个性课程名称： **CUDA高性能科学计算**

|  |  |
| --- | --- |
| 题目名称 | 一种基于CUDA C++混合编程的Minecraft TNT爆炸模拟(2D) |
| **项目背景**  Minecraft以玩家在一个充满着方块的三维空间中自由地创造和破坏不同种类的方块为主题。玩家在游戏中可以在单人或多人模式中通过摧毁或创造精妙绝伦的建筑物和艺术，或者收集物品探索地图以完成游戏的成就（进度）。玩家也可以尝试在创造模式下(打开作弊)红石电路和指令等玩法。  在Minecraft中，TNT是一种能产生强大破坏力的方块，一旦被激活，就会产生爆炸。它可以用来清除大量的方块或进行构造,**其破坏机制是以自身为中心向四周发射破坏射线，射线触及方块后会摧毁方块，以达到爆炸破坏的效果.**  本程序基于这个机制，在二维世界里模拟了TNT的爆炸机制，并对均匀的测试世界（模拟Minecraft的”超平坦维度”）进行破坏模拟,并保留了丰富的可拓展性.  最后，使用Cuda调用GPU对这个爆炸模拟程序的热点区域进行GPU加速和性能优化，可以降低90%以上的内存占用，代价是占用并不大的显存空间,并对一些生成和判定部分的代码取得10~20倍的加速效果(即使规模并不大时)，并且这个加速效果可以随着模拟规模的增大进一步体现.  **架构设计与算法**  为了调试方便，使用经典的函数式编程,每一步和每一种操作都被封装为函数，使用指针传址操作，以达到方便升级维护的目的.函数按照流程排序，有关的内容的函数被放在一块，便于快速修改.在main()中不使用的函数被注释屏蔽，在需要debug时解除注释即可启动debug模式.(注意：cout非常消耗性能)  存储数据使用了简单的一维Array（非STL）和Vector容器(C++ STL)，用于保证易用性和开发与执行的效率，但是也造成了些许性能的开销(比如索引的存储).  **基础软硬件情况**  **操作系统**:Windows 10专业版(22H2)  **CPU**:Intel Core [i7-11800H @2.30GHz](mailto:i7-11800H@2.30GHz) 8C16T 全核睿频4.2GHz(性能增强)  **显卡**(Cuda Runtime):NVIDIA GeForce RTX3060 Laptop (6GB GDDR6)  **内存**:SK Hynix 32GB DDR4 3200MHz(CL22)  **硬盘**:WD BLACK-SN770 1TB(因为出现了爆内存导致硬盘充当虚拟内存的情况)  受到接口限制,实际最大传输带宽:PCIe 3.0x4 (3500MB/s)  **显卡驱动**:老板驱动(531.68) & 一个较新的驱动(546.29)  用于执行CUDA Runtime的CPU和显卡已经换好了崭新的信越7868导热硅脂，并对散热模具进行了清灰操作，以确保硬件运行在最佳状态(CPU:115W 4.2GHz)(GPU:115W GPU\_Clock=1450MHz,Memory\_Clock=1750MHz)  **核心流程和组件介绍**  **>>>2BeAccByCUDA (纯CPU部分:代码标准:C++ 17,编译工具包:Visual Studio)**  **方块**  方块使用结构体数组存储，这是存储方块的结构体的有关代码:  **MCExplosion\_CPU.cpp**  **#include <iostream> #include <fstream> #include <ctime> #include <cmath> #include <random> #include <string> #include <vector> #include<algorithm>**  **using std::string; using std::cout; using std::endl;   struct Blocks{  *//模拟的二维世界的方块的基础信息* string Block\_Name; *//方块的名称* int Block\_Id; *//方块Id(用于debug)* int Block\_HP; *//方块的血量* bool is\_Destoryed=false; *//是否被摧毁* int Direct\_XY[2]; *//定位坐标（方块的左下角)* int Relative\_Position; *//表示和爆炸中心的相对位置关系  /\*从x轴正方向开始逆时针旋转，一共有12种情况:  \* 0=正右(k>0)  \* 1=右上(K>0)  \* 2=正上(k>0)  \* 3=正上(k<0)  \* 4=左上(k<0)  \* 5=正左(k<0)  \* 6=正左(k>0)  \* 7=左下(k>0)  \* 8=正上(k>0)  \* 9=正上(k<0)  \* 10=右下(k<0)  \* 11=正右(k<0)  \* \*/* double K\_Min; *//二维相交求解:最小K值* double K\_Mid; *//二维相交求解:中间K值* double K\_Max; *//二维相交求解:最大K值* };**  爆炸中心和方块有十二种相对位置关系(这是由射线破坏的判定机制决定的;解决问题的基本思路是，直接求解出点和方块的位置关系(这一步因为逻辑判断分支的开销过于严重，会导致试图使用GPU全部加速反而会加速**-**75%,因此我将任务进行了一些简单的分解，具体请见后文)，得出”可能发生相交”的斜率区间，随后在执行相交判定时可以提高判定效率(不用计算而是直接看斜率是否落在区间内))  这种做法的使得GPU的核心线程优势在执行相交判定算法时得以凸显.    **初始化方块世界**  因为要直接和Cuda对比运行效率，因此这里使用了模拟GPU的方式来运行，最后进行移植操作（使用cuda加速）,不过这种操作和两层for循环来比的话，可读性和效率方面我认为差不多.因此我使用了单层for循环.  **void GenerateBlocksLikeGPU(string BlockName,int BlockHP,Blocks\* Block\_Array,int ArraySize,int Depth,int Width,int\* TNTPosition){  clock\_t GenBlock\_Start=clock();  *//用于验证GPU的函数是否正确...然后转移到GPU上* int Left\_Side=-Width/2; *//定位左侧和右侧* int Right\_Side=Width/2-1; *//定位左侧和右侧* for(int i=0;i<ArraySize;i++){  Block\_Array[i].Block\_HP=BlockHP;  Block\_Array[i].Block\_Id=i;  *//在CPU程序里，我是先定位X坐标再定位Y坐标，生成方式是X从左往右,Y从下往上  //求X的方式是用第n个的索引除去深度（Y的长度），就可以获得X的位置  //获得X的位置后，就知道走过了几个列，此时减去之前走过的列的量，就可以知道我现在的深度在哪里* Block\_Array[i].Direct\_XY[0]=Left\_Side+i/abs(Depth); *//使用abs()是为了处理Depth为负数的情况  //cout<<i<<"/"<<Depth<<"="<<i/Depth<<endl;* Block\_Array[i].Direct\_XY[1]=Depth+i%abs(Depth);  *//cout<<-Depth<<"+"<<i<<"%"<<Depth<<"="<<-Depth+i%Depth<<endl;  //cout<<"Index:"<<i<<"Gen:X-Y="<<BlockArray[i].Direct\_XY[0]<<" "<<BlockArray[i].Direct\_XY[1]<<endl;   //判断相对位置(相对于爆炸点)* int Relative\_Position[2]={  Block\_Array[i].Direct\_XY[0]-TNTPosition[0]  *//x(方块定位坐标）-x(爆炸点),判断是左(<0)还是右(>0)* ,  Block\_Array[i].Direct\_XY[1]-TNTPosition[1]  *//y(方块定位坐标)-y(爆炸点),判断是下(<0)还是上(>0)* };   *//方块的另外几个坐标  //先生成由定位坐标来确定的其他三个点的坐标值  //左上点:X=X,Y=Y+1* int DirectXY\_LT[2]={Block\_Array[i].Direct\_XY[0],**  **Block\_Array[i].Direct\_XY[1]+1};  *//右上点:X=X+1,Y=Y+1* int DirectXY\_RT[2]={Block\_Array[i].Direct\_XY[0]+1,  Block\_Array[i].Direct\_XY[1]+1};  *//右下点:X=X+1,Y=Y* int DirectXY\_RD[2]={Block\_Array[i].Direct\_XY[0]+1,  Block\_Array[i].Direct\_XY[1]};   *//判断分支  //正右(k>0)* if(Relative\_Position[0]>0 && Relative\_Position[1]>0 && abs(Relative\_Position[1])<=1){  *//k最小值为定位坐标点;k最大值为左上;没有中间k值* Block\_Array[i].Relative\_Position=0;   Block\_Array[i].K\_Min=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  }  *//右上(k>0)/正上(k>0)(需要计算中判定)* else if(Relative\_Position[0]>0 && Relative\_Position[1]>0){  *//k最小值为右下;k中间值为定位坐标点;k最大为左上* Block\_Array[i].Relative\_Position=1;   Block\_Array[i].K\_Min=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);   *//正上(k>0)的情况需要计算上式来判定* if(isinf(Block\_Array[i].K\_Mid) && isinf(Block\_Array[i].K\_Max)){  Block\_Array[i].Relative\_Position=2;  }  }  *//正上(k<0)/左上(k<0)(正上需要计算中判定)*** **else if(Relative\_Position[0]<0 && Relative\_Position[1]>0 && abs(Relative\_Position[1])>1){  *//先判断左上和正上，排除正左的情况;  //对于左上，k最小值为左下;k中间为右下;k最大为右上(注意负数)* Block\_Array[i].Relative\_Position=4;   Block\_Array[i].K\_Min=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);   *//正上(k<0)的情况需要计算上式来确定* if(isinf(Block\_Array[i].K\_Mid) && isinf(Block\_Array[i].K\_Max)){  Block\_Array[i].Relative\_Position=3;  }  }  *//正左(k<0)* else if(Relative\_Position[0]<0 && Relative\_Position[1]>0 && abs(Relative\_Position[1])<=1){  *//k最小值为右下;k最大值为右上;没有中间k值* Block\_Array[i].Relative\_Position=5;   Block\_Array[i].K\_Min=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  }  *//正左(k>0)* else if(Relative\_Position[0]<0 && Relative\_Position[1]<0 && abs(Relative\_Position[1])<=1){  *//k最小值为右上;k最大值为右下;没有中间k值* Block\_Array[i].Relative\_Position=6;   Block\_Array[i].K\_Min=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);**  **Block\_Array[i].K\_Max=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  }  *//左下(k>0),正上(k>0)* else if(Relative\_Position[0]<0 && Relative\_Position[1]<0){  *//先按照左下来判定;k最小值为左上;k中间为右上;k最大为右下* Block\_Array[i].Relative\_Position=7;   Block\_Array[i].K\_Min=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);   if(isinf(Block\_Array[i].K\_Mid) && isinf(Block\_Array[i].K\_Max)){  *//正上方(k>0)* Block\_Array[i].Relative\_Position=8;  }  }   *//补充:x=0的情况（正上)* else if(Relative\_Position[0]==0 && Relative\_Position[1]<0){  Block\_Array[i].Relative\_Position=9;   *//最大：垂直(-无限，这里给-99999),最小:右侧.无中间k值* Block\_Array[i].K\_Min=-99999;  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  }   *//正右(k<0)* else if(Relative\_Position[0]>0 && Relative\_Position[1]<0 && abs(Relative\_Position[1])<=1){  *//k最小值为定位点;k最大值为左上;没有中间k值* Block\_Array[i].Relative\_Position=11;   Block\_Array[i].K\_Min=(double)**  **(Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  }   *//正上(k<0),右下* else if(Relative\_Position[0]>0 && Relative\_Position[1]<0){  *//先按照右下来判定;(注意负数)k最小为定位点;k中间为左上;k最大为右上* Block\_Array[i].Relative\_Position=10;   Block\_Array[i].K\_Min=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/(Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  }  else{}  }   *//接下来，修复nan(ind)的bug(-1,-1直接怼脸爆炸源了)(替换为0)* for(int i=0;i<ArraySize;i++){  *//cout<<"Checking"<<i<<endl;* if(isnan(Block\_Array[i].K\_Min)){  Block\_Array[i].K\_Min=-99999;  }  if(isnan(Block\_Array[i].K\_Mid)){  Block\_Array[i].K\_Mid=0;  }  if(isnan(Block\_Array[i].K\_Max)){  Block\_Array[i].K\_Max=99999;  }  }   */\*  for(int i=0;i<All\_Blocks\_Number;i++){  cout<<"Name:"<<Block\_Array[i].Block\_Name<<" "  <<"HP:"<<Block\_Array[i].Block\_HP<<" "  <<"X:"<<Block\_Array[i].Direct\_XY[0]<<" "***  ***<<"Y:"<<Block\_Array[i].Direct\_XY[1]<<" "  <<"Relative\_position:"<<Block\_Array[i].Relative\_Position<<" "  <<"K-To-(0,0)-KMin:"<<Block\_Array[i].K\_Min<<" "  <<"K-To-(0,0)-KMid:"<<Block\_Array[i].K\_Mid<<" "  <<"K-To-(0,0)-KMax:"<<Block\_Array[i].K\_Max<<" "  <<endl;  //debug:用于检查  }  \*/* clock\_t GenBlock\_End=clock();  cout<<"GenBlock\_Time:"<<(double)((GenBlock\_End-GenBlock\_Start)/CLOCKS\_PER\_SEC)<<endl; }**  在后面的Cuda加速中，我选取了一定的内容分离，用GPU加速一些”简单的部分”,同时也提高了CPU的运行效率.  **输出方块信息**  这个代码非常简单，判断方块是否被摧毁，然后将没有被摧毁的方块的坐标信息输出，用于给绘图引擎进行绘制，打印模拟结果，也可以在生成时进行检查.  **void ExportBlockFile(Blocks\* BlockArray,int ArraySize){  *//导出方块信息文件(用于生成时检查)* cout<<"Start Exporting Files for Blocks..."<<endl;  clock\_t Export\_Start=clock();  std::ofstream ofstream;  ofstream.open("BlockFile.txt"); *//打开文件用于输出* for(int PrintIndex=0;PrintIndex<ArraySize;PrintIndex++){  if(BlockArray[PrintIndex].is\_Destoryed==false){  ofstream<<BlockArray[PrintIndex].Direct\_XY[0]<<" "  <<BlockArray[PrintIndex].Direct\_XY[1]<<endl;  }  }  clock\_t Export\_End=clock();  double Export\_Time=(double)(Export\_End-Export\_Start)/CLOCKS\_PER\_SEC;  cout<<"Export Dur(Block):"<<Export\_Time<<endl; }**  在文件较大时，这个输出的性能开销非常巨大，因此我建议后续升级时使用并发操作.  **射线**  考虑到开销问题，射线除了存储自身的基础信息外，还承担起了存储相交数据的任务，如果把存储相交信息的变量放在方块里，**会导致开销非常巨大**，并导致了很多没有被占用的无用空间，因此射线会承担起这个任务.(不过，我发现这好像可以直接把存储任务交给另一个结构体，因为我后面也是这样做的（单独用了一个结构体数组）,这样可以进一步降低开销,优化性能.(不过,这个时候我已经在写报告了).  **struct Rays{  *//模拟的二维世界的射线的基础信息   //考虑到性能问题，我将相交数据存储到射线里以节省空间***  ***//但是如上文所说，我完全可以单独新建一个数组用于存储.  //两个存储和方块的相交坐标，一个存储距离，用索引来同步存取   //数组长度由模拟规模决定;因此每次修改模拟规模，这里也要修改  //长度为(对角线长度向上取整)x2+4 (+2其实就够了，我怕出事多加2)  //模拟规模半边长:10,则为34(15x2+4)* int Ray\_HP; *//射线的威力* double Ray\_K; *//射线的斜率   //相交数据* int BlockId[1000000]={0}; *//相交方块的Id* double Cross\_X[1000000]={0}; *//相交点:X* double Cross\_Y[1000000]={0}; *//相交点:Y* double Cross\_Len[1000000]={0}; *//交点-射线距离* };**  **->为什么我使用了1000000?**  ***//数组长度由模拟规模决定;因此每次修改模拟规模，这里也要修改  //长度为(对角线长度向上取整)x2+4 (+2其实就够了，我怕出事多加2)  //模拟规模半边长:10,则为34(15x2+4)***  **好吧，这一段注释（我的设计意图）其实是有问题的**，因为一些未知原因（直到我写这个报告时仍未找到原因，但是我找到了解决问题的方法）,我发现它需要比我预想的更多的空间占用，如果你把缓存空间开大一点，就不会出现bug,比如这预设为34(世界大小20x10)的情况，把数组长度拉到160以上，就可以正常运行.如果不加大，就会出现**”穿墙打击”bug**，表现为射线穿过方块破坏了后面的方块,使用Python的matplotlib绘图展示bug如下（爆炸点是(0,0)）:    **绘图程序(Python 3.9.10,第三方库:Matplotlib)**  别用，运行较大规模模拟时效率太低.建议用别的绘图引擎来显示.  **MCExplodeDraw.py**  ***#coding=utf-8 #MC-Explosion绘图展示程序  #Sample Codee #square = plt.Rectangle(xy=(0.2, 0.2), width=0.2, height=0.2, alpha=0.8, angle=0.0) # xy: 左下角位置，width, height：长，宽，angle：逆时针旋转角度，color：设置颜色 #axes1.add\_patch(square) #把图形加载到绘制区域  #square1 = plt.Rectangle(xy=(0.5, 0.5), width=0.2, height=0.4, alpha=0.8, angle=60) # 逆时针旋转60度 #square2 = plt.Rectangle(xy=(0.5, 0.2), width=0.4, height=0.2, alpha=0.8, angle=0.0, linestyle='--') # 虚线边 #axes1.add\_patch(square1) #axes1.add\_patch(square2)* import os import matplotlib.pyplot as plt  def Load\_File(File\_Path):  *"""  读取指定的Block\_File，返回一个处理后的二维列表  每一个列表都包含一个方块的信息......虽然我觉得字典效率更高，但是为了可拓展性，先这样吧  """* with open(File\_Path,"r",encoding="utf-8") as fl:  Out\_List=[line.split() for line in fl.readlines()] *#读取所有数据* return Out\_List  *#基础信息* Path\_Now=os.getcwd() *#获取工作路径  #绘图工作* fig1 = plt.figure(num=1, figsize=(5, 5)) *#确保正方形在屏幕上显示一致，固定figure的长宽相等* axes1 = fig1.add\_subplot(1, 1, 1) *#设置绘图区域* Data\_Blocks=Load\_File(Path\_Now+"\\"+"BlockFile.txt") *#指定BlockFile* for line in Data\_Blocks:  X\_Y=[float(i) for i in line] *#转换为浮点数据用于绘图  #然后逐个绘入图中,最后进行展示* square=plt.Rectangle(xy=(X\_Y[0],X\_Y[1]),width=1, height=1, alpha=0.8, angle=0.0,ec="black")  axes1.add\_patch(square)  xmin=[float(i) for i in Data\_Blocks[0]][0] xmax=[float(i) for i in Data\_Blocks[-1]][0] ymin=[float(i) for i in Data\_Blocks[0]][1] ymax=[float(i) for i in Data\_Blocks[-1]][1]  *#auto 自动设置模式 #plt.xlim(xmin=xmin-0.2\*(xmax-xmin),xmax=xmax+0.2\*(xmax-xmin)) #plt.ylim(ymin=ymin-0.2\*(ymax-ymin),ymax=ymax+0.2\*(ymax-ymin))* plt.xlim(xmin=-120,xmax=120) #请手动修改 plt.ylim(ymin=-120,ymax=20)  plt.grid() *#绘制网格线* plt.show() *#展示图案  #os.system("pause") #如果你想编译成.exe，就把这个注释解除。***  **初始化射线的K值**  **定向爆破（Y方向）**  因为显而易见的数学原因，y=x左右的斜率分布极不均匀，0~1和1~∞的斜率分布区间是相等的，因此如果用随机抽取的话，基本上就是垂直爆破了,因此这段代码被弃用，但是被保留在了我的文件中.  **void GenerateTNT\_Y(Rays\* Raylist,int list\_len,int Ray\_HP){  */\*初始化爆炸物，暂定以原点(0,0)为爆炸中心  \* 因此解析式很简单,y=kx,这里生成一系列的  \* k，代表不同的发射射线  \* 该发射方式使用随机抽取来初始化k  \* Y方向杀伤力很大，X方向较小  \* \*/  // 定义随机数生成器* std::random\_device rd;  std::mt19937 gen(rd());  *//定义范围* std::uniform\_real\_distribution<double> distrib(-list\_len,list\_len);  *//正负均匀分布* for(int i=0;i<list\_len;i++){  Raylist[i].Ray\_K=distrib(gen);  Raylist[i].Ray\_HP=Ray\_HP;**  } }  **圆方程生成法**(但是事实上受到精度限制问题，也只能短程模拟)  不过我看了一眼游戏内的爆炸画面……这个远距离的射线发散问题也是同样存在的，因此**为了尊重游戏代码设计(特性),我决定不对此进行修改**.  另外这个设计真的太适合Cuda加速了,可以轻易地达到微秒级,比CPU快出了几个数量级.    **Minecraft 自定方块:威力=197 爆炸时远距离出现了散射射线的情况**    图中爆炸中心为左下角靠中间,爆炸时外部边缘出现了这样的散射射线条纹.  因此我尊重了这个特性，模拟结果（威力无限大时）效果如下:      **MCExplosion\_CPU.cpp**  **void GenerateTNT\_Round(Rays\* Rays,int list\_len,int Ray\_HP){  */\*初始化爆炸物，暂定以原点(0,0)为爆炸中心  \* 因此解析式很简单,y=kx,这里生成一系列的  \* k，代表不同的发射射线  \* 为了均匀化，模拟球面坐标来分配射线的k  \*即使用圆的方程R^2=X^2+Y^2，来生成k  \* 这样做比较均匀  \* \*/* clock\_t GenTNT\_Start=clock();  double R=100000.0; *//初始化圆的半径；考虑精度问题，使用100000  //事实上最后也是输出Y=kx的斜率组，但是会比上个函数更加均匀* double dx=(double)(2\*R/list\_len); *//步长* for(int i=0;i<list\_len;i++){  double x=-R+i\*dx; *//每一次都更新X的值，以求出对应的y和k* double y=sqrt(pow(R,2)-pow(x,2));  *//由R和x求出对应的y,由方程R^2=X^2+Y^2,然后就可以求出k* double k=y/x; *//求出对应的斜率  //cout<<"x="<<x<<":"<<"y="<<y<<","<<"k="<<k<<endl;  //最后，把k写入数组* Rays[i].Ray\_K=k;  Rays[i].Ray\_HP=Ray\_HP;  }  clock\_t GenTNT\_End=clock();  cout<<"GenerateTNT\_Round:"<<(double)(GenTNT\_End-GenTNT\_Start)/CLOCKS\_PER\_SEC<<endl; }**  **检查K值生成情况(debug工具)**  **一个简单的打印程序……打印K的生成情况.**  **void ExportRayFile\_GenCheck(Rays\* AllRays,int list\_len){  *//导出射线信息文件(用于生成时检查)* cout<<"Start Exporting Files for Rays..."<<endl;  clock\_t Export\_Start=clock();  std::ofstream ofstream;  ofstream.open("RayFile.txt"); *//打开文件用于输出* for(int PrintIndex=0;PrintIndex<list\_len;PrintIndex++){  ofstream<<AllRays[PrintIndex].Ray\_K<<endl;  }  clock\_t Export\_End=clock();  double Export\_Time=(double)(Export\_End-Export\_Start)/CLOCKS\_PER\_SEC;**  **cout<<"Export Dur(Ray):"<<Export\_Time<<endl; }**  **相交判定**  根据我上文所说……检测斜率落在哪个区间，然后就直接执行对应的操作，计算出d(射线发射源和方块的距离)后返回方块和射线Id和距离，这样就可以执行最后的毁伤判定.  对于每一条射线都进行逐条判定(第一层不给方块是因为并发读写时必然出现线程锁的情况，同时也会出现并发写入(多对一)的情况，因此无法使用GPU这样做),不过就算第一层for循环给射线，第二层给方块，也可以获得十几倍的加速效果(因为我认为这个逻辑判定和简单的计算赋值GPU也是没问题的).  如果我把射线量从100增加至1000时，GPU对CPU的加速优势会更加明显(5.7s-6s 变为7.2s-32s)  **void Cross\_Check(Rays\* RaysArray,const Blocks\* BlocksArray,  int Rays\_Len,int Blocks\_Len){  */\*检查射线和方块的相交情况  \* 判定方法:逐个判断射线和每一个方块是否相交  \* 第一层for循环:每一个射线  \* 第二层for循环:判断射线和方块是否相交(根据斜率判定算法,这是配合方块的K值的)  \* 如果相交，则根据k值区间确定相交了哪个面，由此代入方程(x=a/y=b)求解出交点  \* 最后，写入相交交点和距离进Ray的结构体数组中.  \*  \* Rays\_len是为了第一层for循环，配合Rays\*取出数值... (Acc by CUDA...)  \* Blocks\_len是为了第二层for循环，配合Blocks\*取出数值判断...(Acc by CUDA...)  \* \*/* clock\_t Generate\_Start\_Time=clock(); *//记录相交判定的开始时间   //第一层for循环* for(int Ray\_Index=0;Ray\_Index<Rays\_Len;Ray\_Index++){  double Ray\_K\_temp=RaysArray[Ray\_Index].Ray\_K; *//临时缓存K值用于判断* int Ray\_Write\_Index=0; *//初始化指针，用于写入相交数据   //cout<<"Checking Ray:"<<Ray\_Index<<endl;  //cout<<"Ray\_K="<<Ray\_K\_temp<<endl;  /\*这两句注释是因为一个神奇bug  \* 因为C++不允许if(1<x<2)这样的语法，但是奇怪的是,CLion对这种语法不报错  \* 但是运行时爆出了一大堆奇怪bug，包括但不限于爆栈，堆损坏，数值被莫名篡改等  \* 因此，我将下面的代码更正为if(1<x && x<2)这样的格式后就正常工作了，程序正常退出...  \* \*/   //第二大层for循环(判断和方块的关系)* for(int Block\_Index=0;Block\_Index<Blocks\_Len;Block\_Index++){  *//和上文一样，首先你要判断射线和方块的相对关系  //因为不同的相对关系对应了不同的对应意义，所以代码不能被复用    //也是因为这个原因，你需要在方块的结构体里加入"相对位置"这个值，否则这个代码工作不了  //同时，你还要记录"和哪个方块相交，因此还需要记录相交的方块的信息(在Ray结构体里)    //为了代码阅读的一致性,这里的分支和上文保持一致  //正右* if(BlocksArray[Block\_Index].Relative\_Position==0){  if(Ray\_K\_temp>BlocksArray[Block\_Index].K\_Min && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//单值条件:X=定位坐标X,带入y=kx求出y* double fx\_x=(double)(BlocksArray[Block\_Index].Direct\_XY[0]);  double fx\_y=fx\_x\*Ray\_K\_temp;  *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//右上* else if(BlocksArray[Block\_Index].Relative\_Position==1){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//交于下方;固定Y=定位坐标Y,带入y=kx求出x坐标* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//交于左侧;固定x=定位坐标x,带入y=kx,求出y坐标* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0];  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  else{}  }  *//正上(k>0)* else if(BlocksArray[Block\_Index].Relative\_Position==2){  *//大于最小k值，小于99999* if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<=99999){  *//固定y=定位坐标y;根据y=kx求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//正上(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==3){  *//小于最小k值就行* if(Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Min){  *//固定y=定位坐标y，由y=kx求出相交的x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//左上* else if(BlocksArray[Block\_Index].Relative\_Position==4){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//固定y=定位坐标y;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x=定位坐标x+1;求出y* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0]+1;  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//正左(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==5){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x值为定位x值+1;求出y* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0]+1;  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//正左(k>0)* else if(BlocksArray[Block\_Index].Relative\_Position==6) {  if (BlocksArray[Block\_Index].K\_Min <= Ray\_K\_temp && Ray\_K\_temp <= BlocksArray[Block\_Index].K\_Max) {  *//固定x值为定位x值+1;求出y* double fx\_x = (double) BlocksArray[Block\_Index].Direct\_XY[0] + 1;  double fx\_y = fx\_x \* Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }   *//修复一个特殊情况(-1,-1)* if(BlocksArray[Block\_Index].Direct\_XY[0]==-1 && BlocksArray[Block\_Index].Direct\_XY[1]==-1){  if(Ray\_K\_temp>=0){  double fx\_x=0;double fx\_y=0;double fx\_d=0;  RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  }   }  *//左下* else if (BlocksArray[Block\_Index].Relative\_Position == 7) {  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)(BlocksArray[Block\_Index].Direct\_XY[1]+1);  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x=定位坐标x+1;求出y* double fx\_x=(double)(BlocksArray[Block\_Index].Direct\_XY[0]+1);  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  }  *//正上(k>0)* else if(BlocksArray[Block\_Index].Relative\_Position==8){  *//k大于最小值就可以* if(Ray\_K\_temp>=BlocksArray[Block\_Index].K\_Min){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1]+1;  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;   *//cout<<"Re:8,xyd="<<fx\_x<<","<<fx\_y<<" "<<fx\_d<<endl;* }  }  *//正上(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==9){  if(Ray\_K\_temp<=-BlocksArray[Block\_Index].K\_Max){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1]+1;  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;   }  }  *//右下(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==10){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//固定x=定位坐标x;求出y* double fx\_x=BlocksArray[Block\_Index].Direct\_XY[0];  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  *//cout<<"Re:10,XYD="<<fx\_x<<","<<fx\_y<<" "<<fx\_d<<endl;* }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1]+1;  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  *//cout<<"Re:10,XYD="<<fx\_x<<","<<fx\_y<<" "<<fx\_d<<endl;* }  else{}  }  *//正右(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==11){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x=定位坐标x;求出y* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0];  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  }  else{}  }  }   clock\_t Generate\_End\_Time=clock(); *//记录相交判定的结束时间* double Generate\_Time=(double)(Generate\_End\_Time-Generate\_Start\_Time)/CLOCKS\_PER\_SEC;  cout<<"Cross Check Time:"<<Generate\_Time<<"s"<<endl; }**  **打印检查(debug工具)**  根据数据的合法性打印出所有的相交数据,可以进行一些数据比对(配合后面的一个函数使用,通过前后文比对，就知道发生什么错误了)  **void ShowCrossData(Rays\* RaysArray,int Rays\_Len,int Depth,int Width,int ArraySize){  *//检查判定结果(查看射线内部的数据赋值情况)  //同样地，这里的for循环数值要和结构体预留的数组长度一致   //如果修改了模拟规模，这里也要修改* for(int Ray\_Index=0;Ray\_Index<Rays\_Len;Ray\_Index++){  *//第一大层循环:遍历每一条射线* for(int Data\_Index=0;Data\_Index<=1000000;Data\_Index++){**  ***//如果有数据，就打印，没有则忽略  //也是因为这样，初始化结构体数组内的数据为0.  //根据BlockId现场解析出Block坐标(代码来自于GenBlockLikeGPU)* int Left\_Side=-Width/2; *//定位左侧和右侧* int Right\_Side=Width/2-1; *//定位左侧和右侧* int BLock\_Index=RaysArray[Ray\_Index].BlockId[Data\_Index]; *//方块编号* int BLock\_X=Left\_Side+BLock\_Index/abs(Depth);*//使用abs()是为了处理Depth为负数的情况* int BLock\_Y=Depth+BLock\_Index%abs(Depth);   if(abs(RaysArray[Ray\_Index].BlockId[Data\_Index])>ArraySize){  *//清除异常数据* cout<<"Maybe Something Crashed,This is Crash\_Report:"<<endl;  cout<<"Ray\_Index:"<<Ray\_Index<<endl;  cout<<"Block\_Index and XY:"<<RaysArray[Ray\_Index].BlockId<<" "  <<BLock\_X<<","<<BLock\_Y<<endl;  cout<<"Cross\_XYD:"<<RaysArray[Ray\_Index].Cross\_X[Data\_Index]<<" "  <<RaysArray[Ray\_Index].Cross\_Y[Data\_Index]<<" "  <<RaysArray[Ray\_Index].Cross\_Len[Data\_Index]<<" "  <<endl;   RaysArray[Ray\_Index].Cross\_X[Data\_Index]=0;  RaysArray[Ray\_Index].Cross\_Y[Data\_Index]=0;  RaysArray[Ray\_Index].Cross\_Len[Data\_Index]=0;  RaysArray[Ray\_Index].BlockId[Data\_Index]=0;  }   if(RaysArray[Ray\_Index].BlockId[Data\_Index]!=0||  RaysArray[Ray\_Index].Cross\_Len[Data\_Index]>0 *//考虑BlockId=0的情况  //因为一些未知的虚空打击bug，展示时不得不这样做...我还是先把问题数据删了吧* ){  *//打印输出结果* cout<<"Ray:"<<Ray\_Index<<" "  <<"Ray\_K:"<<RaysArray[Ray\_Index].Ray\_K<<" "  <<"Corssed Block:"<<RaysArray[Ray\_Index].BlockId[Data\_Index]<<" "  <<"BlockPosition(DirectXY):"<<BLock\_X<<","<<BLock\_Y<<" "  <<"CrossPosition(Accurate):"<<RaysArray[Ray\_Index].Cross\_X[Data\_Index]<<","  <<RaysArray[Ray\_Index].Cross\_Y[Data\_Index]<<" "  <<"D="<<RaysArray[Ray\_Index].Cross\_Len[Data\_Index]<<" "<<endl;  }  }  } }**  **显示没有被摧毁的方块,并检查debug信息(debug工具)**  **我的程序在debug时遇到了一些灵异事件**，比如就算射线威力设置为无限大，依然会有方块判定不到，无法摧毁，因此我写了一个这样的工具，用于查出没有被摧毁的方块的详细信息.  因此，它可以配合射线威力预设为无限大时使用，检查判定程序是否出了问题.  **void Debug\_For\_No\_Destroryed(Blocks\* BlockArray,Rays\* RayArray,  int BlockArray\_Size,int RayArray\_Size){  *//有一些方块即使射线威力设置为无限大，也没被摧毁，这是为什么呢？  //所以，我打算用这个函数进行debug,检测一下有关信息.   //设置射线威力为无限大后下方应该被全部摧毁，这里收集所有没有被摧毁的方块的信息并进行检索* std::vector<Blocks> Problem\_Blocks; *//用于存储有问题的方块* std::vector<int> Problem\_Blocks\_Ids; *//存储方块Id* for(int i=0;i<BlockArray\_Size;i++){ *//收集有问题的方块并进行诊断* if(BlockArray[i].is\_Destoryed==false) {  Problem\_Blocks.push\_back(BlockArray[i]);  }  }  for(auto it=Problem\_Blocks.begin();it!=Problem\_Blocks.end();it++){  cout<<"Problem Block: "<<it->Direct\_XY[0]<<" "<<it->Direct\_XY[1]  <<" Index= "<<it->Block\_Id<<" "<<"Relative:"<<it->Relative\_Position<<endl; *//问题方块定位坐标  //由定位坐标XY方向定位BlockId:  //检查是否与射线有相交信息* for(int r=0;r<RayArray\_Size;r++){   *//如果要修改模拟规模，这里也要改* for(int rb=0;rb<34;rb++){  if (RayArray[r].BlockId[rb]==it->Block\_Id){  cout<<"Found Cross,This is Cross\_Data:"<<endl;  cout<<"Ray\_Id:"<<r<<" Ray\_K:"<<RayArray[r].Ray\_K<<" "  <<"Cross At:"<<RayArray[r].Cross\_X[rb]<<","<<RayArray[r].Cross\_Y[rb]<<" "  <<"D="<<RayArray[r].Cross\_Len[rb]<<endl;  }  }  }  cout<<"\n\n\n"<<endl;  } }**  因为打印检查中有判断数据是否合法的代码，因此它同样适合于将数据载入到Vector容器中，用于最后一步的摧毁判定.因此我对代码进行了一定的改造，利用Vector容器来存储这些数据.  将相交数据载入Vector容器  **void LoadDataToVector(Rays\* RaysArray,int Rays\_Len,int Depth,int Width,int ArraySize,  std::vector<Cross\_Data>\* Data\_Vector){  *//检查判定结果并将数据写入Vector容器，用于最后的判定  //同样地，这里的for循环数值要和结构体预留的数组长度一致  //如果修改了模拟规模，这里也要修改* for(int Ray\_Index=0;Ray\_Index<Rays\_Len;Ray\_Index++){  *//第一大层循环:遍历每一条射线* for(int Data\_Index=0;Data\_Index<=1000000;Data\_Index++){  *//如果有数据，就打印，没有则忽略  //也是因为这样，初始化结构体数组内的数据为0.* if(abs(RaysArray[Ray\_Index].BlockId[Data\_Index])>ArraySize){  *//清除异常数据* RaysArray[Ray\_Index].Cross\_X[Data\_Index]=0;  RaysArray[Ray\_Index].Cross\_Y[Data\_Index]=0;  RaysArray[Ray\_Index].Cross\_Len[Data\_Index]=0;  RaysArray[Ray\_Index].BlockId[Data\_Index]=0;  }   *//好吧，我觉得不全为0就行.* if(RaysArray[Ray\_Index].BlockId[Data\_Index]!=0||  RaysArray[Ray\_Index].Cross\_Len[Data\_Index]>0){  *//因为一些未知的虚空打击bug，展示时不得不这样做...我还是先把问题数据删了吧   //将数据塞入vector容器* Data\_Vector->push\_back({Ray\_Index,  RaysArray[Ray\_Index].BlockId[Data\_Index],  RaysArray[Ray\_Index].Cross\_Len[Data\_Index]});  }  }  }  *//给容器排个序(按照D从小到大排序，后面直接顺序判定即可）* sort(Data\_Vector->begin(),Data\_Vector->end(), Sort\_Vector);   *//cout<<"Sorted:"<<endl;  //Print\_Vector(Data\_Vector);* }**  **配套代码:Vector容器按照d排序(用于最后的判定)**  **#include<algorithm> //纯C++不用导入，编译器(VS)自动优化，但是NVCC需要**  **bool Sort\_Vector(const Cross\_Data &a,const Cross\_Data &b){  *//按照D的大小排序* return a.D<b.D; *//从小到大排序* }**  **Vector容器存储的结构体**  **struct Cross\_Data{  *//存储射线和方块的相交数据，D越短表示越早相交  //最后按照D从小到大排序，逐个判定即可* int Ray\_Id; *//射线Id* int Block\_Id; *//方块Id* double D; *//射线和方块的距离* };**  **配套代码:打印Vector内容(debug工具)**  **void Print\_Vector(std::vector<Cross\_Data>\* Data\_Vector){  for(auto it=Data\_Vector->begin();it!=Data\_Vector->end();it++){  cout<<"Ray\_Id:"<<it->Ray\_Id<<" "  <<"Block\_Id:"<<it->Block\_Id<<" "  <<"D:"<<it->D<<endl;  } }**  用于判定或者载入异常时就可以用这个来debug.  一个debug方法是将输出数据的控制台的信息Ctrl+A全选粘贴到txt文件中，测量行数就知道数据是否被全部载入Vector容器了(配合上文的debug工具使用)  **执行摧毁**  根据前文得到的数据逐个相互判定扣血，即可将被摧毁的方块标注为已摧毁，这样打印时就可以打印出模拟结果了.判定逻辑很简单吗，不要犯顺序错误就行.  因为并发读写的问题，我不觉得这不具备用Cuda加速的意义.  **void Perform\_Destruction(Rays\* RaysArray,Blocks\* BlocksArray,  std::vector<Cross\_Data>\* Data\_Vector){  *//逐个读取Vector的数据并判定执行方块摧毁......  //Vector已经排序好了,逐个判定就行了.* clock\_t Destruction\_Start=clock();  for(auto it=Data\_Vector->begin();it!=Data\_Vector->end();it++){  int Temp\_Ray=it->Ray\_Id;  int Temp\_BLock=it->Block\_Id;  *//先判断是否存在(Block/Ray HP>0)   //cout<<"Comparing Data:Ray: "<<Temp\_Ray<<" HP= "<<RaysArray[Temp\_Ray].Ray\_HP  // <<" Block: "<<Temp\_BLock<<" HP= "<<BlocksArray[Temp\_BLock].Block\_HP<<endl;* if(RaysArray[Temp\_Ray].Ray\_HP>0 && BlocksArray[Temp\_BLock].Block\_HP>0){  *//再判断血量关系  //Ray>Block,执行摧毁,标记为摧毁.* if(RaysArray[Temp\_Ray].Ray\_HP>BlocksArray[Temp\_BLock].Block\_HP){  *//注意，一定是先扣血，再标记为0，否则会变成无损耗.* RaysArray[Temp\_Ray].Ray\_HP-=(BlocksArray[Temp\_BLock].Block\_HP); *//扣血* BlocksArray[Temp\_BLock].Block\_HP=0; *//血量标记为0* BlocksArray[Temp\_BLock].is\_Destoryed=true; *//标记为已摧毁* }  *//Ray=Block,同归于尽* else if(RaysArray[Temp\_Ray].Ray\_HP==BlocksArray[Temp\_BLock].Block\_HP){  RaysArray[Temp\_Ray].Ray\_HP=0; *//标记为0（能量耗尽)* BlocksArray[Temp\_BLock].Block\_HP=0; *//血量标记为0* BlocksArray[Temp\_BLock].is\_Destoryed=true; *//标记为已摧毁* }  else if(RaysArray[Temp\_Ray].Ray\_HP<BlocksArray[Temp\_BLock].Block\_HP){  *//注意，一定是先扣血，再标记为0，否则会变成无损耗.* BlocksArray[Temp\_BLock].Block\_HP-=(RaysArray[Temp\_Ray].Ray\_HP); *//扣血* RaysArray[Temp\_Ray].Ray\_HP=0; *//标记为0（能量耗尽)* }  }  *//debug用* else{  if(BlocksArray[it->Block\_Id].Block\_HP==0){}*//cout<<"Not\_Performed For BlockHP=0:"<<" "<<it->Block\_Id<<" "<<it->Ray\_Id<<" "<<it->D<<endl;* else if(RaysArray[it->Ray\_Id].Ray\_HP==0){}*//cout<<"Not\_Performed For RayHP=0:"<<" "<<it->Block\_Id<<" "<<it->Ray\_Id<<" "<<it->D<<endl;* else cout<<"Not\_Performed For Something Crashed:"<<" "<<it->Block\_Id<<" "<<it->Ray\_Id<<" "<<it->D<<endl;  }  }   clock\_t Destruction\_End=clock();  double Destruction\_Time=(double)(Destruction\_End-Destruction\_Start)/CLOCKS\_PER\_SEC;  cout<<"Destruction Dur(Block):"<<Destruction\_Time<<endl; }**  **核心流程**  **int main(){  system("chcp 65001"); *//UTF-8 ON* cout<<"C++Program:Minecraft Explosion"<<endl;   *//爆炸中心坐标(该程序只设定一个爆炸中心)* int Explode\_Postion[2]={0,0}; *//设定爆炸位置   //方块  //这里为了方便，确定顶点坐标,框选区域为方块* int World\_Size\_LD[2]={-2000,-2000}; *//左下角* int World\_Size\_RT[2]={1999,-1}; *//右上角* int All\_Blocks\_Number=(World\_Size\_RT[0]-World\_Size\_LD[0]+1)\*  (World\_Size\_RT[1]-World\_Size\_LD[1]+1);  cout<<"Data Length:"<<All\_Blocks\_Number<<endl;  *//方块的总数量* cout<<"DataSize:"<<sizeof(Blocks)<<endl; *//Blocks结构体大小* size\_t All\_Blocks\_Size=sizeof(Blocks)\*All\_Blocks\_Number; *//计算需要分配的空间* cout<<"NEW:"<<All\_Blocks\_Size<<endl; *//debug   //生成方块数据数组，用于存放方块信息* Blocks\* All\_Blocks=new Blocks[All\_Blocks\_Size]; *//分配空间:方块*** ***//GenerateBlocks("Minecraft Stone",10,All\_Blocks,  // World\_Size\_LD,World\_Size\_RT,Explode\_Postion);* GenerateBlocksLikeGPU("Minecraft Stone",10,All\_Blocks,  All\_Blocks\_Number,World\_Size\_LD[1],  World\_Size\_RT[0]-World\_Size\_LD[0]+1,Explode\_Postion);   *//射线  //生成射线斜率数组，代表射线的斜率* int All\_Rays\_len=500;  Rays\* All\_Rays=new Rays[All\_Rays\_len]; *//分配空间:射线* GenerateTNT\_Round(All\_Rays,All\_Rays\_len,200);   *//生成时输出的检查文件* ExportRayFile\_GenCheck(All\_Rays,All\_Rays\_len);   *//判断碰撞关系* Cross\_Check(All\_Rays,All\_Blocks,All\_Rays\_len,All\_Blocks\_Number);  *//ShowCrossData(All\_Rays,All\_Rays\_len,World\_Size\_LD[1],World\_Size\_RT[0]-World\_Size\_LD[0]+1,All\_Blocks\_Number);   //创建相交数据存储用Vector容器，准备进行最后的射线判定* std::vector<Cross\_Data> Cross\_Data\_Vector;  LoadDataToVector(All\_Rays,All\_Rays\_len,World\_Size\_LD[1],World\_Size\_RT[0]-World\_Size\_LD[0]+1,  All\_Blocks\_Number,&Cross\_Data\_Vector);  sort(Cross\_Data\_Vector.begin(),Cross\_Data\_Vector.end(), Sort\_Vector);  *//容器排序   //debug  //Print\_Vector(&Cross\_Data\_Vector);   //射线判定* Perform\_Destruction(All\_Rays,All\_Blocks,&Cross\_Data\_Vector);   *//摧毁后检查  //ExportBlockFile(All\_Blocks,All\_Blocks\_Number);   //Debug:异常判定处理*** ***//Debug\_For\_No\_Destroryed(All\_Blocks,All\_Rays,All\_Blocks\_Number,All\_Rays\_len);* delete[] All\_Blocks; *//释放方块数据的内存* delete[] All\_Rays; *//释放射线数据的内存* system("pause");  return 0; }**  **>>>Acced By CUDA (Cuda Runtime:代码标准:C++ 17,编译工具包:NVCC)**  代码的可加速点有:生成方块（混合加速），生成射线（纯GPU加速）,相交判定(射线视角),以下是使用Cuda的加速代码:  加速的原理其实很简单，把简单的i++类型的for循环分配给不同的线程后，利用多核心和多线程来操作不同的数据，这样可以大大提高效率.  **MCExplosion\_CUDA.cu**  完全GPU加速:  **\_\_global\_\_ void GenerateTNT\_Round(Rays\* Rays,int list\_len,int Ray\_HP){  */\*初始化爆炸物，生成时以原点(0,0)为爆炸中心  \* 因此解析式很简单,y=kx,这里生成一系列的  \* k，代表不同的发射射线  \* 为了均匀化，模拟球面坐标来分配射线的k  \*即使用圆的方程R^2=X^2+Y^2，来生成k  \* 这样做比较均匀  \* \*/  //CUDA配置* int Index=threadIdx.x+blockIdx.x\*blockDim.x;  int stride=blockDim.x\*gridDim.x;   double R=100.0; *//初始化圆的半径；考虑精度问题，使用100* double dx=2\*R/list\_len; *//步长* for(int i=Index;i<list\_len;i+=stride){  double x=-R+i\*dx; *//每一次都更新X的值，以求出对应的y和k* double y=sqrt(pow(R,2)-pow(x,2));  *//由R和x求出对应的y,由方程R^2=X^2+Y^2,然后就可以求出k* double k=y/x; *//求出对应的斜率  //cout<<"x="<<x<<":"<<"y="<<y<<","<<"k="<<k<<endl;  //最后，把k写入数组* Rays[i].Ray\_K=k;  Rays[i].Ray\_HP=Ray\_HP;  } }**  部分加速:如果你直接把任务全部丢给GPU，此时会加速高达-75%,还不如纯CPU任务了,术业有专攻，重逻辑推断+计算的部分交给CPU来做，GPU做一些简单的初始化即可.  **“这也充分证明了程序的可加速性和GPU的能力是有限的.正如第一节课所说.”**  **\_\_global\_\_ void GenerateBlocksIndex(int BlockHP,Blocks\* Block\_Array,int ArraySize,  int Depth,int Width,int\* TNTPosition) {  */\*初始化二维的方块世界;  \* 暂时规定，方块的定位坐标是左下角，尺寸全部为1x1;  \* 也就是(0,0)对应的方块的四个坐标点（从左往右，从上到下)是  \* (1,0),(1,1),(0,0),(0,1)  \* \*/* int Left\_Side = -Width / 2; *//定位左侧  //int Right\_Side=Width/2-1; //定位右侧* int Index = threadIdx.x + blockIdx.x \* blockDim.x;  int stride = blockDim.x \* gridDim.x;  for (int i = Index; i < ArraySize; i += stride) {  Block\_Array[i].Block\_HP = BlockHP;  Block\_Array[i].Block\_Id = i;  *//在CPU程序里，我是先定位X坐标再定位Y坐标，生成方式是X从左往右,Y从下往上  //求X的方式是用第n个的索引除去深度（Y的长度）再加上X的起点坐标(Left\_Side)，就可以获得X的位置  //获得X的位置后，就知道走过了几个列，此时减去之前走过的列的量，就可以知道我现在的深度在哪里* Block\_Array[i].Direct\_XY[0] = Left\_Side + i / abs(Depth);  Block\_Array[i].Direct\_XY[1] = Depth + i % abs(Depth);  *//printf("i=%d,X=%d,Y=%d\n",i,BlockArray[i].Direct\_XY[0],BlockArray[i].Direct\_XY[1]);* } }**  给GPU干这样的活还是比较简单的.  这个部分仍然使用CPU：  **void GenerateBlocksDataOnCPU(int BlockHP,Blocks\* Block\_Array,  int ArraySize,int Depth,int Width,int\* TNTPosition){  clock\_t GenBlock\_Start=clock();  for(int i=0;i<ArraySize;i++){  *//判断相对位置(相对于爆炸点)* int Relative\_Position[2]={  Block\_Array[i].Direct\_XY[0]-TNTPosition[0]  *//x(方块定位坐标）-x(爆炸点),判断是左(<0)还是右(>0)* ,  Block\_Array[i].Direct\_XY[1]-TNTPosition[1]  *//y(方块定位坐标)-y(爆炸点),判断是下(<0)还是上(>0)* };   *//方块的另外几个坐标  //先生成由定位坐标来确定的其他三个点的坐标值  //左上点:X=X,Y=Y+1* int DirectXY\_LT[2]={Block\_Array[i].Direct\_XY[0],  Block\_Array[i].Direct\_XY[1]+1};  *//右上点:X=X+1,Y=Y+1* int DirectXY\_RT[2]={Block\_Array[i].Direct\_XY[0]+1,  Block\_Array[i].Direct\_XY[1]+1};  *//右下点:X=X+1,Y=Y* int DirectXY\_RD[2]={Block\_Array[i].Direct\_XY[0]+1,  Block\_Array[i].Direct\_XY[1]};   *//判断分支  //正右(k>0)* if(Relative\_Position[0]>0 && Relative\_Position[1]>0 && abs(Relative\_Position[1])<=1){  *//k最小值为定位坐标点;k最大值为左上;没有中间k值* Block\_Array[i].Relative\_Position=0;   Block\_Array[i].K\_Min=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  }  *//右上(k>0)/正上(k>0)(需要计算中判定)* else if(Relative\_Position[0]>0 && Relative\_Position[1]>0){  *//k最小值为右下;k中间值为定位坐标点;k最大为左上* Block\_Array[i].Relative\_Position=1;   Block\_Array[i].K\_Min=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);   *//正上(k>0)的情况需要计算上式来判定* if(isinf(Block\_Array[i].K\_Mid) && isinf(Block\_Array[i].K\_Max)){  Block\_Array[i].Relative\_Position=2;  }  }  *//正上(k<0)/左上(k<0)(正上需要计算中判定)* else if(Relative\_Position[0]<0 && Relative\_Position[1]>0 && abs(Relative\_Position[1])>1){  *//先判断左上和正上，排除正左的情况;  //对于左上，k最小值为左下;k中间为右下;k最大为右上(注意负数)* Block\_Array[i].Relative\_Position=4;   Block\_Array[i].K\_Min=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);   *//正上(k<0)的情况需要计算上式来确定* if(isinf(Block\_Array[i].K\_Mid) && isinf(Block\_Array[i].K\_Max)){  Block\_Array[i].Relative\_Position=3;  }  }  *//正左(k<0)* else if(Relative\_Position[0]<0 && Relative\_Position[1]>0 && abs(Relative\_Position[1])<=1){  *//k最小值为右下;k最大值为右上;没有中间k值* Block\_Array[i].Relative\_Position=5;   Block\_Array[i].K\_Min=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  }  *//正左(k>0)* else if(Relative\_Position[0]<0 && Relative\_Position[1]<0 && abs(Relative\_Position[1])<=1){  *//k最小值为右上;k最大值为右下;没有中间k值* Block\_Array[i].Relative\_Position=6;   Block\_Array[i].K\_Min=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);  }  *//左下(k>0),正上(k>0)* else if(Relative\_Position[0]<0 && Relative\_Position[1]<0){  *//先按照左下来判定;k最小值为左上;k中间为右上;k最大为右下* Block\_Array[i].Relative\_Position=7;   Block\_Array[i].K\_Min=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RD[1]-TNTPosition[1])/(DirectXY\_RD[0]-TNTPosition[0]);   if(isinf(Block\_Array[i].K\_Mid) && isinf(Block\_Array[i].K\_Max)){  *//正上方(k>0)* Block\_Array[i].Relative\_Position=8;  }  }   *//补充:x=0的情况（正上)* else if(Relative\_Position[0]==0 && Relative\_Position[1]<0){  Block\_Array[i].Relative\_Position=9;   *//最大：垂直(-无限，这里给-99999),最小:右侧.无中间k值* Block\_Array[i].K\_Min=-99999;  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  }   *//正右(k<0)* else if(Relative\_Position[0]>0 && Relative\_Position[1]<0 && abs(Relative\_Position[1])<=1){  *//k最小值为定位点;k最大值为左上;没有中间k值* Block\_Array[i].Relative\_Position=11;   Block\_Array[i].K\_Min=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/  (Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  }   *//正上(k<0),右下* else if(Relative\_Position[0]>0 && Relative\_Position[1]<0){  *//先按照右下来判定;(注意负数)k最小为定位点;k中间为左上;k最大为右上* Block\_Array[i].Relative\_Position=10;   Block\_Array[i].K\_Min=(double)  (Block\_Array[i].Direct\_XY[1]-TNTPosition[1])/(Block\_Array[i].Direct\_XY[0]-TNTPosition[0]);  Block\_Array[i].K\_Mid=(double)  (DirectXY\_LT[1]-TNTPosition[1])/(DirectXY\_LT[0]-TNTPosition[0]);  Block\_Array[i].K\_Max=(double)  (DirectXY\_RT[1]-TNTPosition[1])/(DirectXY\_RT[0]-TNTPosition[0]);  }  else{}  }   *//接下来，修复nan(ind)的bug(-1,-1直接怼脸爆炸源了)(替换为0)* for(int i=0;i<ArraySize;i++){  *//cout<<"Checking"<<i<<endl;* if(isnan(Block\_Array[i].K\_Min)){  Block\_Array[i].K\_Min=-99999;  }  if(isnan(Block\_Array[i].K\_Mid)){  Block\_Array[i].K\_Mid=0;  }  if(isnan(Block\_Array[i].K\_Max)){  Block\_Array[i].K\_Max=99999;  }  }   */\*  for(int i=0;i<All\_Blocks\_Number;i++){  cout<<"Name:"<<Block\_Array[i].Block\_Name<<" "  <<"HP:"<<Block\_Array[i].Block\_HP<<" "  <<"X:"<<Block\_Array[i].Direct\_XY[0]<<" "  <<"Y:"<<Block\_Array[i].Direct\_XY[1]<<" "  <<"Relative\_position:"<<Block\_Array[i].Relative\_Position<<" "  <<"K-To-(0,0)-KMin:"<<Block\_Array[i].K\_Min<<" "  <<"K-To-(0,0)-KMid:"<<Block\_Array[i].K\_Mid<<" "  <<"K-To-(0,0)-KMax:"<<Block\_Array[i].K\_Max<<" "  <<endl;  //debug:用于检查  }  \*/* clock\_t GenBlock\_End=clock();  cout<<"CPU\_GenBlock\_Time:"<<(double)((GenBlock\_End-GenBlock\_Start)/CLOCKS\_PER\_SEC)<<endl; }**  上述代码组合使用，初始化2000x1000的方块时，从纯CPU的32s-35s加速到3s完成(10x+)；如果你扩大模拟规模，这会更快,并且，这可以大幅度降低空间开销,得益于混合编程的NVCC编译器和较新版的显卡驱动（我发现老版本显卡驱动(531.68)的显存占用会比较大）,  这是纯C++(CPU运行)时的性能开销(不知道还以为是Python魅力时刻):    如果将射线数量增加到1000条，会直接吃满32GB，下方的”已提交”会增加到76GB  然后…….    **但是如果你使用NVCC编译器进行混合编程，协作运行:**    此时内存的占用会大幅度降低，并且显存开销也不大.  因此，使用CUDA的另一个意义（第一个意义是加速）,可以提供更好的空间占用优化，提示更好的性能，更大的模拟规模等.  同时，CPU因为被显卡分担了很多简单的任务，也可以专心用于计算和逻辑判断任务，这使得CPU也可以高效地运行.  **完全GPU加速:相交判定(我只改了黄色部分)**  加速效果由射线条数直接决定;100条时加速效果可以被视为误差;1000条时会直接快4倍,条数越多，加速效果越好.  **\_\_global\_\_ void Cross\_Check(Rays\* RaysArray,Blocks\* BlocksArray,int Rays\_Len,int Blocks\_Len){  */\*检查射线和方块的相交情况  \* 判定方法:逐个判断射线和每一个方块是否相交  \* 第一层for循环:每一个射线  \* 第二层for循环:判断射线和方块是否相交(根据斜率判定算法,这是配合方块的K值的)  \* 如果相交，则根据k值区间确定相交了哪个面，由此代入方程(x=a/y=b)求解出交点  \* 最后，写入相交交点和距离进Ray的结构体数组中.  \*  \* Rays\_len是为了第一层for循环，配合Rays\*取出数值... (Acc by CUDA...)  \* Blocks\_len是为了第二层for循环，配合Blocks\*取出数值判断...(Acc by CUDA...)  \* \*/   //Cuda\_Runtime配置* int Index=threadIdx.x+blockIdx.x\*blockDim.x;  int stride=blockDim.x\*gridDim.x;   *//第一层for循环(使用CUDA加速)* for(int Ray\_Index=Index;Ray\_Index<Rays\_Len;Ray\_Index+=stride) {  double Ray\_K\_temp = RaysArray[Ray\_Index].Ray\_K; *//临时缓存K值用于判断* int Ray\_Write\_Index = 0; *//初始化指针，用于写入相交数据   //cout<<"Checking Ray:"<<Ray\_Index<<endl;  //cout<<"Ray\_K="<<Ray\_K\_temp<<endl;  /\*这两句注释是因为一个神奇bug  \* 因为C++不允许if(1<x<2)这样的语法，但是奇怪的是,CLion对这种语法不报错  \* 但是运行时爆出了一大堆奇怪bug，包括但不限于爆栈，堆损坏，数值被莫名篡改等  \* 因此，我将下面的代码更正为if(1<x && x<2)这样的格式后就正常工作了，程序正常退出...  \* \*/   //第二大层for循环(判断和方块的关系)* for(int Block\_Index=0;Block\_Index<Blocks\_Len;Block\_Index++){  *//和上文一样，首先你要判断射线和方块的相对关系  //因为不同的相对关系对应了不同的对应意义，所以代码不能被复用    //也是因为这个原因，你需要在方块的结构体里加入"相对位置"这个值，否则这个代码工作不了  //同时，你还要记录"和哪个方块相交，因此还需要记录相交的方块的信息(在Ray结构体里)    //为了代码阅读的一致性,这里的分支和上文保持一致  //正右* if(BlocksArray[Block\_Index].Relative\_Position==0){  if(Ray\_K\_temp>BlocksArray[Block\_Index].K\_Min && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//单值条件:X=定位坐标X,带入y=kx求出y* double fx\_x=(double)(BlocksArray[Block\_Index].Direct\_XY[0]);  double fx\_y=fx\_x\*Ray\_K\_temp;  *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//右上* else if(BlocksArray[Block\_Index].Relative\_Position==1){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//交于下方;固定Y=定位坐标Y,带入y=kx求出x坐标* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//交于左侧;固定x=定位坐标x,带入y=kx,求出y坐标* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0];  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  else{}  }  *//正上(k>0)* else if(BlocksArray[Block\_Index].Relative\_Position==2){  *//大于最小k值，小于99999* if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<=99999){  *//固定y=定位坐标y;根据y=kx求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//正上(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==3){  *//小于最小k值就行* if(Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Min){  *//固定y=定位坐标y，由y=kx求出相交的x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//左上* else if(BlocksArray[Block\_Index].Relative\_Position==4){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//固定y=定位坐标y;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1];  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x=定位坐标x+1;求出y* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0]+1;  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//正左(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==5){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x值为定位x值+1;求出y* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0]+1;  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d=sqrt(pow(fx\_x,2)+pow(fx\_y,2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index]=fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index]=fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index]=fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index]=Block\_Index;  Ray\_Write\_Index++;  }  }  *//正左(k>0)* else if(BlocksArray[Block\_Index].Relative\_Position==6) {  if (BlocksArray[Block\_Index].K\_Min <= Ray\_K\_temp && Ray\_K\_temp <= BlocksArray[Block\_Index].K\_Max) {  *//固定x值为定位x值+1;求出y* double fx\_x = (double) BlocksArray[Block\_Index].Direct\_XY[0] + 1;  double fx\_y = fx\_x \* Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }   *//修复一个特殊情况(-1,-1)* if(BlocksArray[Block\_Index].Direct\_XY[0]==-1 && BlocksArray[Block\_Index].Direct\_XY[1]==-1){  if(Ray\_K\_temp>=0){  double fx\_x=0;double fx\_y=0;double fx\_d=0;  RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  }   }  *//左下* else if (BlocksArray[Block\_Index].Relative\_Position == 7) {  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)(BlocksArray[Block\_Index].Direct\_XY[1]+1);  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x=定位坐标x+1;求出y* double fx\_x=(double)(BlocksArray[Block\_Index].Direct\_XY[0]+1);  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  }  *//正上(k>0)* else if(BlocksArray[Block\_Index].Relative\_Position==8){  *//k大于最小值就可以* if(Ray\_K\_temp>=BlocksArray[Block\_Index].K\_Min){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1]+1;  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;   *//cout<<"Re:8,xyd="<<fx\_x<<","<<fx\_y<<" "<<fx\_d<<endl;* }  }  *//正上(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==9){  if(Ray\_K\_temp<=-BlocksArray[Block\_Index].K\_Max){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1]+1;  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;   }  }  *//右下(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==10){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<BlocksArray[Block\_Index].K\_Mid){  *//固定x=定位坐标x;求出y* double fx\_x=BlocksArray[Block\_Index].Direct\_XY[0];  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  *//cout<<"Re:10,XYD="<<fx\_x<<","<<fx\_y<<" "<<fx\_d<<endl;* }  else if(BlocksArray[Block\_Index].K\_Mid<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定y=定位坐标y+1;求出x* double fx\_y=(double)BlocksArray[Block\_Index].Direct\_XY[1]+1;  double fx\_x=fx\_y/Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  *//cout<<"Re:10,XYD="<<fx\_x<<","<<fx\_y<<" "<<fx\_d<<endl;* }  else{}  }  *//正右(k<0)* else if(BlocksArray[Block\_Index].Relative\_Position==11){  if(BlocksArray[Block\_Index].K\_Min<=Ray\_K\_temp && Ray\_K\_temp<=BlocksArray[Block\_Index].K\_Max){  *//固定x=定位坐标x;求出y* double fx\_x=(double)BlocksArray[Block\_Index].Direct\_XY[0];  double fx\_y=fx\_x\*Ray\_K\_temp;   *//根据D=(x^2+y^2)\*\*0.5求出d，写入数组* double fx\_d = sqrt(pow(fx\_x, 2) + pow(fx\_y, 2));   *//写入结构体数组* RaysArray[Ray\_Index].Cross\_X[Ray\_Write\_Index] = fx\_x;  RaysArray[Ray\_Index].Cross\_Y[Ray\_Write\_Index] = fx\_y;  RaysArray[Ray\_Index].Cross\_Len[Ray\_Write\_Index] = fx\_d;  RaysArray[Ray\_Index].BlockId[Ray\_Write\_Index] = Block\_Index;  Ray\_Write\_Index++;  }  }  else{}  }  } }**  **CUDA版本主程序流程**  **int main(){  system("chcp 65001"); *//UTF-8 ON* cout<<"CUDA Program:Minecraft Explosion"<<endl;   */\*  //配置CUDA环境  int deivceId;  int Number\_Of\_SMs;   cudaGetDevice(&deivceId);  cudaDeviceGetAttribute(&Number\_Of\_SMs, cudaDevAttrMultiProcessorCount, deivceId);  //获取设备Id和SM数   cudaSetDevice(0);  cudaError\_t cudaStatus;  //Choose which GPU to run on, change this on a multi-GPU system.  cudaStatus = cudaSetDevice(0);  if (cudaStatus != cudaSuccess) {  fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");  }  else cout<<"OK,Device Id:"<<deivceId<<endl;   size\_t Threads\_Per\_Block=1024;  size\_t Number\_Of\_Blocks=Number\_Of\_SMs\*32;   \*/   //设置线程数和线程块数* size\_t Threads\_Per\_Block=1024;  size\_t Number\_Of\_Blocks=896;   *//定位世界坐标* int World\_Size\_LD[2]={-2000,-2000}; *//左下角* int World\_Size\_RT[2]={1999,-1}; *//右上角* int World\_Depth=World\_Size\_LD[1]; *//维度深度* int World\_Width=World\_Size\_RT[0]-World\_Size\_LD[0]+1; *//维度宽度   //定义爆炸点* int\* Explosion\_Point; *//爆炸位置* cudaMallocManaged(&Explosion\_Point,sizeof(int)\*2);  Explosion\_Point[0]=0;Explosion\_Point[1]=0;   *//定义方块HP* int Block\_HP=100;   Blocks\* All\_Blocks; *//存放方块数据* int All\_Blocks\_Number=(World\_Size\_RT[0]-World\_Size\_LD[0]+1)\*(World\_Size\_RT[1]-World\_Size\_LD[1]+1);  cout<<"Data Length:"<<All\_Blocks\_Number<<endl;  *//方块的总数量* cout<<"DataSize:"<<sizeof(Blocks)<<endl; *//Blocks结构体大小* size\_t All\_Blocks\_Size=sizeof(Blocks)\*All\_Blocks\_Number; *//计算需要分配的空间* cout<<"Mallocing:"<<All\_Blocks\_Size<<endl; *//debug* cudaMallocManaged(&All\_Blocks,All\_Blocks\_Size); *//分配空间  //cudaMemPrefetchAsync(All\_Blocks,All\_Blocks\_Size,deivceId); //预取入GPU* cudaError\_t GenErr;  cudaError\_t asyncErr;   clock\_t Start\_Time=clock();  GenerateBlocksIndex<<<Number\_Of\_Blocks,Threads\_Per\_Block>>>(  Block\_HP,All\_Blocks,All\_Blocks\_Number,  World\_Depth,World\_Width,Explosion\_Point);  cudaDeviceSynchronize();   GenErr = cudaGetLastError();  if(GenErr != cudaSuccess) printf("GenError: %s\n", cudaGetErrorString(GenErr));   clock\_t End\_Time=clock();  double Duration=(double)(End\_Time-Start\_Time)/CLOCKS\_PER\_SEC;  cout<<"GPU-GenTime:"<<Duration<<"s"<<endl;   asyncErr = cudaDeviceSynchronize();  if(asyncErr != cudaSuccess) printf("AsyncError: %s\n", cudaGetErrorString(asyncErr));   GenerateBlocksDataOnCPU(Block\_HP,All\_Blocks,All\_Blocks\_Number,  World\_Depth,World\_Width,Explosion\_Point);   *//爆炸射线* int All\_Rays\_len=500; *//发射100条爆炸射线* int Ray\_HP=200; *//定义射线威力* Rays\* All\_Rays; *//分配空间:射线* size\_t Ray\_Size=sizeof(Rays)\*All\_Rays\_len;  cudaMallocManaged(&All\_Rays,Ray\_Size); *//cudaMalloc* GenerateTNT\_Round<<<Number\_Of\_Blocks,Threads\_Per\_Block>>>(All\_Rays,All\_Rays\_len,Ray\_HP);  cudaDeviceSynchronize();   ExportRayFile\_GenCheck(All\_Rays,All\_Rays\_len);   *//CHECK  /\*  for(int i=0;i<All\_Blocks\_Number;i++){  cout<<"HP:"<<All\_Blocks[i].Block\_HP<<" "  <<"Relative\_Postion:"<<All\_Blocks[i].Relative\_Position<<" "  <<"X:"<<All\_Blocks[i].Direct\_XY[0]<<" "  <<"Y:"<<All\_Blocks[i].Direct\_XY[1]<<" "  <<"K-To-(0,1)-KMin:"<<All\_Blocks[i].K\_Min<<" "  <<"K-To-(0,1)-KMid:"<<All\_Blocks[i].K\_Mid<<" "  <<"K-To-(0,1)-KMax:"<<All\_Blocks[i].K\_Max<<" "  <<endl;  }  \*/* cudaError\_t Cross\_Check\_Err;  Cross\_Check<<<Number\_Of\_Blocks,Threads\_Per\_Block>>>(  All\_Rays,All\_Blocks,All\_Rays\_len,All\_Blocks\_Number);  cudaDeviceSynchronize();  Cross\_Check\_Err=cudaGetLastError();  if(Cross\_Check\_Err!=cudaSuccess)cout<<"Cross\_Check\_Err:"<<cudaGetErrorString(Cross\_Check\_Err)<<endl;   *//ShowCrossData(All\_Rays,All\_Rays\_len,World\_Size\_LD[1],  // World\_Size\_RT[0]-World\_Size\_LD[0]+1,All\_Blocks\_Number);   //创建相交数据存储用Vector容器，准备进行最后的射线判定* std::vector<Cross\_Data> Cross\_Data\_Vector;  LoadDataToVector(All\_Rays,All\_Rays\_len,World\_Size\_LD[1],World\_Size\_RT[0]-World\_Size\_LD[0]+1,  All\_Blocks\_Number,&Cross\_Data\_Vector);  *//Print\_Vector(&Cross\_Data\_Vector);   //射线判定* Perform\_Destruction(All\_Rays,All\_Blocks,&Cross\_Data\_Vector);   *//摧毁后检查  //ExportBlockFile(All\_Blocks,All\_Blocks\_Number);   //摧毁后展示  //ExportBlockFile(All\_Blocks,All\_Blocks\_Number);   //Debug:异常判定处理  //Debug\_For\_No\_Destroryed(All\_Blocks,All\_Rays,All\_Blocks\_Number,All\_Rays\_len);* cudaFree(Explosion\_Point);  cudaFree(All\_Blocks);  cudaFree(All\_Rays);   system("pause");  return 0; }**  总结  GPU,NPU,XPU等新的计算单元正在逐渐进入人们的视线，并且会伴随着未来厂商的更新和推广进入人们的生活,2023-12-16日，Intel Core Ultra正式上市会直接集成NPU单元，实现人们远程AI终端的梦想;AMD的搭载Ryzen AI模块的8040系列处理器也将在2024年1月的CES上问世.除了集成显卡外，NPU也进入了个人终端.这确实是个好事，标志着AI时代的正式到来，同时高性能计算的路也会越走越广(本质上，还是处理不同类型计算的单元).  我不认为Cuda是护城河,相反，Cuda也许是引领程序员走向未来的钥匙.同时，地球之大，容得下NVIDIA,AMD等一众公司,也不止Cuda可以用于计算;现在我们处于一个万物竞发的AI起源时代.现在一些高校已经将cuda作为选修课进入了教育体系;也许,随着处理单元的多样化，高性能计算也会成为科研的必修课，并且随着计算机越来越普及，逐步进入很多人的生活，这些课程也会焕发出真正的光芒.也许……也许我们可以作为这个领域的先驱,去推广更多人去拥抱未来吧.  写于2023.12.17. | |