UNAT

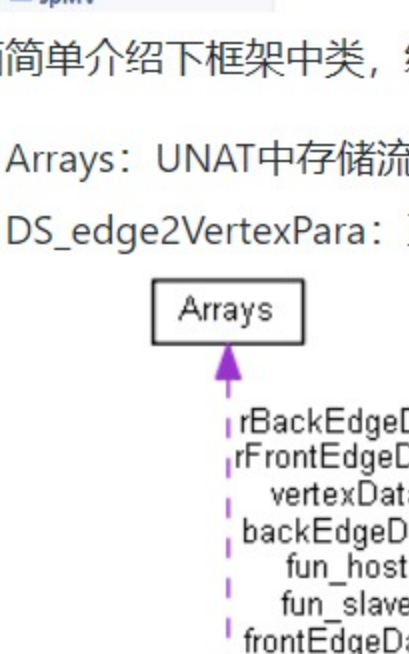
UNAT是一款运行在神威平台上的非结构网格遍历器框架，用户可直接使用此框架对自有代码进行加速，无需关心申威26010处理器架构及加速特性，只需给定网格拓扑及运算操作即可获得相对于主核4-6倍的加速效果。

UNAT代码结构

本框架代码主要由C++语言编写，从核部分代码由C语言编写，文件结构如下图所示：include文件夹中包含了所有头文件；iterator文件夹下包含了遍历器的声明及实现；test文件夹用于测试代码的正确性，用户也可通过此文件夹的内容了解各个遍历器的使用方法；tools文件夹包含了寄存器通信接口，从核代码DMA接口以及框架总体宏定义文件；topology文件夹下包含了网格拓扑的创建操作；wrappedInterface文件夹下为相应运算操作的函数指针实现。

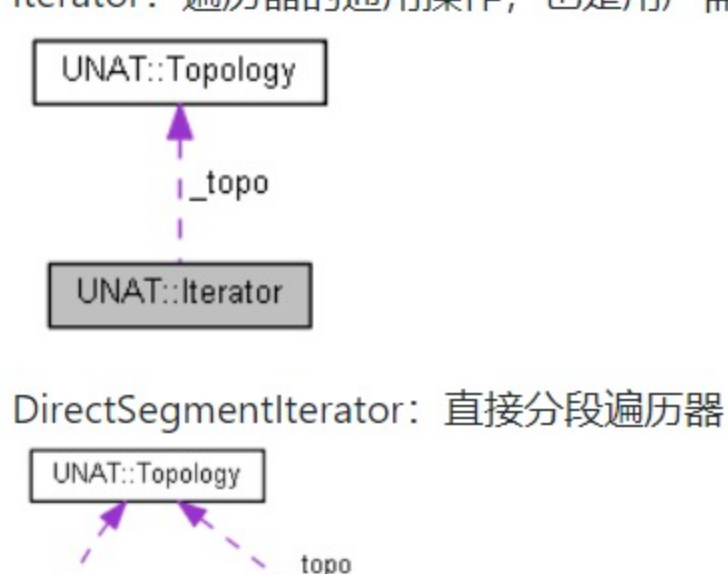
下面简单介绍下框架中类，结构体以及接口：

- Arrays：UNAT中存储流场数据的结构体，后面将会介绍；
- DS_edge2VertexPara：直接分段(Direct Segment)遍历器中函数指针所需的数据，包含结构体Arrays；



- Topology：网格拓扑的创建，生成与转化；
- Iterator：遍历器的通用操作，也是用户需要关心的接口，包含网格拓扑Topology；

- DirectSegmentIterator：直接分段遍历器，遍历器的一种，继承自Iterator，包含Topology；



- Pack：寄存器通信中发送与接收数据的结构体
- RlmpiInfo：用于寄存器通信接口
- Table：用于寄存器通信接口

遍历器

本框架包含多种遍历器，适用于不同应用场景，用户可根据需要及实际加速效果自行选择

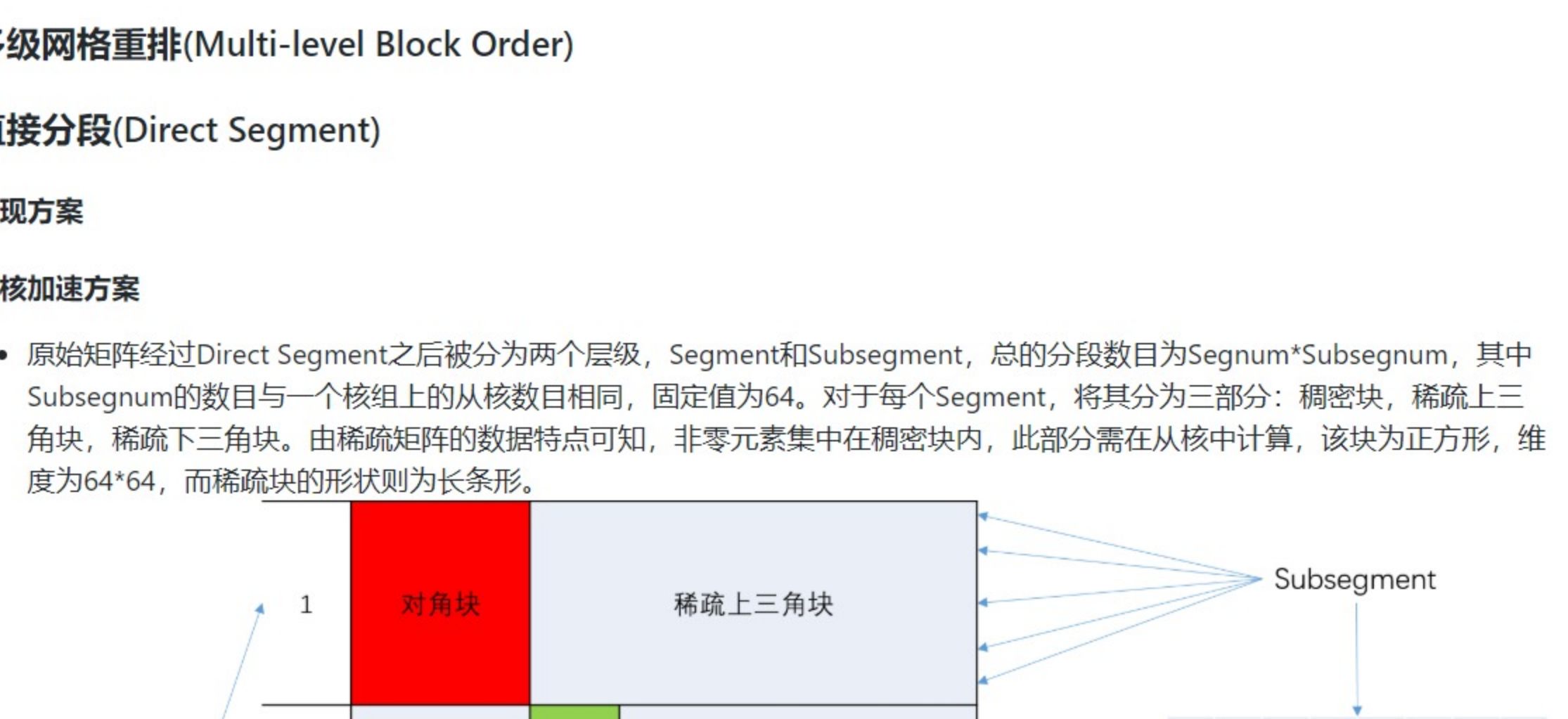
多级网格重排(Multi-level Block Order)

直接分段(Direct Segment)

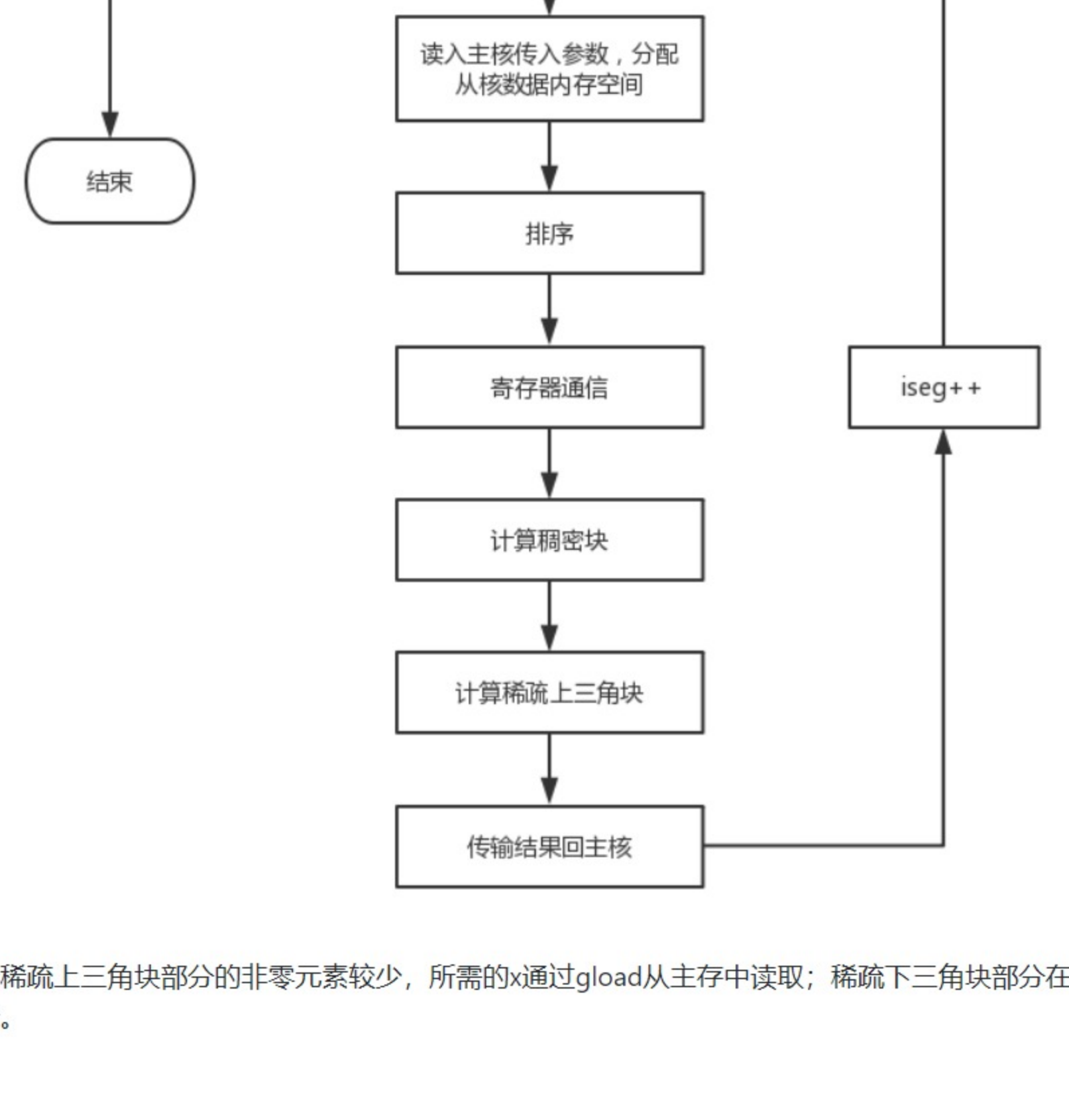
实现方案

从核加速方案

- 原始矩阵经过Direct Segment之后被分为两个层级，Segment和Subsegment，总的分段数目为Segnum*Subsegnum，其中Subsegnum的数目与一个核组上的从核数目相同，固定值为64。对于每个Segment，将其分为三部分：稠密块，稀疏上三角块，稀疏下三角块。由稀疏矩阵的数据特点可知，非零元素集中在稠密块内，此部分需在从核中计算，该块为正方形，维度为64*64，而稀疏块的形状则为长条形。

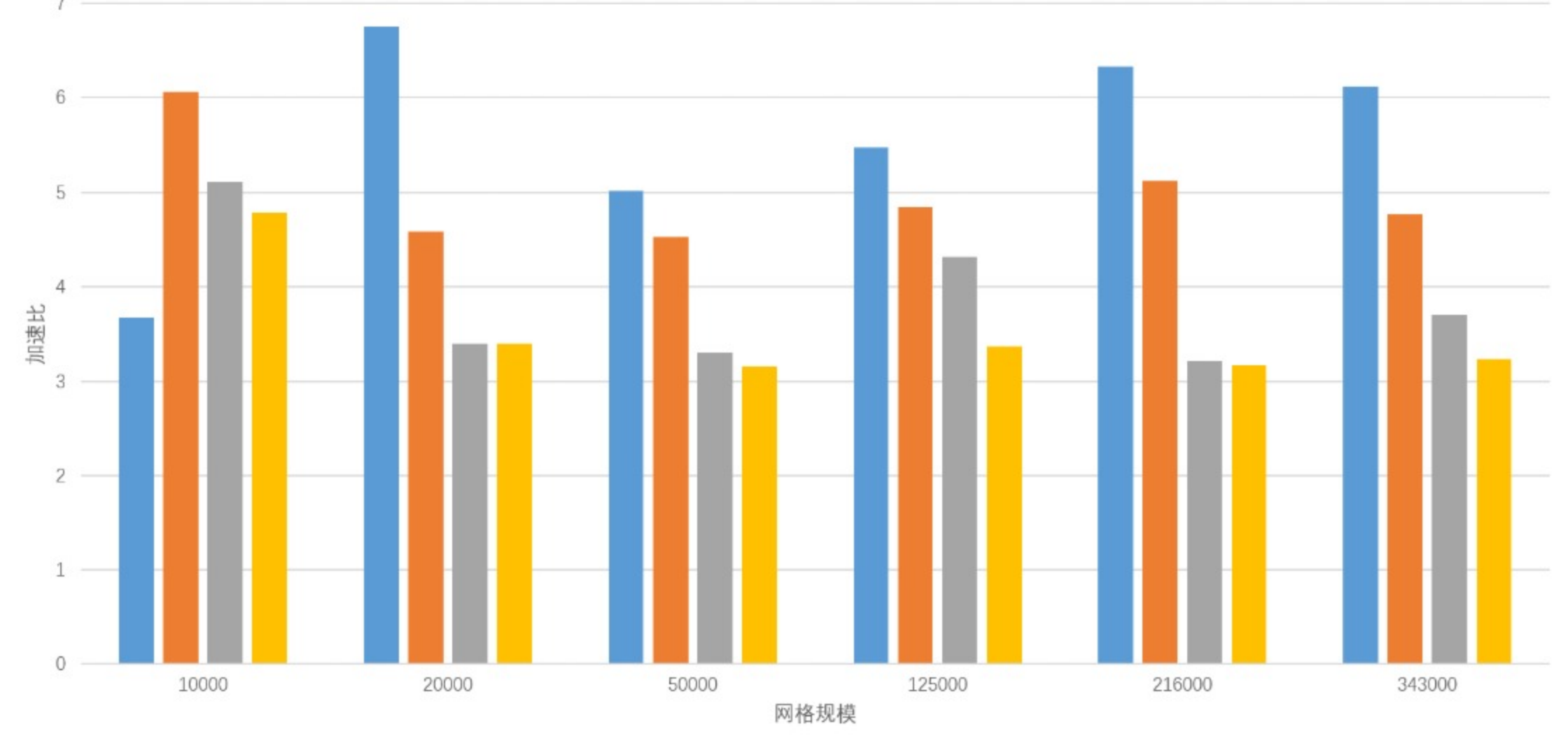


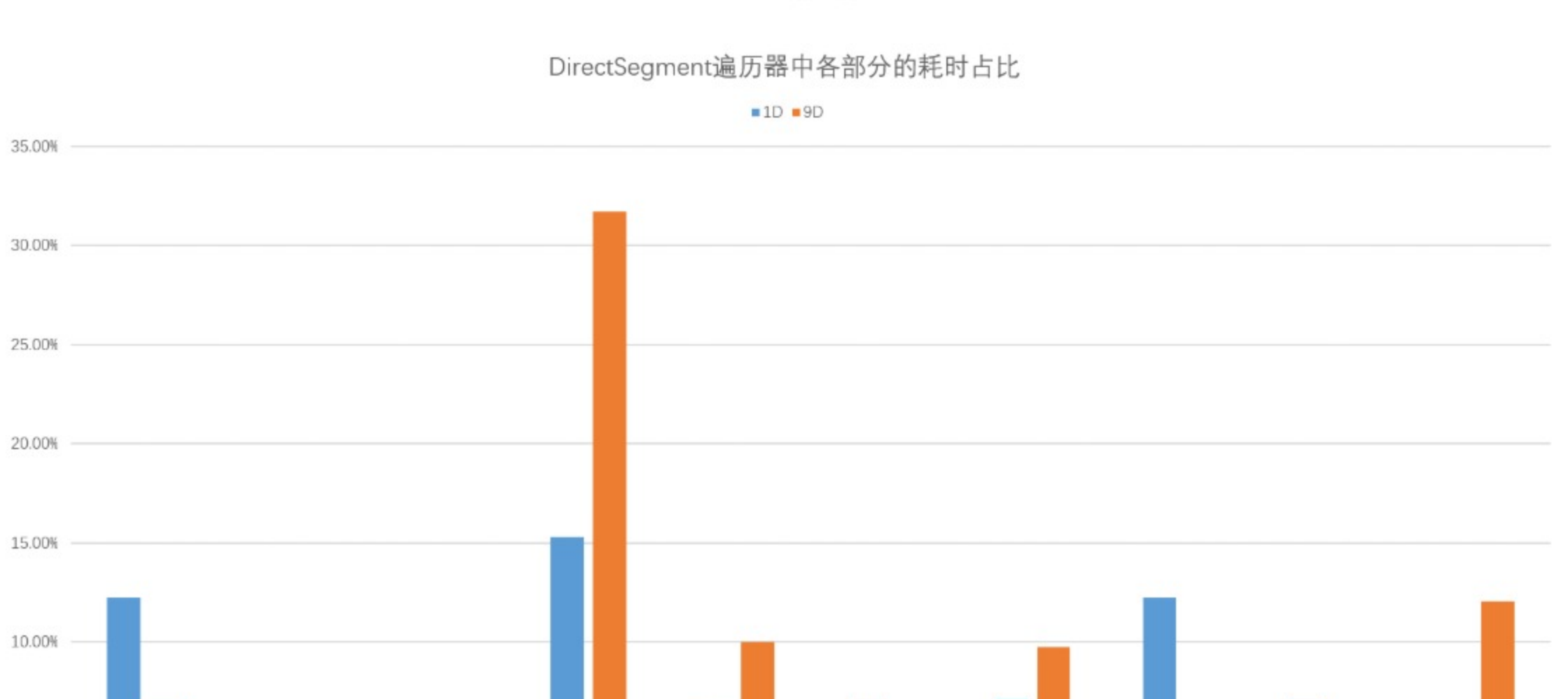
- 从核的计算过程如下图所示，一个核组一次计算一个segment，每个从核负责计算其中的一个subsegment，以稀疏矩阵向量乘操作(spMV)为例，读入主核传入的参数后，为A,x,b开辟内存空间，由于从核LDM空间有限，不能将x完全读入，因此每个从核只读取稠密块对应subsegment的部分x，稠密块其余subsegment的x通过寄存器通信方式获取。使用寄存器通信接口需要矩阵A以block方式存储，由于原有矩阵A的存储方式为LDU方式，需要对其进行重新排序.



- 由于稀疏上三角块部分的非零元素较少，所需的x通过gload从主存中读取；稀疏下三角块部分在主核中计算，以实现主从核并行。

性能





最小带宽排序(Mini-band Order)

数据结构

本框架中流场中的数据存储在Arrays结构体中，其定义可参见下方代码。

```
typedef struct
{
    // float data array holder
    swFloat** floatArrays;
    // integer data array holder
    swInt** intArrays;
    // float data array dimensions
    swInt*    fArrayDims;
    // integer data array dimensions
    swInt*    iArrayDims;
    // float data array action
    swInt*    fArrayInOut;
    // integer data array action
    swInt*    iArrayInOut;
    // float data array number
    swInt    fArrayNum;
    // integer data array number
    swInt    iArrayNum;
    // float data array sizes
    swInt    fArraySizes;
    // integer data array sizes
    swInt    iArraySizes;
} Arrays;
```

网格域的数据被分成四种类型

- backEdgeData：下三角网格面
- frontEdgeData：上三角网格面
- selfConnData: 对角线网格面
- vertexData：网格单元数据 用户可由float类型或者double类型的数据构造上面四类数据

```
constructSingleArray(backEdgeData, dimensions, faceNum, COPYIN, lower);
constructSingleArray(frontEdgeData, dimensions, faceNum, COPYIN, upper);
constructSingleArray(selfConnData, dimensions, faceNum, COPYIN, diag);
constructSingleArray(vertexData, dimensions, vertexNum, COPYIN, x);
addSingleArray(vertexData, dimensions, vertexNum, COPYOUT, b);
```

使用方法

配置项目

修改项目Makefile，该文件位于PROJECT根目录下，打开文件修改其中的第4行

```
PROJECT=YOUR_DIR/UNAT
```

通过make命令编译该框架即可得到libUNAT.a，库文件和头文件位于文件位于PROJECT/lib和PROJECT/include文件夹内，用户在自有代码里引用及链接相应文件

```
make clean
make
```

创建网格拓扑

本框架中可以通过两种数据结构创建网格拓扑，LDU和CSR，对于LDU形式的数据结构，给定网格面左右两边的网格单元编号及网格面数目，即可创建网格拓扑

```
Topology* topo = Topology::constructFromEdge(startVertices,endVertices,faceNum);
```

- **startVertice和endVertices需严格以LDU方式存储，即startVertices[i]<endVertices[i]**
- **startVertices和endVertices需要以行优先的方式按顺序存储**

指定运算操作

这里通过宏定义方式指定函数指针，该文件位于PROJECT/wrappedInterface/文件夹下，用户可为每种运算操作创建新的文件夹，这里以稀疏矩阵向量乘(spMV)为例，该文件夹下包含三个文件

- spMV.h spMV_host.cpp spMV_slave.c
  分别为头文件，主核函数指针文件和从核函数指针文件，其中主核函数指针和从核函数指针文件内容相同，用户目前需要写两份相同的代码，下一版本将对Makefile进行改进。
  下面以spMV_slave.c为例，该函数指针的实现包含三个部分，从上到下依次为对角元素，上三角元素，下三角元素的计算，用户可参照代码中获取指针及网格面数量的代码，自定义其他运算操作。

```
define_e2v_hostFunPtr(spMV_slave)
{
    //selfConn computation
    swFloat* diag   = accessArray(selfConnData, 0);
    swFloat* x      = accessArray(vertexData, 0);
    swFloat* b      = accessArray(vertexData, 1);
    swInt iDim,dims;

    swInt vertexNum = getArraySize(selfConnData);
    dims = getArrayDims(selfConnData, 0);
    swInt ivertex;
    for( ivertex = 0; ivertex < vertexNum; ivertex++)
    {
        for(iDim=0;iDim<dims;iDim++)
        {
            b[ivertex*dims+iDim]
                += diag[ivertex*dims+iDim]*x[ivertex*dims+iDim];
        }
    }

    //frontEdge computation
    swFloat* upper  = accessArray(frontEdgeData, 0);
    swInt edgeNumber = getArraySize(frontEdgeData);
    dims = getArrayDims(frontEdgeData, 0);
    swInt iedge;
    for( iedge = 0; iedge < edgeNumber; iedge++)
    {
        for(iDim=0;iDim<dims;iDim++)
        {
            b[startVertices[iedge]*dims+iDim]
                += upper[iedge*dims+iDim]*x[endVertices[iedge]*dims+iDim];
        }
    }
```

对于spMV.h文件，从核函数的声明增加了slave_xxx前缀，这是因为从核在编译时在目标文件中的名字增加了slave_xxx前缀，因此函数声明要进行相应修改

```
void slave_spMV_slave(Arrays* backEdgeData, Arrays* frontEdgeData,Arrays* selfConnData, Arrays* vertexData, swInt* sta
void spMV_host(Arrays* backEdgeData, Arrays* frontEdgeData,Arrays* selfConnData, Arrays* vertexData, swInt* startVerti
```

调用遍历器

本框架包含多种遍历器，用户可自行选择，这里以DirectSegmentIterator为例进行说明

- 构造遍历器 构造遍历器需要网格拓扑信息以及网格面和网格单元权重，网格面的权重即为一个网格面上的数据量，网格单元同理。对于spMV操作，每个网格面上包括lower[i]和upper[i]两个数据，每个网格单元上包括x[i]和b[i]两个数据，所有这里cellWeights和edgeWeights均设为2*dimensions。

```
std::vector<swInt> cellWeights(topo->getVertexNumber(), 2*dimensions);
std::vector<swInt> edgeWeights(topo->getEdgeNumber(), 2*dimensions);
DirectSegmentIterator iterator(*topo, &cellWeights[0], &edgeWeights[0]);
```

- 调用遍历器 遍历器包括两种 ** edge2VertexIteration：按网格面循环的遍历方式 ** vertex2EdgeIteration：按网格单元寻魂的遍历方式 这里以edge2VertexIteration为例，给定四种类型的流场数据以及函数指针

```
iterator.edge2VertexIteration( &backEdgeData, &frontEdgeData,&selfConnData, &vertexData, spMV_host, slave_spMV_slave);
```

注意这里的从核函数指针，spMV_slave.c文件内的名字为spMV_slave，但是由于从核编译特点，函数指针在目标文件中的名字增加了slave_xxx前缀，因此调用时需要进行相应修改

样例

以上的使用过程已在PROJECT/test/directSegment/test.cpp文件中实现，用户可对比该文件理解框架的使用过程