

矩阵乘在 CPU、GPU、FPGA 上的实现及优化实验代码及笔记

jx

2020 年 11 月 3 日

1 CPU 上的矩阵乘

CPU 矩阵乘主要用的是 c++ 在 Windows 系统上的 `<thread>` 包, 因为 CPU 上的矩阵乘只用作多线程加速的练习, 所以并没有做到完全的优化。比如这里矩阵用的是 `vector`, 在绝对的运算速度上不如数组。矩阵的初始化使用的是每一个元素的值等于它的行 + 列, 即 $A[i][j]=i+j$ 。没有使用随机生成矩阵的原因是, 本次矩阵乘任务要在 CPU, GPU, FPGA 上分别执行, 所以我觉得最好是使用相同的矩阵, 这样可以便于对比它们的运行速度。

但是在查阅矩阵乘优化的资料和练习过程中, 发现了一个有趣的问题。

假设我们封装了矩阵类, 现在要实现矩阵乘法 $F = DE$ 。为了讨论方便, 设所有矩阵都是 n 阶矩阵, 且假定矩阵是按行存储的。最普通的实现方式如下:

```
for(int i=0;i<n;++i)
    for(int j=0;j<n;++j)
        for(int k=0;k<n;++k)
            F[i][j]+=D[i][k]*E[k][j];
```

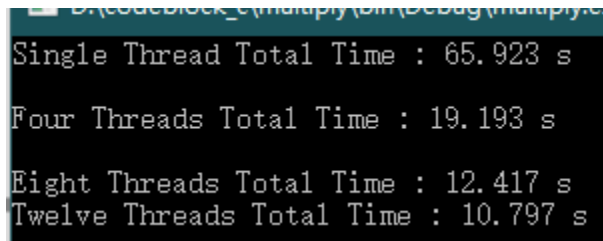
这种方式其实是速度最慢的方式, 有一个能够把速度提升五倍左右的最简单的办法, 就是把原来的 `ijk` 顺序改为 `ikj` 顺序:

```
for(int i=0;i<n;++i)
    for(int k=0;k<n;++k)
        s=D[i][k];
        for(int j=0;j<n;++j)
            F[i][j]+=s*E[k][j];
```

下面分析为什么效果这么显著, 主要就是一个内存访问不连续, 导致 `cache` 命中率不高的问题。为了加速, 就要尽可能使内存访问连续。前面提到, 矩阵是按行存储的, 那么如果顺序为 `ijk`, 则计算 `F` 中的每一个元素时 (即每次 `++k` 时), `B` 中被访问的元素会

沿着相应的列向下移动，这样在内存中就会进行一次跳跃访问，也就是访问不连续。当顺序为 ikj 时，访问的元素移动顺序都是按行进行，总体速度来说 ikj 是最快的。

使用以上方法优化的同时，还使用了 `<thread>` 库进行多线程的并发执行。因为矩阵按行存储，所以每个线程的任务是通过矩阵的行数进行分配的，比如四线程的时候，每个线程的任务量就是矩阵的行数 m 除以 4，即 $m/4$ 。因为实验室的电脑最多能支持 12 线程，所以在程序中设计了四线程、八线程、十二线程分别对 2000×2000 的矩阵进行运算，最后观察它们的运行速度，发现对于 2000×2000 的矩阵乘运算，使用十二线程能够得到最快的运算速度。



```
Single Thread Total Time : 65.923 s
Four Threads Total Time : 19.193 s
Eight Threads Total Time : 12.417 s
Twelve Threads Total Time : 10.797 s
```

图 1: 在 windows 系统上的多线程矩阵乘法运行结果

源代码:

```
#include<iostream>
#include<vector>
#include<string>
#include<algorithm>
#include<thread>
#include<ctime>

using namespace std;

// f mn = d mt * e tn
vector<vector<int>> > d;
vector<vector<int>> > e;
vector<vector<int>> > f;

//single thread
void mulSingle()
{
    int m = d.size(); // d.size()表示矩阵d的行数
    int t = d[0].size(); // d[0].size()表示矩阵d的列数
    int n = e[0].size();
    // ikj is faster than ijk (memory access issues)
    for (int i = 0; i < m; ++i)
        for (int k = 0; k < t; ++k)
        {
```

```

        int s = d[i][k];
        for (int j = 0; j < n; ++j)
            f[i][j] += s * e[k][j];
    }
}

//multi thread
void mulMulti(int rowStart, int rowEnd)
{
    int m = d.size();
    int t = d[0].size();
    int n = e[0].size();
    // ikj is faster than ijk (memory access issues)
    for (int i = rowStart; i < rowEnd; ++i)
        for (int k = 0; k < t; ++k)
        {
            int s = d[i][k];
            for (int j = 0; j < n; ++j)
                f[i][j] += s * e[k][j];
        }
}

// create a matrix
vector<vector<int>> > createMat(int m, int n) {
    vector<vector<int>> > ans(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            ans[i][j] = i + j; // arbitrary value
    return ans;
}

int main()
{
    clock_t startTime, endTime;

    // initializing matrices
    d = createMat(2000, 2000);
    e = createMat(2000, 2000);

    // f (m*n) = d (m*t) * e (t*n)
    int m = d.size();
    int t = d[0].size();
    int n = e[0].size();

    f.resize(m);
    for (int i = 0; i < m; ++i)
        f[i].resize(n);

```

```
//single thread
startTime = clock();
mulSingle();
endTime = clock();

//display

cout << "Single Thread Total Time : " << (double)(endTime - startTime)\
    / CLOCKS_PER_SEC << " s" << endl;

// initializing matrix
f.clear();
f.resize(m);
for (int i = 0; i < m; ++i)
    f[i].resize(n);
cout << endl;

//multiple thread
startTime = clock();
int div = m / 4;
thread t1(mulMulti, 0, div);
thread t2(mulMulti, div, 2 * div);
thread t3(mulMulti, 2 * div, 3 * div);
thread t4(mulMulti, 3 * div, m);
t1.join();
t2.join();
t3.join();
t4.join();
endTime = clock();

//display

cout << "Four Threads Total Time : " << (double)(endTime - startTime)\
    / CLOCKS_PER_SEC << " s" << endl;
cout << endl;

startTime = clock();
int divv = m / 8;
thread th1(mulMulti, 0, divv);
thread th2(mulMulti, divv, 2 * divv);
thread th3(mulMulti, 2 * divv, 3 * divv);
thread th4(mulMulti, 3 * divv, 4 * divv);
thread th5(mulMulti, 4 * divv, 5 * divv);
thread th6(mulMulti, 5 * divv, 6 * divv);
thread th7(mulMulti, 6 * divv, 7 * divv);
thread th8(mulMulti, 7 * divv, 8 * divv);
```

```

    th1.join();
    th2.join();
    th3.join();
    th4.join();
    th5.join();
    th6.join();
    th7.join();
    th8.join();
    endTime = clock();

    cout << "Eight Threads Total Time : " << (double)(endTime - startTime)\
        / CLOCKS_PER_SEC << " s" << endl;

    startTime = clock();
    int divvv = m / 12;
    thread thr1(mulMulti, 0, divvv);
    thread thr2(mulMulti, divvv, 2 * divvv);
    thread thr3(mulMulti, 2 * divvv, 3 * divvv);
    thread thr4(mulMulti, 3 * divvv, 4 * divvv);
    thread thr5(mulMulti, 4 * divvv, 5 * divvv);
    thread thr6(mulMulti, 5 * divvv, 6 * divvv);
    thread thr7(mulMulti, 6 * divvv, 7 * divvv);
    thread thr8(mulMulti, 7 * divvv, 8 * divvv);
    thread thr9(mulMulti, 8 * divvv, 9 * divvv);
    thread thr10(mulMulti, 9 * divvv, 10 * divvv);
    thread thr11(mulMulti, 10 * divvv, 11 * divvv);
    thread thr12(mulMulti, 11 * divvv, m);
    thr1.join();
    thr2.join();
    thr3.join();
    thr4.join();
    thr5.join();
    thr6.join();
    thr7.join();
    thr8.join();
    thr9.join();
    thr10.join();
    thr11.join();
    thr12.join();
    endTime = clock();

    cout << "Twelve Threads Total Time : " << (double)(endTime - startTime)\
        / CLOCKS_PER_SEC << " s" << endl;
    //this_thread::sleep_for(chrono::seconds(5));
    return 0;
}

```

2 GPU 上的矩阵乘

在 GPU 上的矩阵乘是通过 CUDA 编程实现的。CUDA 是建立在 NVIDIA 的 CPUs 上的一个通用并行计算平台和编程模型，基于 CUDA 编程可以利用 GPU 的并行计算引擎来更加高效地解决比较复杂的计算难题。GPU 并不是一个独立运行的计算平台，而需要与 CPU 协同工作，可以看成是 CPU 的协处理器，因此所谓的 GPU 并行计算，其实是指的是基于 CPU+GPU 的异构计算架构。在此架构中，CPU 与 GPU 通过 PCIe 总线连接在一起协同工作，CPU 所在位置称为主机端 (host)，而 GPU 所在位置称为设备端 (device)，如下图。

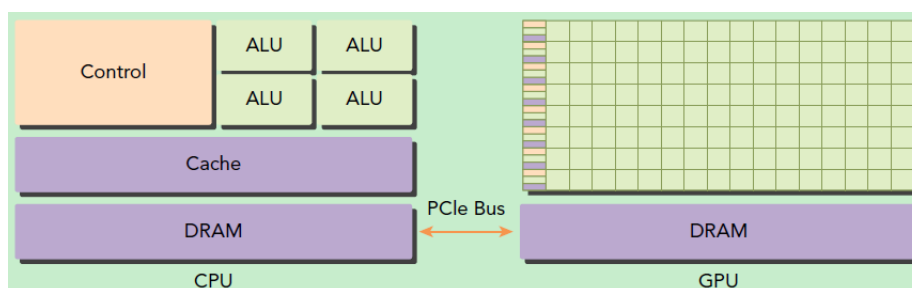


图 2: 基于 CPU+GPU 的异构计算. 来源:《CUDA C 编程》

在 CUDA 中, *host* 和 *device* 是两个重要的概念, 我们用 *host* 指代 CPU 及其内存, 而用 *device* 指代 GPU 及其内存。CUDA 程序中既包含 *host* 程序, 又包含 *device* 程序, 它们分别在 CPU 和 GPU 上运行。同时, *host* 与 *device* 之间可以进行通信, 这样它们之间可以进行数据拷贝。典型的 CUDA 程序的执行流程如下:

1. 分配 *host* 内存, 并进行数据初始化;
2. 分配 *device* 内存, 并从 *host* 将数据拷贝到 *device* 上;
3. 调用 CUDA 的核函数在 *device* 上完成指定的运算;
4. 将 *device* 上的运算结果拷贝到 *host* 上;
5. 释放 *device* 和 *host* 上分配的内存; 上面流程中最重要的一个过程是调用 CUDA 的核函数来执行并行计算, *kernel* 是 CUDA 中一个重要的概念, *kernel* 是在 *device* 上线程中并行执行的函数, 核函数用 `__global__` 符号声明, 在调用时需要用 `<<< grid, block >>>` 来指定 *kernel* 要执行的线程数量, 在 CUDA 中, 每一个线程都要执行核函数, 并且每个线程会分配一个唯一的线程号 *thread ID*, 这个 ID 值可以通过核函数的内置变量 *threadIdx* 来获得。

由于 GPU 实际上是异构模型, 所以需要区分 *host* 和 *device* 上的代码, 在 CUDA 中是通过函数类型限定词来区别 *host* 和 *device* 上的函数, 主要的三个函数类型限定词如下:

- `__global__`: 在 *device* 上执行, 从 *host* 中调用 (一些特定的 GPU 也可以从 *device* 上调用), 返回类型必须是 `void`, 不支持可变参数参数, 不能成为类成员函数。注意用

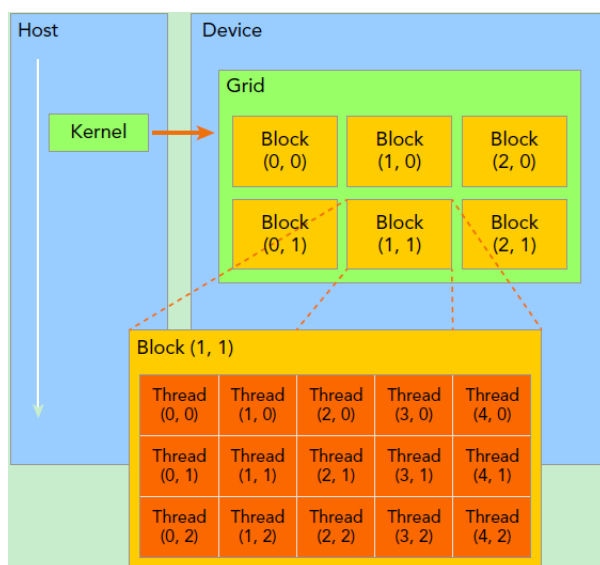


图 3: kernel 上的两层线程组织结构 (2-dim)

global 定义的 kernel 是异步的，这意味着 host 不会等待 kernel 执行完就执行下一步。

- `__device__`: 在 device 上执行，单仅可以从 device 中调用，不可以和 global 同时用。
- `__host__`: 在 host 上执行，仅可以从 host 上调用，一般省略不写，不可以和 global 同时用，但可和 device 同时用，此时函数会在 device 和 host 都编译。

kernel 在 device 上执行时实际上是启动很多线程，一个 kernel 所启动的所有线程称为一个网格 (grid)，同一个网格上的线程共享相同的全局内存空间，grid 是线程结构的第一层次。而网格又可分为很多的线程块 (block)，一个线程块里面包含很多线程，这是第二个层次。线程两层组织如图 3 所示，这是一个 grid 和 block 均为 2-dim 的线程组织。grid 和 block 都是定义为 dim3 类型的变量，dim3 可以看成是包含三个无符号整数 (x, y, z) 成员的结构体变量，在定义时，缺省值初始化为 1。因此 grid 和 block 可以灵活地定义为 1-dim, 2-dim 以及 3-dim 结构，对于图中结构 (主要水平方向为 x 轴)，定义的 grid 和 block 如下所示，kernel 在调用时也必须通过执行配置 `<<grid, block>>` 来指定 kernel 所使用的线程数及结构。

```
dim3 grid(3, 2);
dim3 block(5, 3);
kernel_fun<<< grid, block >>>(prams...);
```

所以，一个线程需要两个内置的坐标变量 (blockIdx, threadIdx) 来唯一标识，它们都是 dim3 类型变量，其中 blockIdx 指明线程所在 grid 中的位置，而 threadIdx 指明线程所在 block 中的位置，如图中的 Thread (1,1) 满足：

```
threadIdx.x = 1
threadIdx.y = 1
blockIdx.x = 1
blockIdx.y = 1
```

一个线程块上的线程是放在同一个流式多处理器 (Streaming Multiprocessor) 上的, 但是单个 SM 的资源有限, 这导致线程块中的线程数是有限制的, 现代 GPUs 的线程块可支持的线程数可达 1024 个。由于 SM 的基本执行单元是包含 32 个线程的线程束, 所以 block 大小一般要设置为 32 的倍数。同时, 网格和线程块只是逻辑划分, 一个 kernel 的所有线程其实在物理层是不一定同时并发的。所以 kernel 的 grid 和 block 的配置不同, 性能会出现差异, 这点是要特别注意的。

在实验开始之前, 我先通过如下程序查看了实验室的 GPU 硬件配置, 可以看到 GTX 1050Ti 的显卡每个线程块支持的最大线程数为 1024 个, 有 6 个 SM。

源代码:

```
int count;

//取得支持Cuda的装置的数目
cudaGetDeviceCount(&count);
if (count == 0)
{
    fprintf(stderr, "There is no device.\n");

    return false;
}
int i;
for (i = 0; i < count; i++)
{

    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);
    //打印设备信息
    printf("Device Name : %s.\n", prop.name);
    printf("totalGlobalMem : %d.\n", prop.totalGlobalMem);
    printf("sharedMemPerBlock : %d.\n", prop.sharedMemPerBlock);
    printf("regsPerBlock : %d.\n", prop.regsPerBlock);
    printf("warpSize : %d.\n", prop.warpSize);
    printf("memPitch : %d.\n", prop.memPitch);
    printf("maxThreadsPerBlock : %d.\n", prop.maxThreadsPerBlock);
    printf("maxThreadsDim[0 - 2] : %d %d %d.\n", prop.maxThreadsDim[0], prop.
        maxThreadsDim[1], prop.maxThreadsDim[2]);
    printf("maxGridSize[0 - 2] : %d %d %d.\n", prop.maxGridSize[0], prop.maxGridSize
        [1], prop.maxGridSize[2]);
    printf("totalConstMem : %d.\n", prop.totalConstMem);
```



```
printf("major.minor : %d.%d.\n", prop.major, prop.minor);
printf("clockRate : %d.\n", prop.clockRate);
printf("textureAlignment : %d.\n", prop.textureAlignment);
printf("deviceOverlap : %d.\n", prop.deviceOverlap);
printf("multiProcessorCount : %d.\n", prop.multiProcessorCount);

if (cudaGetDeviceProperties(&prop, i) == cudaSuccess)
{
    if (prop.major >= 1)
    {
        break;
    }
}
}

// 输出如下
Device Name : GeForce GTX 1050 Ti.
totalGlobalMem : 0.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 6.1.
clockRate : 1392000.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 6.
```

矩阵乘算法的设计和 CPU 上类似，区别就是在分配线程的方面。首先矩阵的初始化为为了使生成的矩阵和 CPU 上相同，还是使用了 $A[i]=i+j$ 的方法。在分配线程方面，由于 GPU 支持很多线程，所以根据矩阵的大小设置了 grid 的数量，使每个线程执行一个乘法运算。

但是消耗我大部分时间的不是分配线程的过程，而是怎么在 GPU 运算过程中进行计时。在开始我使用了 c 语言中的 `clock()`，但是发现在核函数中输出的 `clock()` 值是一个负数，这个问题的原因到现在也没找到。后来通过查阅资料和别人编写的程序，发现了另一种计时方法，使用 `cudaevent` 进行计时，最终的运行速度为 2s 左右，如图 4 所示，比 CPU 上十二线程的执行时间快了 5 倍左右。

当然，目前的程序并没有完全优化到最好，之后会考虑 cache 命中率问题以及使用 CUBLAS 库，不但可以屏蔽掉复杂的底层实现以及不同计算设备带来的参数设计的影响，还有机会把矩阵乘法的效率进一步提升。

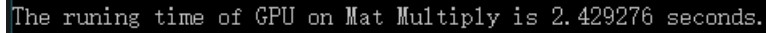


图 4: 在 GPU 上通过 cuda 实现的多线程矩阵乘法运行结果

源代码:

```
#include <stdio.h>
#include <stdlib.h>
//CUDA RunTime API
#include <cuda_runtime.h>
using namespace std;
#define THREAD_NUM 1024
#define MATRIX_SIZE 2000

//定义线程块的个数 (向上取整)
const int blocks_num = (MATRIX_SIZE * MATRIX_SIZE + THREAD_NUM - 1) / THREAD_NUM;

//CUDA 初始化
bool InitCUDA()
{
    int count;

    //取得支持Cuda的装置的数目
    cudaGetDeviceCount(&count);
    if (count == 0)
    {
        fprintf(stderr, "There is no device.\n");

        return false;
    }
    int i;
    for (i = 0; i < count; i++)
    {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        if (cudaGetDeviceProperties(&prop, i) == cudaSuccess)
        {
            if (prop.major >= 1)
            {
                break;
            }
        }
    }
    if (i == count)
    {

```

```
        fprintf(stderr, "There is no device supporting CUDA 1.x.\n");
        return false;
    }
    cudaSetDevice(i);
    return true;
}

//矩阵初始化
void matgen(int* a, int n)
{
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            a[i * n + j] = i + j;
        }
    }
}

// __global__ 函数 并行计算矩阵乘法
__global__ static void matMultCUDA(const int* a, const int* b, int* c, int n)
{
    //表示目前的 thread 是第几个 thread (由 0 开始计算)
    const int tid = threadIdx.x;

    //表示目前的 thread 属于第几个 block (由 0 开始计算)
    const int bid = blockIdx.x;

    //从 bid 和 tid 计算出这个 thread 应该计算的 row 和 column
    const int idx = bid * THREAD_NUM + tid;
    const int row = idx / n;
    const int column = idx % n;
    int i;
    if (row < n && column < n)
    {
        int t = 0;
        for (i = 0; i < n; i++)
        {
            t += a[row * n + i] * b[i * n + column];
        }
        c[row * n + column] = t;
    }
}

// 主函数
```

```

int main()
{
    //CUDA 初始化
    if (!InitCUDA()) return 0;

    //定义矩阵
    int* a, * b, * c;
    int n = MATRIX_SIZE;

    //分配内存
    a = (int*)malloc(sizeof(int) * n * n);
    b = (int*)malloc(sizeof(int) * n * n);
    c = (int*)malloc(sizeof(int) * n * n);

    //生成矩阵
    matgen(a, n);
    matgen(b, n);

    /*把数据复制到显卡内存中*/
    int* cuda_a, * cuda_b, * cuda_c;

    //cudaMalloc 取得一块显卡内存
    cudaMalloc((void**)&cuda_a, sizeof(int) * n * n);
    cudaMalloc((void**)&cuda_b, sizeof(int) * n * n);
    cudaMalloc((void**)&cuda_c, sizeof(int) * n * n);

    //cudaMemcpy 将产生的矩阵复制到显卡内存中
    //cudaMemcpyHostToDevice - 从内存复制到显卡内存
    //cudaMemcpyDeviceToHost - 从显卡内存复制到内存
    cudaMemcpy(cuda_a, a, sizeof(int) * n * n, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_b, b, sizeof(int) * n * n, cudaMemcpyHostToDevice);

    // 计时
    cudaEvent_t gpuStart, gpuFinish;
    float elapsedTime;
    cudaEventCreate(&gpuStart);
    cudaEventCreate(&gpuFinish);
    cudaEventRecord(gpuStart, 0);

    // 在CUDA 中执行函数 语法: 函数名称<<<block 数目, thread 数目, shared memory 大小>>>(
        参数...);
    matMultCUDA << < blocks_num, THREAD_NUM, 0 >> > (cuda_a, cuda_b, cuda_c, n);
    cudaEventRecord(gpuFinish, 0);
    cudaEventSynchronize(gpuStart);
    cudaEventSynchronize(gpuFinish);
    cudaEventElapsedTime(&elapsedTime, gpuStart, gpuFinish);

```

```

printf("\nThe runing time of GPU on Mat Multiply is %f seconds.\n", elapsedTime /
      1000.0);

//cudaMemcpy 将结果从显存中复制回内存
cudaMemcpy(c, cuda_c, sizeof(int) * n * n, cudaMemcpyDeviceToHost);

cudaFree(cuda_a);
cudaFree(cuda_b);
cudaFree(cuda_c);
return 0;
}

```

3 FPGA 上的矩阵乘

在 FPGA 上的矩阵乘是在实验室的 PYNQ-Z1 板子上实现的，因为是第一次使用 hls 编程，所以对程序的优化方法不太熟练，最后程序的性能并不尽人意。实验总体分为三个步骤，实验的前两个步骤都是不需要板子就能完成的，第三步需要板子连接电脑，在 Jupyter Notebook 中通过 python 实现。

3.1 Vivado HLS 生成矩阵乘法加速 IP

通过 hls 生成矩阵乘法加速 ip 的过程就是一套具体的 hls 流程，即 C 仿真、C 综合、C 联合 RTL 仿真、输出 ip 核。hls 中新建一个工程，选择的 part 为 xc7z020clg400-1, topfunction 和 testbench 的代码如下。

```

#include <stdio.h> // top function代码
#include <stdlib.h>
#include "matrix.h"

void matrix_mul(int A[1000][1000], int B[1000][1000], int C[1000][1000])
{
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE m_axi depth=1000000 port=C offset=slave bundle=C_P
    #pragma HLS INTERFACE m_axi depth=1000000 port=B offset=slave bundle=B_P
    #pragma HLS INTERFACE m_axi depth=1000000 port=A offset=slave bundle=A_P
    int i, j, k;
    for(i=0; i<1000; i++)
    {
        for(j=0; j<1000; j++)
        {
            int sum=0;
            for(k=0; k<1000; k++)
            {

```

```
        sum += A[i][k] * B[k][j];
    }
    C[i][j] = sum;
}
}
```

```
#include <stdio.h> // testbench代码
#include <stdlib.h>
#include "matrix.h"

static int A[1000][1000], B[1000][1000], test[1000][1000];

int main()
{
    int i, j;
    for(i = 0; i < 1000; i++)
    {
        for(j = 0; j < 1000; j++)
        {
            A[i][j] = 1;
            B[i][j] = 1;
            test[i][j] = 1000;
        }
    }
    static int C[1000][1000] = {0};
    matrix_mul(A, B, C);
    int errcount = 0;
    for(i = 0; i < 1000; i++)
    {
        for(j = 0; j < 1000; j++){
            if(test[i][j] != C[i][j])
            {
                errcount += 1;
            }
        }
    }
    if(errcount != 0){
        printf("test failed");
    }
    else
        printf("test passed!");
    return 0;
}
```

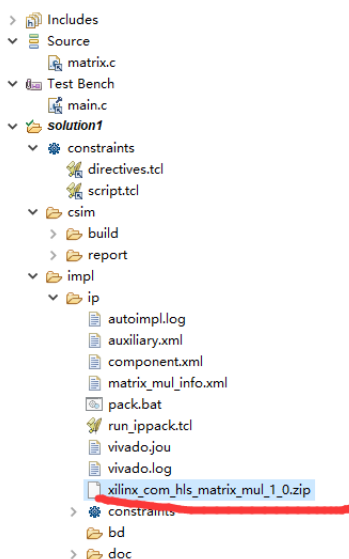


图 5: 在 hls 中找到导出的 ip 核

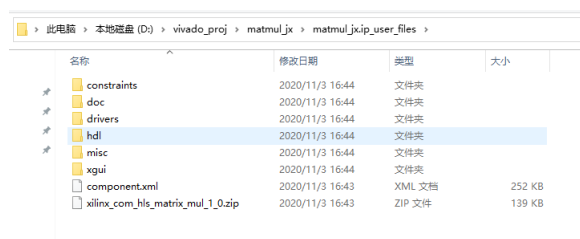


图 6: 将 ip 压缩包复制到 vivado 项目文件夹并解压

3.2 Vivado 进行 Block Design

成功执行四个步骤后，会在 hls 界面里或对应文件夹中找到图 5 中标出的 ip 核。

接着在 vivado 里新建工程，选择的 part 同上。工程建立后，在如图 6 的文件路径中找到该文件夹，将 ip 核压缩包复制至这里并解压。接着进入 vivado 界面，进入 setting 界面，添加 ip 的路径。添加成功后，选择 Create Block Design，首先添加 ZYNQ 的 IP，如图 8 所示，并点击最上面的 Run Block Automation 进行自动连线。

自动连线完毕后，双击 ZYNQ 的 ip，修改配置，如图 9 所示。然后继续添加 hls 自定义的 ip，名字即为 top function 的名字，这里是 matrix_mul。最后进行两次自动连线，并验证正确性，如图 11、12。

验证成功后，按图 14-17 依次导出相应文件。至此已经完成了 vivado 相关的工作，接下来的工作需要 PYNQ-Z1 的板子来完成。

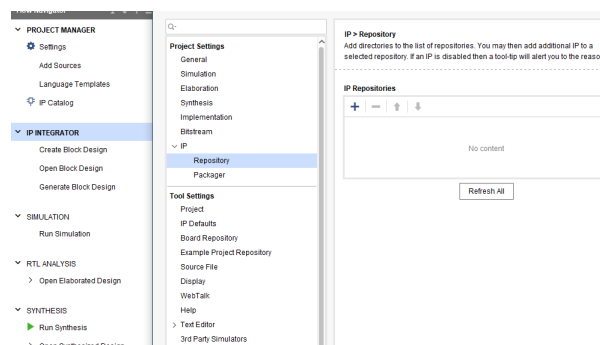


图 7: 在 vivado 中添加 hls 生成的 ip

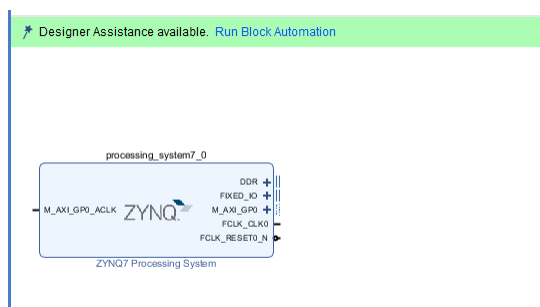


图 8: 在 block design 中添加 zynq 的 ip

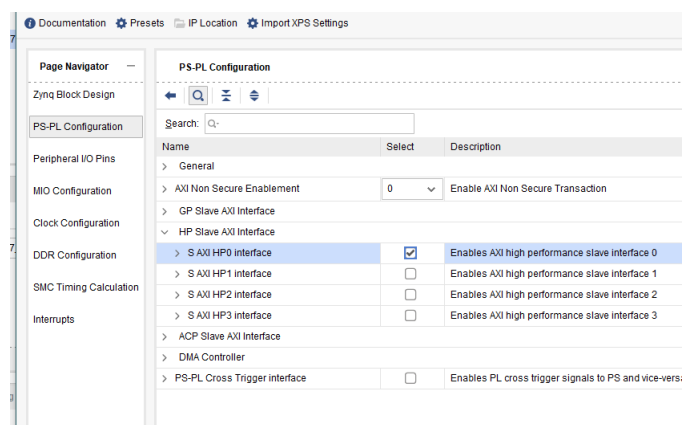


图 9: 修改 zynq 的 ip 核的配置，添加 HP 接口

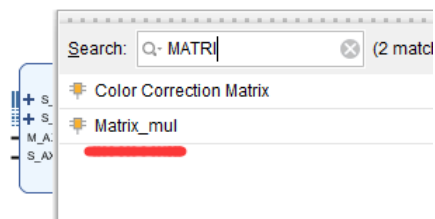


图 10: 在 block design 中添加 hls 自定义的 ip

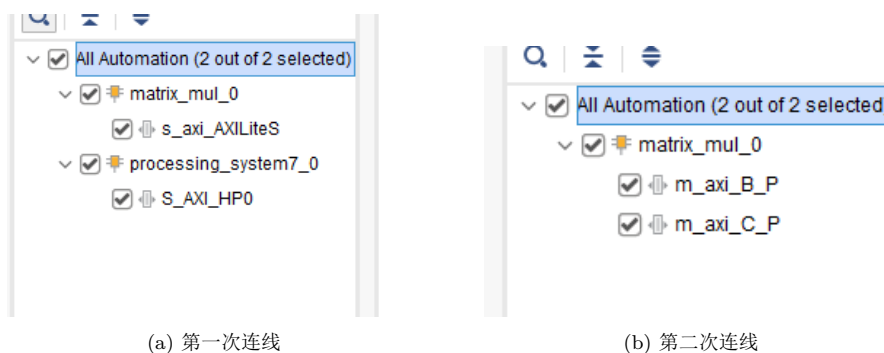


图 11: 两次自动连线

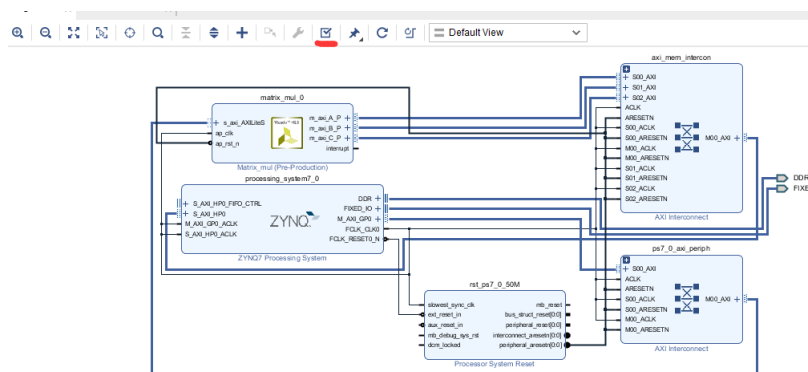


图 12: 进行验证

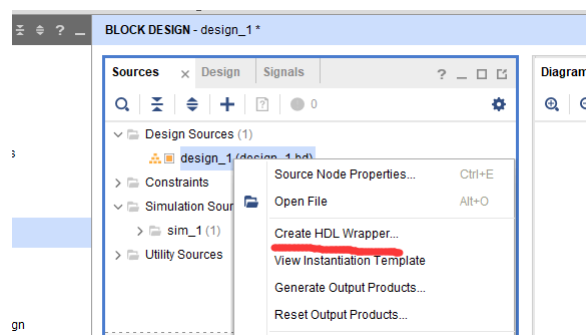
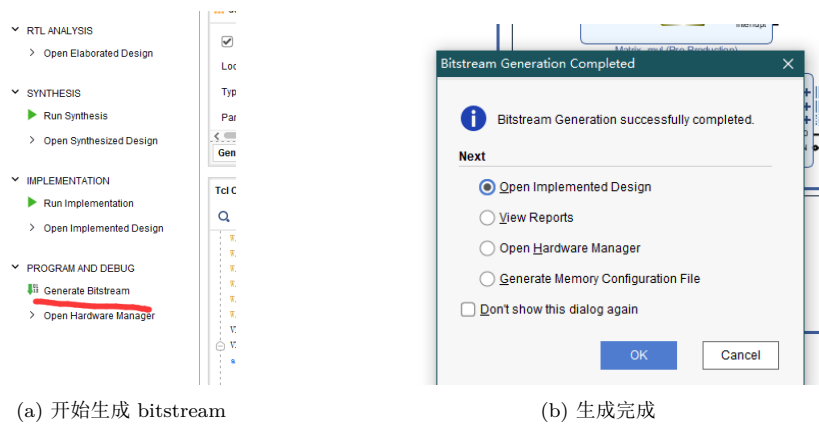


图 13: 生成顶层调用文件



(a) 开始生成 bitstream

(b) 生成完成

图 14: 生成比特文件并等待生成完成

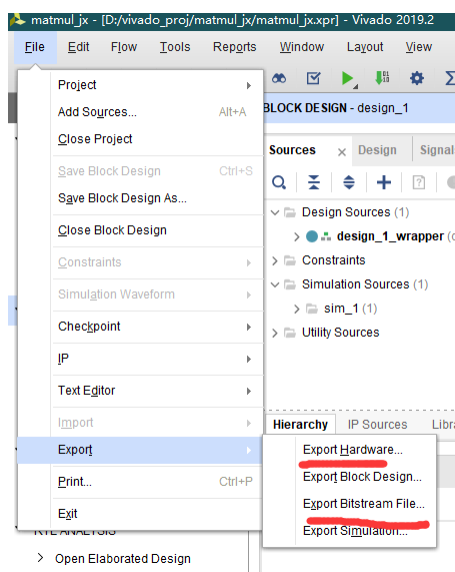
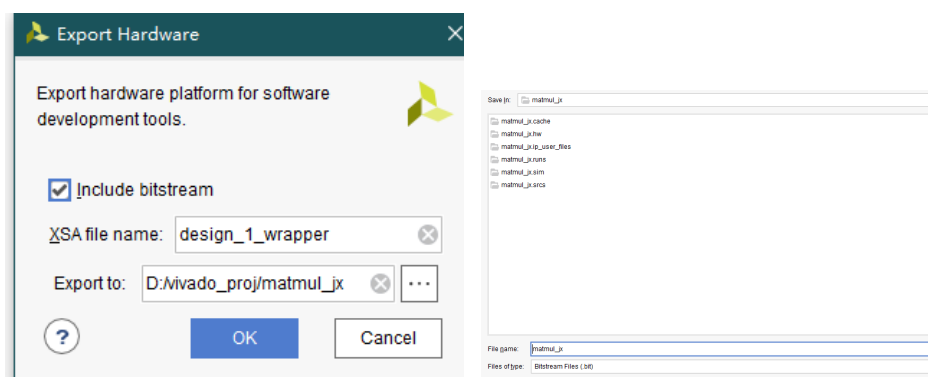


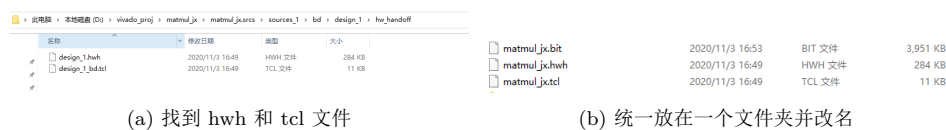
图 15: 导出文件



(a) hardware

(b) bitstream

图 16: 导出 hardware 和 bitstream



(a) 找到 hwh 和 tcl 文件

(b) 统一放在一个文件夹并改名

图 17: 将导出文件整理



图 18: 将文件上传到 jupyter notebook

3.3 Jupyter Notebook 进行矩阵乘法加速 IP 的调用

首先必须用电脑连接 PYNQ，要完成连接需要改动电脑网络 ip 的配置。配置成功后，进入板子的 jupyter notebook 主页，完成如图 18 所示的文件上传。然后点击右上角的 new-python，新建一个 python 文件，两段代码如下所示。

```
from pynq import Overlay # 第一段代码（生成初始矩阵a和b）
overlay = Overlay("./matmul_jc.bit")
mul_ip = overlay.matrix_mul_0

from pynq import Xlnk
xlnk = Xlnk()

import numpy as np
inputa = xlnk.cma_array(shape=(1000, 1000), dtype=int)
inputb = xlnk.cma_array(shape=(1000, 1000), dtype=int)
for i in range(1000):
    for j in range(1000):
        inputa[i][j] = i + j
        inputb[i][j] = i + j
print(inputa)
print(inputb)
```

```
mul_ip.write(0x10, inputa.physical_address) # 第二段代码（调用ip核执行矩阵乘法）
mul_ip.write(0x18, inputb.physical_address)
output_c = xlnk.cma_array(shape=(1000, 1000), dtype=int)
#output_c.physical_address = array_add_ip.read(0x20)
mul_ip.write(0x20, output_c.physical_address)
mul_ip.write(0x00, 0x81)
print(output_c)
```

实验结果如图 19，其中因为程序并没有进行优化，导致运行时间非常长，所以在第二段代码刚开始执行时输出的结果还是 0，等待 3s 后再输出时，程序也没有完全运行完成，最后等待了接近 15 分钟才完全输出了执行乘法之后的结果矩阵。

```

[[ 0  1  2 ..., 997 998 999]
 [ 1  2  3 ..., 998 999 1000]
 [ 2  3  4 ..., 999 1000 1001]
 ...,
 [ 997 998 999 ..., 1994 1995 1996]
 [ 998 999 1000 ..., 1995 1996 1997]
 [ 999 1000 1001 ..., 1996 1997 1998]]
[[ 0  1  2 ..., 997 998 999]
 [ 1  2  3 ..., 998 999 1000]
 [ 2  3  4 ..., 999 1000 1001]
 ...,
 [ 997 998 999 ..., 1994 1995 1996]
 [ 998 999 1000 ..., 1995 1996 1997]
 [ 999 1000 1001 ..., 1996 1997 1998]]

```

(a) 第一段代码运行结果

```

mul_ip.write(0x10, inputa.physical_address)
mul_ip.write(0x18, inputb.physical_address)
output_c = xlnk.cma_array(shape=(1000, 1000), dtype=int)
#output_c.physical_address = array_add_ip.read(0x20)
mul_ip.write(0x20, output_c.physical_address)
mul_ip.write(0x00, 0x81)
print(output_c)

[[0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 ...,
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]]

```

(b) 第二段代码运行结果

```

print(output_c)
[[332833500 333333000 333832500 ..., 830835000 831334500 831834000]
 [333333000 333833500 334334000 ..., 832331500 832832000 833332500]
 [333832500 334334000 334833500 ..., 833828000 834329500 834831000]
 ...,
 [ 0 0 0 ..., 0 0 0]
 [ 0 0 0 ..., 0 0 0]
 [ 0 0 0 ..., 0 0 0]]

```

(c) 3s 后运行结果

```

: print(output_c)
[[ 332833500 333333000 333832500 ..., 830835000 831334500
 831834000]
 [ 333333000 333833500 334334000 ..., 832331500 832832000
 833332500]
 [ 333832500 334334000 334833500 ..., 833828000 834329500
 834831000]
 ...,
 [ 830835000 832331500 833828000 ..., -1972121796 -1970625296
 -1969128796]
 [ 831334500 832832000 834329500 ..., -1970625296 -1969127796
 -1967630296]
 [ 831834000 833332500 834831000 ..., -1969128796 -1967630296
 -1966131796]]

```

(d) 最终结果

图 19: 运行结果