放置存储器件，从而进行传输速率匹配。

我最终选择使用 BRAM 作为 PL 和 PS 之间的传输缓冲，选择的原因主要是 BRAM 十分适合与处理这种存储需求较大，同时对于时钟具有较高要求的任务；事实上 BRAM 在 FPGA 之中被用在 ILA 分析仪之中进行数据存储，因此对于本任务是十分合适的。

因此在 PL 端的“后处理计算”模块计算得到最优视差之后，将使用我自主编写的 BRAM 读写模块写入一个较大的 BRAM 之中，在完成一次完整的左右眼全部像素的视差计算之后，我编写的 BRAM 模块会产生一个“中断脉冲”，这个中断脉冲将会被送入 PS 的中断处理单元 GIC 进行读取。

PL 端有中断、AXI 总线两个接口与 PS 相接。PS 端的操作在下文介绍。

【PS 部分介绍】

在上一段之中介绍到，PL 端在完成一帧图像的所有 PL 流水线计算之后使用 BRAM 缓冲、由 AXI 总线和中断线与 PS 连接。

PS 在进入主程序之后，首先进行 AXI 总线使能、网口使能、串口使能以及最后进行中断使能；接着，PS 端始终等待来自 PL 端固定地址的中断上升沿，探测到这个中断上升沿说明 PL 端已经完成了一帧的完整计算、并且将视差结果已经稳定存入了 BRAM 之中；此时 PS 端会使用 AXI 总线按照 BRAM 的地址次序读出图像，每读出一个像素，就将该像素的视差值使用网口和串口转发至上位机。

在本次项目之中保留了串口回传的方式，但是事实上在实际应用之中应当抛弃串口回传。串口回传本质上仅仅是为了展示，因为串口回传效率太低，无法跟上 PL 端下一帧的计算效率。也就是：等待串口回传会耽误 PL 下一帧的接收和转发。

本次项目之中，我们最终能够保证：针对 200*400 大小的左眼和右眼灰度原始图像，在 0.042s 之内能够完成整张图像（200*400*32bit）的视差计算，因此必须保证 0.042 秒之内将 BRAM 数据发送出去。对于发送速度为 100MHz 的网口，这是轻松的；但是对于发送速率仅 115200Hz 的串口，这个转发速度显然不够。

【上位机部分介绍】

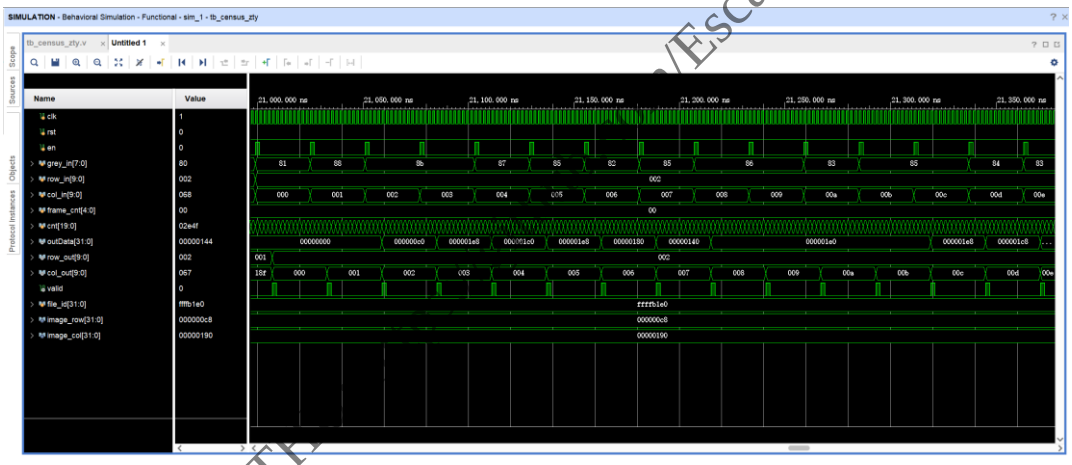
在获得 PS 通过网口发送上来的视差图之后，上位机使用 python 的工具进行视差图优化、以及点云图的绘制。本部分本质上并不属于嵌入式设计，因此不再赘述，基本原理请参考上文“3.6 视差-点云转换”。

接下来将分别对于 PL、PS 的各个部分进行详细介绍。

5. census 变换仿真、调试与改进

Census 代码的调试结果在下图之中进行展示，下图展示的是 census 模块进行的 vivado 仿真效果。以及对应的保存下来的文件进行对比。可以看到，census 得到的结果是与 Ubuntu-C++的结果完全一致的。

仿真和调试的代码在本文件夹之中的 project_census 项目之中，其中的 census.v 文件是计算 census 的模块，仿真文件顶层是 tb_census_zty.v 为仿真文件。直接运行 simulation 可以得到结果，产生的文件在 image_data 的 mem2image.txt 之中。



mem2image.txt

```
00000000
000000c0
000001e8
000001c0
000001e8
00000180
00000140
000001e0
000001e0
000001e0
000001e0
000001e8
000001c8
000001c0
000001e0
000001e0
000001e8
000001c8
000001c0
000001e1
000001eb
000001cf
000000c7
000001e7
000001cf
00000187
000001e7
000001ef
0000014e
000000c4
000001e8
00000100
00000068
```

census_left_X.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
00000000
000000C0
000001E8
000001C0
000001E8
00000180
00000140
000001E0
000001E0
000001E0
000001E0
000001E8
000001C8
000001C0
000001E0
000001E0
000001E8
000001C8
000001C0
000001E1
000001EB
000001CF
000000C7
000001E7
000001CF
00000187
000001E7
000001EF
0000014E
000000C4
000001E8
00000100
00000068
```


图 12：census 仿真结果与写入文件后和 Ubuntu-C++产生结果的对比（左侧文本文档为

verilog 写入后产生的文档、右侧为 Ubuntu-C++仿真的结果，均以右目为例）

census 直接编写出来的代码本质上距离实际形成系统还是有一定的距离的，本质上 census 采用的算法是连续存储三行的数据，每一次输入一个像素时，自动将3×3的计算区域向后推移一格；这样的计算方法要求对于输入的灰度数据连续三行存储，过程之中输出顺序需要进行仔细的设计。

我对于代码的调试与改进是将代码的输出次序和有效值进行了精细化设计，在灰度原始值开始输入第二行时，此时输出开始变得有效，输出的 census 值是第一行的 census 值，应当全部置为 0；在灰度原始值开始输入第三行时，输出值开始变为 census 按照公式计算出来的结果。这样的代码控制流程才能使得输出的 census 与 Ubuntu-C++的仿真结果互相对应。

除此之外，我为 census 代码增添了 en 数据有效端口输入、以及对应的 valid 数据有效端口输出，从而使得前面的模块数据稳定有效的时候才被 census 计算模块读入、并且 census 模块只有输出值稳定的时候输出 valid 脉冲。

我的改进除了这些之外，还有对于 census 全并行计算设计的构想，这一部分将在“12.FPGA 的高度并行化图像处理展望”之中进行介绍。

6. 原始代价计算代码编写、仿真、调试与改进

【原始代价计算代码编写】

我采用“打拍子”的方式对于代码进行编写，从而保证流水线的设计。按照拍子的顺序，原始代价编写主要分为三个拍子，下面分别进行介绍。

第一拍：

如果输入端口上的 en 端口处于高电平，则读入输入端口 census_left 和 census_right 上的上一步计算出来的左眼 census 有效值和右眼 census 有效值、并且使用寄存器存储。

这里需要注意，我在这里设计了右眼长度为 128 个单元的移位寄存器；而左眼仅仅使用单个寄存器。这里这样设计的主要原因是算法以左眼为主视眼，左眼当前 census 值为$Q_l[i,j]$、右眼当前 census 值为$Q_r[i,j]$，那么按照之前的理论“3.3. SGBM 算法原始代价计算”部分的论述，左眼需要分别进行如下的计算：

$$OriginCost[i,j,d] = Hamming\{Q_l[i,j],\ Q_r[i,j-d]\},\ d \in [d_{min}, d_{max}]$$

也即是说，对于新输入的左眼，右眼需要存储从d_{max}时钟之前开始的 census 计算值。所以在第一拍之中我为右眼设计了长度为$d_{max} = 128$的移位寄存器，对于左眼则仅仅是一个寄存器。

第二拍：

使用 for 生成器语法，左眼$Q_l[i,j]$分别与$Q_r[i,j-d]$进行异或计算。之所以此处使用的是 for 生成器语法，主要原因就是“左眼$Q_l[i,j]$分别与$Q_r[i,j-d]$进

行异或计算”对于各个 d 的取值是并行的。使用生成器能够并行计算。

第三拍：

Hamming 距离在异或计算之后，需要统计异或结果各个位上 1 的数量。这个过程在 C++语言之中往往是使用循环的思路完成的，但是循环并不是硬件思维。这里仍然是使用 for 生成器，直接将各个位加起来，就是 1 的数量。这是很直观的并行，因为各个计算结果之间互不相干。

以上就是三拍的计算，除了三拍的计算之外，我们同时对于输入的 row_in 行号和 col_in 列号以及 en 使能信号按照 3 拍进行了延迟、并输出出去。

【原始代价计算仿真编写】

仿真的编写我直接将 census 变换的左眼和右眼的结果作为原始代价的输入，输出写入一个文档之中便于进行核对。

【原始代价计算调试与改进】

原始代价计算的代码调试结果如下图示，以及对应的保存下来的文件进行对比。可以看到，原始代价计算 verilog 仿真得到的结果是与 Ubuntu-C++的结果完全一致的。

仿真和调试的代码在本文件夹之中的 origin_cost 项目之中，其中的 origin_cost.v 文件是计算原始代价的模块，仿真文件顶层是 tb_origin_cost.v 为仿真文件。直接运行 simulation 可以得到结果，产生的文件在 image_data 的 mem2image.txt 之中。

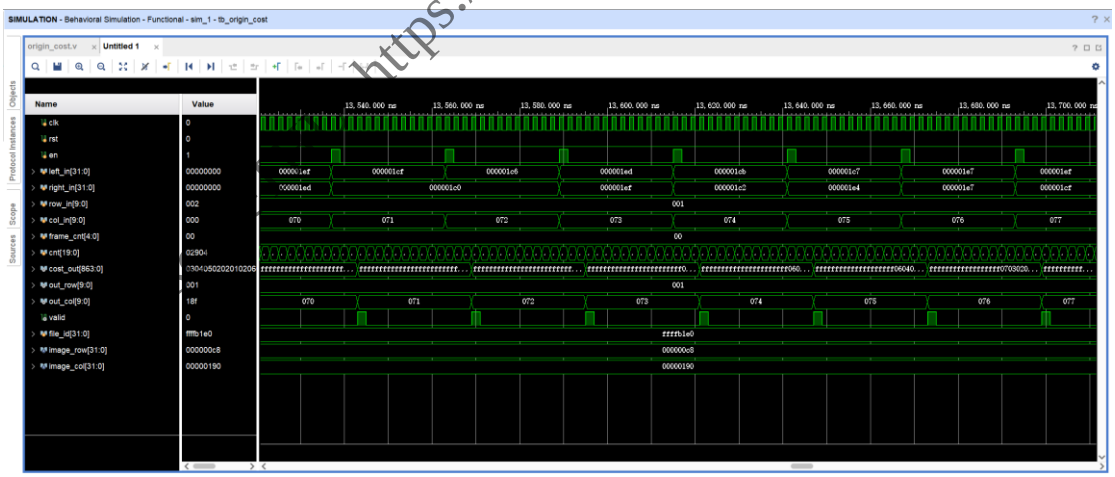


图 13：原始代价仿真结果与写入文件后和 Ubuntu-C++产生结果的对比（左侧文本文档为 Ubuntu-C++仿真的结果、右侧文本文档为 verilog 仿真生成的文档）

7. 第二版聚合代价计算代码编写、仿真、调试与改进

我还是采用“打拍子”的思想进行代码编写，从而保证流水线的效率。

【第二版聚合代价计算代码编写 1——代价聚合求取】

功能介绍：

完成代价聚合求取的主要计算。

输入输出介绍：

输入 1：原始代价$OriginCost[i,j,:]$，就是每一个像素处所有的视差对应的原始代价。输入的方式是每一个时刻输入一个 864 位的 origin_cost，864 位的原因是每个原始代价是 8 位数字、视差的范围是[20,128]，因此对每一个像素位置一共是$8\times(128-20)=864\ bit$。

输入 2：路径上一个位置的聚合代价$AggrCost[i,j-1,:]$，就是上一个路径点的所有聚合代价。输入的方式同样是一个 864 位的 aggr_last_cost。

输入 3：路径上一个位置的聚合代价的最小值$MinAggr[i,j-1]$，就是上一个路径点的最小聚合代价。输入的方式是一个 8 位的 min_aggr_last。

输入 4~6：当前输入点的 i、j 以及使能信号 en。

输出 1：当前像素位置的聚合代价$AggrCost[i,j,:]$，就是当前路径点的所有聚合代价。输出的方式是一个 864 位的 cost_aggr。

输出 2~4：当前像素位置的 i、j 以及使能信号 valid。

代码计算逻辑拍子介绍：

第一拍：

将输入的 row_in、col_in、en、origin_cost、aggr_last_cost、min_aggr_last 保存在寄存器之中。

第二拍：

使用第一拍保存的数据，分别按照上文理论部分“3.4. SGBM 算法代价聚合计算”计算$l1$、$l2$、$l3$、$l4$的值。这里需要注意的是，由于$l1$、$l2$、$l3$、$l4$本质上都是 8 位寄存器、第一拍的 origin_cost、aggr_last_cost、min_aggr_last 也都是 8 位寄存器，同时$l1$、$l2$、$l3$、$l4$计算过程之中涉及加法操作，一定要对于溢出的情况进行判断和防止。

第三拍：

求出$l1$、$l2$、$l3$、$l4$之中的最小值的第一步。分别将$l1$、$l2$和$l3$、$l4$各自比较、求出最小值。也就是说这一拍子的输出是$\min\{l1, l2\}$和$\min\{l3, l4\}$。

第四拍：

求出$l1$、$l2$、$l3$、$l4$之中的最小值的第二步。这一步的输出是得到的最小值也就是$\min\{\min\{l1, l2\}, \min\{l3, l4\}\}$，将上一拍子的两个输出求取最小值。

第五拍：

计算和输出本位置的聚合代价结果：

$$AggrCost[i,j,:] = OriginCost[i,j,:] + \min\{l1, l2, l3, l4\} - MinAggr[i,j-1]$$

其他的 row_out、col_out 以及 valid 都是 row_in、col_in、en 的 5 拍延时。

【第二版聚合代价计算代码编写 2——最小聚合代价值求取】

功能介绍：

以本位置的聚合代价$AggrCost[i,j,:]$作为输入，求出本位置的最小聚合代价$MinAggr[i,j]$。本模块输出的最小聚合代价将会被接入到上一个模块（代价聚合求取模块）的“上一位置最小聚合代价”输入端口（即输入端口 3）。

输入输出介绍：

输入 1：聚合代价$AggrCost[i,j,:]$，这是一个 864 位的数值。

输出 1：最小聚合代价$MinAggr[i,j]$，这是一个 8 位的数值，代表上面说得输入 864 位之中大小最小的 8 位。

代码计算逻辑拍子介绍：

我在编写代码的时候使用了“最小比较树”这一经典算法。这个经典算法的流程是：每一次都是两两比较，经过多轮的两两比较最终能够得到最小值。如下图所示展示了最小比较树的计算流程，图中以 8 个数值为例，十分类似于“世界杯”淘汰赛环节的对战。

按照图中的最小比较树计算逻辑，我们可以知道如果是 108 个 8 位数字进行比较，则一共需要比较 8 轮输出结果。事实上我们的代码里正是进行了 8 轮比较。具体的每一轮就是两两相比，就不再赘述了。

这样的算法好处在于：算法的时间利用率是 100%，每一个时钟周期本质上都能保证一个有效数字的输出。流水线效率是最高的。除此之外，这是一个

采用“分治”思想的算法，两两比较，这样的方法其实被应用在除了比大小之外的很多领域之中。并且除了求取最小值，这样的思想也可以被用于后面“后处理视差计算模块”之中求取“次小值”的计算之中！

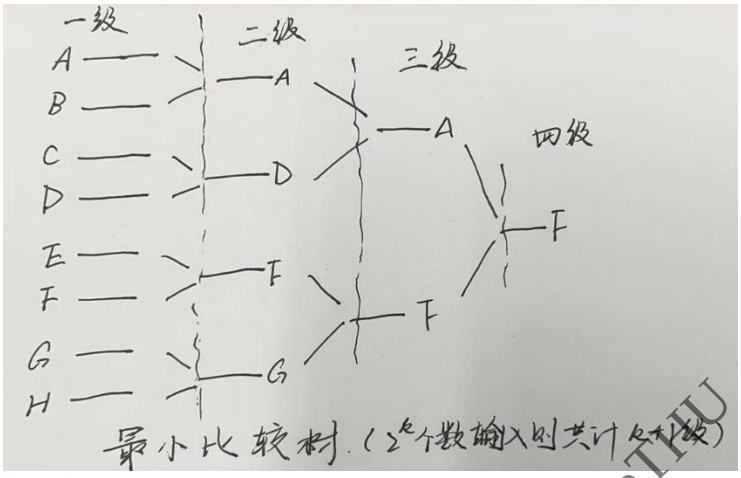


图 14：最小比较树算法示意图

【第二版聚合代价计算代码编写 3——延迟模块】

由于聚合代价的直接输出其实还需要接回到输入端，作为上一个像素点的聚合代价输入；同时聚合代价的输出还需要接入到上面的最小值模块，求出的最小值接回输入端，作为上一个像素点的最小代价输入。因此我们必须保证这两个接回输入端的值是同步的，并且都与下一个输入的原始代价同步。因此我们必须在输出端的聚合代价上另分出一路、连接一个与最小值模块消耗同样节拍数的“延迟模块”。

延迟模块的编写本质上就是移位寄存器，这里就不再赘述。

【第二版聚合代价计算代码编写 4——代价聚合整合】

将上述的三个部分代码整合起来，就得到了代价聚合的整合完整模块。我们绘制模块的框图如下所示：

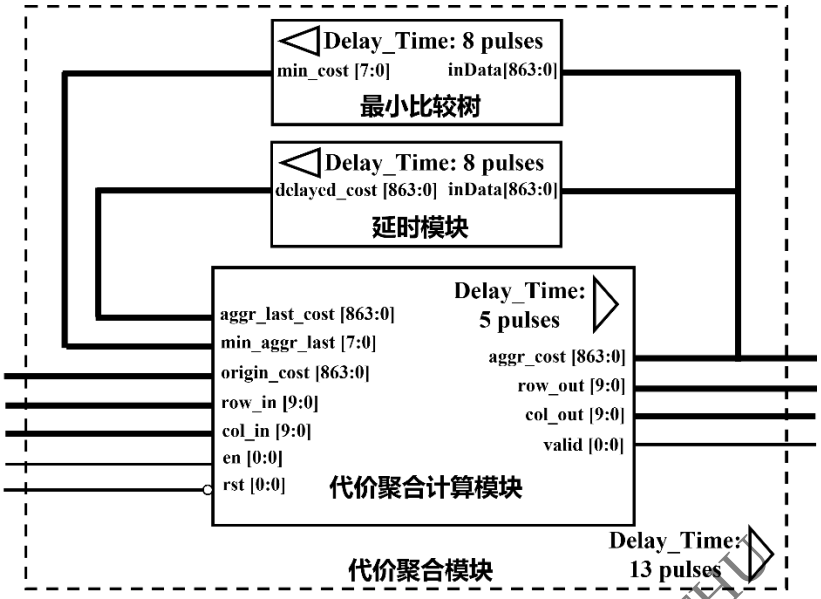


图 15：代价聚合模块架构

从整体上来看，代价聚合模块从本像素原始代价输入，直到本像素的聚合代价输出，共计需要 5+8=13 个周期，这也正是本次项目每 13 个周期输出一次有效值的来源。这个周期不能再缩短，本质原因是：本模块的时间利用率并不取决于硬件算法设计，而是软件算法需要使用上一个时刻的计算结果决定的，也就是说，此时的时间利用率取决于模块的**计算时延。**如果希望能够更快的计算出结果、提升时间利用率，那么最可靠的办法是缩减最小比较树求取最小值的时延。但是按照求取最小值的理论时延最小值（即数值分析之中的算法复杂度），最小比较树已经是最优算法，因此从理论上，这就已经是聚合代价的最高效率、而且无法进一步优化。

【第二版聚合代价计算仿真编写】

仿真部分的编写，我直接将文档之中标准原始代价作为输入，但是这个输入每隔 13 个时钟周期输入一次，当且仅当输出端的 valid 数值为 1 时写入此时的聚合代价输出值到文档之中进行比对。

【第二版聚合代价计算调试与改进】

聚合代价计算的代码调试结果如下图示，以及对应的保存下来的文件进行对比。可以看到，聚合代价计算 verilog 仿真得到的结果是与 Ubuntu-C++的结果完全一致的。

仿真和调试的代码在本文件夹之中的 aggregate_cost 项目之中，其中的 aggregate_cost.v/delay_aggr.v/min_aggr_cost.v/aggregate/v 文件是计算聚合代价的模块，仿真文件顶层是 tb_aggr_cost.v 为仿真文件。直接运行 simulation 可以得到结果，产生的文件在 image_data 的 mem2image.txt 之中。

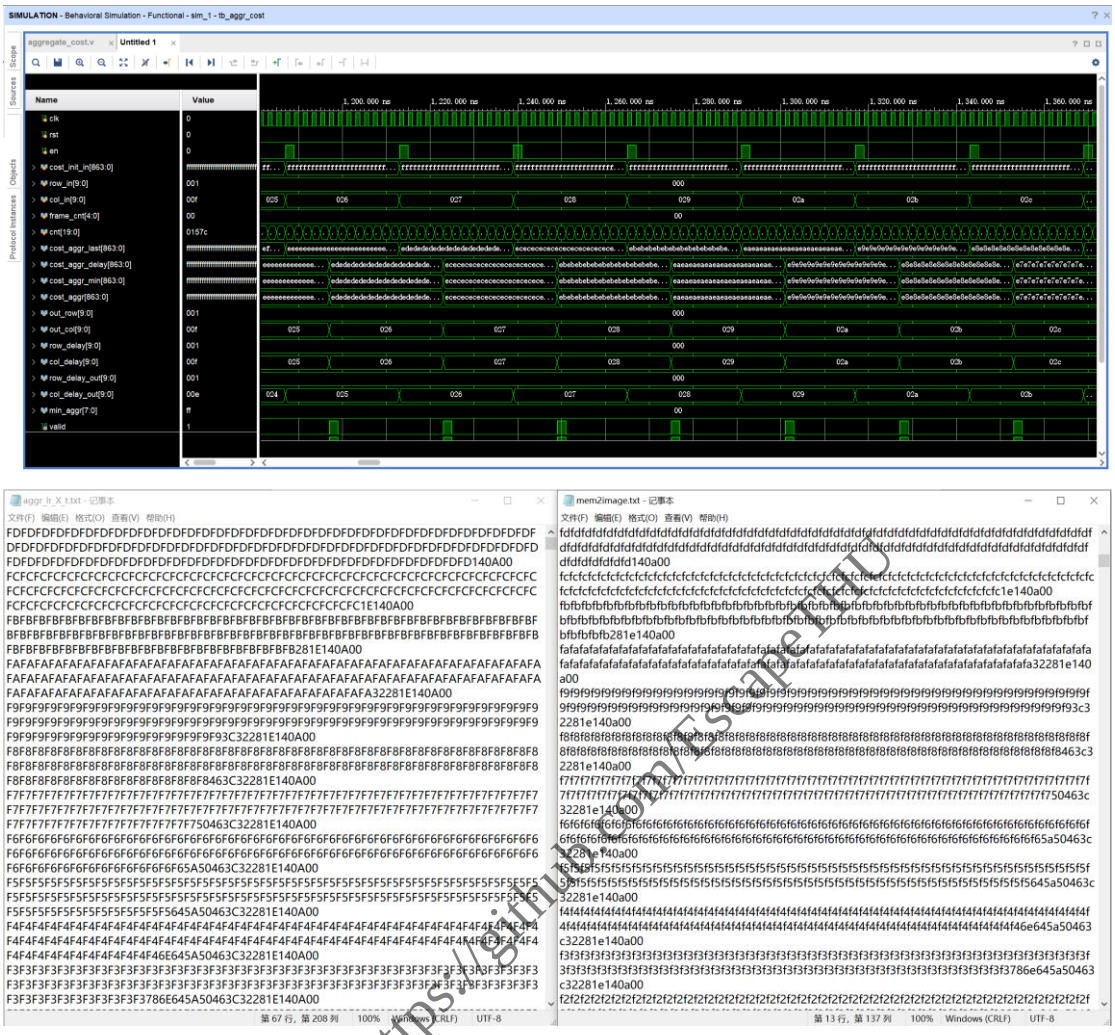


图 16：聚合代价仿真结果与写入文件后和 Ubuntu-C++产生结果的对比（左侧文本文档为 Ubuntu-C++仿真的结果、右侧文本文档为 verilog 仿真生成的文档）

8. 后处理代码编写、仿真、调试与改进

【后处理代码编写】

我采用“打拍子”的方式对于代码进行编写，从而保证流水线的设计。按照拍子的顺序，后处理代码编写主要分为四个阶段共计 19 个拍子，下面按照阶段分别进行介绍。

第一阶段：

首先是求取聚合代价之中各个视差对应代价最小值、以及最小代价位于的视差位置。本质上就是从上一个步骤输出的 864 位$AggrCost[i,j,:]$之中，找到数值大小最小的 8 位数字$AggrCost\left[i,j,d^{(\min)}\right]$、以及这最小的 8 位数字在$AggrCost[i,j,:]$之中的位置$d^{(\min)}$。

不难想到，采用的方法就是最小比较树的方法。为了同时给出位置，我在最小比较树的代码之中增添了一个始终不变的储存位置的寄存器，每一级之中

更新出较小者的坐标，最终就能够输出最小值的坐标。

第二阶段：

将$AggrCost\left[i,j,d^{(\min)}\right]$的数值改为 FF，也就是使得聚合代价之中的最小值不再影响寻找最小值。这是为了第三阶段求取次小值做准备。

第三阶段：

将第二阶段更改之后的$AggrCost[i,j,:]$再一次输入最小比较树，找到次小代价的数值$AggrCost\left[i,j,d^{(\text{secmin})}\right]$和位置$d^{(\text{secmin})}$。

第四阶段：

首先判定下式的正确性：

$$\frac{L_a^{(\min)}\left[i,j,d^{(min)}\right]-L_a^{(secmin)}[i,j,d^{(secmin)}]}{L_a^{(\min)}[i,j,d^{(min)}]}<1-\eta\%$$

这个式子左侧实际上使用了除法，原则上应当是一个浮点数。但是 FPGA 之中不存在浮点数，因此我们将左右同时乘 100、再同时乘$L_a^{(\min)}[i,j,d^{(min)}]$，就可以进行整数比较。

如果上式不成立，直接将输出的视差置为 FFFF，也就是 32 位数之中的最大值、我将其设置为无效值。

如果上式成立，进行下一步运算：

$$d[i,j]=d^{(min)}+\frac{L_a\left[i,j,d^{(min)}-1\right]-L_a\left[i,j,d^{(min)}+1\right]}{2\cdot denom[i,j]}$$

这个运算本质上也是浮点运算，但是为了简化，我们简单地判断分式是否超过$[-0.5,+0.5]$的范围，如果超过，那么相应的在$d^{(min)}$上-1 或者+1。

【后处理仿真编写】

仿真部分的编写，我直接将文档之中标准聚合代价作为输入，这个输入同样是每隔 13 个时钟周期输入一次，当且仅当输出端的 valid 数值为 1 时写入此时的最佳视差输出值到文档之中进行比对。

【后处理调试与改进】

后处理的代码调试结果如下图示，以及对应的保存下来的文件进行对比。可以看到，后处理 verilog 仿真得到的结果是与 Ubuntu-C++的结果完全一致的。

仿真和调试的代码在本文件夹之中的 project_sgbm 项目之中，其中的 disparity_calc.v 文件是计算后处理的模块，仿真文件顶层是 tb_disparity_impl.v 为仿真文件。直接运行 simulation 可以得到结果，产生的

文件在 image_data 的 disparity_res_impl.txt 之中。

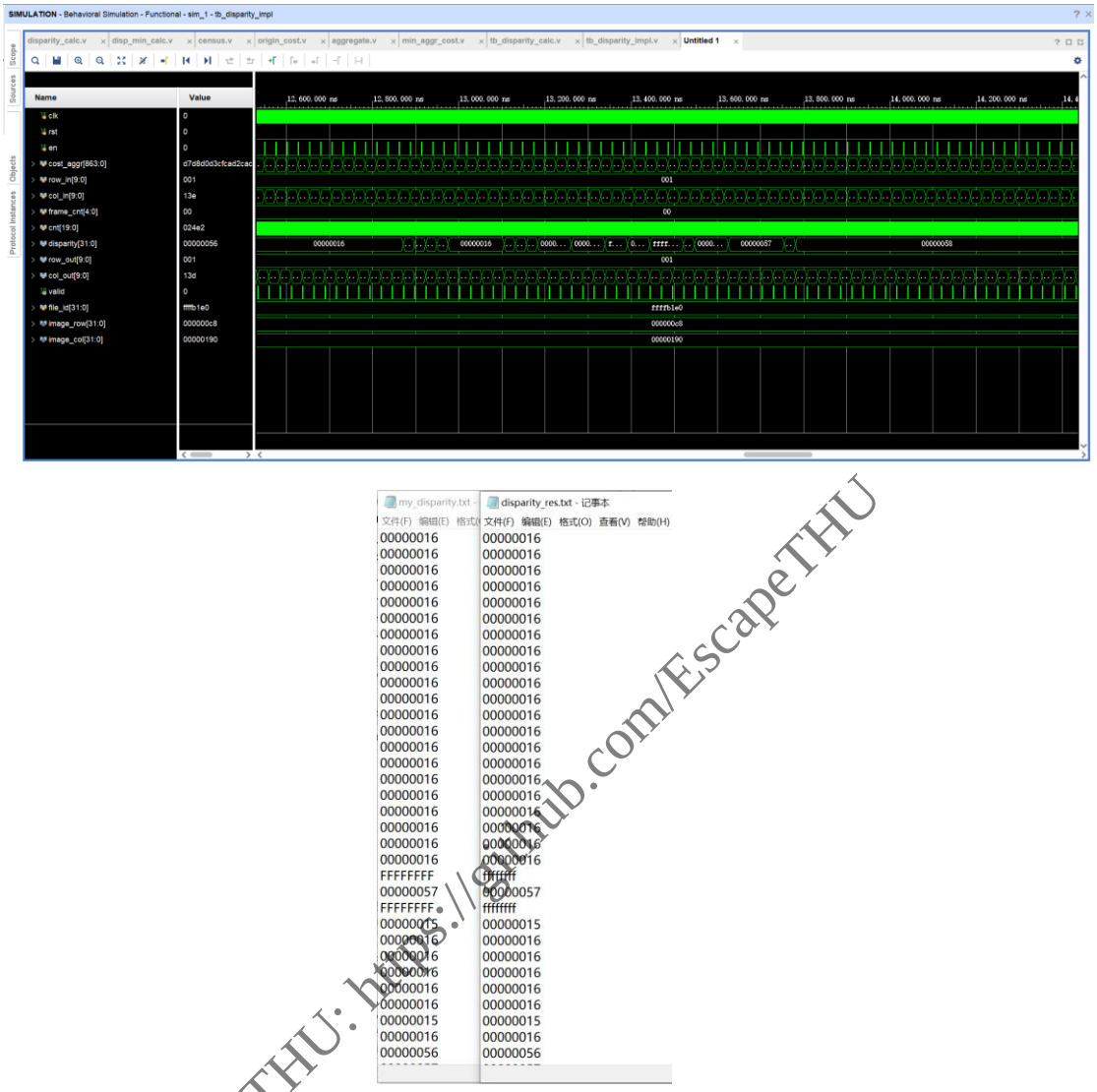


图 17：后处理仿真结果与写入文件后和 Ubuntu-C++产生结果的对比（左侧文本文档为 Ubuntu-C++仿真的结果、右侧文本文档为 verilog 仿真生成的文档）

9. PL 系统设计、编写、仿真、调试

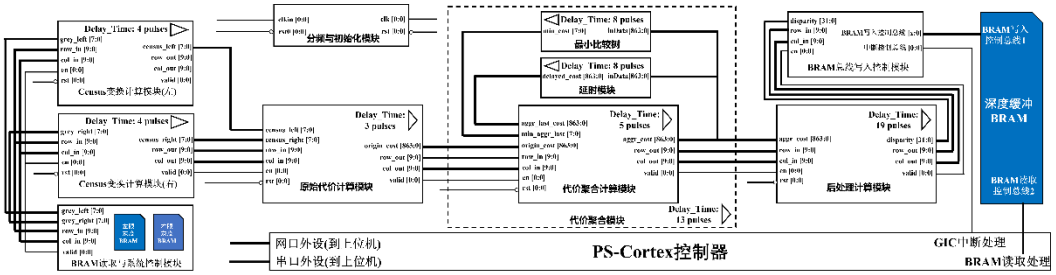


图 18：PS-PL 系统全图

根据上述的所有模块，我绘制了如上图所示的 PL 系统整体示意图。（图像可能稍小，完整版请见文件夹下的“整体架构.png”）。

可以看到，除了上面提到的“census 变换模块”、“原始代价计算模块”、“代

价聚合模块”、“后处理模块”之外，还有“BRAM 读取与系统控制模块”、“BRAM 总线写入控制模块”以及“分频与初始化模块”这三个控制类型的模块，下面分别进行介绍。

【BRAM 读取与系统控制模块】

这个模块对于左眼和右眼的灰度图像存储 BRAM 进行了初始化、并根据输入的时钟进行灰度图像灰度像素输出的控制。由于之前介绍的，第二版实际使用的代价聚合代码必须存在 13 个周期的等待时间，因此要求灰度像素必须每间隔 13 个周期才能给进去一个新的像素。在本模块之中，为了实现每 13 个时钟周期输入一次，我设计了一个计数器，根据计数器的计数，每 13 个计数产生一次有效灰度读取、并在 valid 端口上产生一个持续一个时钟周期的“使能有效脉冲”，作为后续模块输入端口数据有效的依据。这个模块在输出左眼和右眼灰度值的同时，输出本时刻像素的行号和列号，为后续模块的计算与判断提供方便。

【BRAM 总线写入控制模块】

这个模块看似是 Xilinx IP 核，实际上是我独立自主编写的。由于最终的视差 BRAM 必须要与 PS 进行交互缓冲，端口必须设置为“BRAM Controller”格式，而这个端口格式要求 PL 端写入 BRAM 的时候必须使用符合 Xilinx 规定的总线。PS 端由于具备 AXI 总线到 BRAM 总线的转换器 Xilinx IP 核（名称就是 AXI BRAM Controller），因此不必担心（就是上图之中 PS 端“BRAM 读取处理”是使用 AXI BRAM Controller 这个 IP 核实现的）。但是 PL 端不存在这样一个封装好的 IP 核，所以需要我自主设计编写一个模拟 BRAM 总线的模块。

我编写这个 BRAM 总线写入控制模块接收视差、行号、列号与有效使能信号作为输入，首先判定本像素是否是图像的最后一个，如果是最后一个，那么输出的“中断控制总线”上将会产生一个正脉冲，来给 PS 产生中断，开启 PS 对于视差 BRAM 的读取转发流程。如果不是最后一个，那么激活 BRAM 总线写入当前视差。

【分频与初始化模块】

本质上就是一个分频器，并且产生 reset 信号。这个模块比较简单，在此不再赘述。

【PL 仿真代码编写】

已经向 PL 集成了 BRAM，因此我们直接给 clkin 信号与 rst0 信号就可开始仿真。

【PL 整体调试结果】

PL 系统整体调试结果如下图示，以及对应的保存下来的文件进行对比。可

以看到，PL 整体调试 verilog 仿真得到的结果是与 Ubuntu-C++的结果完全一致的。

仿真和调试的代码在本文件夹之中的 project_sgbm 项目之中，其中的 project_sgbm.bd 文件是全部的 SGBM 算法模块顶层文件，仿真文件顶层是 tb_sgbm_after.v 为仿真文件。直接运行 simulation 可以得到结果。

图 19：SGBM 的 PL 部分全部模块仿真（与 Ubuntu-C++完全一致）

10.PS 系统设计、编写、调试

【PS 设计思想介绍】

PS 端进行设计的基本要求主要是两点：首先，快速的使用 AXI 总线与 PL 端进行交互，并且能够不漏过任何求出的像素；第二，快速的将从 PL 获得的像素转发给上位机以便上位机处理。

针对要求的第一点，由于之前介绍到 AXI 总线的读取速率为数个微秒才能得到一个有效数据，因此必须使用 BRAM 作为缓冲。那么在 PL 端设计时就相应的必须在一帧计算完成之后及时通知 PS，为了使得 PS 不忽略这个通知，我选用中断的方式通知 PS 端。

针对要求的第二点，由于 PS 需要在下一帧计算完成之前及时将上一帧图像的视差结果回传上位机，每一帧在 PL 端仅仅花费 0.042 秒即可计算完成，因此只有网口能够承担如此的数据量。

除了上述的要求之外，还有一个隐含的要求，即作为课程项目设计，必须展示出 PS 向上位机回传信息的过程，而网口并不利于展示，因此此处设计我保留了串口回传的方式，来对于回传的数据进行展示。使用串口调试助手就可以检查数据的正确性。网口的回传则直接被上位机写入一个文件之中。

【PS 中断配置】

PS 端使用 Vitis 对于中断进行配置，其关键在于找到 XSA-BSP 固件之中的中断地址。我本次 vitis 的中断地址为 xparameters.h 之中的 XPAR_SCUGIC_0_DEVICE_ID 以及 XPAR_FRABRIC_RAM_RW_0_INTR_INTR 两个地址分别代表 GIC 中断地址和 RAM_RW 中断编号。之后进行正常的中断配置即可。

【PS 网口配置】

网口配置参考了 Xilinx Vitis 的标准例程，使用了 lwip 标准 UDP 网络接口，最高传输速率为百兆级别。

【PS 串口配置】

串口直接采用 xil_printf 即可。

【PS 调试】

由于与下文的“11.PL-PS-上位机联合调试”息息相关，统一在下一小节叙述。

11.PL-PS-上位机联合调试

PL-PS-上位机的代码配置这里就不再赘述，关于联合调试的效果，本人制作了视频进行讲解，请参考本文件夹下的“SGBM 双目匹配系统展示.avi”。仿真和调试的代码在本文件夹之中的 project_image 项目之中，其中正常打开 project_image.xpr 就是 Vivado 工程，使用 Vitis 打开 vitis_image 文件夹的工程，其中就是 PS 端的全部代码，可以看到 main 函数之中注释十分详细，其中包括 AXI 总线激活、GIC 中断处理激活、lwip 网口激活等。

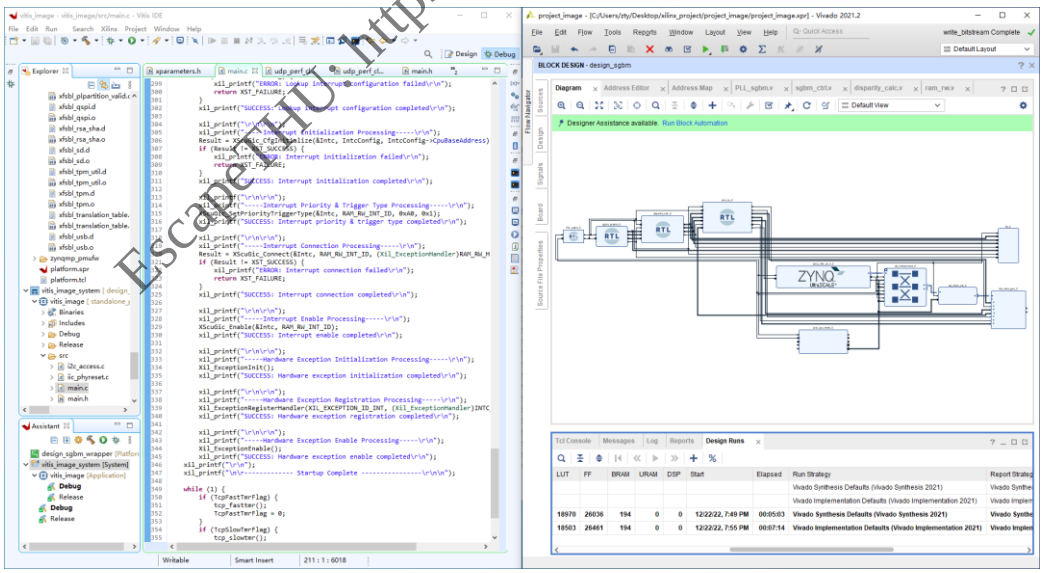


图 20：PS-PL 联合代码

12.FPGA 的高度并行化图像处理展望

在本次项目之中，经过上述描述不难发现，本组编写的模块都可以按照行的顺序实现并行化。

FPGA 对于图像的处理，其中更加高效的关键就在于按照行的顺序，不同行之间并行处理，从而对底层的代码效率进行进一步提升。例如 census 变换各行可以同时并行计算，只要输入变为三行数值同时输入，就可以计算中间一行的 census 变换结果，因此同时按照N组三行三行的输入顺序，就可以同时进行N行的并行 census 计算。原始代价计算、聚合代价计算以及最优视差的计算亦复如是！那么不难想象，在 FPGA 资源允许的情况下，这样的做法能够在本报告的基础上进一步提升！

当然，事实上我们并没有真的这样做，“非不能也，实不可为也”。对于 FPGA 和集成化电路设计，成本思想也是同样重要的。经过本文上述的分析，本文采用的硬件设计其实已经达到了单条流水线能够达到的最高效率。这样的运算效率实际上是用资源来“堆砌”出来的，例如我编写的最小比较树算法，采用了最高的并行效率，这是相当消耗 LUT 运算资源的；此外，原始代价计算、census 变换，我都选用了效率最高、同时也是最消耗资源的运算方式。综合和部署之后，LUT 资源消耗达到了 18970！这个 LUT 消耗占用 A-7 的 100%、占用 K-7 的约 25%（当然仅仅占用我使用的板子 Zynq Ultrascale+ ZCU208-xczu48dr 的 5%），对于普通的低成本 FPGA 已经是很高的计算资源消耗！

除了上面提到的 LUT 用量问题之外，本项目为了高速处理大量数据的 PS-PL 交互，使用了很多的 BRAM 资源。按照综合与部署的结果来看，共计消耗了 194 片 32Kbit 的 RAM 资源。按照 Zynq-7000 Soc 系列的配置，仅仅具备 2 片 36KB 的 RAM 资源，这个资源量远远不能满足项目的需求！（当然实际上我使用的板子具有 1080 片 32Kbit 的 RAM 资源，这样的用量仅占我使用开发板的 18%）。

我希望实现的算法是双目视觉匹配的 SGBM 算法，我的构想是希望这样的一套硬件算法能够广泛部署到所有的双目视觉设备上。如果以这个目标为导向，显然这个资源开销已经略显得有点多了，基于现有算法，使用的芯片起码是 Xilinx-K7 系列芯片，这样的成本已经在千元级别。

回到本小节开始时的讨论，在上完这样一门结构化集成电路设计课之后，我认为有一个关键点深入我的思想之中，那就是：结构化电路设计首先要面向需求、面向用户、面向人。电路设计确实是一门技术，但是怎样设计、怎样权衡、如何产品化，这些问题并不是简单的技术就能讲明白的，而是需要设计者首先具备充足的工程经验、能够对这些问题做出预判；接着设计者要具备长足的目光、能够正确分析紧要的要求并留出裕量；最后则是设计者要具备洞察的捷思和刚毅的勇气、把握机会，能够使自己的产品具备独树一帜的能力。

13.结构化集成电路课程总结

这门课程确实给我留下了很多的回忆。期中之前做仿真时从不明白 SGBM 算法、到逐渐熟练掌握其中精髓；期中之后趁着两周赶紧拉经纬和卓尔开始写硬件代码；世界杯的梅西夺冠之夜，更让我振奋的是 PL 端整机调试完成；一边顶着新冠肺炎的低烧和嗓子疼痛和浑身酸软，一边调试 PS-PL 交互。这些经历其实已经能够浓缩我的整个 2022 下半年。

我本科时遇到的老师曾经讲过一句话：“我们做学术的，要么让自己的知识摆在货架上；要么让自己的智慧留在书本里”。这句话已经是我考量技术的关键标准，经过这一门课程，我认为我对于我的技术不论是上货架、还是进书本都有了更加深刻的理解。