先进编译实验室
Advanced Compiler

程序性能优化理论与方法

韩林　高伟　著

先进编译实验室
Advanced Compiler

目录

上篇

目录

下篇

先进编译实验室
Advanced Compiler

第十二章 多层次并行程序优化

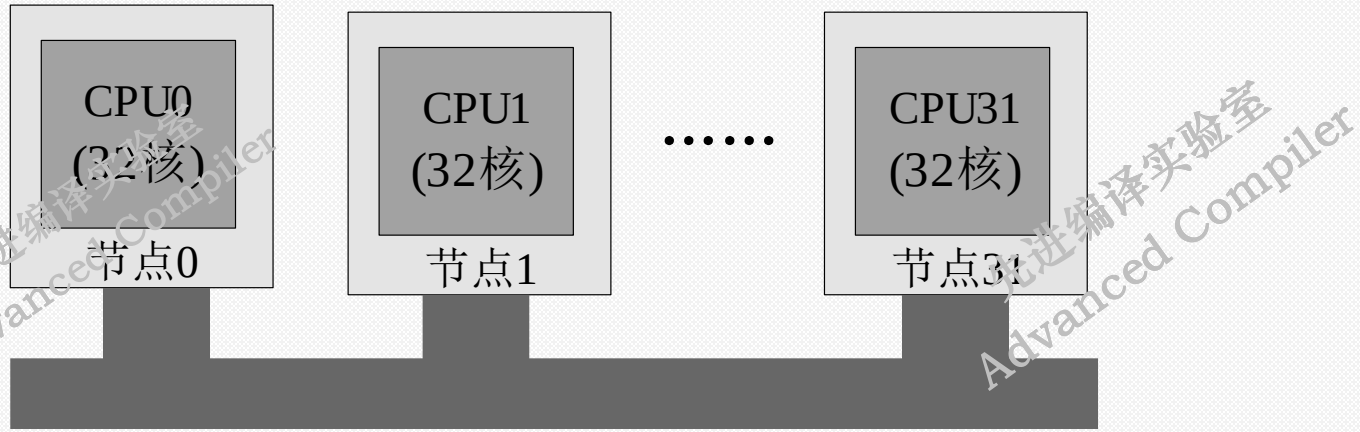
12.1Hygon C86同构多核平台

- 将在节点同构的海光Hygon C86多核平台上以矩阵乘法为例介绍同构多层次并行程序的编写及优化方法。主要有以下内容：

  ① 简单介绍Hygon C86平台的架构特点

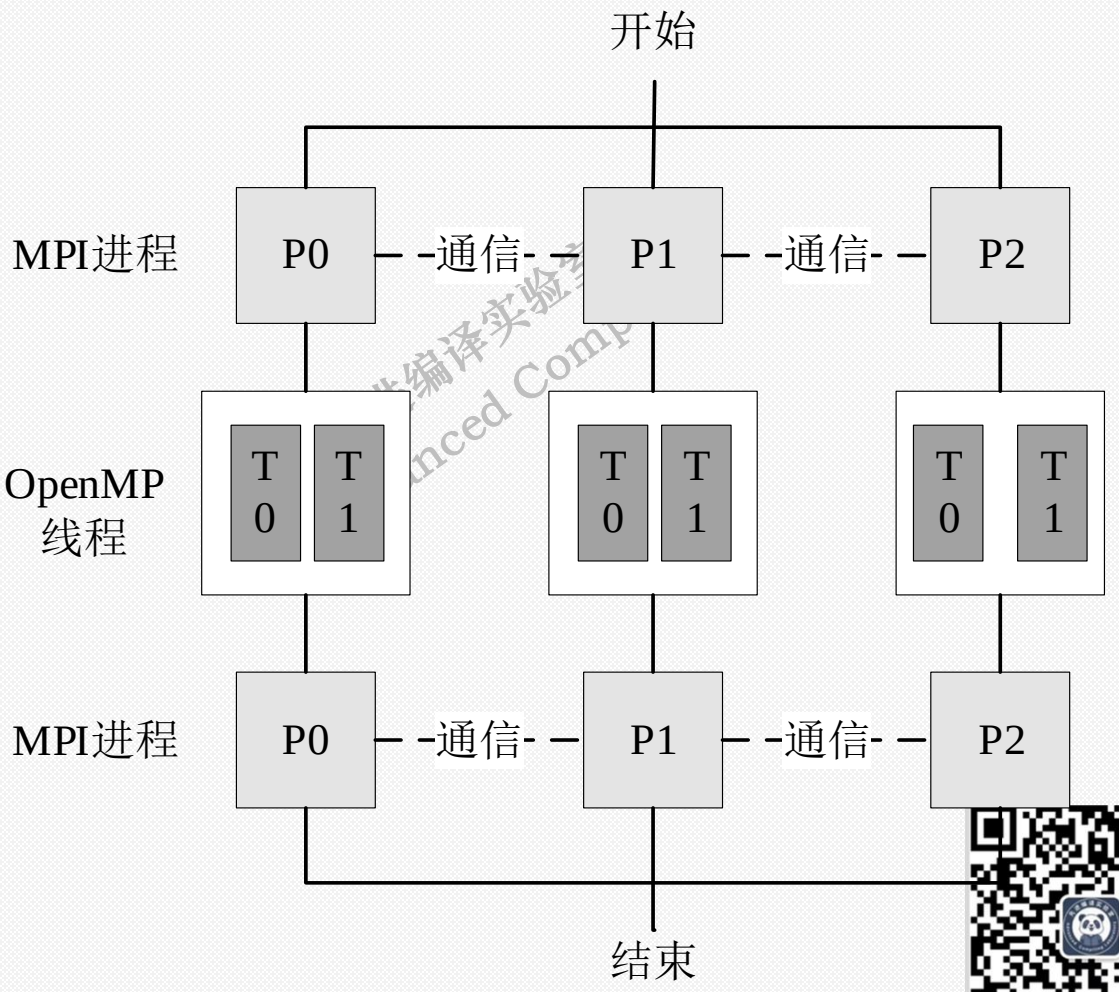
  ② 按照并行粒度由大到小逐层优化的思路，从程序编写角度展示矩阵乘法在Hygon C86平台上的多层次并行优化的过程

- Hygon C86平台上的每个节点以1路32核CPU处理器为计算资源，各节点采用网络互联。

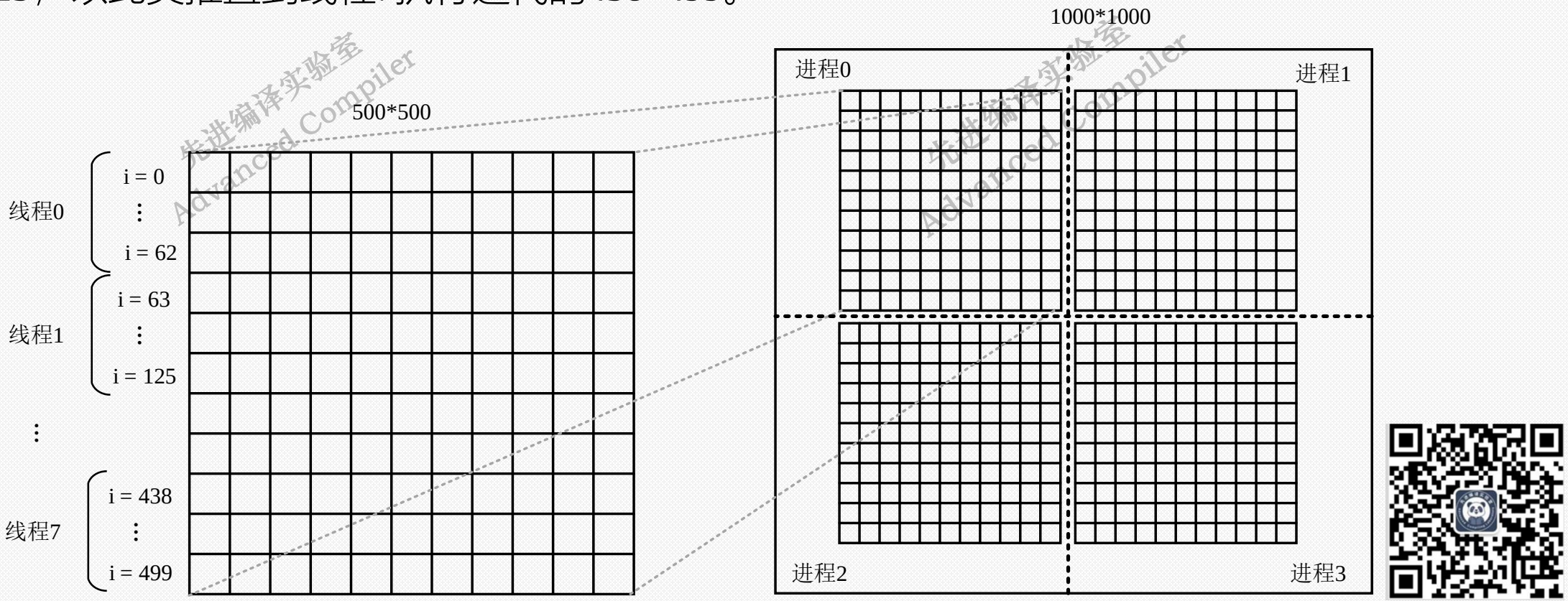


- 前面已经介绍过，OpenMP是用于在共享内存前提下使用多线程编程的接口，因此其在大规模并行机中只适合在单一节点内的一个或多个CPU上使用。而MPI是专门用于分布式内存系统中的消息传递模型，每个进程在内存中都拥有自己独立的地址空间。
- 但单纯的依靠MPI并行模型，在每个节点上通过增加MPI进程获取更多并行，考虑到进程级并行所带来的大量内存消耗，将导致程序性能的下降。
- 因此，充分利用消息传递模型MPI开发节点间并行，利用共享内存模型OpenMP进行节点内并行在Hygon C86平台上编写并行程序是一个不错的选择。因为它可以减少不同节点之间的通信，并在不增加内存需求的情况下提高每个节点的并行性。此外针对MPI加OpenMP混合编程[illegible]，每个线程又能够在处理器中的向量单元中利用SIMD指令进行数据级的细粒度并行，进一步提升程序性能。

- 基于第十一章使用棋盘式分解方式的MPI版本矩阵乘法，使用MPI+OpenMP混合编程的方式进行优化。
- 在各进程执行过程中再分别开启一定数量的线程，去共同执行进程内的计算任务来实现对性能的提升。
- 在该混合编程架构中，不同进程的线程不能随时访问其它进程中的线程，这使得代码可维护性高，编程难度低，易保证程序的正确性。

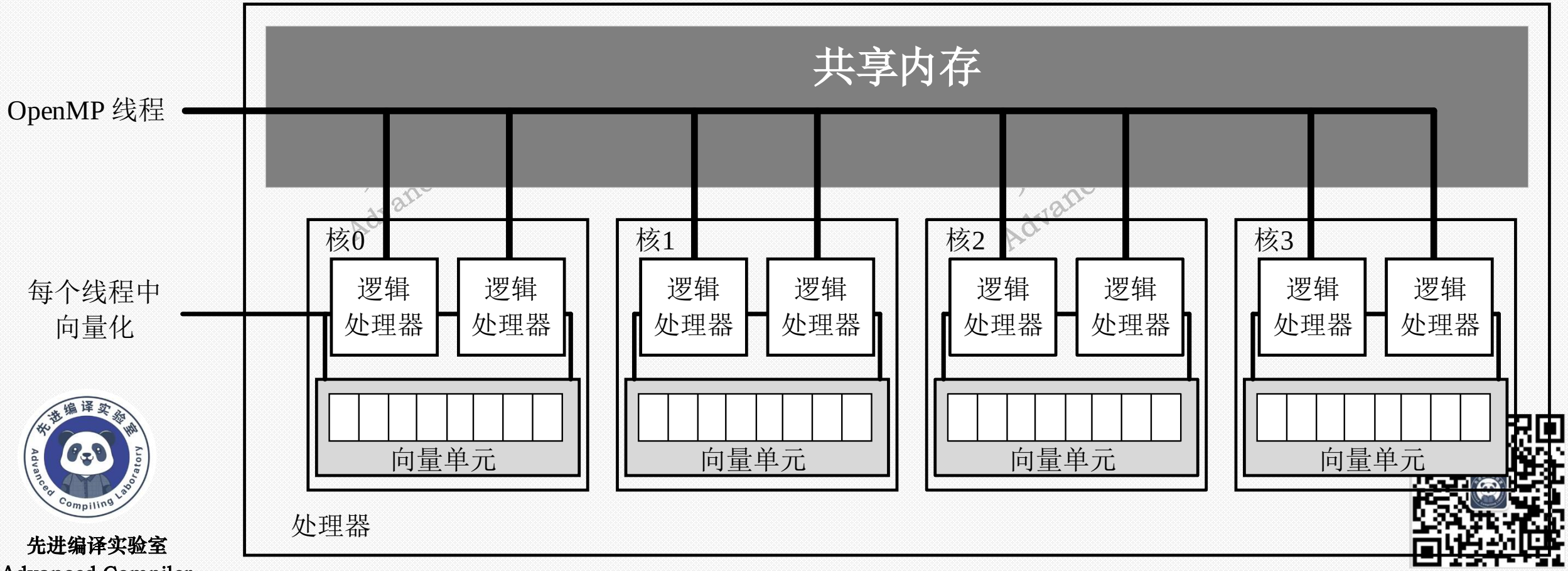


- 使用MPI+OpenMP的多层次并行方案后如下图所示，假设每个进程创建8个线程，对应分配到的循环为最外层i层的500次迭代。因为每层i循环的运算不存在数据依赖，所以在执行过程中可以直接将迭代次数平均分为8份，每个线程约执行63次，即线程0执行迭代的0~62，线程1执行迭代的63~125，以此类推直到线程7执行迭代的438~499。

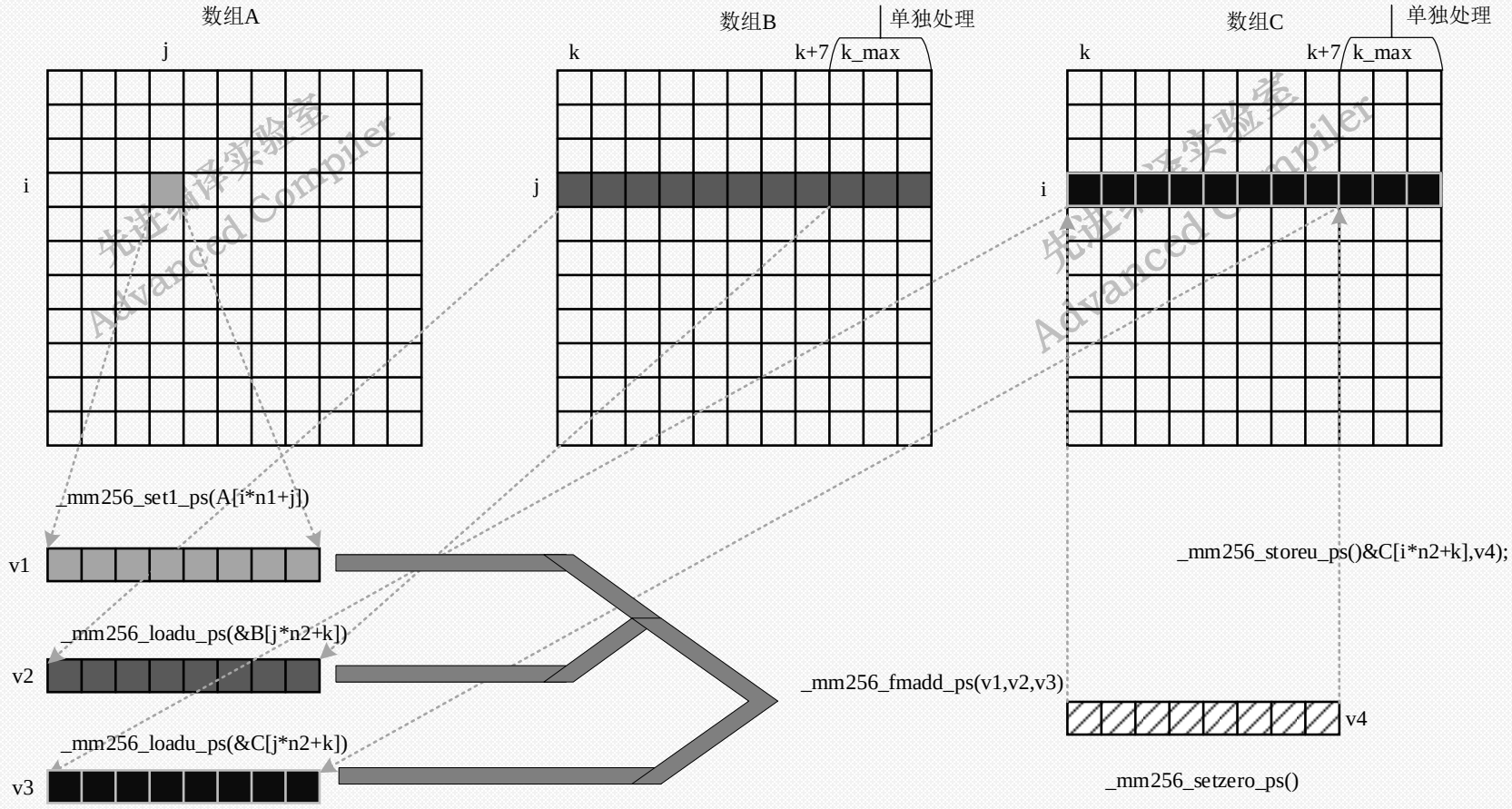


- SIMD并行能够通过单条向量指令对内存中的多个连续数据进行并行运算，因此可以使用SIMD并行对MPI+OpenMP混合编程实现的棋盘式分解矩阵乘进一步的优化，从更小的粒度即数据级并行层次上去考虑程序优化。
- 对基于MPI+OpenMP混合代码的矩阵乘法使用SIMD指令进行向量化以进一步优化。Hygon C86处理器同样采用X86架构，其指令集与intel的AVX指令集相同，使用SIMD指令对矩阵乘法进行向量化重写，该指令集支持256位数据的向量运算。

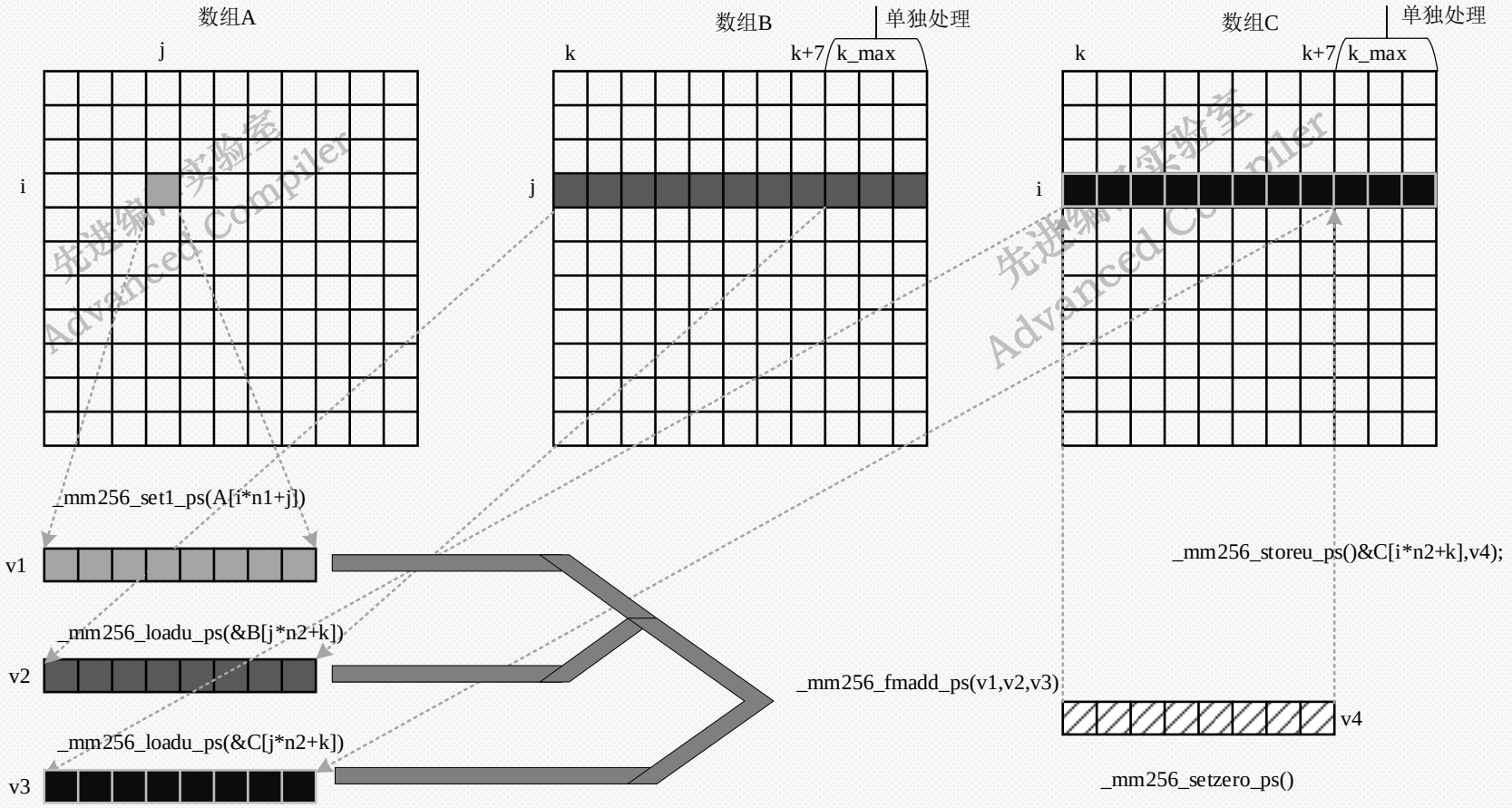
- 如下图所示，在一个多核处理器内，每个核心都使用了超线程技术，能够将一个处理器虚拟为两个逻辑处理器从而做到2个线程并行计算，核内线程可以使用向量单元进行向量化计算。



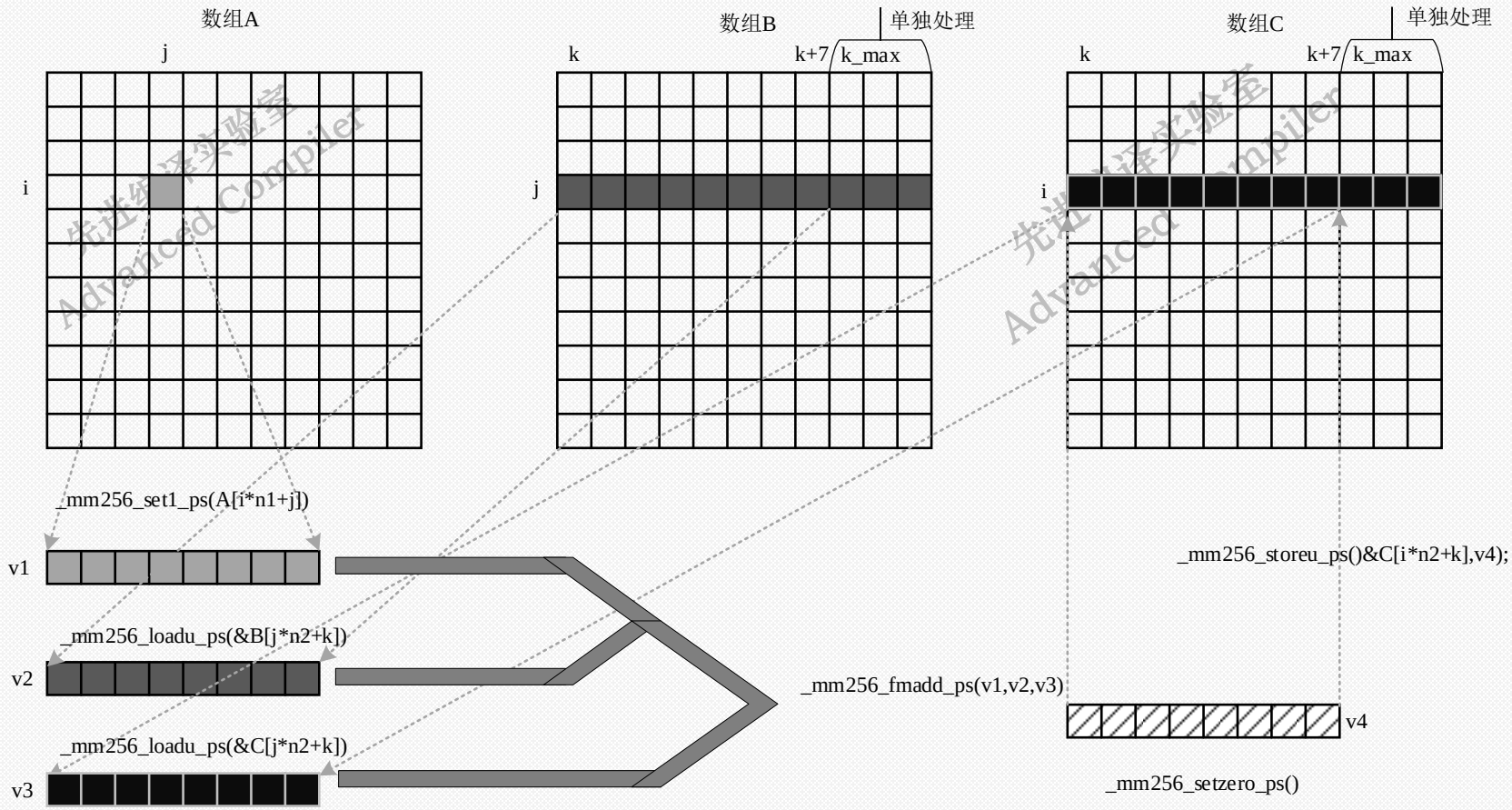
- 对矩阵乘法进行向量化的过程如下图所示，首先需要定义v1、v2、v3、v4四个数据类型为_m256的单精度浮点寄存器变量，存储大小为256位即8个32位的单精度浮点数据，四个变量分别用于存储每次读取数组A、B、C的数据以及临时计算结果。



- 在每次进行j层循环计算时需要完成第i行的计算任务，将最内层循环k设置循环步长为8来保证向量一次读入8个数据。



- 在每次迭代中，首先需要将存储C[i][k]~C[i][k+7]临时结果的向量v4初始化为0，之后将v1向量的数据全部置为A[i][j]与存储B[j][k]~B[j][k+7]数据的向量v2相乘，最后将结果向量与加载v3中的C[i][k]~B[i][k+7]相加后得到向量v4，再将v4保存至C[i][k]~B[i][k+7]中。

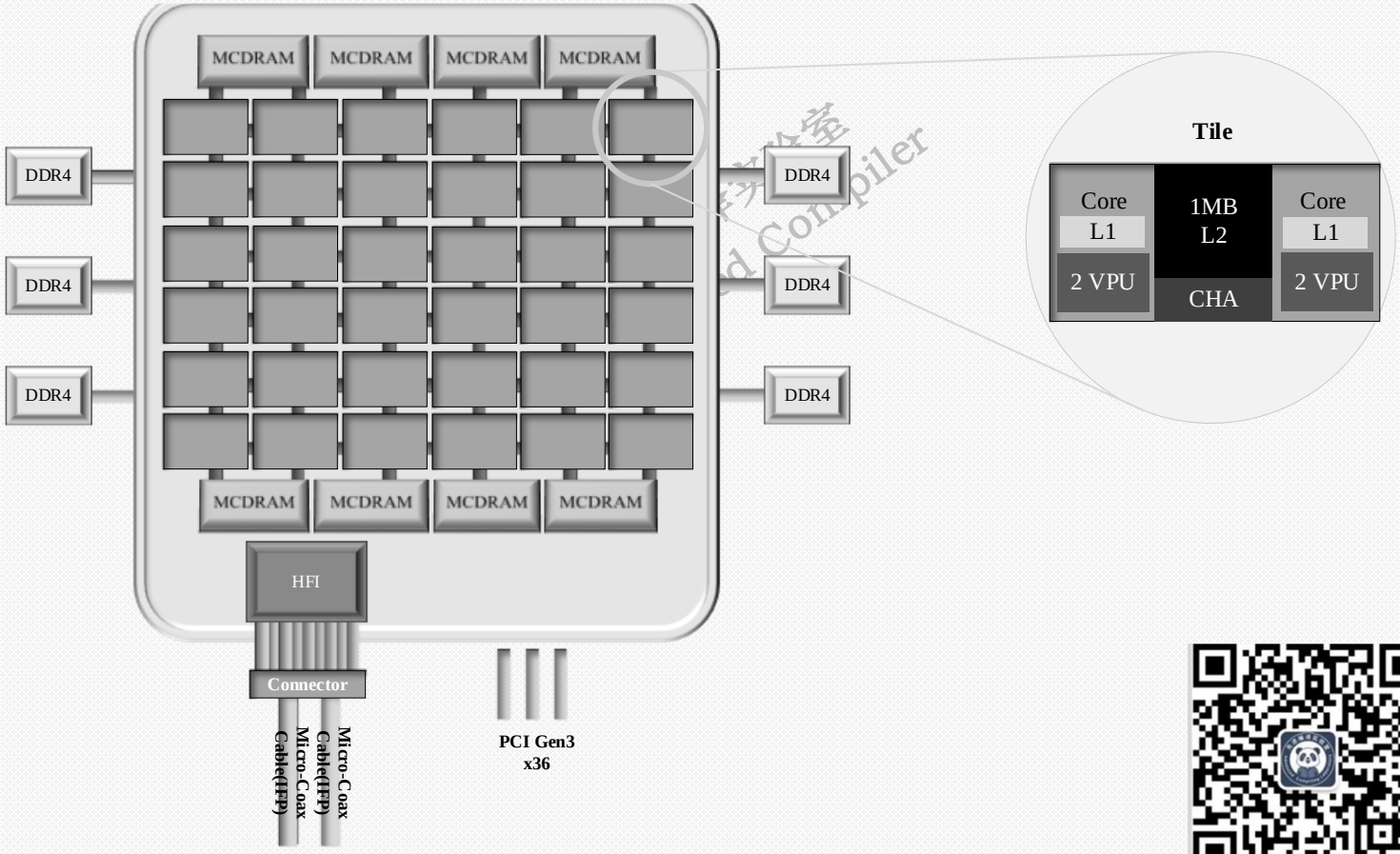


- 计算过程中所涉及的初始化、加载、保存、相乘相加运算等操作都通过使用下表中的AVX指令集部分内置函数来实现。

函数	功能
__m256 _mm256_setzero_ps (void)	将所有元素值置为0
__m256 _mm256_set1_ps (float a)	将所有元素全部设置为a值
__m256 _mm256_loadu_ps (float const * mem_addr)	将mem_addr指向的256位元素从内存中以非对齐方式加载至返回向量
__m256 _mm256_fmadd_ps (__m256 a , __m256 b , __m256 c)	将a和b中的元素相乘后，将中间结果与c中的元素相加
void _mm256_storeu_ps (float * mem_addr , __m256 a)	将a中的256位元素以非对齐方式存储到mem_addr指向的内存中

- KNL中有超过60个核心，每两个核心组成一个组，为便于描述将组简称为Tile。所有Tile中的处理器核通过片上的2D网络网格进行互连，每一行与每一列都是一个双向环形网络，有效地提升了处理器核间的通信效率。

- 在内存支持方面，KNL拥有6通道DDR4控制器，最高可支持384GB DDR4内存，除此之外，KNL片上还配备了16GB带宽超过400GB/s的多通道DRAM，即MCDRAM。在I/O支持方面，KNL支持36个用于I/O的PCIe Gen3通道。

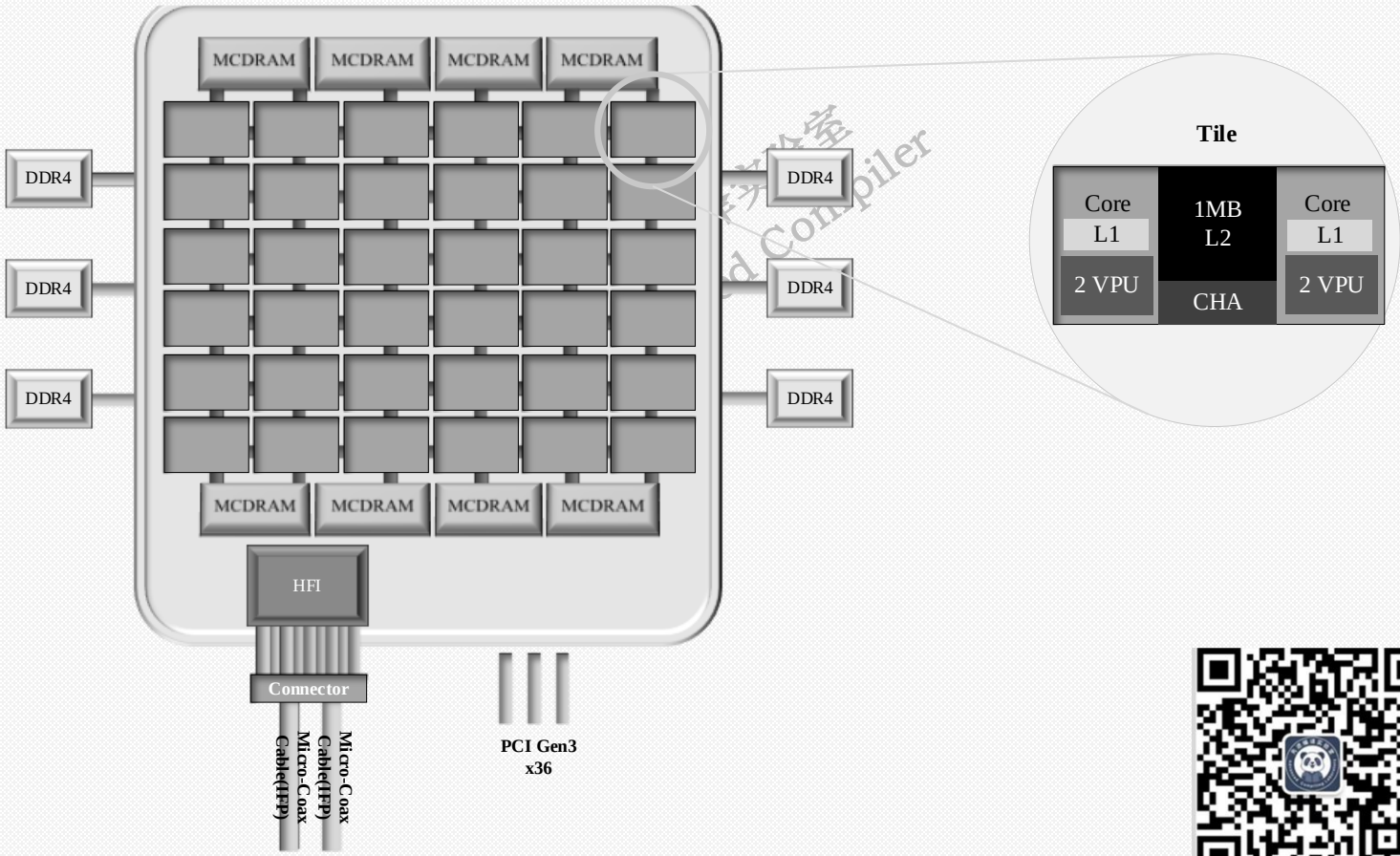


(a)

(b)

- Tile内部架构中，每两个核心组成一个Tile，每个核心拥有32KB的私有一级缓存，并共享1MB的二级缓存，通过基于目录的MESIF协议由缓存或本地代理CHA实现全局一致性。

- KNL的核心还包含32个512位的向量寄存器，2个支持512bit SIMD操作的向量处理单元VPU，使得其可以在一个周期内执行16个单精度或8个双精度的SIMD指令。同时针对高性能计算做了相应优化，支持4路超线程技术。

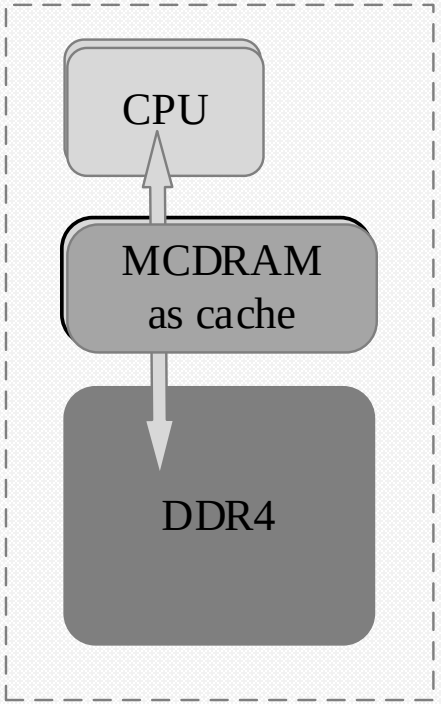


(a)

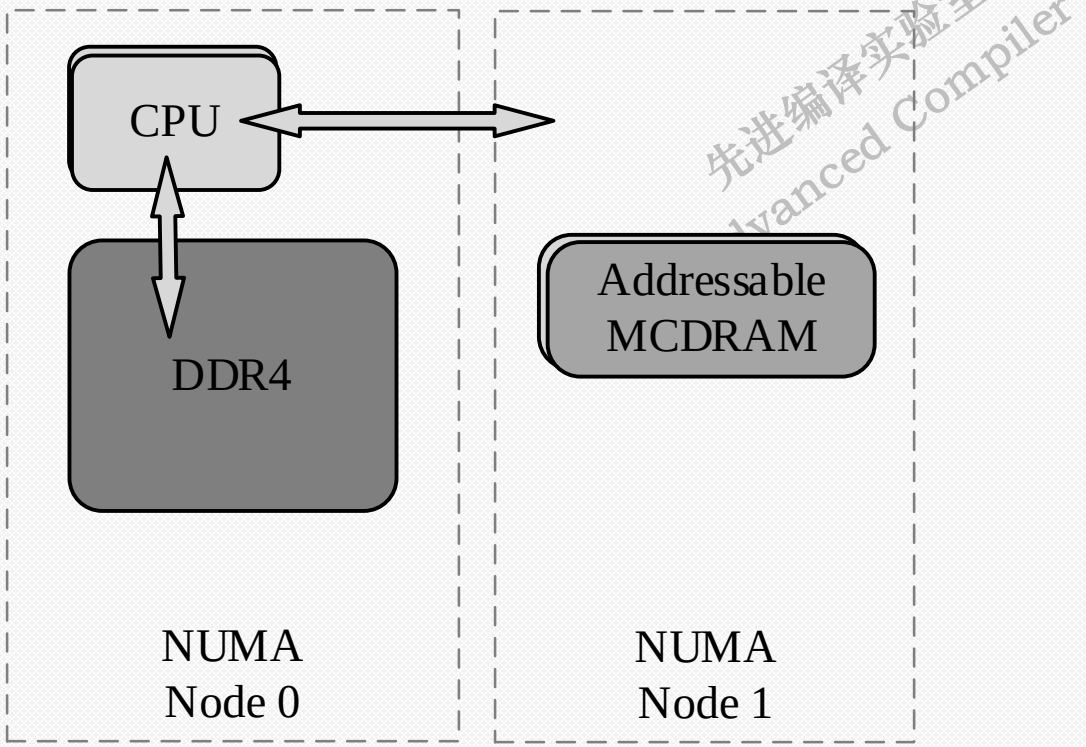
(b)

- 计算机系统中的内存带宽受限是计算应用中性能常见瓶颈之一，
- 在解决该问方面，KNL支持有MCDRAM内存，该内存可以在三种不同的模式下工作：
  - ✓ Cache模式
  - ✓ Flat模式
  - ✓ Hybrid模式
- 优化人员通过在启动时的BIOS设置中选择不同的MCDRAM内存模式，作为缓存或可编址高带宽内存以满足不同类型计算程序的内存需要。

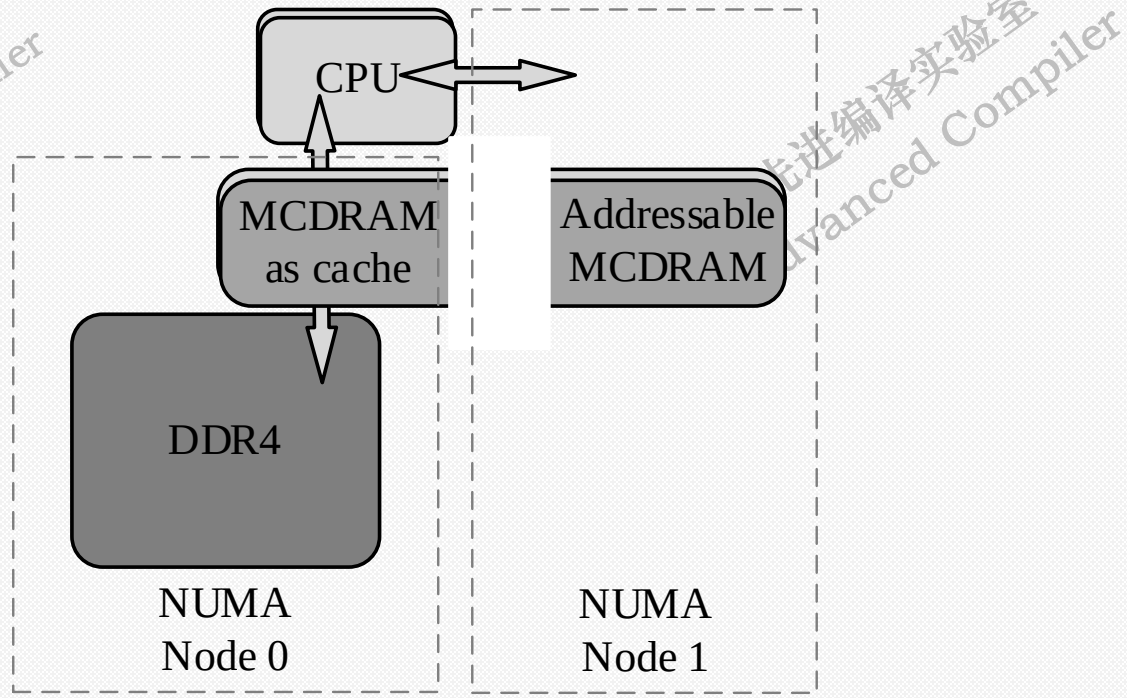
- 在Cache模式下的MCDRAM，采用直接映射的方式作为DDR4内存的缓存，对用户来说完全透明，由硬件来管理如何使用。在该模式下，所有请求将首先进入MCDRAM进行缓存查找，然后在未命中的情况下被发送到DDR4中。对于内存占用量非常大、内存带宽要求非常高且内存访问模式正常的应用程序，设置为这种模式往往是最佳的。



- 在Flat模式下的MCDRAM将作为独立的NUMA结点呈现给操作系统，与DDR4内存映射至同一系统物理地址空间。操作时需要明确地将内存分配到MCDRAM，用户可以通过numactl或memkind库函数管理分配。对于内存带宽要求高但内存占用量适中的应用程序，设置为这种模式往往是最佳的。

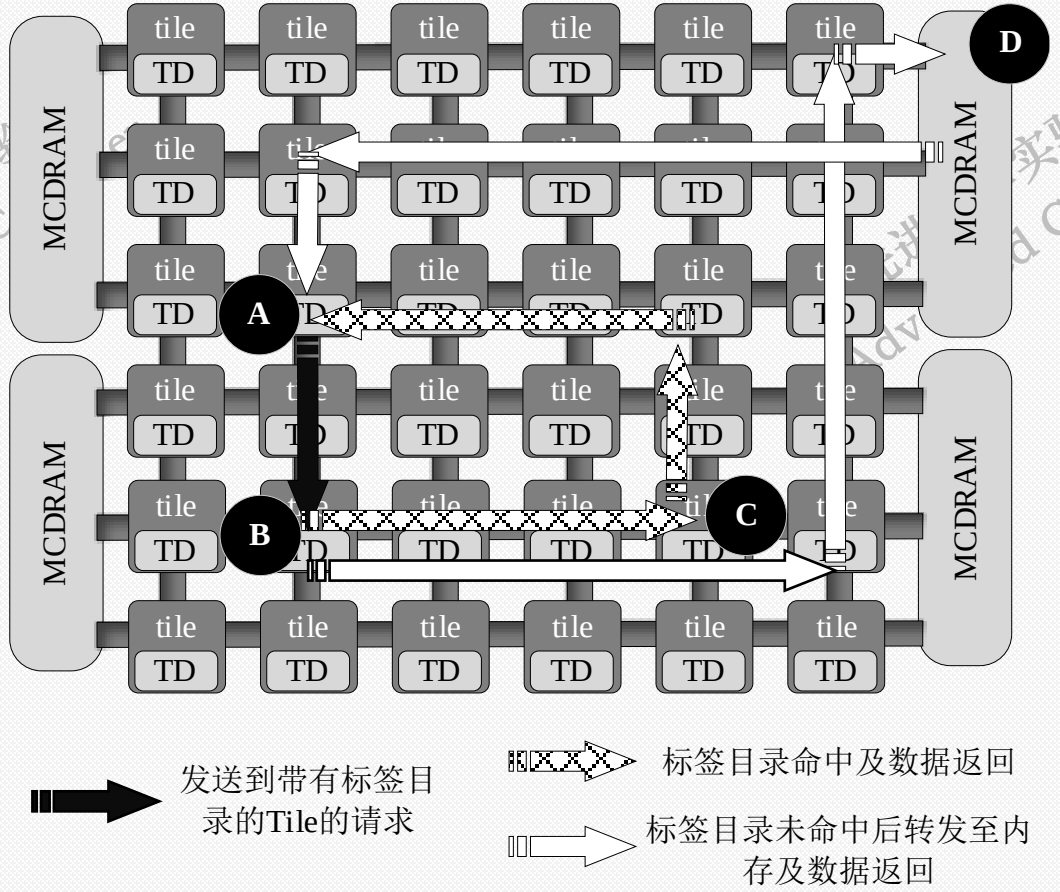


- Hybrid模式则是将Cache模式和Flat模式混合起来的模式。MCDRAM被分为两部分，一部分工作在Cache模式，一部分工作在Flat模式。可以在BIOS中选择设置MCDRAM的25%、50%、75%三种比例，此模式适合希望完全优化其代码或工作流的高级用户。

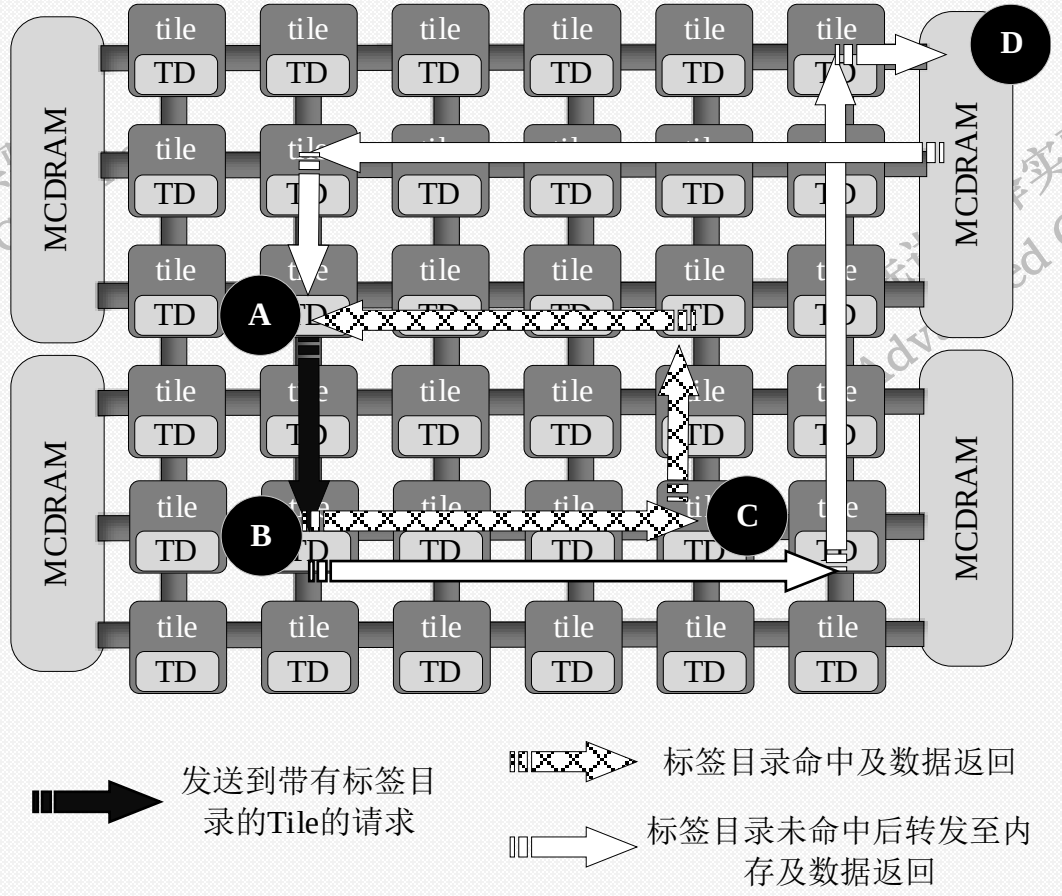


- KNL中的每个内核都有一个一级缓存，被组织为一个Tile的两个内核共享二级缓存，每个二级缓存通过网格连接到其它内核且使用MESIF协议保持缓存一致性。
- 具体实施为KNL有一组块标签目录TD（tag directory）组成的分布式标签目录DTD(distributed tag directory)用于标识任何缓存行的状态及其在芯片上的位置。
- 对于任何内存地址，硬件都可以通过哈希函数识别负责该地址的TD。

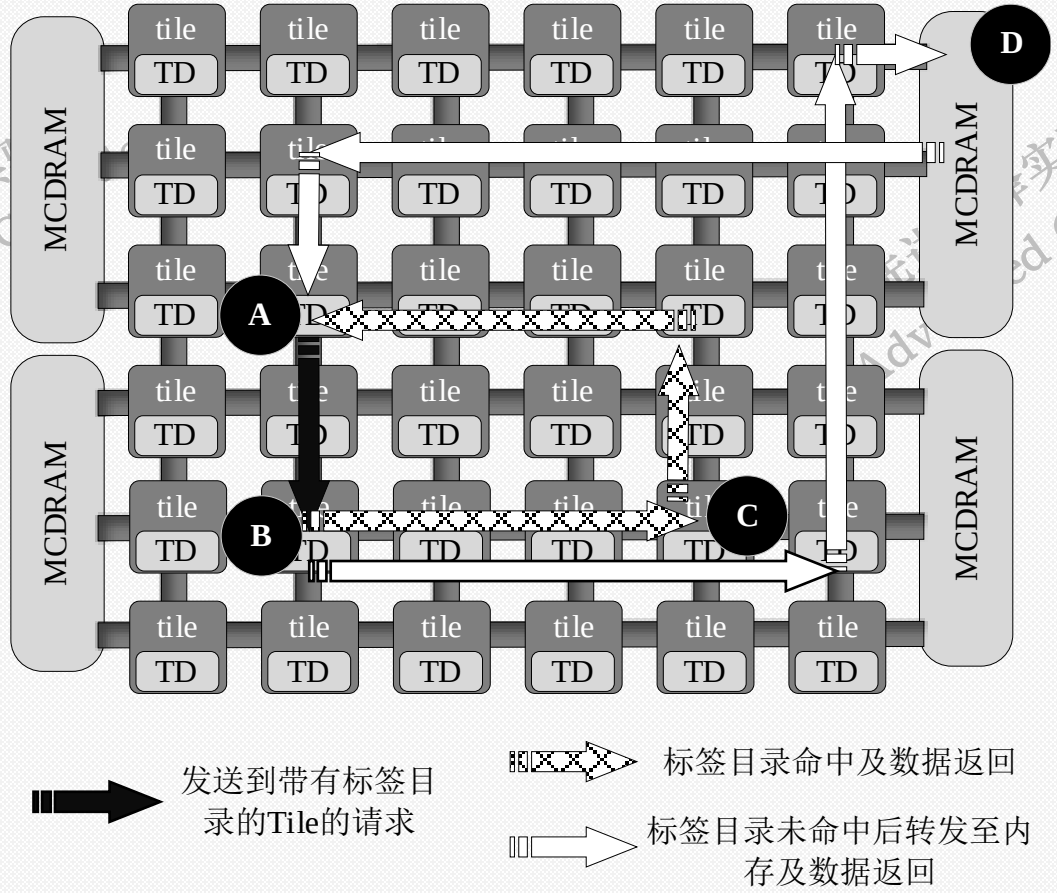
- 内存访问过程如下图所示，当应用程序从内存地址请求数据时，正在处理的Tile A将先查询本地缓存以查看请求的内存地址是否存在，若存在则计算将以最小的数据访问延迟进行，否则Tile A将在DTD中查询包含该数据的缓存行，如图中黑色填充的箭头所示。



- 如果根据该TD得知，该缓存行存在于其它Tile C的二级缓存中，则另一条消息将从Tile B发送到Tile C，最后Tile C将数据发送到Tile A，如图中网格填充的箭头所示；



- 如果请求的内存地址没有被缓存，Tile B会将请求转发给负责该地址的内存控制器D，D是基于MCDRAM的内存控制器，也可能是基于DDR4的内存控制器，从内存控制器中读取数据后发送给Tile A，如图中白色填充的箭头所示。



- 由此可见，随着芯片硬件复杂性的增加，KNL缓存组织也十分复杂，使得内存访问过程的复杂性随之增加。

- 为了降低内存访问的复杂性，针对不同的计算应用程序，KNL支持有All-to-All、Quadrant、Hemisphere、SNC-4和SNC-2五种不同的集群模式以尽可能降低这种访存开销，集群模式的设置也必须通过BIOS来进行。

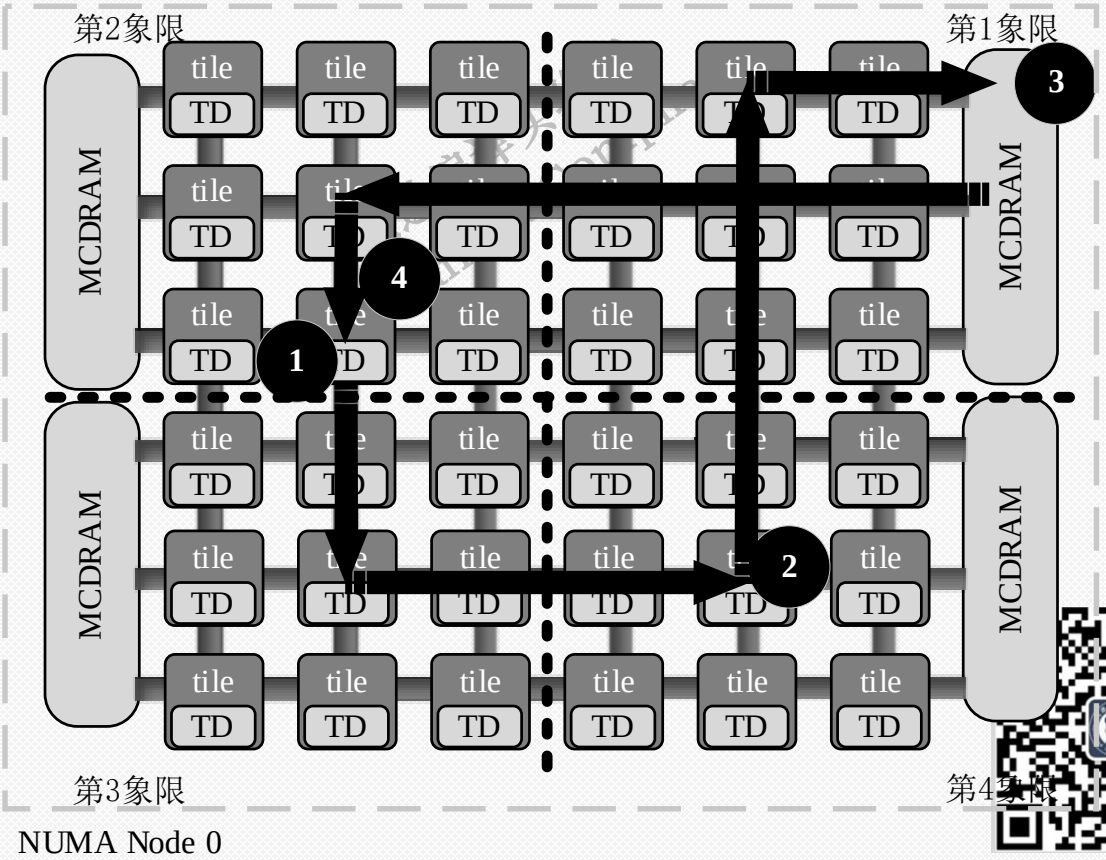
  ✓ All-to-All模式下，Tile、分布式目录和内存之间没有任何关联，内存、分布式目录均匀分布在所有核中。这是最通用的模式，对软件的内存配置没有特定的要求，但它的性能通常低于其它集群模式。

- ✓ Quadrant或Hemisphere模式将所有核心分为4个或2个虚拟象限，分布式目录与该目录对应的内存数据处于同一个象限中，但Tile与分布式目录、内存之间没有关联，来自任何Tile的请求都可以到达任何目录，但该目录将仅访问其所在象限中的内存。该模式提供比All-to-All模式更好的延迟，并且对软件的支持是透明的。
- ✓ SNC-4或SNC-2模式进一步扩展了Quadrant或Hemisphere模式，将Tile、分布式目录以及内存三者关联起来，所有核心被划分为4个或2个NUMA域。来自Tile的请求将访问所在集群的目录，然后该目录也将访问所在集群的内存。因为大多数流量将包含在本地集群中，在该集群模式下，尤其是在负载操作下，为所有模式中最佳延迟。对于利用此模式性能的程序，必须运行在同一个NUMA集群中分配内存，以进行NUMA优化发挥最优性能。

- 在All-to-All模式下，二级缓存缺失且请求内存地址未缓存时，系统进行的相应操作。

- 二级缓存在第2象限发生缺失。首先在第4象限找到对应的分布式目录，之后在第1象限读取内存数据，最后将数据返回第2象限，整个过程跨越距离是这三种模式中最长的。

- 数字1表示二级缓存缺失
- 数字2表示直接访问，
- 数字3表示内存访问，
- 数字4表示数据返回，
- 黑色填充箭头代表数据流向，灰色虚线边框表示NUMA节点，黑色虚线代表逻辑象限。

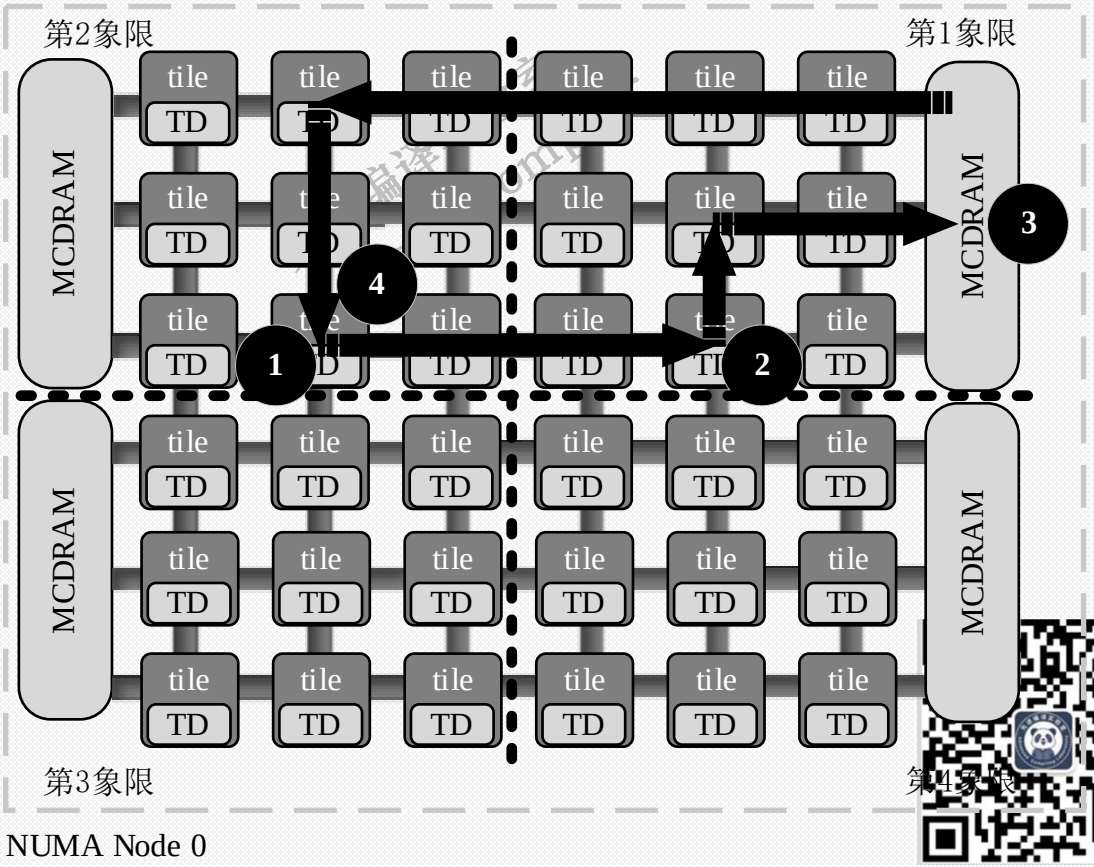


NUMA Node 0

- Quadrant或Hemisphere模式下，二级缓存缺失且请求内存地址未缓存时，系统进行的相应操作。

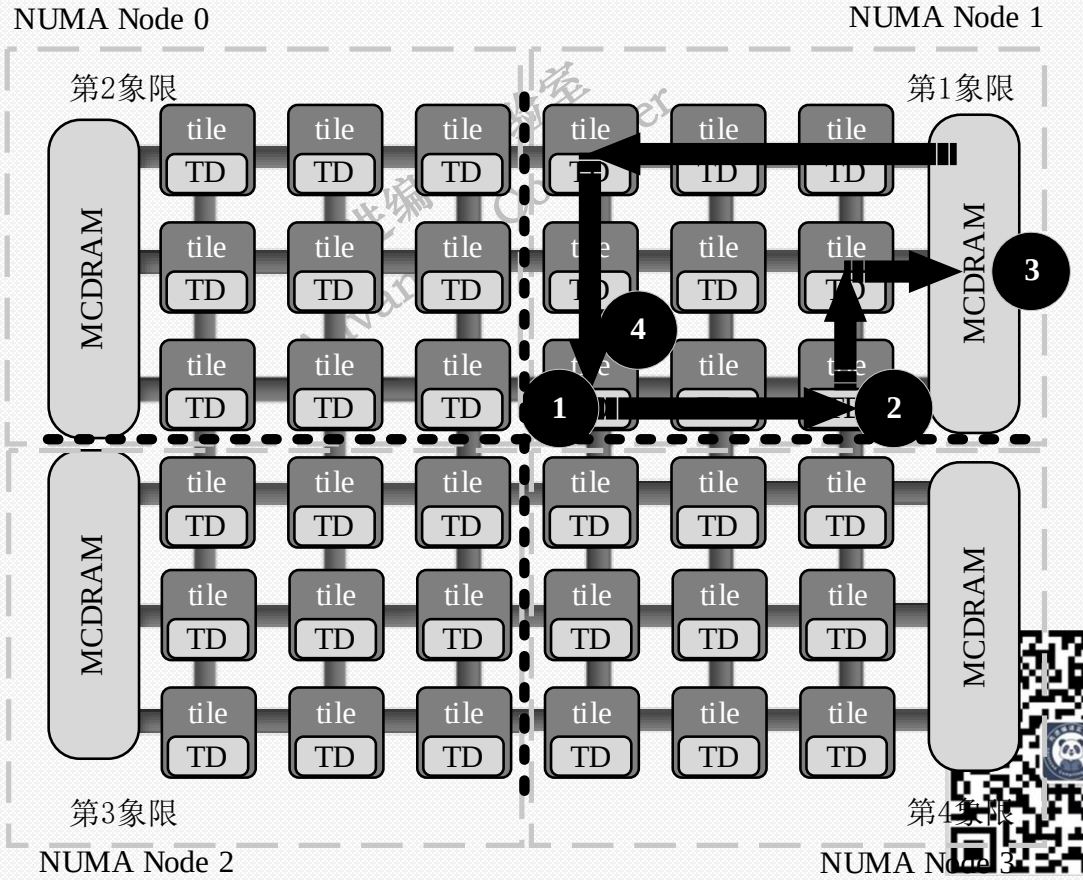
- 二级缓存在第2象限发生缺失。首先在第1象限找到对应的分布式目录，由于本模式下分布式目录与对应内存数据处在同一象限，因此在分布式目录所在的第1象限读取内存数据，最后将数据返回第2象限，整个过程跨越距离相对较短。

  - 数字1表示二级缓存缺失
  - 数字2表示直接访问，
  - 数字3表示内存访问，
  - 数字4表示数据返回，
  - 黑色填充箭头代表数据流向，灰色虚线边框表示NUMA节点，黑色虚线代表逻辑象限。



- SNC-4/SNC-2模式下，二级缓存缺失且请求内存地址未缓存时，系统进行的相应操作。

- 第1象限发生二级缓存缺失。由于CPU物理上被分为4个象限，因此在发生缓存缺失的第1象限内找到分布式目录并在本象限内获取内存数据，整个过程跨越距离最短。

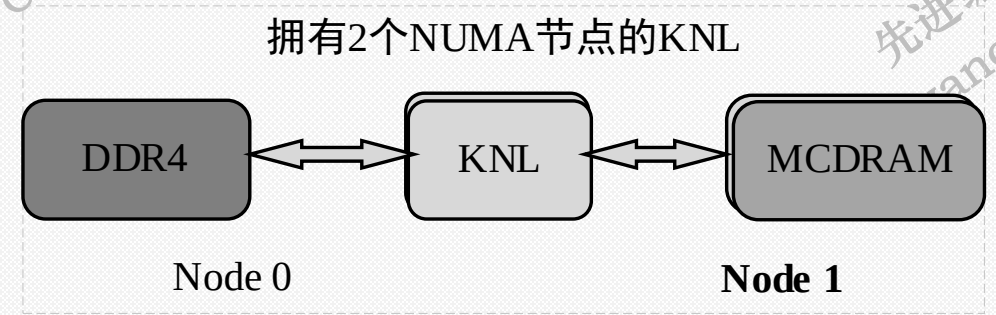


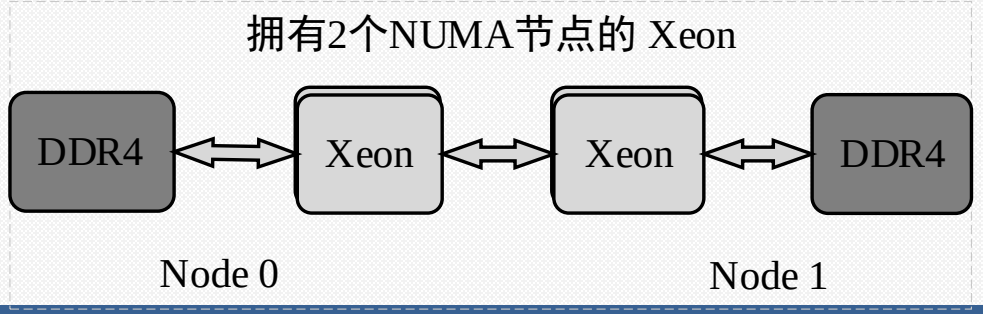
- 数字1表示二级缓存缺失
- 数字2表示直接访问，
- 数字3表示内存访问，
- 数字4表示数据返回，
- 黑色填充箭头代表数据流向，灰色虚线边框表示NUMA节点，黑色虚线代表逻辑象限。

- KNL支持AVX-512扩展指令集，AVX-512是x86指令集架构ISA（Instruction Set Architecture）的512位SIMD指令扩展，能够一次处理512位的数据，即进行16个单精度或8个双精度数据的并行处理。
- AVX-512由多个扩展组成，支持AVX-512的处理器并不要求支持AVX-512的所有扩展，只要求支持其核心部分AVX-512F。
- AVX-512F使用EVEX编码方案扩展大多数基于32位和64位的AVX指令，以支持512位寄存器以及掩码操作、广播、移位、异常处理等。
- 为了充分利用KNL的硬件特性，需要进一步将Mul_Matrix函数使用AVX-512指令集进行优化改写使得性能进一步提升，改写的过程较为简单，需要将程序中定义的寄存器变量的向量长度由原来的256位，即__m256，改为512位的__m512。

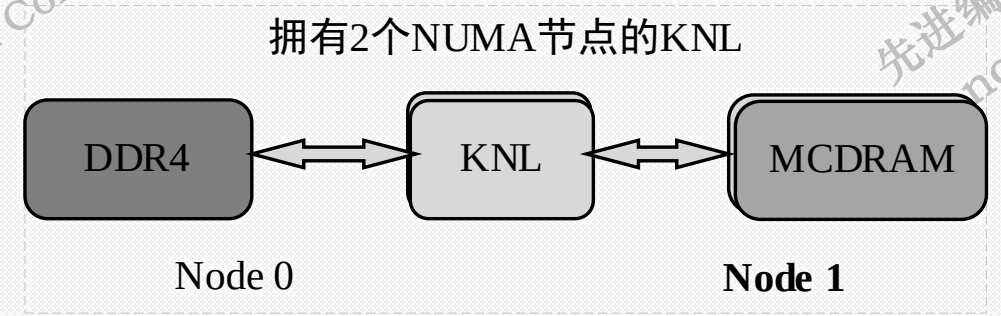
- KNL中内存模式与集群模式均可自行调整，其中Flat模式需要软件明确地将内存分配到MCDRAM，可以借助HBW_malloc库中的内存分配函数分配内存，使得编写的程序具有可移植性。

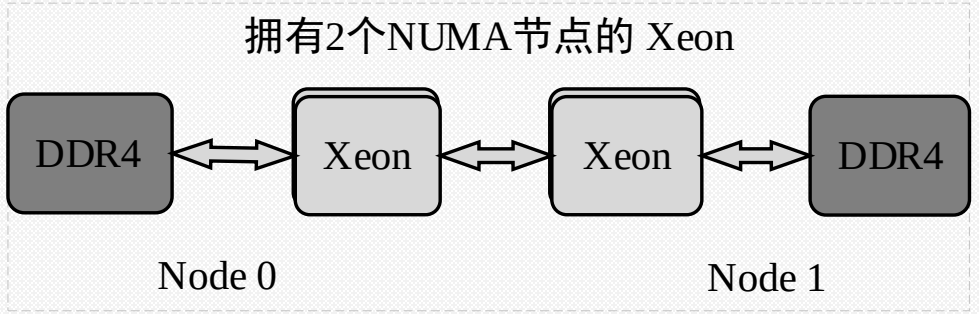
- 使用HBW_malloc库编写程序具有可移植性的原因是其利用NUMA机制对两种内存进行寻址。如图所示，对于没有MCDRAM内存的系统，将默认使用标准内存分配策略，而对于两种内存都存在的系统，MCDRAM就像双插槽系统中常规的NUMA内存。





- 具体地来说，在DDR中分配内存的函数名为malloc，在MCDRAM中进行内存分配的函数名为hbw_malloc，在Flat模式下运行上述矩阵乘法程序，则需要将内存分配的函数进行相应替换，同时与之配套的free内存释放函数也需要替换为hbw_free，在编写程序时需要包含<hbwmalloc.h>头文件，编译时需要链接-lmemkind库。





12.3 Hygon DCU异构众核平台

- **异构架构**是指在系统上存在着两种或两种以上架构，不同架构上的编程模型、计算与访存模式、通信方式等不同。
- 本节将在拥有Hygon DCU设备的异构平台上，继续以矩阵乘法为例介绍多层次并行程序的一般编写方法以及优化方法。

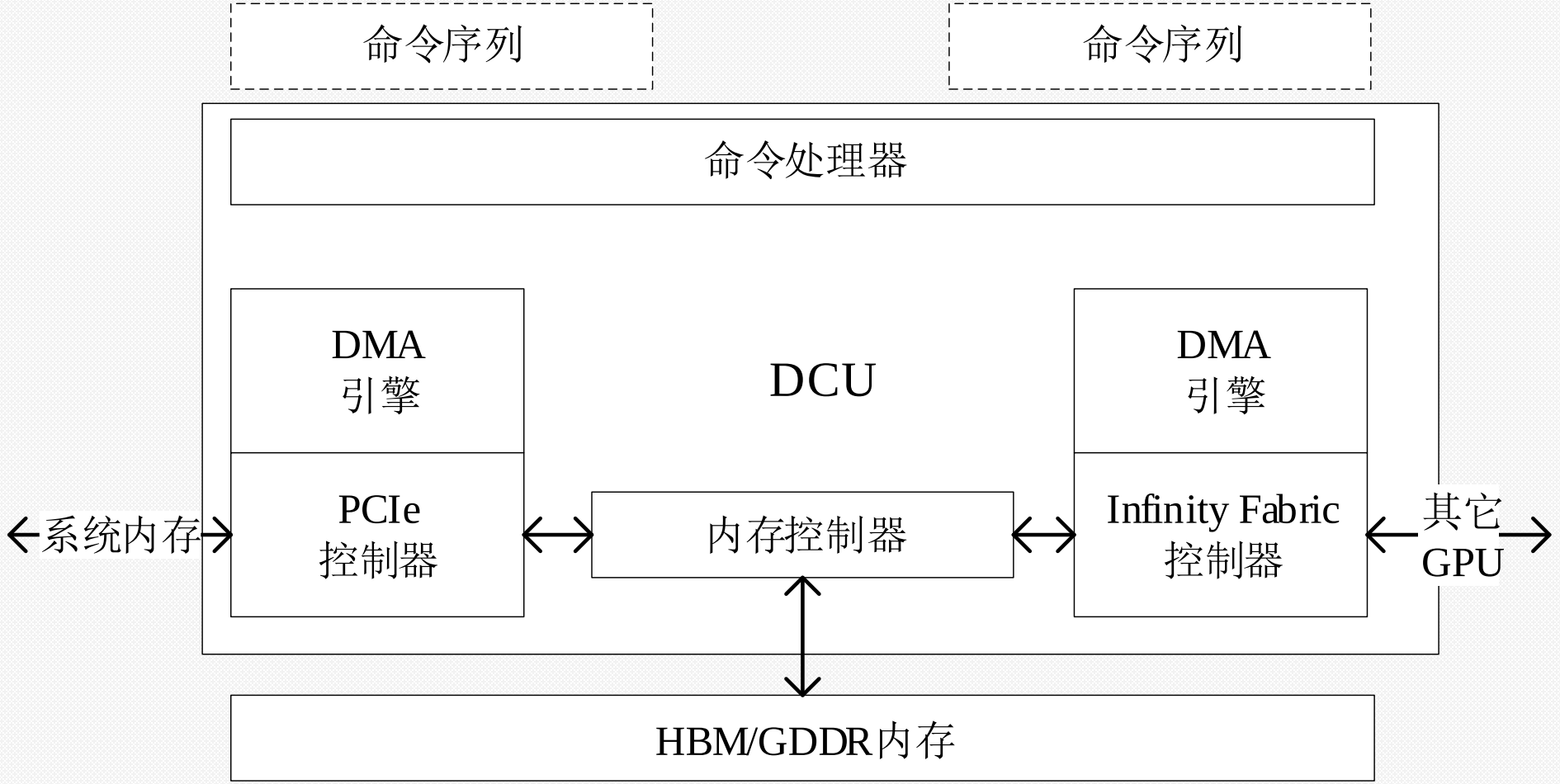
- 深度计算器（Deep Computing Unit,DCU）是海光(HYGON)推出的一款专门用于人工智能和深度学习的加速卡。该加速卡以GPGPU架构为基础，精简了用于逻辑判断、分支跳转和中断处理等功能的控制单元，在芯片上设计了数量众多的算术运算单元，其主要特点有：

① 强大的计算能力。基于GPU的大规模并行计算微结构的设计不但使其具备强大的双精度浮点计算能力，同时在单精度、半精度、整型计算方面表现同样优异。

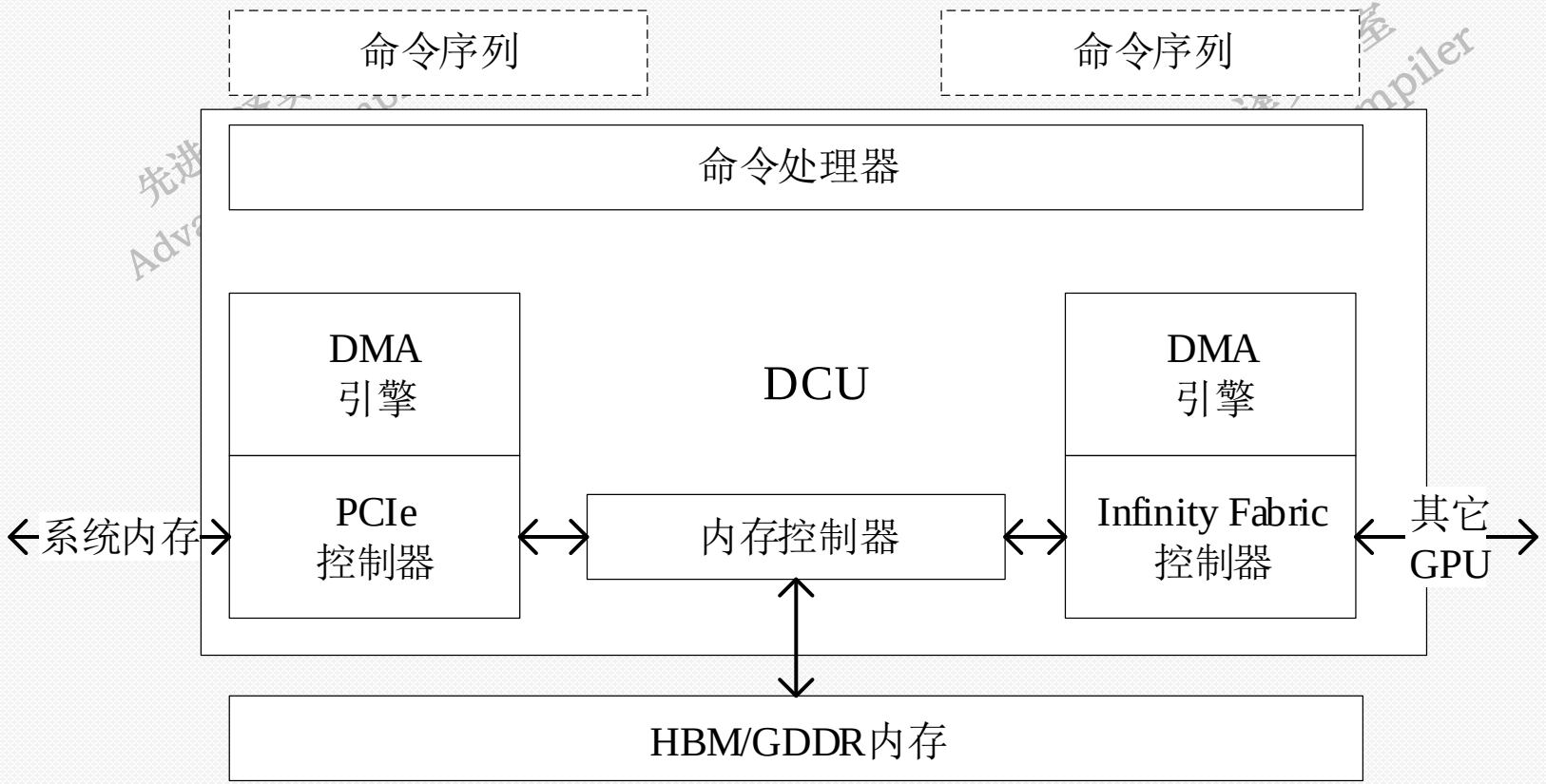
② 高速并行数据处理能力。海光DCU集成片上具有高带宽内存芯片，可以在大规模数据计算过程中提供优异的数据处理能力，使其可以适用于广泛的应用场景。

③ 良好的软件生态环境。海光DCU采用GPGPU架构，兼容类CUDA环境，实现了与AMD GPGPU主流开发平台的兼容。

- DCU硬件架构如图所示，其中Infinity Fabric为AMD公司开发的一种总线技术，可以用于CPU-CPU、CPU-GPU、GPU-GPU之间传输数据。HBM以及GDDR是用于制造GPU显存的架构类型。



- 由图可以看出，在程序执行时DCU的指令处理器可以以串行的方式从命令序列中读取并处理新的命令。在获取命令所需的数据时既可以通过PCIe控制器与系统内存交换数据，也可以通过Infinity Fabric控制器与其它DCU交换数据，不过最终这些通过内存控制器获取的数据都将通过总线传输到HBM或GDDR类型的片外全局内存中。



- 与英伟达GPU中的CUDA编程模型类似，DCU加速卡也需要使用专门的编程模型才能调用。
- DCU使用的是HIP编程模型，其是一种显式并行编程模型。
- HIP异构编程模型与CUDA类似，可以较简便的对已有异构算法进行迁移。
- 同时，也可以使用自动化程序迁移工具hipify将CUDA编写的异构并行程序快速切换至HIP编程模型。

- 与CUDA编程模型类似，HIP编程模型同样有主机端和设备端之分，其区别和联系如下：

① 主机端是指CPU设备，设备端是指DCU设备。

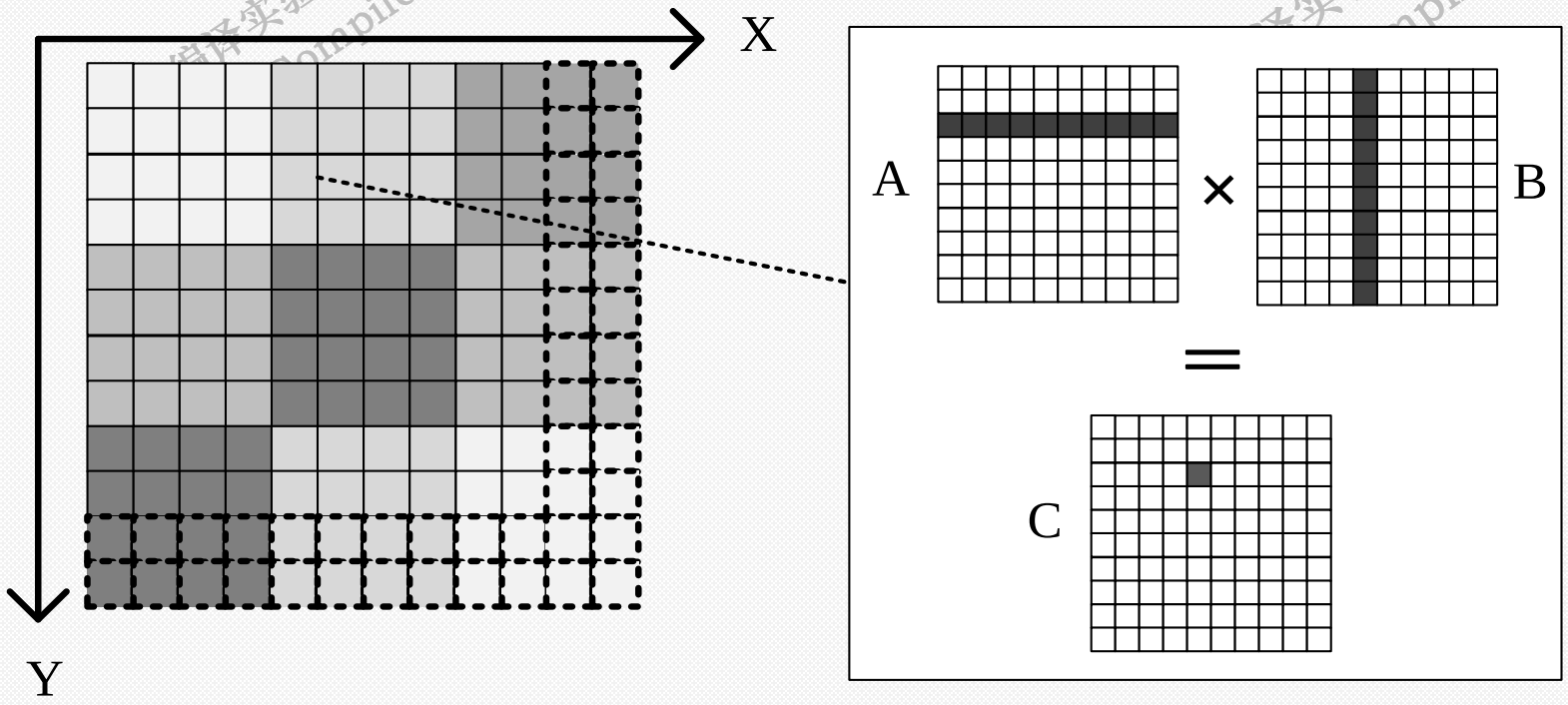
② 主机端代码在CPU上运行，入口函数是main，设备端代码在DCU上运行，由主机端通过核函数进行调用。

③ 主机端使用C++语言编写，设备端代码使用扩展的C语法编写。

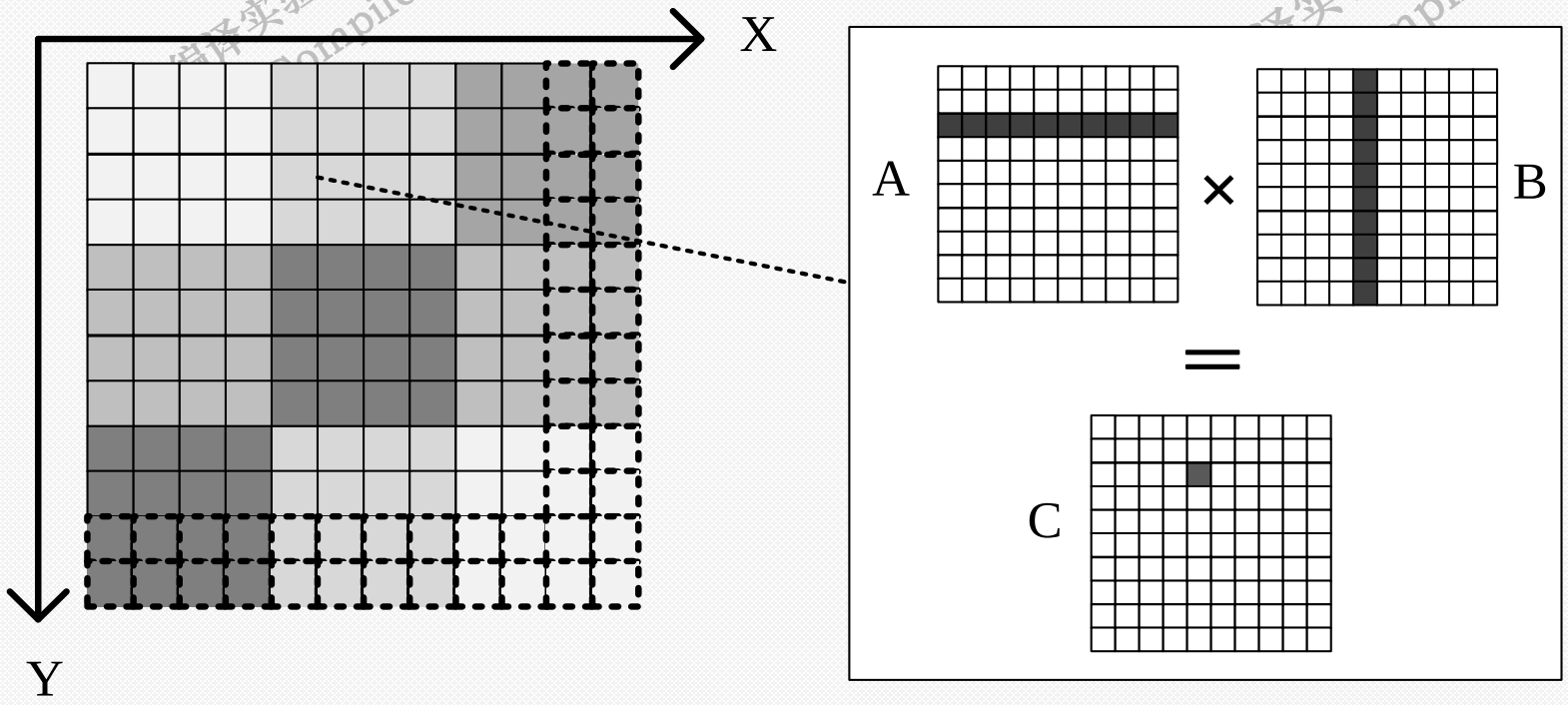
④ HIP使用Runtime API在主机端分配设备显存，管理主机端和设备端的内存拷贝，运行设备端核函数等。

⑤ 主机端以流的方式向设备端提交指令。

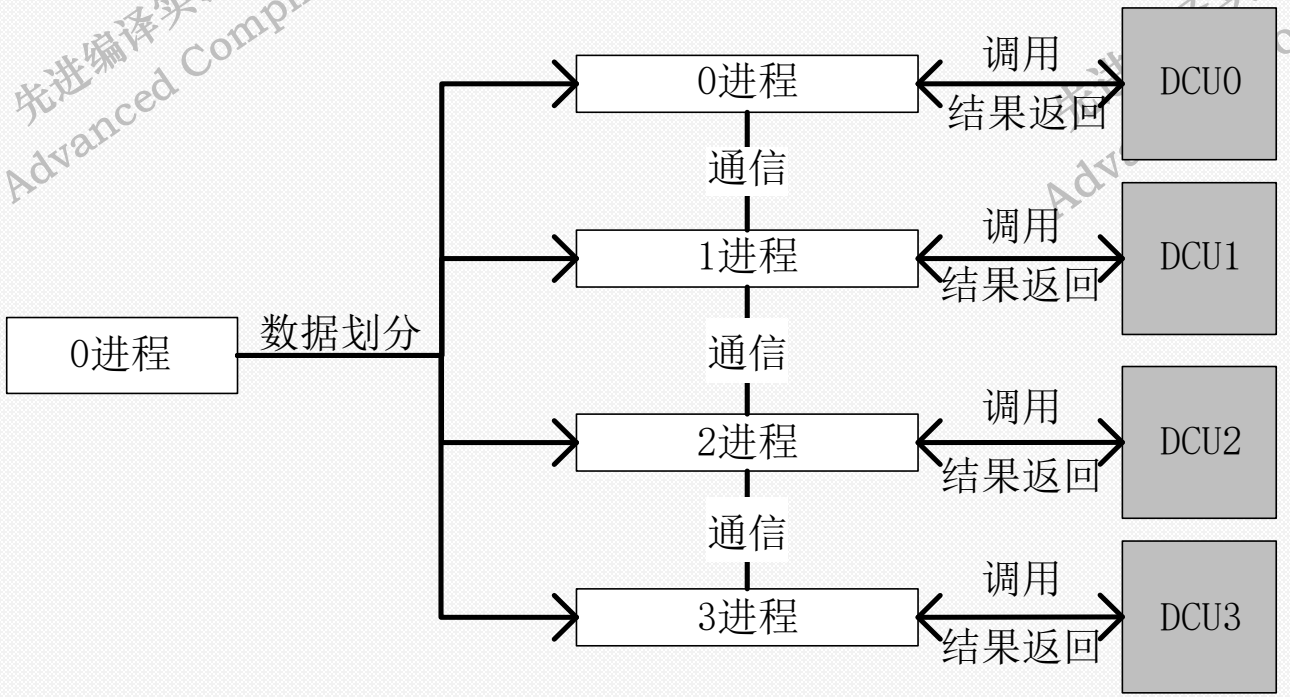
- HIP编程与CUDA编程类似，在执行任务时需要将任务按照网格进行划分，之后每个网格上使用一个线程进行计算。下图是当块大小为4时使用二维网格进行2个10*10的矩阵相乘时的映射规则。由于每个块必须是4*4的，因此在映射时必须虚拟出多余的2行以及2列的网格，以满足将任务划分为一个3*3的网格大小。



- 一共并发出(10+2)*(10+2)个线程去执行任务。其中对于横纵坐标都小于10的网格，例如在图中标注的第3行第5列的网格上，DCU设备将会并发出一个线程去执行A矩阵的第3行与B矩阵的第5列的向量乘法，之后将结果写入C矩阵的第3行第5列的元素中。对于该例网格中的最后2行以及2列，在并发出线程后不参与数据计算。



- 实现了基础执行版本的HIP矩阵乘之后，可以利用MPI技术将多个DCU设备组织起来共同完成计算任务，实际编程时的程序组织方法如图所示。其中具体的任务划分方法与MPI章节中按行分解矩阵乘法相同，首先由0进程将完整的矩阵B以及平均划分之后的矩阵A分发到包括本进程在内的各个进程中，之后每个MPI进程分别调用一个DCU设备进行计算，最后每个进程在接收到DCU计算结果后将其发送到0进程。



- 同时也可以通过流和事件在一个或多个进程中实现多DCU矩阵乘法，首先将划分后的矩阵A和完整的矩阵B通过异步传输函数拷贝到对应流控制的DCU中，然后调用核函数进行计算，最后将结果从对应流控制DCU返回到矩阵C对应位置。流和事件实现多DCU应用程序并行计算。其工作流程如下：

① 选择这个应用程序将使用的DCU集；

② 为每个设备创建流和事件；

③ 为每个设备分配设备资源；

④ 通过流在每个DCU上启动任务；

⑤ 使用流和事件来查询和等待任务的完成；

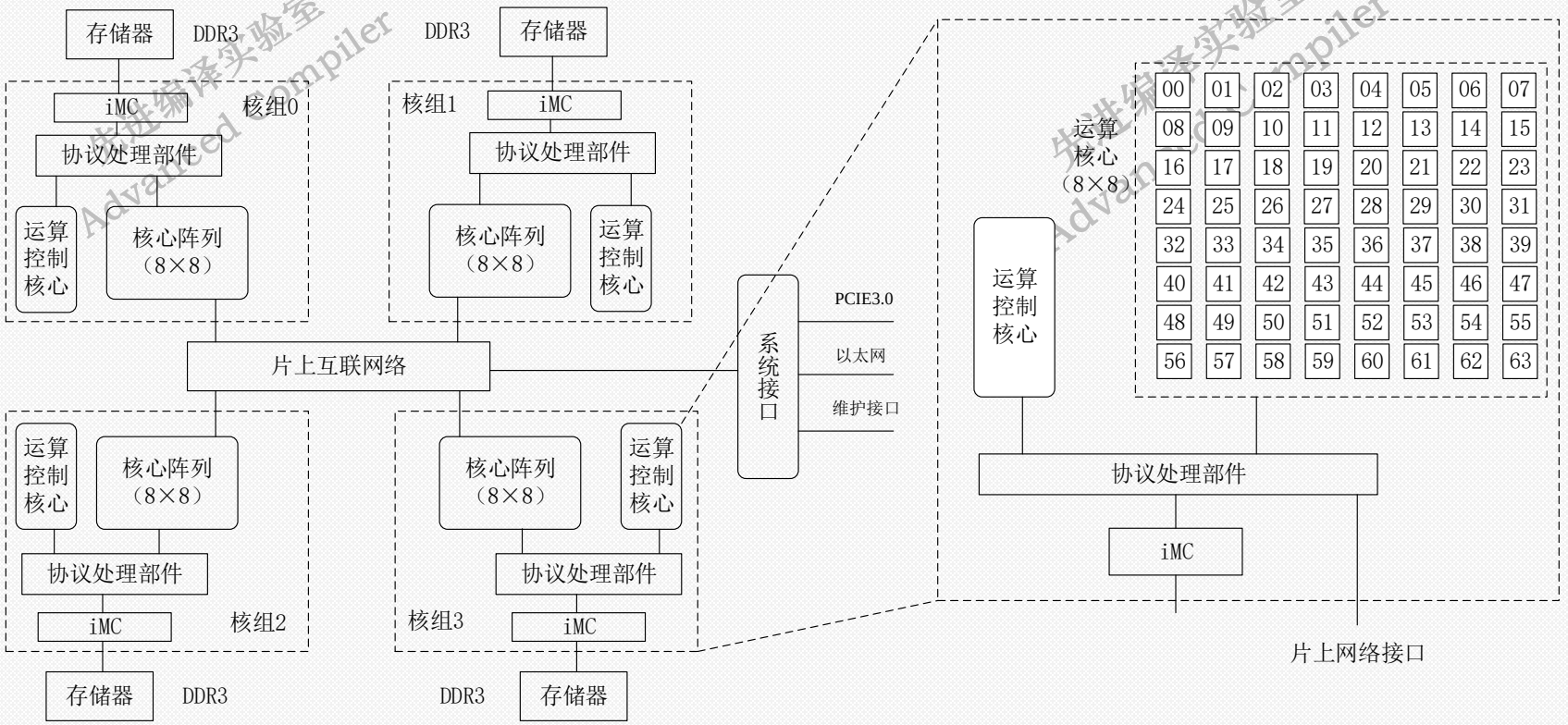
⑥ 清空所有设备的资源。

12.4 申威26010异构众核平台

- 在上一节中，通过对Hygon DCU异构多层次并行的介绍，基本了解了异构结构的并行编程方法。
- 本节将介绍国产异构众核申威26010处理器，除了对该处理器的架构、专用加速线程库等进行介绍外，还会详细讲解在此平台上程序的编写和优化。

12.4.1 平台介绍

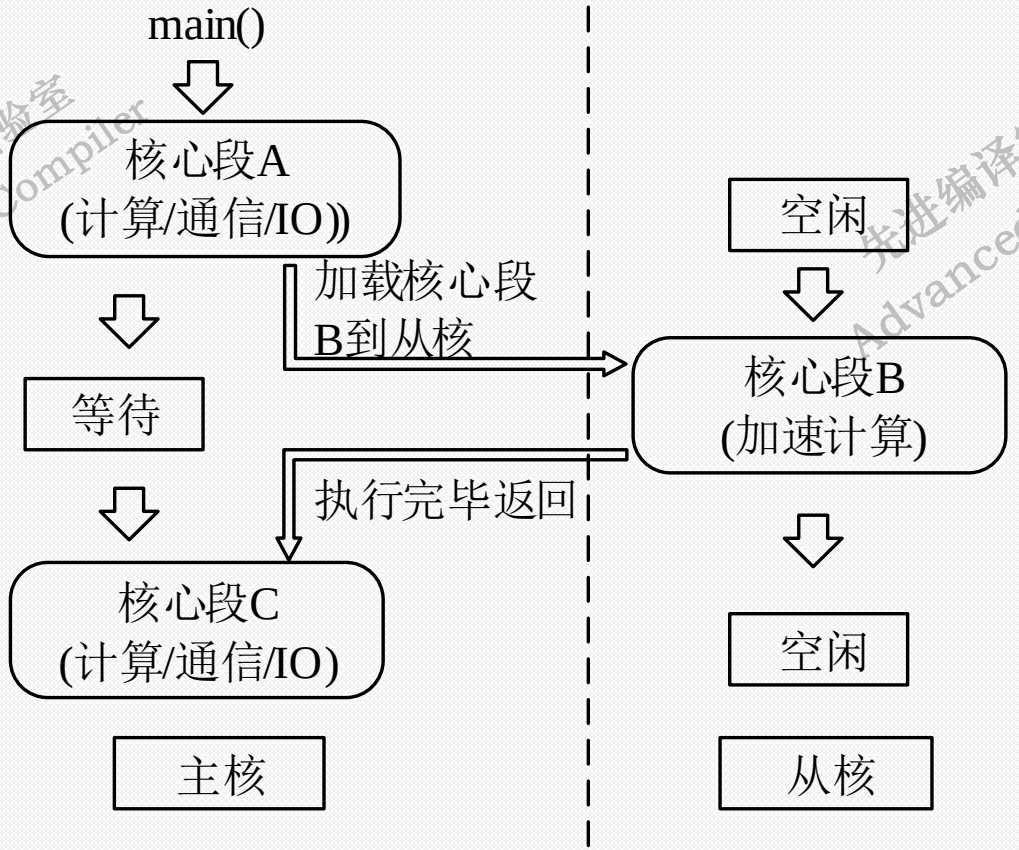
- “神威•太湖之光”超级计算机所搭载的核心计算部件为国产申威26010异构众核处理器，该处理器集成了4个运算核组共260个计算核心。每个运算核组包括1个主核和1个运算核心阵列。运算核心阵列也称为从核阵列，其由64个运算从核核心、阵列控制器和二级指令Cache等构成。4个核组的物理空间统一编址，每个核组上的主核和从核均可以访问芯片上的所有主存空间



12.4.1 平台介绍

- “神威•太湖之光”的每个计算节点包含一颗申威26010众核处理器，其内存为32GB。该众核处理器包括四个核组，每个核组有8GB的本地内存。从核可以直接离散访问主存，也可以通过DMA方式批量访问主存。从核阵列之间采用寄存器通信方式进行通信。每个从核局部存储空间LDM大小为64KB，指令存储空间为16KB。
- “神威•太湖之光”主要采用主从加速并行、主从协同并行、主从异步并行和主从动态并行四种异构并行方式，本章节以主从加速并行模式进行讲述。

- **主从加速并行**是指，应用的计算核心被加载到从核上进行加速计算，每个从核绑定一个线程，而主核只需完成应用程序的通信、I/O和部分串行代码的计算。从核在计算过程中，主核处于等待状态，其具体实现流程如图所示。



- “神威•太湖之光”通过加速线程库Athread改写程序使其在从核阵列上进行加速计算，该加速线程库是针对主从加速编程模型所设计的程序加速库，其目的是为了用户能够方便、快捷地对核组内的线程进行灵活的控制和调度，从而更好地发挥核组内多从核并发执行的加速性能。

- Athread库是对DMA源语的一种封装。DMA直接内存访问，是一种高速的数据传输操作，利用DMA可以在外部设备和内部存储器之间直接读写数据，在数据传输过程中不需要CPU参与。在数据传输开始之前，需由DMA控制器设定此次数据传输的起始地址、目标地址、传输数据量大小等参数，一旦控制器初始化完成，数据开始传送，DMA就可以脱离CPU独立完成数据传送，同时CPU可以在数据传输过程中执行别的任务。DMA的数据传输也是在数据总线、地址总线、控制总线上进行的，在没有DMA请求时CPU占有总线。DMA请求出现后，DMA控制器向CPU申请总线的使用权，希望CPU把所需要的总线让出来，由DMA控制器来负责接管。当数据传输结束后，DMA控制器将总线的控制权交还给CPU。因此，一个完整的DMA传输过程包括DMA请求、DMA响应、DMA传输和DMA结束四个步骤。

- 本节将继续以矩阵乘法为例，利用Athread线程库，实现单主核多从核上的矩阵乘法。对部分运行在主核上的代码中的Athread接口函数简单讲解如下：

  - ✓ athread_init()：无参数，完成加速线程库的初始化。
  - ✓ athread_spawn(func,void *arg)：在当前进程中添加新的线程组，执行的任务由函数func指定，函数func的参数由arg提供。
  - ✓ athread_join()：无参数，显式阻塞调用该线程组，直到该线程组终止。
  - ✓ athread_halt()：无参数，确定线程组所有从核无相关作业后，停滞从核组流水线，关闭从核组。

- 对后续从核程序示例代码中出现的部分数据类型、Athread接口函数讲解如下：

  - ✓ __thread_local：该属性表示它所修饰的数据对象存储在从核的局部存储器上。
  - ✓ athread_get_id()：获得本地单线程的逻辑线程标识号，参数-1默认本地从核。

- DMA只能由从核发起，即无论是主核主存空间传输数据到从核局存空间，还是从核局存空间传输数据到主核主存空间，DMA都是由从核主动发起，主核被动接收数据。Athread加速线程库中封装好了两个DMA函数，即athread_get()和athread_put()函数，分别对应了上述内容中主核主存传输数据到从核局存、从核局存传输数据到主核主存两种功能。

athread_get(dma_mode mode, void *src, void *dest, int len, void *reply, char mask,int stride,int bsize)：从核局存LDM接收主存MEM数据，进行主存MEM到从核LDM的数据get操作，将MEM的数据get到LDM指定位置

dma_mode mode：DMA传输命令模式
void *src：DMA传输主存源地址
void *dest：DMA传输本地局存目标地址
int len：DMA传输数据量，以字节为单位
void *reply：DMA传输回答字地址，必须为局存地址，地址4B对界
char mask：DMA传输广播有效向量，有效粒度为核组中一行，某位为1表示对应的行传输有效，作用于广播模式和广播行模式
int stride：主存跨步，以字节为单位
int bsize：行集合模式下，必须配置，用于指示在每个从核上的数据粒度大小。其它模式下，在DMA跨步传输 时有效，表示DMA传输的跨步向量块大小，以字节为单位

- DMA只能由从核发起，即无论是主核主存空间传输数据到从核局存空间，还是从核局存空间传输数据到主核主存空间，DMA都是由从核主动发起，主核被动接收数据。Athread加速线程库中封装好了两个DMA函数，即athread_get()和athread_put()函数，分别对应了上述内容中主核主存传输数据到从核局存、从核局存传输数据到主核主存两种功能。

int **athread_put**(dma_mode mode, void *src, void *dest, int len, void *reply,int stride,int bsize)：从核局存LDM往主存MEM发送数据，进行从核LDM到主存MEM的数据put操作，将 LDM的数据put到MEM指定的位置

dma_mode mode：DMA传输命令模式
void *src：DMA传输局存源地址
void *dest：DMA传输主存目的地址
int len：DMA传输数据量，以字节为单位
void *reply：DMA传输回答字地址，必须为局存地址，地址4B对界
int stride：主存跨步，以字节为单位
int bsize：行集合模式下，必须配置，用于指示在每个从核上的数据粒度大小。其它模式下，在DMA跨步传输时有效，表示DMA传输的跨步向量块大小，以字节为单位

- 利用单层次并行矩阵乘法的算法核心思路比较简单，就是利用多个从核读取主核内存中矩阵的部分行到从核的64KB局存空间，每个从核完成各自的计算任务后将数据写回主存空间。
- Athread版本主核程序

- 在代码中，主核上对A、B两个矩阵初始化后，利用第15行的athread_init()函数完成加速线程库的初始化，
- 使用第16行中的athread_spawn(Mul_Matrix,0)函数开启线程组，其中Mul_Matrix为每个从核线程上所执行的任务函数。
- 17、18行的athread_join()和athread_halt()函数用于等待线程组的任务完成后关闭线程组。

```
1//主核代码Master.c
2#include<stdlib.h>
3#include<mympi.h>
4#include<athread.h>
5#define J 100//列数，可自定义，注意矩阵规模大小不要大于局存大小即可
6#define I 100//行数，可自定义
7float matrix_a[I*J];//矩阵A
8float matrix_b[I*J];//矩阵B
9float matrix_c[I*J];//矩阵C，存储A×B的结果
10extern SLAVE_FUN(Mul_Matrix)();//引入运行在从核Slave.c上的外部函数
11int main(void){
12              Init_Matrix(matrix_a,I*J,10);//对各矩阵进行初始化，下同
13              Init_Matrix(matrix_b,I*J,10);
14              Init_Matrix(matrix_c,I*J,10);
15              athread_init();
16              athread_spawn(Mul_Matrix,0);//Mul_Matrix为运行在从核上的计算程序
17              athread_join();
18              athread_halt();
19              return 0;
20}
```

- Athread版本从核程序

- 在代码中，从核利用第19行和24行的athread_get()函数，从主核内存上读取进行计算所需的整个B矩阵和A矩阵中的某几行后。

```
//从核代码如下
1#include<stdio.h>
2#include<stdlib.h>
3#include"slave.h"
4#define J 100
5#define I 100
6#define ROW_NUM 20 //从核分得所需计算的行数，可自定义
7__thread_local float my_id;//线程号
8__thread_local float local_A[I];/从核局存上暂存从主存取来的a矩阵的某一行
9__thread_local float local_B[I*J];//用来接收从主存取来的矩阵b
10__thread_local volatile float local_C[I];//用来暂存计算后的某一行，传回主存矩阵c中
11extern float matrix_a[I*J],matrix_b[I*J],matrix_c[I*J];//引用外部变量的方式访问主存地址
12__thread_local volatile unsigned long get_reply,put_reply;//用于DMA传输时的标志位
13void Mul_Matrix(){
14    float temp; //用于计算，暂存某矩阵元素的结果
15    my_id=athread_get_id(-1); //获取从核逻辑id号
16    get_reply=0;//读取数据时的标志位
17    put_reply=0;//返回数据时的标志位
18    //从主存中读取完整矩阵B到从核局存的matr_b中
19    athread_get(PE_MODE,&matrix_b[0],&local_B[0],I*J*4,&get_reply,0,0,0);
20    while(get_reply!=1);
21    //依次从主存中读取矩阵A的某一行数据存入从核局存local_A中
22    for(int i=0;i<ROW_NUM;i++){
23        get_reply=0;
24        athread_get(PE_MODE,&matrix_a[my_id*ROW_NUM*J+*J*i],&local_A[0],I*4,
25                                    &get_reply,0,0,0);
```

12.4.2 程序编写

- 利用28到34行的代码段进行计算。
- 计算结果通过第36行的athread_put()函数发送到主核内存指定位置。

```
26                  while(get_reply!=1);//等待传输完成
27                  //开始计算C的某一行的数据
28                  for(int k=0;k<J;k++){
29                          int temp=0;
30                          for(int l=0;l<J;l++){
31                                  temp=temp+local_A[l]*local_B[k+l*J];
32                          }
33                          local_C[k]=temp;
34                  }
35                  //将计算完成的某一行的数据传输回主存中的指定位置。
36
        athread_put(PE_MODE,&local_C[0],&matrix_c[my_id*J*i],J*4,&put_reply,0,0);
37                  while(put_reply!= 1);//等待传输完成
38                  put_reply=0;
39}
```

- 需要注意的是，因从核的局存空间仅为64KB，在本例中若仍以1000×1000的矩阵规模进行计算，float型数据占用4B，那么每个从核需要存储的矩阵B的大小为1000×1000×4B，约为3906KB大小，这将远超从核局存64KB的容量。
- 此处为了演示方便，将矩阵维度缩小为100×100，这样就满足了64KB局存大小的限制。

- 对上述代码进行编译时，主核程序代码和从核程序代码需要先分别编译为.o文件后再进行混合链接成可执行文件。
- “神威•太湖之光”系统中提供有重新设计的编译器sw5cc，编译和作业提交命令如下。
- 其中，-n指定所要使用的主核数量；-cgsp指定需要使用的从核数目，该参数必须≤64；-q向指定的队列中提交作业；-o将作业的结果输出到指定文件。

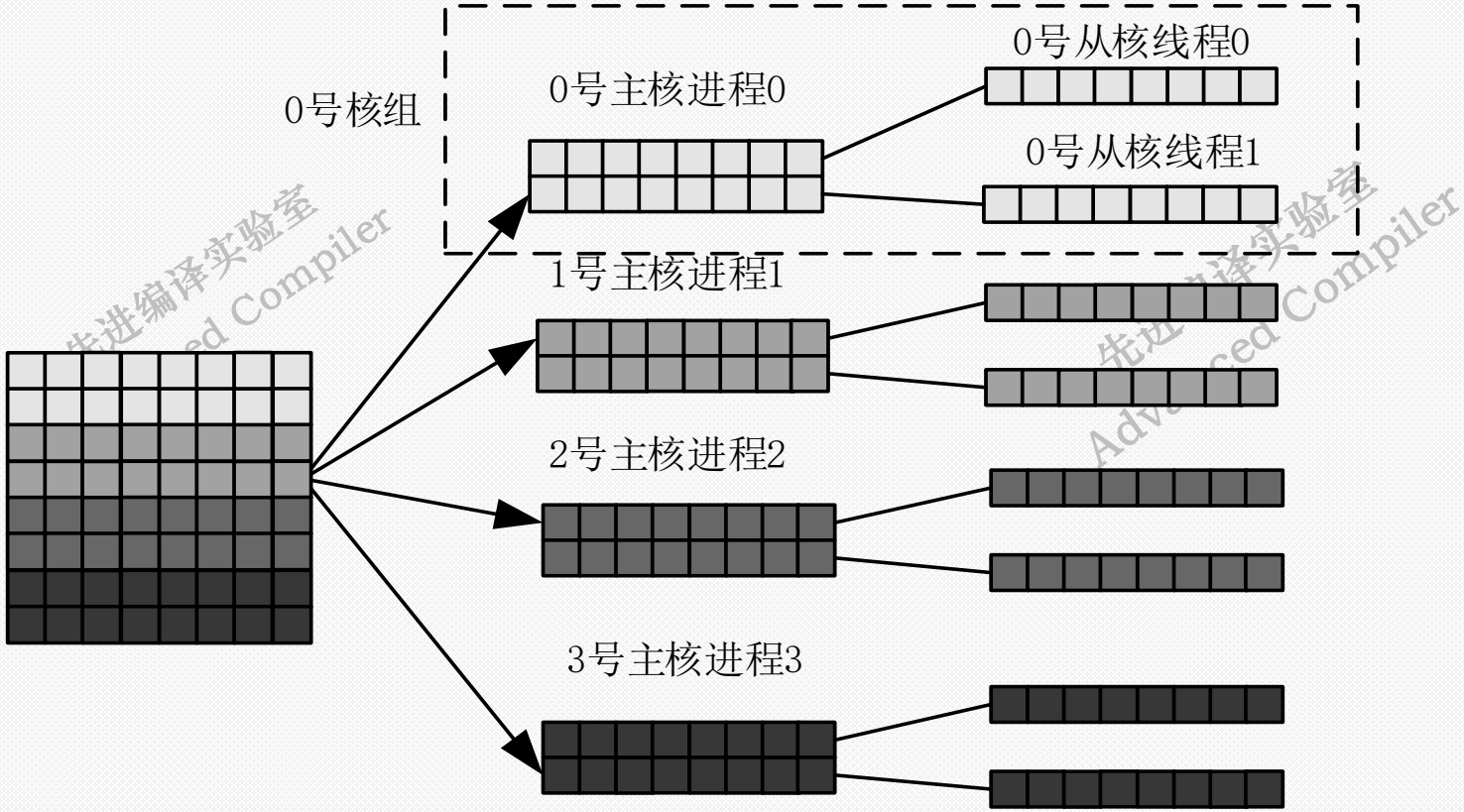
```
sw5cc -host -c Master.c
sw5cc -slave -c Slave.c
sw5cc -hybrid  Master.o  Slave.o -o [可执行文件]
bsub -n 1 -cgsp 64 -q q_sw_expr  -I  [可执行文件]
```

- 上述程序是利用从核实现了单核心阵列的并行，接下来将利用多个核心阵列一起完成计算任务，即利用MPI开启多进程，再利用Athread库调用多线程进行加速计算。

- 多层次并行的矩阵乘算法相较于单层次并行的矩阵乘算法并没有复杂太多。依旧以按行分解的矩阵乘法为例，其主要思想为：

  - 在MPI所开启的每个进程基础上，利用Athread加速线程库对各进程获得的几行数据进行细粒度的划分从而开启多个从核线程，
  - 每个线程利用athread_get()函数从它所属的进程内获取矩阵A的某一行和整个矩阵B后，
  - 在从核上进行计算任务，每个从核线程得出矩阵C某一行的计算结果，
  - 最后利用athread_put()函数将结果数据发送回其所属的主核进程。

- 具体的数据划分方法如图所示：



- MPI+Athread版本主核程序

```
1#include<stdio.h>
2#include<stdlib.h>
3#include"mpi.h"
4#include"mympi.h"
5#include<athread.h>
6#define I 100//定义矩阵的维数，可自定义
7#define J 100
8float * A,* B,* C;//声明矩阵A、B、C
9float *tempA,*tempC;//声明tempA和tempC，存A分发的几行数据和C的几行结果
10extern SLAVE_FUN(Mul_Matrix)();
11int main(int argc,char **argv){
12              int id;//进程号
13              int p;//进程数
14              int count;//每个进程要处理的矩阵A的行数
15              MPI_Init(&argc,&argv);
16              MPI_Comm_size(MPI_COMM_WORLD,&p);
17              MPI_Comm_rank(MPI_COMM_WORLD,&id);
18              B=(int *)malloc(I*J*sizeof(float));
19              //进程0负责对各矩阵进行初始化
20              if(id==0){
21                              A=(float *)malloc(I*J*sizeof(float));
22                              C=(float *)malloc(I*J*sizeof(float));
23                              Init_Matrix(A,I*J,10);
24                              Init_Matrix(B,I*J,10);
25                              Init_Matrix(C,I*J,1);
26              }
```

- MPI+Athread版本主核程序

- 在代码中，0号进程利用第31行的MPI_Scatter()函数使各进程获得矩阵A的某几行数据，
- 之后通过第32到36行中的Athread线程库函数开启从核线程，负责对本进程所获得的数据进行计算。
- 当从核计算任务Mul_Matrix完成后，37行的MPI_Gather()函数对散落在各进程的计算结果进行聚集收回。主核0号进程负责数据分发和收集。
- 这里利用了主从加速并行方法，主核只负责通信，计算全都放在从核上的思想。

```
27          MPI_Bcast(B,I*J,MPI_FLOAT,0,MPI_COMM_WORLD);//进程0广播B至各进程
28          count=I/p;//每个进程所分得的行数
29          tempA=(float *)malloc(count*J*sizeof(float));
30          tempC=(float *)malloc(count*J*sizeof(float));
31
            MPI_Scatter(A,count*J,MPI_FLOAT,tempA,count*J,MPI_FLOAT,0,MPI_COMM
_WORLD);
32          athread_init();//开始利用从核组对某进程分得的部分矩阵进行计算
33          athread_spawn(Mul_Matrix,0);
34          athread_join();
35          athread_halt();
36          //收集各进程的结果至进程0
37
            MPI_Gather(tempC,count*J,MPI_FLOAT,C,count*J,MPI_FLOAT,0,MPI_COMM_
WORLD);
38          free(B);
39          free(tempA);
40          free(tempC);
41          return 0;
42}
```

- MPI+Athread版本从核程序

- 该版本的从核程序与Athread单层次并行中的从核程序相比并没有太大的改动，代码相关注释和讲解可参考前面的，这里不再赘述。
- 这也符合了从核只负责开启线程用于计算，而主核负责开启进程用于进程间通信这一思路。

```
1#include<stdio.h>
2#include"slave.h"
3#include<stdlib.h>
4#define I 100
5#define J 100
6#define ROW_NUM 100 //每个从核所分得的所需计算的矩阵行数，可自定义
7extern float *A，*B，*C;
8extern float *tempA，*tempC;
9__thread_local int my_id;
10__thread_local volatile unsigned long get_reply,put_reply;
11__thread_local float local_A[I];
12__thread_local float local_B[I*J];
13__thread_local float local_C[I];
14void Mul_Matrix(){
15              int temp;
16              my_id=athread_get_id(-1);
17              get_reply=0;
18              athread_get(PE_MODE,&B[0],&local_B[0],4*I*J,&get_reply,0,0,0);
19              while(get_reply!=1);
20              for(int i=0;i<ROW_NUM;i++){
21                              get_reply=0;
22
                athread_get(PE_MODE,&tempA[my_id*ROW_NUM*J+J*i],&local_A[0],4*J,
23                              &get_reply,0,0,0);
24                              while(get_reply!=1);
```

- MPI+Athread版本从核程序

- 该版本的从核程序与Athread单层次并行中的从核程序相比并没有太大的改动，代码相关注释和讲解可参考前面的，这里不再赘述。
- 这也符合了从核只负责开启线程用于计算，而主核负责开启进程用于进程间通信这一思路。

```
25                      for(int k=0;k<J;k++){
26                              float temp=0.0;
27                              for(int l=0;l<J;l++){
28                                      temp=temp+local_A[l]*local_B[k+l*J];
29                              }
30                              local_C[k]=temp;
31                      }
32                      put_reply=0;
33
        athread_put(PE_MODE,&local_C[0],&tempC[my_id*J*i+J*i],I*4,&put_reply,0,0);
34                      while(put_reply!=1);
35              }
36}
```

- 在“神威•太湖之光”计算机系统上的编译命令如下：

```
mpicc -host -c Master.c
sw5cc -slave -c Slave.c
mpicc -hybrid master.o slave.o -o matrix
```

- 进行作业提交时，需指定部分参数以确定开启的进程数和线程数，以矩阵A为100×100为例，若开启四个进程，那么每个进程可以得到矩阵A的25行数据，若每个从核负责矩阵A的每一行，则每个进程下需要25个从核进行加速计算，作业提交的命令如下：

```
bsub -n 4 -cgsp 25 -q q_sw_expr -o record  ./matrix
```

- 仅增加计算核心的数量所带来的性能提升是有限的，甚至并不一定能带来加速效果，优化人员不能简单地靠堆砌硬件数量来加速计算，在众核编程中性能提升的关键在于如何提高访存的性能。

- 在编写代码层面可以从以下几个方面进行优化，如减少主从核间通信次数、实现计算与访存重叠、用从核间通信取代主从核间通讯等，众核访存常用的优化思路有以下几种：

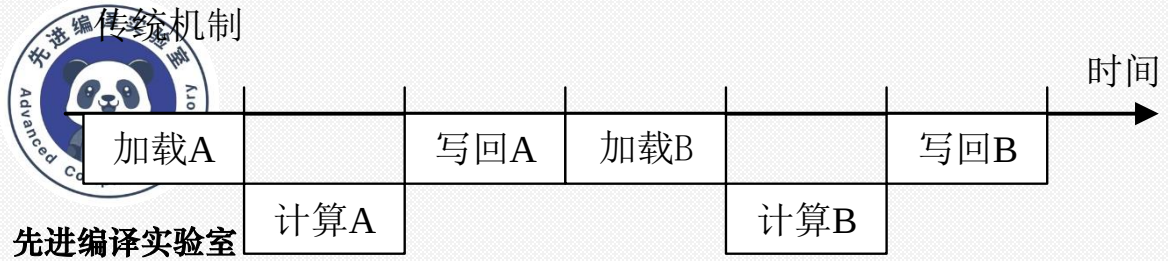
  ① 利用双缓冲机制实现计算与访存时间上的重叠。

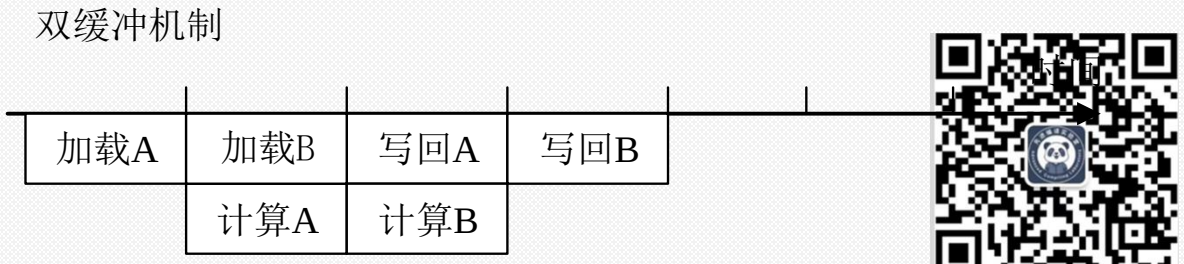
  ② 对数据布局进行优化，减少从核离散访问主存的次数。

  ③ 充分发挥DMA的带宽优势。

- 提高众核加速性能的关键是如何降低或隐藏从核通信开销。
- 所谓双缓冲机制，就是当需要多次的DMA读写操作时，在从核的局部存储空间上申请2倍于通信数据大小的存储空间，以便存放两份同样大小且互为对方缓冲的数据，即一方存储空间计算时另一方存储空间用于传输数据，依次交替进行。

- 双缓冲机制通过编程来控制和实现，具体过程如下：
  - 在从核从主核内存读取数据时，除了第一轮次读入数据的通信过程之外，当从核进行本轮次数据计算的同时，进行下一轮次读入数据的通信；
  - 在从核写回数据到主核内存时，除了最后一轮次写回数据的通信过程之外，从核进行本轮次数据计算的同时，进行上一轮次写回数据的通信。
  - 此时从核数据通信部分开销分为两部分，一部分是不可隐藏部分，另外则是可以与计算开销相互隐藏部分。其中不可隐藏部分开销为第一轮次读入与最后一轮次写回的数据通信开销之和。
- 其具体过程如图所示：





- 通过众核实践编程的经验，DMA双缓冲机制在众核编程时，有着固定的框架和模式，在采用双缓冲机制时，通常需要定义额外的双缓冲标识。
- 如代码中第21行所示，index表示当前轮次的变量，next表示下一轮次的变量，而last表示上一轮次的变量。由于采用双缓冲机制，用于存储通信数据的存储空间变为原来的两倍大小，即额外申请了一个数组，两个数组互为彼此的缓冲区。

```
//实现双缓冲后的从核程序代码
1#include<stdio.h>
2#include"slave.h"
3#include<stdlib.h>
4#include"mympi.h"
5#define I 1000
6#define J 100
7#define ROW_NUM 10
8extern float *A;
9extern float *B;
10extern float *C;
11extern float *tempA;
12extern float *tempC;
13__thread_local int my_id;
14__thread_local volatile unsigned long get_reply[2],put_reply;//注意get_reply[2]
15__thread_local float local_A[J],local_A_buffer[J];//开辟另一块空间，与local_A互为缓冲区
16__thread_local float local_B[J*J];
17__thread_local float local_C[J];
18__thread_local float *slave[2];//设置指针数组，用于转换计算数组与接收数据数组的身份
19void Mul_Matrix(){
20        float temp;
21        int index,next;
22        slave[0]=&local_A[0];
23        slave[1]=&local_A_buffer[0];
24        my_id=athread_get_id(-1);
25        get_reply[0]=0;
```

- 因此需要利用第18行的声明建立一个大小为2的指针数组slave[2]，存储两个数组的首地址，其中slave[0]与slave[1]指向的存储空间互为缓冲区。
- 注意此时的回答标志位用第14行所定义的get_reply[2]数组代替原来的reply变量。

```
1#include<stdio.h>
2#include"slave.h"
3#include<stdlib.h>
4#include"mympi.h"
5#define I 1000
6#define J 100
7#define ROW_NUM 10
8extern float *A;
9extern float *B;
10extern float *C;
11extern float *tempA;
12extern float *tempC;
13__thread_local int my_id;
14__thread_local volatile unsigned long get_reply[2],put_reply;//注意get_reply[2]
15__thread_local float local_A[J],local_A_buffer[J];//开辟另一块空间，与local_A互为缓冲区
16__thread_local float local_B[J*J];
17__thread_local float local_C[J];
18__thread_local float *slave[2];//设置指针数组，用于转换计算数组与接收数据数组的身份
19void Mul_Matrix(){
20        float temp;
21        int index,next;
22        slave[0]=&local_A[0];
23        slave[1]=&local_A_buffer[0];
24        my_id=athread_get_id(-1);
25        get_reply[0]=0;
```

- 在利用第28行的athread_get()函数获取到第一轮所需的数据后，开始利用for循环进行多轮次的数据传输。
- 在for循环的迭代过程中，使用第32、33行的求模运算不停地转换index和next的指向，其中index指向将要参与计算的数据，而next指向待传入数据的数组空间。

```
26              athread_get(PE_MODE,&B[0],&local_B[0],4*J*J,&get_reply[0],0,0,0);
27              //传输第一轮数据，需等待传输完成，get_reply修改置位。
28            athread_get(PE_MODE,&tempA[my_id*ROW_NUM*J],&local_A[0],4*J,&get_reply
29              [0],0,0,0);
30      while(get_reply[0]!=2);
31      for(int m=0;m<ROW_NUM;m++){
32            index=m%2;
33            next=(m+1)%2;
34                          get_reply[next]=0;
35            //在传输next轮数据时，无需等待get_reply，开始上一轮次的计算
36             athread_get(PE_MODE,&tempA[my_id*ROW_NUM*J+J*(m+1)],slave[next],4*J
37                          ,&get_reply[next],0,0,0);
38             for(int k=0;k<J;k++){
39                  temp=0.0;
40                  for(int l=0;l<J;l++){
41                        temp=temp+(*(slave[index]+l))*local_B[k+l*J];//注意解指针的用法。
42                  }
43                  local_C[k]=temp;
44            }
45            while(get_reply[next]!=1);
46                  put_reply=0;
47
athread_put(PE_MODE,&local_C[0],&tempC[my_id*ROW_NUM*J+J*m],J*4,
48                          &put_reply,0,0);
49             while(put_reply!=1);
50            }
51}
```

- 在第36行向next指向的数组空间中传入数据后，37行的代码不再是等待传输数据完成的while循环，而是直接开始了index指向的数组数据的计算过程。
- 通过这种机制，next与index指向的数组空间可以同时进行传输数据和计算数据的任务。

```
26                athread_get(PE_MODE,&B[0],&local_B[0],4*J*J,&get_reply[0],0,0,0);
27                //传输第一轮数据，需等待传输完成，get_reply修改置位。
28              athread_get(PE_MODE,&tempA[my_id*ROW_NUM*J],&local_A[0],4*J,&get_reply
29                [0],0,0,0);
30        while(get_reply[0]!=2);
31        for(int m=0;m<ROW_NUM;m++){
32              index=m%2;
33              next=(m+1)%2;
34                              get_reply[next]=0;
35              //在传输next轮数据时，无需等待get_reply，开始上一轮次的计算
36              athread_get(PE_MODE,&tempA[my_id*ROW_NUM*J+J*(m+1)],slave[next],4*J
37                              ,&get_reply[next],0,0,0);
38              for(int k=0;k<J;k++){
39                    temp=0.0;
40                    for(int l=0;l<J;l++){
41                          temp=temp+(*(slave[index]+l))*local_B[k+l*J];//注意解指针的用法。
42                    }
43                    local_C[k]=temp;
44              }
45              while(get_reply[next]!=1);
46                    put_reply=0;
47
athread_put(PE_MODE,&local_C[0],&tempC[my_id*ROW_NUM*J+J*m],J*4,
48                              &put_reply,0,0);
49              while(put_reply!=1);
50              }
51}
```

- 双缓冲机制并未减少通信次数。因此，需要从其它角度来优化此程序。
- 在一个核组内，每个计算核心专属的局存空间不大，但计算核心访问LDM局存延迟较小，从核访问主存的延迟约为700-1000拍，而从核访问其专属LDM局存的延迟仅为4拍，因此可考虑对数据布局进行规划，利用一次访存来实现几次离散访存所带来的效果。
- 即在针对存在不可避免的大量离散访存的程序进行众核算法设计时，可以利用众核的特点来避免离散访存以获取性能的提升。

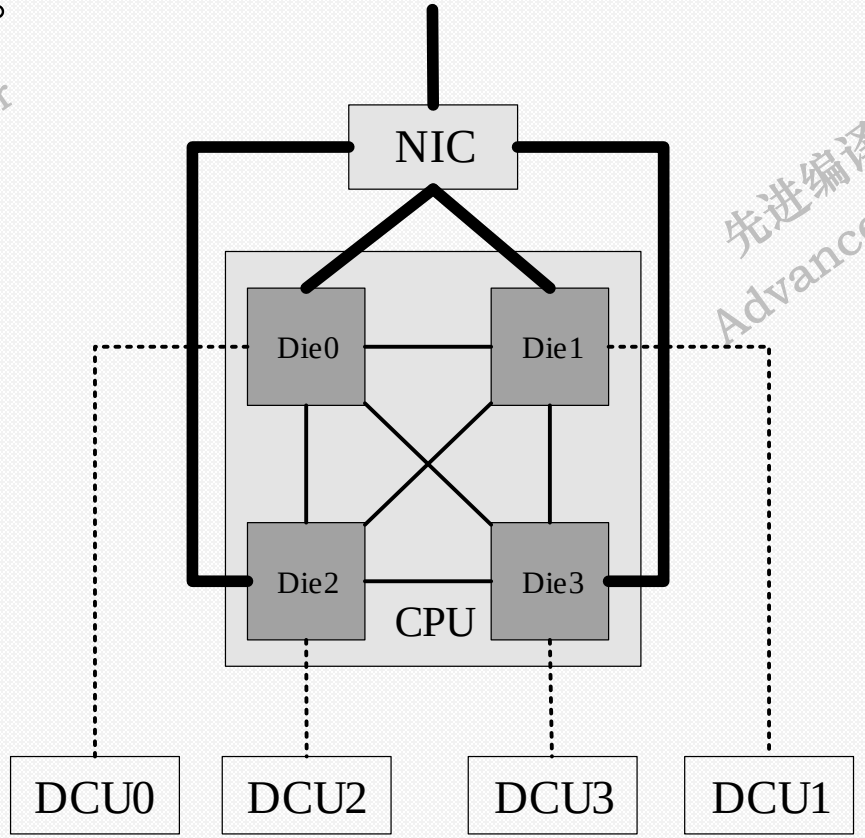
- 针对存在大量离散访存的众核并行设计过程中，应采用如下思路调整：
  - 首先在主核上将原来离散的数组调整成方便通信的读入和写回的存储顺序，
  - 然后计算核心进行通信读入数据、计算和通信写回数据，
  - 最后主核将写回的数据再次调整回原来的存储顺序。
- 尽管相较于原来的算法，增加了前后两个数组存储顺序调整的过程，但由于上述过程都是在主核上来完成的，两个存储顺序调整所导致的计算核心开销增加的不大。但计算方面，由于计算核心对专属局部存储空间访问延迟变小，使得计算核心计算开销大大减小。

- 在对DMA传输数据次数、计算访存重叠进行优化后，思考更进一步的优化方法，即如何才能发挥DMA的最大性能，
- DMA的主要功能是在从核LDM和主存之间进行数据传输，无论是从核从主存读入数据还是从核写回数据到主存，DMA只能由从核发起。
- 经测量，在单次DMA数据传输时，当主存地址为128B对界，且传输的数据粒度大小为128B的倍数时，DMA达到峰值性能。

12.5 “嵩山”超算同构加异构平台

- 之前的小节中以矩阵乘为例介绍了多个平台上的同构多层次并行程序以及异构多层次并行程序的编写以及优化方法，其中同构架构主要使用CPU设备进行计算，异构架构主要使用协处理器设备进行计算。
- 本节将在“嵩山”超级计算机上，继续以矩阵乘法为例，介绍如何同时使用CPU以及加速设备编写多层次并行程序，程序编写完成之后使用一种通用、结合内存模式和硬件特征的任务划分方式对多层次并行程序进行优化以及性能分析。

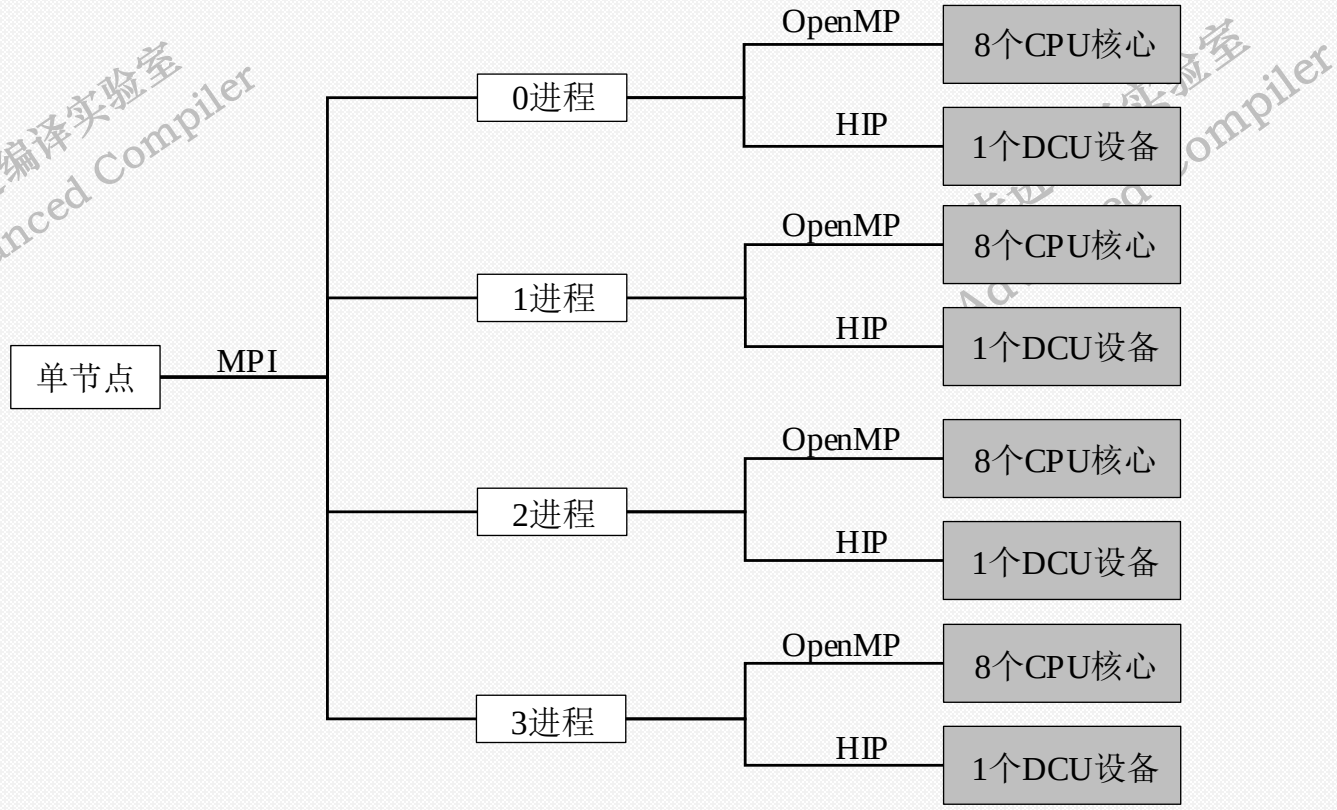
- “嵩山”超级计算机的每个刀片上有两个节点，其中每个节点由一个Hygon C86 7185 32-core CPU和四个DCU组成，单节点硬件架构如图所示。其中NIC指网络适配器，每个节点都通过NIC与其它节点进行通信。Die指CPU物理上的分区，每个分区负责控制一块DCU加速卡。本节将使用一个节点作为同构加异构程序的运行环境。



- Hygon C86 7185 CPU 的具体信息可以看出，其在物理上被分成了4个区，每个区包含连续的8个核心。

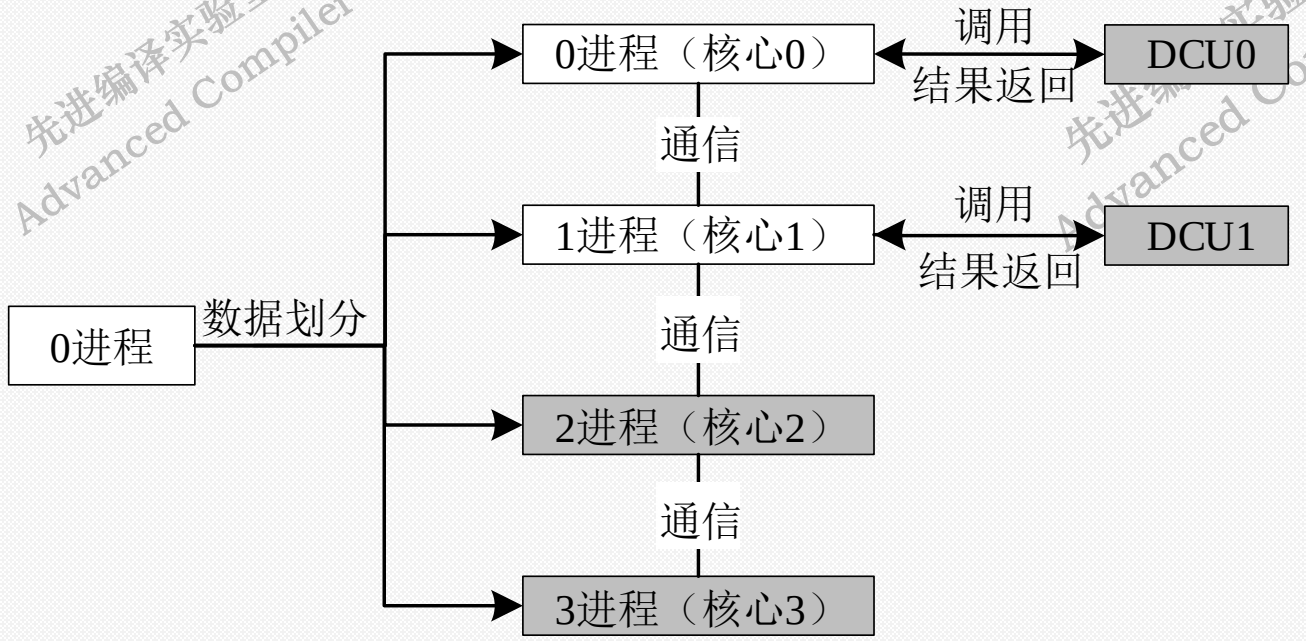
```
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 32677 MB
node 0 free: 408 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 32767 MB
node 1 free: 19686 MB
node 2 cpus: 16 17 18 19 20 21 22 23
node 2 size: 32767 MB
node 2 free: 25652 MB
node 3 cpus: 24 25 26 27 28 29 30 31
node 3 size: 32767 MB
node 3 free: 26765 MB
node distances:
node   0   1   2   3
  0:  10  16  16  16
  1:  16  10  16  16
  2:  16  16  10  16
  3:  16  16  16  10
```

- 如图所示在单个节点上同时使用CPU核心和DCU加速设备进行协同计算的示意图。首先是根据要使用的DCU数量开启同样数量的MPI进程。然后再在每个进程中分别调用CPU核心和DCU设备上进行并行计算，在调用CPU上的8个核心时使用OpenMP进行实现，调用DCU设备时使用HIP编程。



- 本节以按行分解矩阵乘算法为例介绍同构加异构多层次并行程序编写的基本方法。
- 在进行程序编写之前，需要先分析MPI进程、CPU核心以及DCU设备在同构加异构多层次并行中的组织方式。
- 其中每个MPI进程都是一个相对独立的程序，因此需要运行在具有完整逻辑控制功能的CPU核心之上。DCU设备是协处理器，并不具备完整的逻辑控制功能，因此使用DCU设备时必须使用一个MPI进程去调用。
- 在本节的程序编写中，每创建一个MPI进程都为其分配一个独立的CPU核心，每使用一个DCU设备都为其分配一个独立的MPI进程来进行数据交互以及逻辑控制。

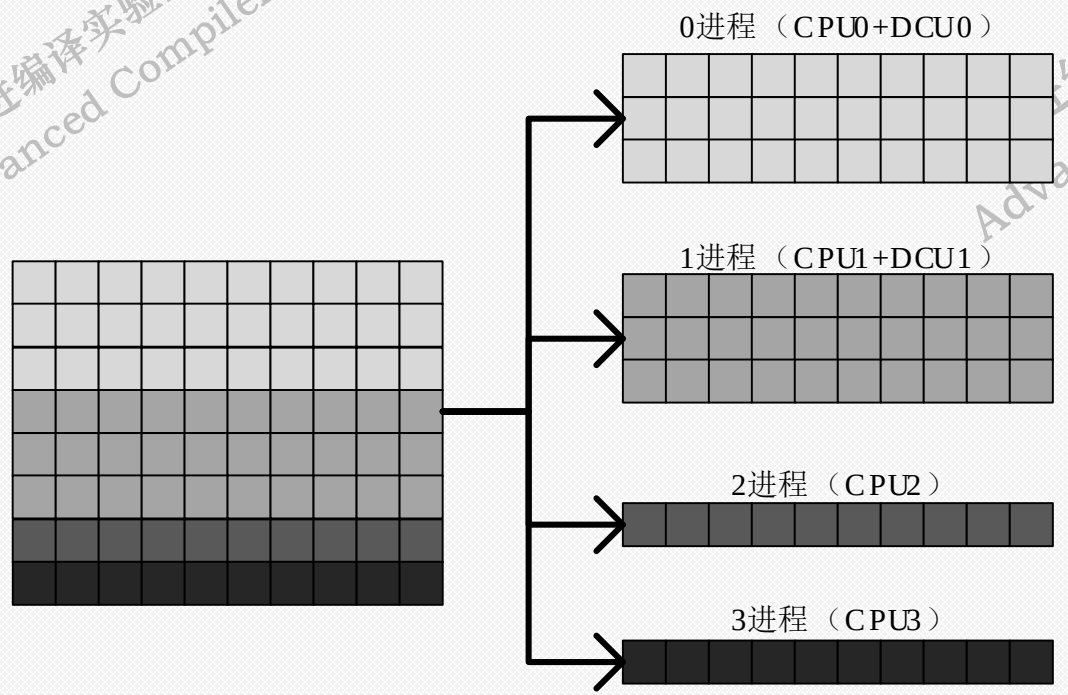
- 下图是在开启4进程时使用2个CPU核心与2个DCU设备进行协同计算时程序的组织方式示例。
- 可以看出，4个MPI进程分别运行在4个独立的CPU核心上，其中2个进程负责调用DCU设备进行计算，剩余2个进程只在CPU核心上进行计算。



- 在进行计算时，由于同时使用了CPU和DCU设备进行计算，那么考虑如何进行任务划分以充分发挥出两者的计算速度成为了首要考虑的问题。
- 1个DCU设备的计算速度是远大于1个CPU核心的，当每个CPU核心或DCU设备接收到的任务量与其计算速度不匹配时，将会出现DCU设备在很短的时间完成计算任务后，花费较长时间等待CPU进行计算，或者出现相反的情况。
- 因此，当同时使用CPU与DCU共同计算时，必须按照其计算速度的比值来划分任务。

- 下图是4进程下使用2个DCU与2个CPU核心进行矩阵相乘时各个进程对一个8*10的A矩阵进行数据划分的示例。

- 为了说明数据划分的规则，假定在进行矩阵乘法运算时单个DCU设备的处理速度是单个CPU核心的3倍，则每块DCU分配到的A矩阵的行数是单个CPU核心行数的3倍左右。



12.6 小结

Hygon C86同构多核平台

基于节点同构的海光Hygon C86多核平台上，以矩阵乘法为例首先介绍该平台架构特点，然后分别基于OpenMP和SIMD实现线程级和数据级的多层并行，最后给出了类似平台上的通用优化经验。

Intel KNL同构众核平台

基于Intel KNL众核架构，结合该平台的内存和集群特性，基于MPI+OpenMP+SIMD程序利用AVX-512指令集、内存模式和集群模式对其进行优化，并给出此类同构架构上进行优化的通用经验。

多层次并行程序优化

“嵩山”超算同构加异构平台

介绍了“嵩山”超级计算机的节点架构图和处理器的基本信息；从宏观层面了解了如何同事使用CPU核心和DCU加速设备进行协同计算；最后给出了开启4进程时使用2个CPU核心和2个DCU设备进行协同计算的程序组织方式、如何充分发挥两者的计算速度以及数据划分的方法。

申威26010异构众核平台

介绍了国产异构众核申威26010处理器的架构、主从加速的异构并行方式以及专用加速线程库；分别使用Athread和MPI+Athread实现了单层次的并行矩阵乘法和多层次的并行矩阵乘法；最后通过计算和访存的重叠、优化数据布局和发挥DMA的带宽优势三个方面进行优化。

Hygon DCU异构众核平台

介绍了什么是异构架构、DCU的硬件架构、CUDA编程和HIP编程模型的区别和联系。实现了基础执行版本的HIP矩阵乘和MPI+HIP版本的矩阵乘，最后通过流和事件实现了一个或多个进程中的DCU矩阵乘。