UNAT项目报告

笔记本： 工作日志
创建时间： 2019/7/31 9:36 **更新时间：** 2019/7/31 16:09
作者： lhb8125@gmail.com
URL： http://www.youdao.com/w/Among%20all%20kinds%20of%20mesh%20typ...

研究背景

在各种网格类型中，非结构化网格在工程仿真场景中占主导地位，在科学计算中发挥着重要作用。有限元法(FEM)和有限体积法(FVM)等离散化方法在求解各种偏微分方程的非结构化网格上已有几十年的良好实践。然而，基于非结构化网格的高保真度应用程序仍然非常耗时，无论是编程还是运行。流体流动模拟是一种典型的应用，主要利用非结构化网格上的FVM离散化，其高保真度结果的计算能力非常强大。另一方面，随着高性能计算机计算能力的不断提高，硬件体系结构的复杂性和编程难度对领域开发人员构成了很高的门槛。

随着新型处理器体系结构特别是多核处理器体系结构的出现，并行编程模型经历了爆炸式的增长。仔细的程序设计和优化是充分利用新兴并行高性能系统的潜力的关键。与多核架构相比，采用英特尔多核集成处理器(MIC)、英伟达gp-gpu等多核架构的处理器成为500强中最受欢迎的高性能计算平台。因此，除了传统的纯并行编程模型(如MPI和OpenMP)外，针对异构多核体系结构的特定于平台的模型(包括CUDAandOpenCLformedan)也非常丰富，并且具有许多不同的和令人困惑的特征。为了获得最佳性能，必须特别注意新体系结构的更复杂的细节，这将导致极高的开发成本和域开发人员额外的维护困难。

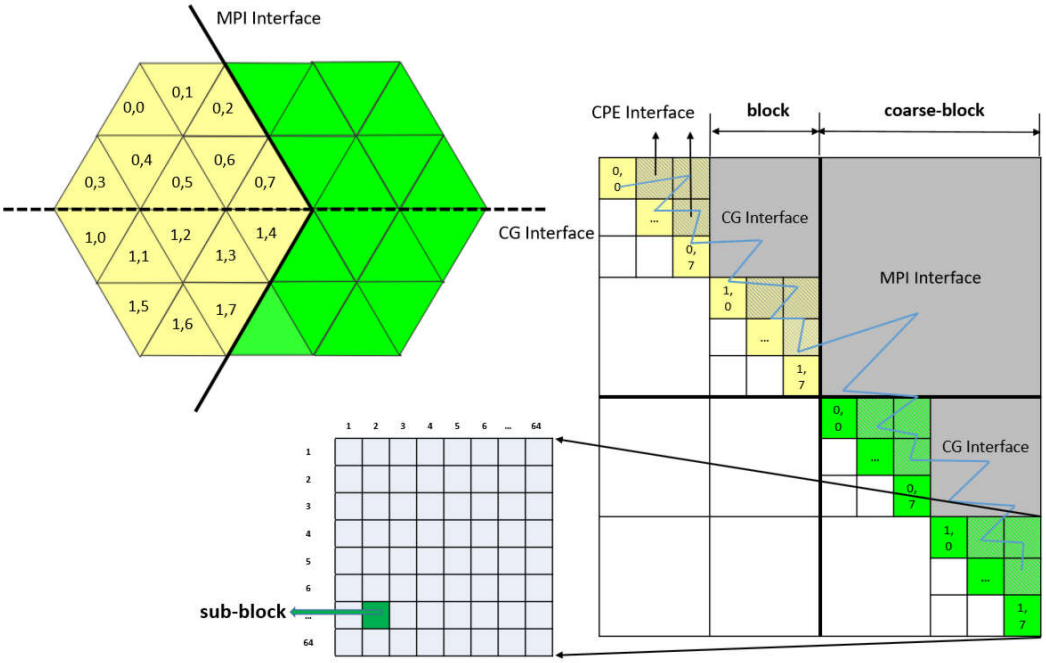
具体地说，多核处理器上的并行计算需要更多的精力来进行非结构化的基于网格的计算。使用一般抽象，非结构化计算可以看作是在相邻图上执行的计算。众所周知，在非结构化计算中，开始点/所有者和结束点/边缘邻居不能同时连续存储，因此不规则和间接的数据访问是不可避免的。在多核体系结构上的细粒度并行计算放大了这一问题。例如，间接数据寻址可能跨越多个线程，导致非统一内存体系结构(NUMA)配置上的内存访问效率低下。同时，竞争条件问题也很突出，因为线程的高并发性使得原子函数等一般技术代价昂贵。此外，不规则的数据访问也意味着在一条高速缓存线路中有很大比例的无效数据，这会降低有效内存带宽。

为了解决上述非结构化计算问题，我们开发了加速工具UNAT, UNAT是基于非结构化网格的加速工具包的缩写，它提供了友好的高级编程接口，同时阐述了较低层次的体系结构感知实现，从而在目标硬件上获得最佳性能。在此工作中，我们在SW26010平台上执行了两种优化策略。第一个是基于多级分块重排序(MLB)的实现，它强调与缓存大小和层次结构一致的数据局部性;第二个是基于精确分段的实现，名为Row-Subsections (RSS)，采用轻量级预处理，具有内核自适应特性。在RSS实现中使用了带着色算法的数据段来避免访存冲突。

实现方法

多级分块重排序方法

如前所述，非结构化网格的计算能力受到间接和不规则寻址的极大限制。为此，提出了多级块(MLB)方法，并设计了一种MPE-CPEs异步并行算法。下图给出了一个非结构化网格的分解及其LDU格式的矩阵形式。MLB中的数据结构是非均匀的多块结构。最粗块表示MPI级别的网格分解。对角粗块上的非零表示分配给单个CG的一个进程内网格单元的内表面，非对角粗块表示相邻进程之间的MPI接口。

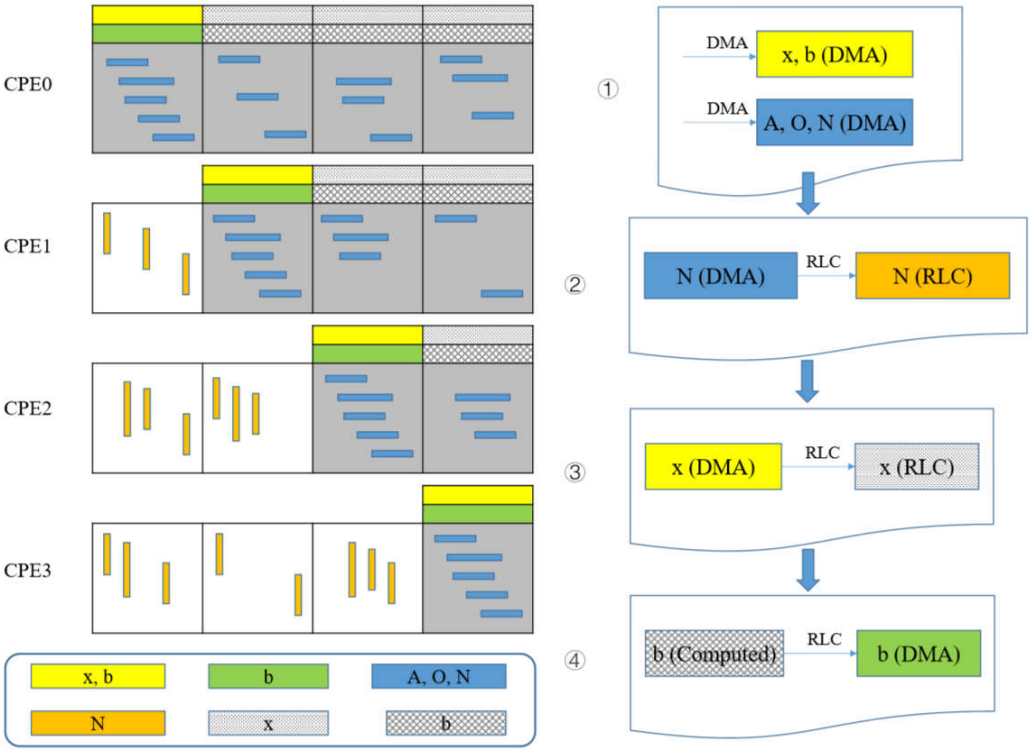


根据LDM大小的限制对中间块(块)进行分区，然后将各块的相关数据完整地存储在LDM中。在这一层，将所有块划分为内部块(对角线块中标记为彩色单元格)和CG接口块(非对角线块中标记为灰色单元格)，分别根据数据密度分配给CPEs和MPE。如果MPE和CPE负责相同的行，那么这个模式将在它们之间引入读写冲突。通过在MPE中相对于CPE的块的行索引中添加一个偏差，可以消除这个问题。

最后，基于CG硬件对最细密块(子块)进行了分解。子块的数量被设置为一个固定的值，64，对应于CG中CPEs的数量。对角子块是对于CPE来说是“当地的”，非对角子块是CPE之间的交界面。由于数据的离散性和稀疏性，直接通过DMA将“当地的”数据加载到LDM中，交界面中的数据通过片上寄存器通信进行传输。上图中的蓝色填充曲线表示不同块在内存中的存储顺序。通常，优先级规则是较细的级别块优先于较粗的级别块，同一级别的行块优先于列块。在子块内部，边缘序列是任意的。利用图形划分工具Metis，可以将每一层的大部分面集中成对角块，考虑负载均衡，可以将界面最小化到一定程度。到目前为止，我们通过MLB得到了一个块结构和有组织的数据结构。

对于CPE中对角块的计算，同一行的子块由一个CPE负责。为了解释CPEs的计算模式，我们以稀疏矩阵向量乘(SpMV)为例。将LDU格式的系数矩阵A分解为下、上、对角三个数组，分别存储下三角、上三角和对角数据。这些数据可以通过DMA加载到

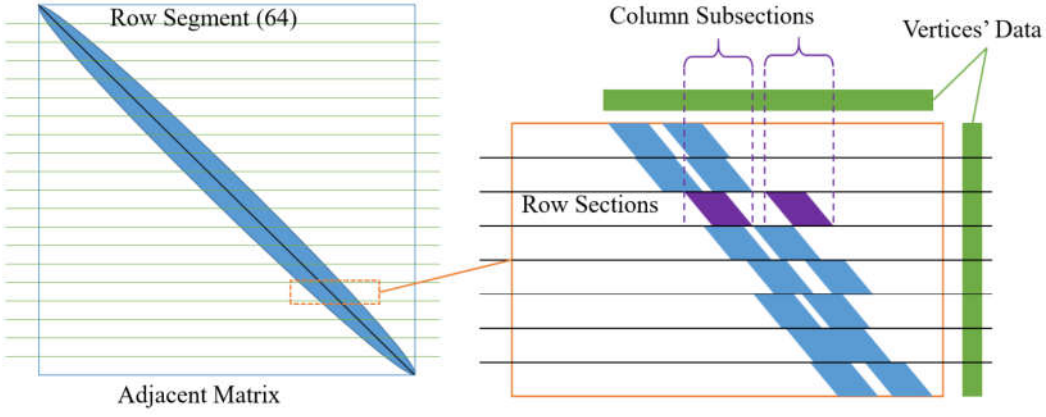
LDM,因为它在内存中连续和稀疏,虽然并不适合向量x和b。斜块的计算过程在下图中展示。



为了简洁性CPEs的数量设定为4。在我们的实现中，对角子块和非对角子块的数据分别被视为“当地的”数据和“全局”数据。首先，数据驻留在相邻图的边缘，包括系数矩阵A和拓扑矩阵(所有者O和邻居N)，通过DMA与局部x和b(1)加载到LDM中。然后通过寄存器级通信(RLC)将相邻数据N从上三角传输到下三角，这对于下一步(2)是必不可少的。对于非对角子块，由于x的冗余性和有限的LDM空间，x不能直接通过DMA加载。全局x存储在其他cpe的LDM中。因此，可以选择x对应于邻居数据N进行通信，得到每个CPE的“稀疏”全局x，布局与A(3)相同)。现在,系数矩阵和向量x是加载到LDM,我们可以执行SpMV操作，得到与x具有相同存储格式的输出向量b。为了避免CPE之间的写冲突,b需要通过RLC转移到相应的CPE上。

行分段方法

准确的数据平铺是该实现的核心策略。在以行优先格式(LDU或COO)存储的相邻矩阵中，行/顶点和条目/边按顺序分解为两个层次，如下图所示。

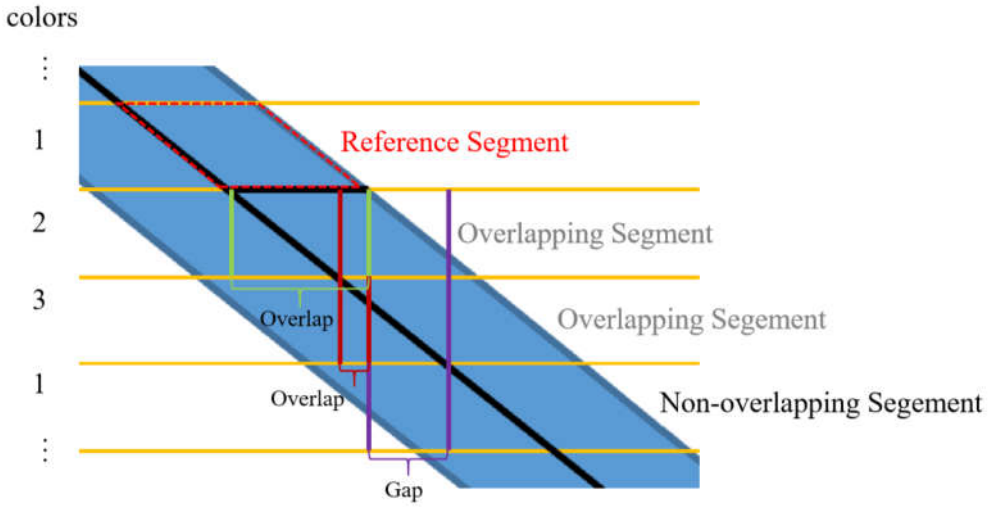


第一级为64段，与一个CG中CPE的数量保持一致，第二级根据单个CPE的LDM容量确定。这是一种不需要对原拓扑进行任何调整的轻量级算法。名称中的行表示行/顶点是

分解的基本单元，名称中的子段来自多个子段，这些子段由平铺顶点数据集生成。在矩阵视图中，子部分是条目列的投影，在上图中标记为ColumnSubsections。在典型的边环计算中，由相关行/所有者标识的边和顶点的数据集被连续有效地访问，而对于列/邻居相关的顶点，则是一个不规则的内存访问问题。利用RSS策略中的列/邻域子节，将相关顶点组织为多个相邻的节，从而实现对列/邻域的有效连续内存访问。

每个行段的列/相邻子段的数量是可变的，子段的跨度在不同的应用场合是自适应的。在列子断面投影中采用了一种多重最近邻算法。在该算法中，边是按行优先顺序遍历的，一个边列是应该放在前一个分段中，还是应该开始一个新的分段取决于给定的跨度。跨度对于性能调优是至关重要的，因为小跨度会导致分段和低内存带宽，而大跨度可能涉及太多的非相关数据，从而降低了有效带宽。在Sunway平台上，为保证最佳带宽效率，给出了一个接近LDM高速缓存线大小的跨距初值，如果触发条件表明子段中有太多的非相关数据，则初始值减半。算法复杂度为O(N)，其中N为条目/边的数量。

RSS策略的一个重要问题是，当采用LDU等对称矩阵格式时，在列/邻域相关子节中无法避免CPEs之间的竞争条件。我们设计了一个着色算法来解决写冲突问题，如下图所示。



典型的邻接矩阵可以在带宽有限的情况下重新排序，因此当参考行段与后续段之间的距离大于相邻矩阵的最大带宽时，它们的条目/边缘不会与后续段重叠。按照这种方法，该算法将尽可能多的非重叠段着色成一个类，以确保写操作的最大并发性。我们的经验,一个代表相邻矩阵的实际的应用程序通常会有一个带宽小于矩阵秩的1/10,所以会有每个颜色平均10多个并发线程类,足以利用4内存隧道cp CG。虽然着色算法通常具有O(N2)复杂度，但在本工作中，N始终为64，因为着色是在段上执行的，段数始终为64，等于单个CG中的CPEs数量。所以RSS在某种程度上仍然是非常轻量级的。实际上，无论是否对相邻矩阵执行带宽最小化的重新排序，RSS策略都是适用的，但是当矩阵过于稀疏时，它会导致低效的内存访问和低写并发性。然而，在大多数情况下，使用宽度优先重排序算法(如RCM)来最小化带宽已经提前执行。因此，我们不采取这种重新排序的RSS预处理程序，以确保其轻量级的性质。

RSS策略的另一个优点是内核的适应性。在实际应用中，数组的数量和结构的组件在不同的内核中是可变的。分区方法必须按照硬件或算法确定的给定大小执行，不能在不同的数据大小可变的内核中保存局部性。对于RSS来说，额外的设计是不必要的，因为在相同的数据布局中，行节大小和列/相邻的子节距离会随着内核的不同而变化。

接口特点

- 具有良好的用户友好度
- 自动化众核细粒度并行功能
- 多种加速方法可供选择
- 最小化程序移植难度
- 支持LDU、COO等多种稀疏矩阵存储格式
- 支持数组结构体（AoS）和结构体数组（SoA）两种数据布局

接口API

非结构网格计算有三大要素：网格拓扑，数据和算子。UNAT的设计理念是将这三大要素分离，以实现统一的计算框架。在使用UNAT时，同样需要对这三部分分别考虑。

构建网格拓扑及遍历器

- MLB遍历器

```
// 创建网格拓扑（LDU格式）
Topology* topo = constructFromEdge(owner, neighbor, edgeNumber);
// 创建MLB遍历器
UNAT::MLBIterator mlbIter(topo, vertexWeights, edgeWeights);
// 重排原始网格拓扑
mlbIter.reorderEdges(owner, neighbor, edgeNumber, vertexNumber);
```

- RSS遍历器

```
// 创建网格拓扑（LDU格式）
Topology* topo = constructFromEdge(owner, neighbor, edgeNumber);
// 创建MLB遍历器，E2V指定该遍历器以网格面为基准遍历所有网格单元，V2E指定以网格单元为基准遍历所有面
UNAT::RSSterator rssIter(topo, vertexWeights, edgeWeights, E2V);
```

封装数据

这里以稀疏矩阵向量乘为例介绍数据封装接口的使用方法，矩阵格式为LDU，因此包含*upper*, *lower*, *diag*, *x*, *b* 等五个数组。

```
// 矩阵上三角的数据（对于CSR矩阵来说为上下三角的数据）
Arrays frontEdgeData;
// 矩阵下三角的数据（对于CSR矩阵来说为空）
Arrays backEdgeData;
// 网格单元上的数据
Arrays vertexData;
constructSingleArray(frontEdgeData, 1, edgeNum, COPYIN, upper);
constructSingleArray(backEdgeData, 1, edgeNum, COPYIN, lower);
constructSingleArray(vertexData, 1, edgeNum, COPYIN, diag);
addSingleArray(vertexData, 1, vertexNum, COPYIN, x);
addSingleArray(vertexData, 1, vertexNum, COPYOUT, b);
```

Arrays为自定义结构体类型，具体定义如下：

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

结构体属性	描述	类型
fArraySizes	数组元素个数	int
fArrayNum	数组个数	int
fArrayInOut	数组类型（COPYIN，COPYOUT， COPYINOUT，UPDATED）	int[]
fArrayDims	数组维度	int[]
floatArrays	数组内容	int[]

其中fArrayInOut表征数组的输入输出类型：

宏定义	描述	值
COPYIN	DMA->LDM	0

宏定义	描述	值
COPYOUT	LDM->DMA	1
COPYINOUT	DMA->LDM->DMA	2
UPDATED	LDM<->LDM	3

接口中提供了以下几种宏构造Array结构体：

```
// 创建空结构体paraData
constructEmptyArray(paraData);
// 创建包含paras数组的结构体paraData
constructSingleArray(paraData, dims, nums, COPYIN, paras);
// 向paraData结构体中添加数组paras2
addSingleArray(paraData, dims, nums, COPYIN, paras2);
```

创建算子

UNAT具有良好的用户友好度表现之一就是其具有与串行程序基本一致的算子定义方法，用户无需关心并行实现细节与众核硬件特性，只需按照上一步数据封装过程将所需数据正确取出即可，下面以spMV算子为例进行介绍

```
// define the SpMV wrapper interface
define_ops(SpMV)
{
        // 取数据过程与封装过程保持一致
        swFloat *diag = accessArray(vertexData, 0);
        swFloat *x = accessArray(vertexData, 1);
        swFloat *b = accessArray(vertexData, 2);
        swFloat *upper = accessArray(frontEdgeData, 0);
        swFloat *lower = accessArray(backEdgeData, 0);
        swInt ivertex, iedge, idim, vertexNum, edgeNum;

        // 首先计算对角线部分
        dims = getArrayDims(vertexData, 0);
        vertexNum = getArraySize(vertexData);
        for(ivertex=0; ivertex<vertexNum; ivertex++)
        {
                for(idim=0; idim<dims; idim++)
                {
                        b[ivertex*dims+idim]
                            = diag[ivertex*dims+idim] *
x[ivertex*dims+idim];
                }
```

```
        }

        // 计算上三角部分
        dims = getArrayDims(frontEdgeData, 0);
        edgeNum = getArraySize(frontEdgeData);
        for(iedge=0; iedge<edgeNum; iedge++)
        {
                for(idim=0; idim<dims; idim++)
                {
                        b[owner[iedge]*dims+idim]
                            = upper[iedge*dims+idim] *
x[neighbor[iedge]*dims+idim];
                }
        }

        // 计算下三角部分，如无下三角（例如CSR矩阵）请置为空
        dims = getArrayDims(backEdgeData, 0);
        edgeNum = getArraySize(backEdgeData);
        for(iedge=0; iedge<edgeNum; iedge++)
        {
                for(idim=0; idim<dims; idim++)
                {
                        b[neighbor[iedge]*dims+idim]
                            = lower[iedge*dims+idim] *
x[owner[iedge]*dims+idim];
                }
        }
}
```

封装算子与执行

以上三步完成了网格拓扑读入，数据封装，算子实现三个主要过程，接下来将整合这三部分完成网格计算过程

```
// 将数据和算子打包到耦合算子中
CoupledOperator spMVOpt;
spMVOpt.data.backEdgeData = &backEdgeData;
spMVOpt.data.frontEdgeData = &frontEdgeData;
spMVOpt.data.vertexData = &vertexData;
spMVOpt.fun_slave = slave_spMV;
spMVOpt.fun_host  = spMV;
// 非流场数据，即控制参数、常量等等.
Arrays paraData;
// spMV无需任何控制参数和常量
```

```
constructEmptyArray(paraData);

// 执行算子完成计算
mlbIter.edge2VertexIteration(&paraData, &spMVOpt, 1);
```

CoupledOperator为自定义结构体类型，将数据和算子耦合在一起，定义如下：

```
typedef struct
{
    e2v_hostFunPtr fun_host;
    e2v_slaveFunPtr fun_slave;
    FieldData *data;
} coupledOperator;
```

结构体属性	描述	类型
fun_host	主核算子函数指针	e2v_hostFunPtr
fun_slave	从核算子函数指针	e2v_slaveFunPtr
data	算子耦合流场数据	FieldData

其中FieldData存储了流场信息，分为网格面和网格单元信息，其定义如下：

```
typedef struct
{
    Arrays *backEdgeData;
    Arrays *frontEdgeData;
    Arrays *vertexData;
} FieldData;
```

结构体属性	描述	元素个数
backEdgeData	网格面数据neighbour->face->owner（除spMV外不常用）	faceNum
frontEdgeData	网格面数据owner->face->neighbour(常用)	faceNum
vertexData	网格单元数据（常用）	cellNum

edge2VertexIteration和vertex2EdgeIteration是遍历器的两大接口，其功能为集合用户定义的网格拓扑，流场数据，算子等三部分进行网格计算，两个接口区别如下：

接口	遍历基准	遍历目标	网格面冗余计算	写冲突
edge2VertexIteration	网格面	相邻网格单元	无	无
vertex2EdgeIteration	网格单元	相邻网格单元	有	有

其参数列表相同

参数列表	描述	类型
paraData	算子所需要的参数化列表	Array[]
cOpt	耦合算子与流场数据后形成的融合算子	coupledOperator[]
optNum	融合算子数目	int

性能报告

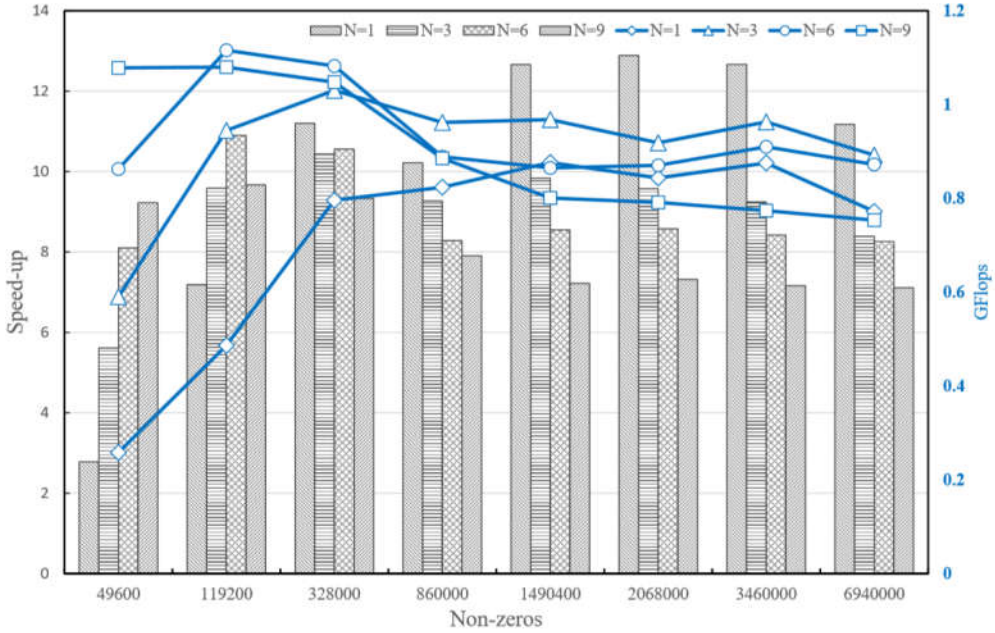
实验设置

在本节中，给出了定量实验结果，并讨论了评估UNAT性能。我们选择稀疏矩阵向量乘(SpMV)作为测量算子。SpMV在线性代数算法中发挥着重要的作用，在计算流体动力学(CFD)、分子动力学(MD)中得到了很好的应用，卷积神经网络(CNN)等机器学习算法对SpMV有着很深的依赖。然而，SpMV由于内存访问和写入冲突的不规则性，对多核体系结构并不友好，这很适合考察UNAT的效率。具体实验设置如下表所示

Iterator	Format	N(Number of components)	Matrices
MLB	LDU	1	boxTube16
		3	
		6	
		9	
	COO	1	
		3	
		6	
		9	
RSS	LDU	1	Goodwin
		3	
		6	
		9	
	OO	1	
		3	
		6	
		9	

MLB遍历器

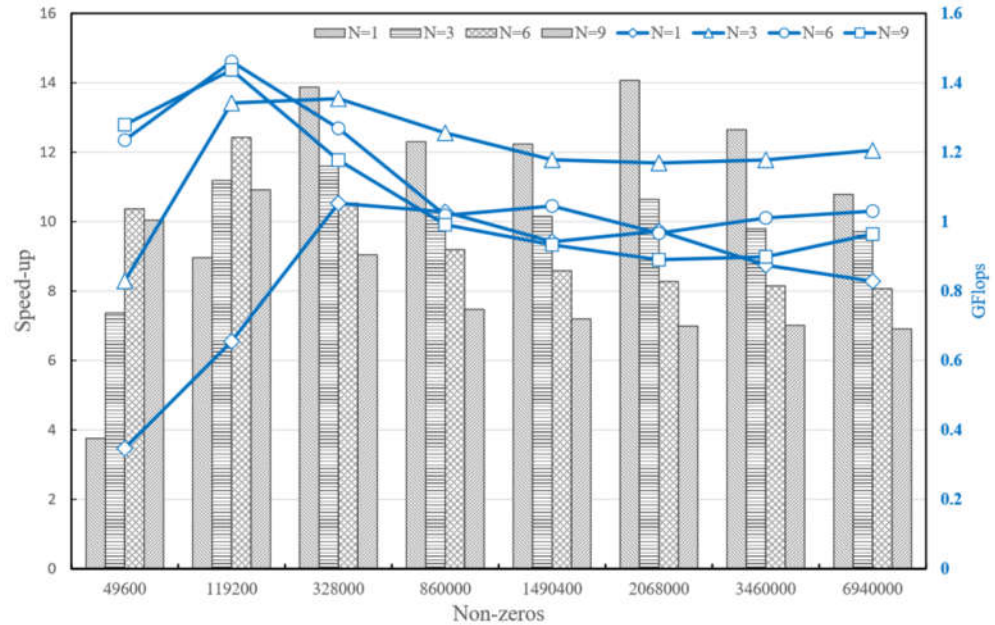
- ***LDU格式下MLB遍历器的加速效果***



MLB方法的加速速度随非零个数的增加而略有提高，其最佳性能在200万网格左右。当N = 1时，报告的加速平均为12，而在其他情况下，可以获得大约70%的性能。原因可以概括为三个因素: 首先，由于结构体内不同元素之间存在跨距，AoS存储模式对多核体系结构不友好；其次，RLC的拓扑只由矩阵的拓扑决定，为了兼容性不依赖于数据布局，所以RLC将连续执行N次，等于组件的数量。最后，在有限的LDM空间内，块的数量与组件的数量成正比变化，而分配给一个CPE的非零则相反。非对角块比例的增加表明CPEs之间的通信边界增加，这给RLC带来了很大的压力。

但是，我们不能简单地从加速比中得出结论：对于MLB迭代器，我们可以从SoA存储模式获得比AoS更好的性能。加速只显示SoA中的一个组件的性能，而这些组件是在外部循环中连续管理的。关于SoA和AoS性能的更精确的度量是每秒浮点运算(Flops)，从上图也可以看出。当矩阵足够大时，除N = 9外，AoS的Flops值略高于SoA，对于最小的矩阵，N = 9的Flops值最高。这意味着，考虑到MLB策略的性能，SoA存储模式是更好的选择。

- ***COO格式下MLB遍历器的加速效果***

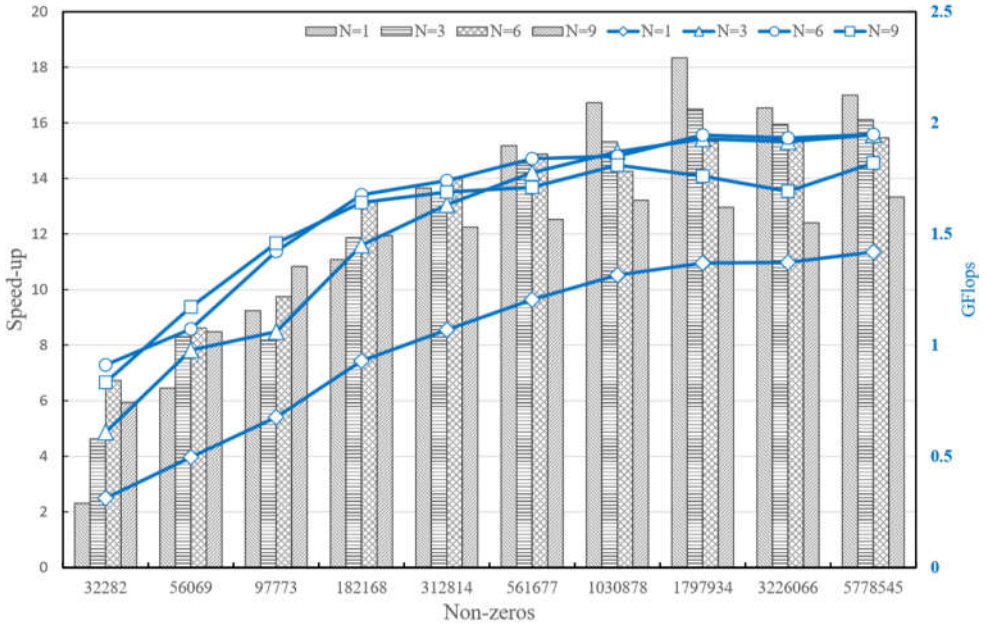


上图展示了采用COO格式的MLB方法的性能。关于AoS和SoA的趋势与LDU相似，但是很明显，无论是AoS还是SoA，我们都可以从COO格式中获得10% ~

20%的性能提升。COO格式的主要优点是没有写冲突。对于LDU格式，将执行两次RLC来传输全局x和b，而对于COO格式，只需要通过RLC传输全局x。然而，这种优势是以冗余计算为代价的，类似于“所有者-计算”模型，而SpMV由于存在下三角而不明显。在一些有限体积方案的通量计算中，稀疏矩阵在结构和数值上都是对称的，无需计算下三角形。但是，对于Sunway处理器来说，COO格式的冗余计算成本并不明显，因为它的计算能力与内存带宽之比更高。这两种矩阵格式对某一问题的性能必须辩证地进行考量和分析

RSS遍历器

- ***LDU格式下RSS遍历器的加速效果***

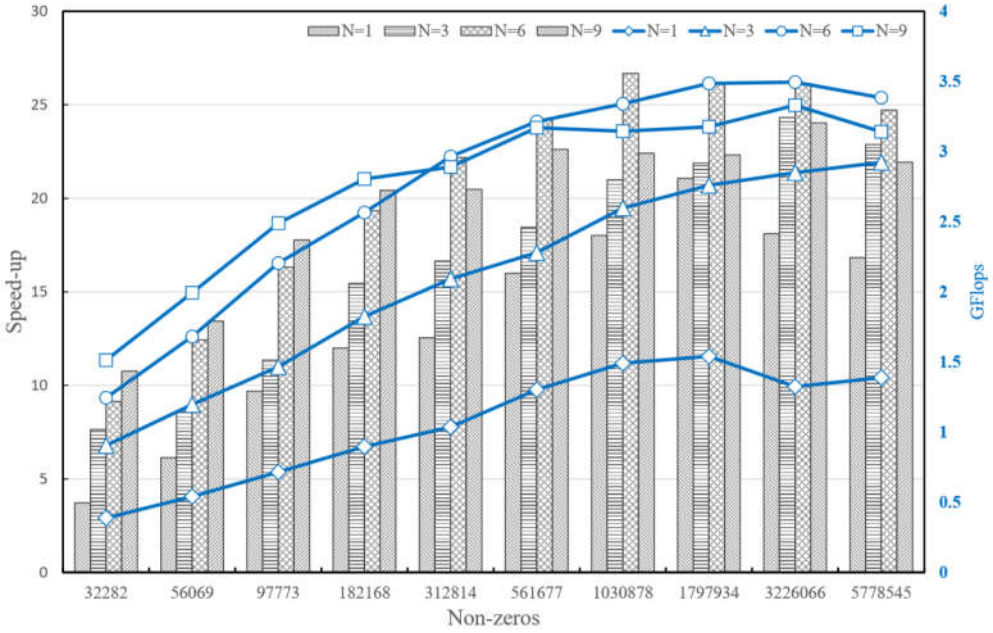


一般来说，我们的实现比基线平均加速15倍，Flops几乎达到2 GFlops。AoS和SoA之间的明显差距主要是由着色算法造成的。对于SW26010，当连续内存访问块的大小达到1024字节时，DMA的带宽逐渐增加，几乎达到性能的峰值。因此，列分段的跨度L由下面公式确定

$$L = \frac{1024}{sizeof(swFloat) \times N)}$$

其中N是struct中的分量数。与SoA模式(N = 1)相比，AoS模式具有更短的列分段，这为DMA提供了两个好处:首先，它在不改变带宽的情况下减少了冗余列的比例;此外，更短的列-分段表示CPEs之间的重叠更少，这减少了DMA轮数。

- ***COO格式下RSS遍历器的加速效果***



上图展示了使用COO格式的RSS方法的性能。最高的报告加速和人字拖分别达到27和3.5 GFlops, COO模式的整体性能是LDU模式的1.5倍。这种明显的差异是由写冲突造成的，这在上一节已经解释过了。同样，SpMV算子中存在下三角形时，冗余计算是“有意义的”。

未来工作展望

- 完善接口，进一步提升用户友好度
- 提升接口稳定性
- 支持进程级并行