9 MIC 性能优化

本章将介绍 MIC 平台应用程序的性能优化方法，包括并行度、内存管理、数据传输、存储器访问、向量化、负载均衡和扩展性等优化手段，并且通过大量的示例代码展示每种优化手段的使用方法。

本章目标

通过本章的学习，你可以：

- 了解 MIC 性能优化策略。
- 了解如何通过并行度、内存管理、向量化、cache 优化等手段优化 MIC 程序的性能。
- 了解如何使用 nocopy、异步、scif 技术优化 CPU 与 MIC 的通信。
- 了解如何对 MIC 的负载均衡和线程扩展性进行优化。

9.1 MIC 性能优化策略

MIC 程序性能优化包含 CPU 与 MIC 端的通信优化以及 MIC 内核计算、访存优化。MIC 程序的优化是一个循环反复的过程，可以用性能优化循环图表示，MIC 性能优化的基本循环过程如图 9-1 所示，包括以下几个步骤：

（1）获取程序的性能数据作为基准。

（2）分析性能数据，通过 VTune 等工具找出性能瓶颈。

（3）根据瓶颈进行分析，并找到相应的优化手段。

（4）实施优化手段，对代码做相应的修改。

（5）测试结果是否正确，如果正确并且性能获得提升就完成一次优化的循环。

（6）进入下次优化循环。

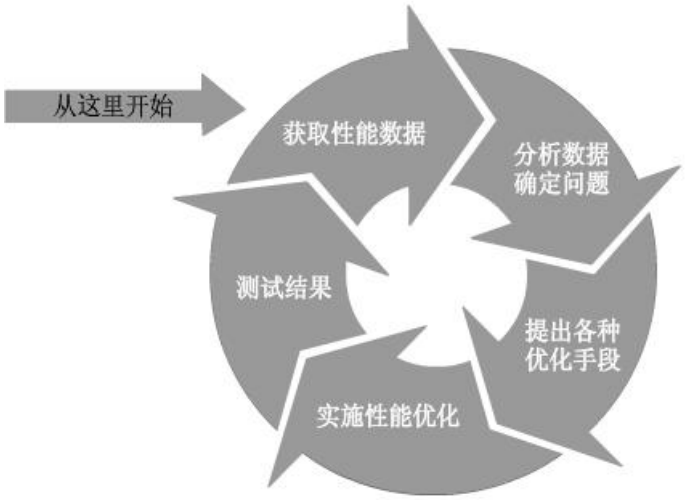

图 9-1　性能优化循环

MIC 性能优化循环中常见问题：

（1）选择合适的测试用例，即 workload。合适的 workload 需要满足几个要求：可测试性、可重现性、稳定性和代表性。合适的测试用例是可以“用例测试”的，即：不能测试时间过短或过长；测试是可以重复进行的，可以复现的；每次复现的性能结果是稳定的，不能偏移太大，而且测试用例具有广泛的代表性，不是和大部分用例大相径庭的。例如，如果所选的测试用例执行的是算法的第 1 条路径，而其他大部分用例执行的是算法的第 2 条路径，则该测试用例的选择是不合适的。

（2）如何得到性能指标。性能指标有很多种，最简单最常用的是计算程序执行时间。在 MIC 程序中，我们可以通过时间函数获取内核计算执行时间，以及数据传递时间，也可以借助 VTune 等工具获取 MIC 内核中每个线程时间等。

（3）分析问题的时候主要考虑热点和关键路径。非主要矛盾可以忽略，把精力放在关键问题的优化上。考虑基准性能指标是什么，最优的性能指标能到多少，可能的潜力有多少。回答这些问题需要考虑制约 MIC 性能的关键因素，如 GFLOPS 值、CPI、程序的并行度、数据的局部性、带宽的压力、向量化程度、IO 是否为主要的瓶颈等。这些关键数据可以借助 VTune 测试得到。

（4）在实施优化过程中，对代码的修改还需兼顾代码的质量。要保证代码的可移植性、可读性、可维护性、可靠性。采用的方法可以是修改编译选项、各种数学库，手动修改热点代码等。

（5）测试用例要完备，要尽可能覆盖所有的情况，并且只有在程序保证正确的前提下测得的 MIC 性能指标才有效。

（6）最后，决定是否进行新一轮的循环优化。考虑的因素有：现有性能是否已经逼近极

限？是否实现了 MIC 硬件的利用率最大化？

MIC 程序的性能优化主要包括系统级和内核级：系统级优化包括节点之间、CPU 与 MIC 之间的负载均衡优化；MIC 内存空间优化；计算与 IO 并行优化；IO 与 IO 并行优化；数据传递优化；网络性能优化；硬盘性能优化等。内核级优化包括并行度优化；负载均衡优化；进程/线程的同步优化；线程扩展性优化；向量化优化；cache 优化；数据对齐优化；库函数的选择等。下面小节我们将详细介绍这些优化手段。

9.2 MIC 优化方法

MIC 作为众核协处理器，具有众多的核，只有程序高度并行化，才能充分发挥出 MIC 的性能。并行化使得程序可以在 MIC 的各个核上同时执行，能够显著提高程序的性能。并行程序常用的两种并行方法为数据并行和任务并行。

数据并行以数据分割为依据，扩展性良好，随着数据规模的增大，线程数也可以增多，并且每个线程的任务量不减少。可以说，数据并行是天然的并行方式。需要注意的是分割不均匀会导致严重的负载不均衡。MIC 是共享内存并行系统众核处理器，因此，数据并行是 MIC 上并行优化的很好的选择。

任务并行是指以多任务为基础，多个任务并行操作，不同任务处理的方式可能不一样，处理的数据本身没有太多的依赖关系。因此，在 CPU 与 MIC 协同计算时，可以采用任务并行的方式，让 CPU 和 MIC 执行不同的任务，以充分发挥各自的优势。

并行程序执行时，由于存在多个任务/线程之间的相互通信和影响使得应用程序行为变得复杂。因此，需要优化 MIC 上的各个方面才能达到性能的最优，MIC 上的并行程序的优化主要包括并行度、内存空间、数据通信/传递、Cache 访问、向量化、负载均衡、线程扩展性等方面的优化。下面对这些优化方法进行展开介绍。

9.2.1 并行度优化

在计算机体系结构中，并行度是指指令并行执行的最大条数。在设计并行程序时，我们可以简单地把并行度认为是在多核/众核处理器上能同时执行的线程数/进程数。对于同一个程序，并行度设计方法的不同将会严重影响到程序的性能。MIC 上的并行度优化主要涉及并行线程/进程的数目、并行层级、并行粒度等方面。

9.2.1.1 并行度

MIC 卡包含众多的物理核，同时每个核上可以开启 4 个线程，因此，程序员只有设计足够多的线程/进程才可以把所有的核利用起来。例如一块 60 个核的 MIC 卡上，我们最多可以开启 240 个线程，最佳线程数一般是每个核设置 3 个或 4 个线程，图 9-2 展示的是某一实际高

性能应用程序在 60 个核的 MIC 卡上设置不同线程数的性能扩展性结果图，从该图可以看出，只有让 MIC 卡上的所有核都充分利用起来才能发挥 MIC 的最大性能。当然，也不是在 MIC 卡上设置的线程数越多越好，线程数太多的话，线程开销比较大，我们只需要让设置的线程数可以保证程序并发度和 MIC 核的高利用率即可。

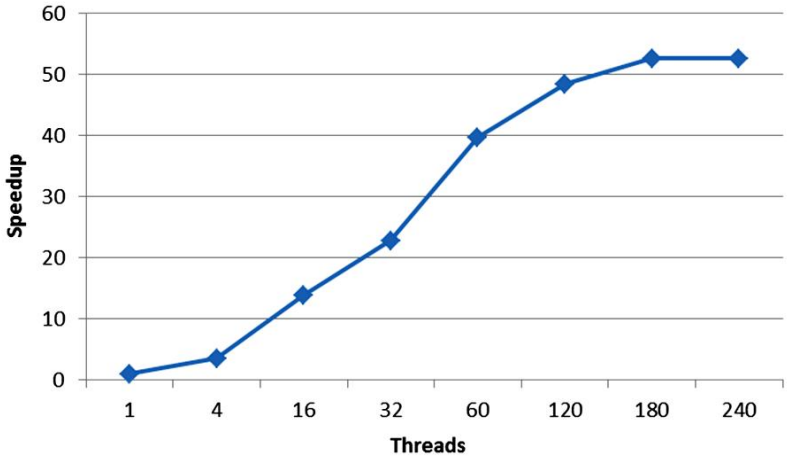

图 9-2　某一高性能应用程序在 MIC 上的性能扩展性

9.2.1.2　并行粒度

并行程序是否选择了合适的层级实现并行，是性能优化中需要关心的重要问题。根据并行程序尽可能使用粗粒度的并行原则，尽可能在最上层并行化代码。在外层上并行除了带来易编程的好处之外，还可以带来好的性能：增加粒度，减少线程调度和销毁的次数，也就是减少线程本身的开销所占的比例，尤其对于 MIC 平台要开启上百个线程，减少线程的开启对性能影响更为重要；同时，隐藏了底层的线程交互，减少了不必要的同步带来的损耗。

下面通过简单的例子说明并行层级，例如程序中有两层循环，并且每层循环都没有数据依赖性，即两层循环都可以并行，根据并行程序尽可能使用粗粒度的并行原则我们可以采用在 i 层循环并行的方式。

```
1    #pragma omp parallel for num_threads(THREAD_NUM)
2    for (i=0; i<M; i++)
3    {
4        for (j=0; j<N; j++)
5        {
6            …
7        }
8    }
```

当然，并不是所有的应用程序都是在外层循环并行效果最佳，外层循环的并行可能会导致线程之间访问的数据跨度比较大，可能会引起 Cache miss，这种情况下采取内层循环的并行效果更佳，同时为了减少线程的开销，我们可以在外层 for 之前开启多线程，在内层 for 进行任务分发，如上面的代码采用下面的并行方式。

```
#pragma omp parallel num_threads(THREAD_NUM)
for (i=0; i<M; i++)
{
#pragma omp for
        for (j=0; j<N; j++)
        {
                …
        }
}
```

在实际的应用程序中也可能出现某一层循环无法达到 MIC 的并行度要求，针对这种情况，我们可以采取多层循环合并的方式。例如上面的代码中 M=20，N=30，无论我们并行哪层 for 都无法达到 MIC 的并行度要求，我们可以合并两层 for，合并之后的循环次数为 600 次，显然可以满足 MIC 平台上的要求。当然，我们也可以采用嵌套并行的方式满足 MIC 的并行度要求。

合并循环：

```
#pragma omp parallel for num_threads(THREAD_NUM)
for (k=0; k<M*N; k++)
{
        i = k/M;
        j = k%M;
        …
}
```

嵌套并行：

```
omp_set_nested(true); //允许嵌套并行
#pragma omp parallel for num_threads(THREAD_NUM1)
for (i=0; i<M; i++)
{
#pragma omp parallel for num_threads(THREAD_NUM2)
        for (j=0; j<N; j++)
        {
                …
        }
}
```

9.2.2 内存管理优化

MIC 作为协处理器，卡上的内存空间相比主机端十分有限，且不可扩展。因此，如何充分利用有限的内存空间，用尽量少的内存，完成尽量多的计算，成为 MIC 优化的重点。另外，由于卡上内存空间有限，导致一些程序在 MIC 移植可行性上，也会面临很大的挑战。

高度并行，也会对内存空间带来新的问题。以前的串行程序，每次迭代所用到的临时内存空间都是可以重复利用的。发展为并行程序以后，需要为每个线程分配独立的临时空间，但由于计算单元核数不多，因此并发线程不多，内存问题并不突出。可是在 MIC 上，由于并发线程达到了 200 个甚至更多，导致所需内存空间急剧增加。假设一个线程私有变量占用空间为 1MB，在 CPU 端，总内存占用可能仅有 4MB 或 8MB 而已，但在 MIC 上，则会根据线程数瞬间扩展到 200MB 以上！因此对内存空间占用的优化成了一个不得不面对的挑战。

另外，MIC 每个核心的时钟频率比 CPU 要低，因此在创建内存空间的效率上，设备端也与主机端有一定的差距。所以对 MIC 卡上的内存空间优化，不仅仅是对空间上的优化，同时还要考虑时间上的优化。

正如上文所述，MIC 内存空间的优化分为两个方面：内存使用容量和申请次数。内存使用容量是指程序对卡上内存空间的使用量，即占用内存的大小，通常关注的是最大情况下的大小。申请次数是指程序运行过程中，静态或动态开辟卡上内存空间的次数。优化关注使用量时，有时会与其他注重性能的优化方法产生冲突。但是，优化空间占用通常事关移植的可行性，因此通常会重视对空间占用的优化并因此牺牲一定的性能。

9.2.2.1 内存占用

卡上内存占用量出现瓶颈，通常有两种情况。其一是，任务本身需要占用大量内存，由于 MIC 端内存容量较小，所以移植中出现困难。其二是，每线程占用临时空间较大，当移植到 MIC 上时，由于线程数较多，导致内存不足。减少内存占用的方法包括：

1. 任务分块

解决内存占用最根本的办法，就是任务分块。如果能把大任务划分为小任务，把一次需要占用的内存空间降下来，则可以解决几乎全部的内存空间不足的问题。但是面对不同的原因，通常会采用不同的任务划分方式，下面分情况进行讨论。

第一种情况，任务本身需要占用大量内存，则通常需要将任务本身分块，一次只处理任务的一个子集。

第二种情况，每个线程需要较多的临时空间，通常可以选择降低并发度，减少线程数，自然可以降低总体内存占用。由于 MIC 卡上可以并发的线程较多，降低一定的并发度，仍然有可能保持性能在可以接受的范围内。

如果任务整体完全无法分割，或 MIC 卡上内存无法满足即使是最小分块的需求，那么

只能认为该程序（通常认为是该算例）无法移植到 MIC 上运行了。即使是主机端，内存也是有限的，如果任务实在太大，在 CPU 端也会无法运行。严格说起来，这种情况也并非完全不能移植，只是需要将任务分成极小的部分，虽然每个部分都仅仅是一个片段，但可以不断地传输－计算－传输－计算……重复这一过程，将整个任务完成。但是，在绝大多数情况下，这种方式所产生的性能都不足以抵消急剧膨胀的开发成本。列此方法，仅供一些极端情况下的参考。

我们显然可以发现，任务分块无论如何也会增加传输次数，并且很有可能减少并发度，这会降低程序的性能。但是正如前文所述，如果不进行分块，程序很有可能无法在 MIC 上运行，因此牺牲一定性能，使程序具有可行性，也是可以接受的。至于其中的“度”，则需要程序员根据具体情况进行把握。

2. 临时空间复用

有些程序中用到的一些临时空间，是可以合并或者节省的。例如：程序前半部分用到数组 a，大小为 100MB，后半部分用到数组 b，大小为 150MB，使用数组 b 时，数组 a 已不再使用。那么可以只开辟 150MB 大小的空间，前面用作数组 a，后面用作数组 b。虽然减少这些临时空间可能会对代码的可读性造成一些不利影响，但是如果这部分对性能影响比较大，则需要权衡可维护性和性能之间的比重，尽量寻找其中的平衡点。

还有一种情况，如代码所示：

```
1    for(i=0;i<N;++i)
2    {
3        c[i]=a[i]+N;
4        d[i]=b[i]+N;
5    }
```

a 和 b 为中间变量，c 和 d 为所求数组。由于 a、c 和 b、d 之间没有依赖关系，因此完全可以将循环分为两个，一个计算 c，一个计算 d。在计算 c 结束，且计算 d 尚未开始时，即可将数组 a 的空间释放，以节省内存占用。

3. 改变算法

从程序算法来看，通常粗粒度并行，即并行层次较高的循环，会占用相对较多的内存资源，而细粒度并行占用的内存资源较少。例如：一个计算 N 个 A*B 的矩阵乘应用（互相无依赖）。如果并行计算 N 个矩阵乘法，则内存占用会比较大。但是如果每次仅并行 A*B，N 个乘法间是串行执行的，这样内存占用就会小很多。与任务分块法不同的是，这里需要改变并行的算法（以前一个线程的功能是串行计算一个矩阵乘法，现在一个线程的功能是计算矩阵的一个元素）和粒度，而任务分块法通常只需要改变任务大小和传输的方式。

从细节代码的算法来看，曾经有一道经典的面试题——“如何在不使用临时变量的情况下，交换两个整型变量的值”。虽然这种方法在 MIC 上未必适用，但也给我们提供了一个思路，即可以试着找寻别的算法，以节省内存空间，完成相同的功能。

除了上述几种方法以外，在编程史的“上古时期”，那时硬件资源比较匮乏，程序员们会竭尽所能，减少哪怕 1 个字节的占用。之后随着硬件的发展，这些技巧被逐渐遗忘，或是不再使用。但是，现在协处理器的出现，面临着与当年同样的问题。因此，在一些比较古老的书籍和文献中，也许会有仍然适用于现在的技巧和方法，读者也不妨一试。

9.2.2.2 申请次数

对内存申请次数的优化不一定会减少内存占用，但是如果注意运用一些技巧，则可以使性能有所提高。

对于内存空间方面的性能优化来说，最关键的一点是：把开辟空间的操作放到循环外面。无论是简单的循环，还是循环中调用的子函数，都有可能根据需要开辟自己私有的空间，尤其是使用 malloc 等函数开辟的大块内存空间。由于 MIC 的时钟频率等原因，开辟空间的操作比主机端要慢。因此，如果在循环内开辟较大的内存空间的话，每次开辟时都会耽误一些时间，而循环次数一多，时间累积起来就会很可观了。虽然这是我们平时不注意的小时间，但是在一些情况下也会造成很大的性能损失。例如我们在某软件移植优化的过程中，就遇到了这种情况。该代码片段非常简单：一个循环，循环体内容为调用计算函数，计算函数内部首先是开辟内存空间，然后计算。我们的任务就是把这个循环移植到 MIC 上。移植本身很简单，但是测试时发现计算函数仅仅占总体运行时间的一半！最开始怀疑是 offload 传输的问题，但自己写测试用例测试时并没有重现问题。后来测试才偶然发现是计算函数内部开辟空间的问题。由于每次开辟的时间比较长，又根据循环开辟了多次，导致运行时间急剧增加。将内存空间全部移到外面一次开辟，程序运行时间即大幅缩短。在转移内存开辟的过程中，我们使用在主机端声明指针，在 offload 时使用 nocopy 开辟空间的方式，进一步节省了不必要的传输时间。在开辟空间的大小上，我们采用线程数乘以单次循环需要的内存大小的容量，运行时各线程根据自己的线程号查找自己“私有”的内存地址。

当需要多次调用 offload 函数，进行一系列操作时，如果不同 offload 函数中有公用的数组，也可以使用 nocopy 等方式一次申请，多次使用。一方面减少了数据传输时间，另一方面也避免了多次申请空间的开销。这种优化方式详见下一节“数据传输优化”。

9.2.3 数据传输优化

数据传输是一个通信的过程。数据传输对并行计算的性能有很大的影响。对于单机节点来说，频繁进行的发送/接收操作将大大降低并行程序的执行性能。对于集群而言，巨大的通信开销对并行效率的影响是致命的。处理器之间的通信是并行程序执行时重要的时间开销来源。作为并行计算额外开销的主要组成部分，降低通信成本可以有效地缩短并行程序的执行时间。所以我们在做并行编程的时候要尽可能地降低 I/O 传输操作的开销。

一般对数据传输会用到的优化方法有：nocopy，offload 异步，SCIF 模型，4K 倍数等。下

面我们将逐一介绍数据传输优化方法。

9.2.3.1 nocopy

CPU 与 MIC 之间通过 PCI-E 通信，PCI-E 的速度较慢，因此，我们需要尽量减少 CPU 与 MIC 之间的数据通信，通过 nocopy 技术可以有效地减少 CPU 与 MIC 之间的通信次数。

offload 中的 in、out 语句默认为每次 offload 开始时申请空间，结束时释放空间，然而在很多应用程序中，数据或空间是可以重复利用的，并不需要每次 offload 时都申请空间、释放空间。nocopy 主要应用在多次调用 offload 语句时，可以减少 CPU 与 MIC 的通信次数，即主要应用在有迭代调用 MIC 的程序中。

没有采用 nocopy 的 MIC 程序可以表示成下面的伪代码：

```
1   …
2   p_c = …; //p_c 在每次迭代中值不改变
3   for(i=0; i<steps; i++)//迭代多次
4   {
5         p_in = …; //每次迭代计算时，p_in 的值变化
6   #pragma offload target(mic) \
7         in(p_in:length(…)) \
8         in(p_c: length(…)) \
9         out(p_out: length(…))
10        {
11              kernel(p_in, p_c, p_out);
12        }
13  }
14  … = p_out; //CPU 端在所有迭代完成之后才用到 p_out 的值
```

上面的代码中，每次迭代 p_in、p_c、p_out 都会在 MIC 端申请空间、进行 CPU 与 MIC 的数据传递、最后释放空间，计算过程如图 9-3 所示。而实际运行中，p_c 的值只需要传递给 MIC 端一次即可，p_out 的值只在迭代完成之后回传给 CPU 端即可，显然上面的代码有很多不必要的 CPU 与 MIC 通信。针对这种情况，我们可以采用 nocopy 的模式减少 CPU 与 MIC 的通信次数。

nocopy 技术往往与 alloc_if()、free_if()联用，nocopy 的使用方法在 MIC 编程章节中进行了详细的介绍。针对上面的 MIC 程序采用 nocopy 之后的伪代码如下：

```
1   …
2   p_c = …; //p_c 在每次迭代中值不改变
3   #pragma offload target(mic) \
4   in(p_c: length(…) alloc_if(1) free_if(0)) \
5   nocopy(p_in:length(…)alloc_if(1) free_if(0)) \
6   nocopy(p_out: length(…) alloc_if(1) free_if(0))
7   {
```

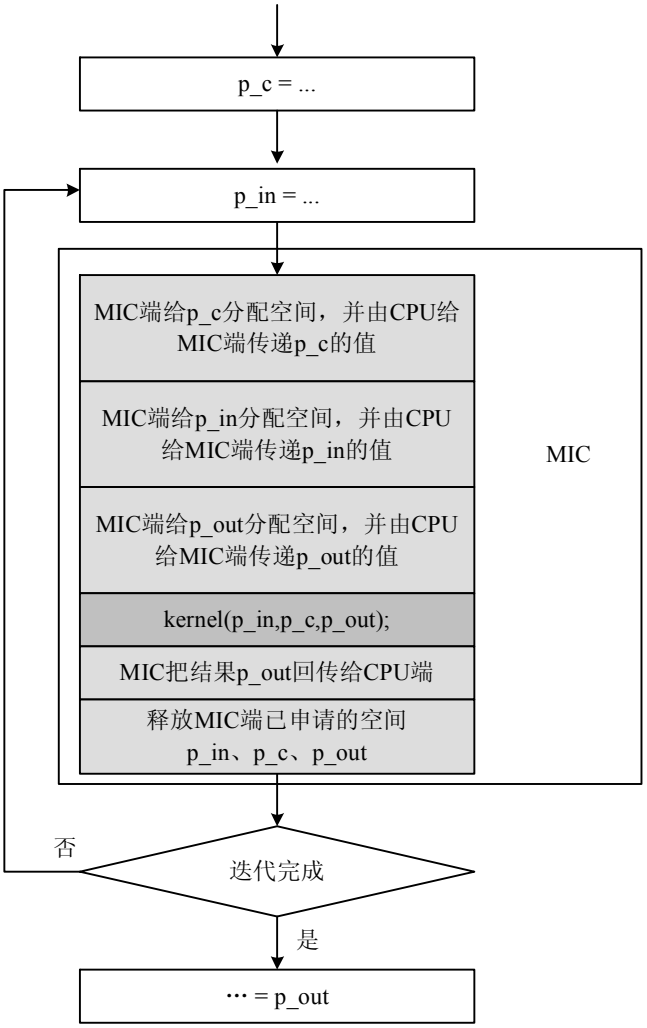

图 9-3　MIC 程序执行过程

```
8    } //申请空间，并且不释放；传递 p_c 的值
9    for(i=0; i<steps; i++)//迭代多次
10   {
11         p_in = …; //每次迭代计算时，p_in 的值变化
12   #pragma offload target(mic) \
13         in(p_in:length(…) alloc_if(1) free_if(0) ) \
14         nocopy(p_c) \
15         nocopy(p_out)     //每次迭代传递 p_in 的值，p_c 和 p_out 采用 nocopy
16         {
17               kernel(p_in, p_c, p_out);
```

```
18          }
19      }
20      #pragma offload target(mic) \
21      nocopy (p_c: length(…) alloc_if(0) free_if(1)) \
22      nocopy(p_in:length(…)alloc_if(0) free_if(1)) \
23      out(p_out: length(…) alloc_if(0) free_if(1))
24      {
25      } //回传 p_out 值到 CPU 端，并释放 MIC 端申请的空间
26      … = p_out; //CPU 端在所有迭代完成之后才用到 p_out 的值
```

采用 nocopy 的 MIC 程序的执行过程如图 9-4 所示，从该图可以看出，采用 nocopy 技术减少了 CPU 与 MIC 的通信次数。nocopy 技术对程序的性能提升起着至关重要的作用，主要应用在需要迭代计算的应用程序中。

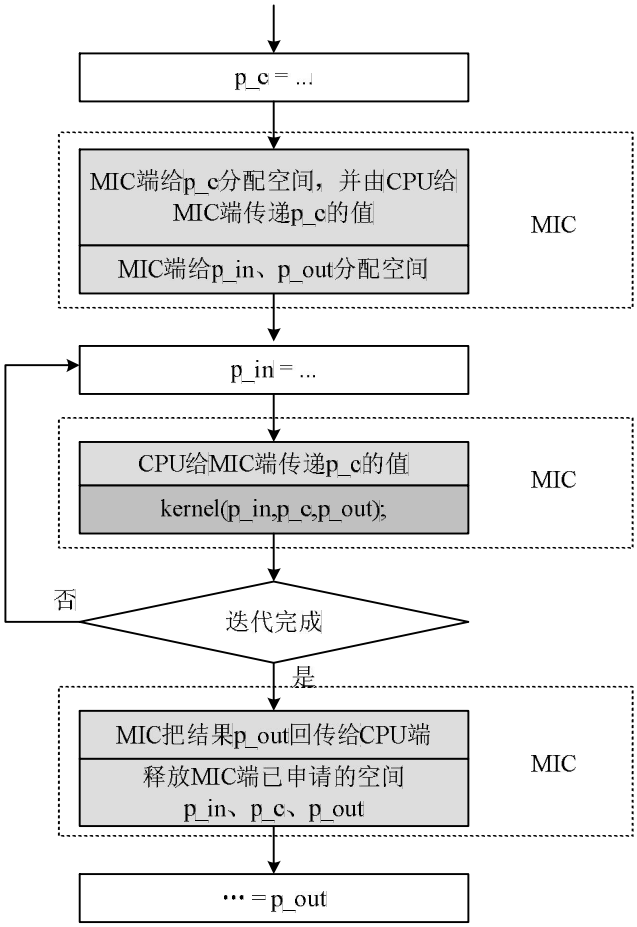

图 9-4　采用 nocopy 的 MIC 程序执行过程

9.2.3.2 offload 异步传输

对于 MIC 的 offload 模式来说，异步有两种含义，其一是数据传输与计算的异步，即 MIC 卡与主机端的数据传输与 MIC 卡上计算的异步，其二是计算与计算的异步，即 MIC 卡与 CPU 计算的异步。

1．数据传输与计算异步

数据传输层面的异步，通常用于需要多次调用 MIC 函数，且相邻调用之间没有依赖关系的情况。出现这种情况有可能是原始算法就是这么写的，也有可能是因为某种原因（例如卡上内存空间不足）对数据进行了分块处理，使得程序需要多次调用没有数据依赖的 MIC 函数。数据传输层面的异步，能够带来的明显好处是：以流水线方式执行传输和计算，可以将绝大部分数据传输时间隐藏，如图 9-5 所示。

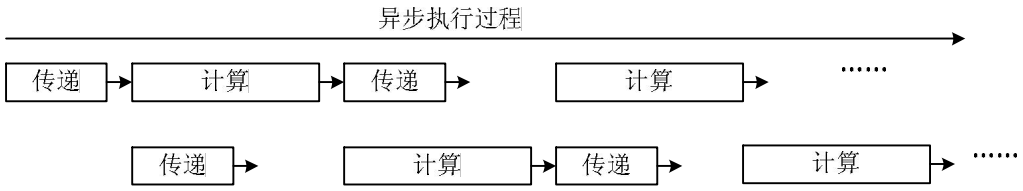

图 9-5　异步执行示意图

因此，在需要对算法进行分块处理的时候，或者数据传输时间比较明显，已经严重影响到程序执行效率的时候，可以使用数据传输异步的方式，对程序进行优化。

数据传输优化主要使用两个 offload 语句的变种：offload_transfer 和 offload_wait。offload_transfer 的作用是传输数据，并在数据传输完成时发送信号。其参数是与传统的 offload 语句完全一致的，区别在于 offload_transfer 语句后面没有代码段，仅有 offload 一条语句。offload_wait 的作用是暂停程序的执行，直到接收到 offload_transfer 发送的信号。举例如下：

```
//C/C++
1    #pragma offload_attribute(push, target(mic))
2    int count = 25000000;
3    int iter = 10;
4    float *in1, *out1;
5    float *in2, *out2;
6    #pragma offload_attribute(pop)
7
8    void do_async_in()
9    {
10           int i;
11   #pragma offload_transfer target(mic:0) \
12           in(in1 : length(count) alloc_if(0) free_if(0) ) signal(in1)
```

```
13
14          for (i=0; i<iter; i++)
15          {
16                  if (i%2 == 0) {
17
18  #pragma offload_transfer target(mic:0) if(i!=iter-1) \
19                  in(in2 : length(count) alloc_if(0) free_if(0) ) signal(in2)
20
21  #pragma offload target(mic:0) nocopy(in1) \
22                  wait(in1) out(out1 : length(count) alloc_if(0) free_if(0) )
23
24                  compute(in1, out1);
25
26                  } else {
27
28  #pragma offload_transfer target(mic:0) if(i!=iter-1) \
29                  in(in1 : length(count) alloc_if(0) free_if(0) ) signal(in1)
30
31  #pragma offload target(mic:0) nocopy(in2) \
32                  wait(in2) out(out2 : length(count) alloc_if(0) free_if(0) )
33
34                  compute(in2, out2);
35                  }
36          }
37  }
```

```
!Fortran
1   integer, parameter :: iter = 10
2   integer,parameter :: count = 25000
3   !dir$ options /offload_attribute_target=mic
4   real(4), allocatable :: in1(:), in2(:), out1(:), out2(:)
5   integer :: sin1, sin2
6   !dec$ end options
7
8   contains
9
10  subroutine do_async_in()
11      integer i
12
13  !dir$ offload_transfer target(mic:0) &
14      in(in1 :    alloc_if(.false.) free_if(.false.) ) signal(sin1)
15
```

```
16  do i = 0, (iter - 1)
17      if (mod(i,2) == 0) then
18
19          !dir$ offload_transfer target(mic:0) if(i/=iter-1) &
20              in(in2 :    alloc_if(.false.) free_if(.false.) ) signal(sin2)
21
22          !dir$ offload target(mic:0) nocopy(in1) wait(sin1) &
23              out(out1 : length(count) alloc_if(.false.) free_if(.false.) )
24
25          call compute(in1, out1);
26
27      else
28
29          !dir$ offload_transfer target(mic:0) if(i/=iter-1) &
30              in(in1 : alloc_if(.false.) free_if(.false.) ) signal(sin1)
31
32          !dir$ offload target(mic:0) nocopy(in2) wait(sin2) &
33              out(out2 : length(count) alloc_if(.false.) free_if(.false.) )
34
35          call compute(in2, out2);
36
37      endif
38   enddo
39
40  end subroutine do_async_in
```

本例仅展示了 do_async_in 一个函数，但这个函数已足以展示异步传输的应用。在本例中，我们开辟了两个数组空间，并等待使用第 1 个空间 in1、out1 计算时，同时在传输第 2 个空间 in2、out2 的数据，当第 2 个空间数据传输完成，且第 1 个空间的计算完成时，开始第 2 个空间的计算，并且传输第 1 个空间的数据，循环往复，直到全部计算完成。即在循环变量为奇数时，传输第 2 个空间数据，并等待第 1 个空间的数据计算完成，循环变量为偶数时反之。由于二者交替进行，因此最终的执行时间大致等于传输和计算二者中较长的时间，而非同步时的二者时间之和。

在进入循环以前，首先使用 offload_transfer 将第 1 个空间填充（C：11～12 行，Fortran：13～14 行），这样在进入循环体时（C：14 行，Fortran：16 行），可以首先启动填充第 2 个空间的工作（C：18～19 行，Fortran：19～20 行），然后等待第 1 个空间传输完毕（C：21～22 行，Fortran：22～23 行），再对第一个空间的内容进行计算（C：24 行，Fortran：25 行）。由于 MIC 卡只有一个，因此对计算来说（也对循环来说）是串行执行的，而只有传输和计算之间才是异步的。代码中需要注意的是，由于本段代码并非完整程序，因而在第一次传输时，并没有开辟空间（alloc_if(0)），这是因为在该函数外，已经在 MIC 卡上开辟空间了（代码中未

体现）。这种方式可以避免在循环中重复开辟空间，节省了时间。

另外一个值得注意的地方是 signal 的参数。在 C/C++中，参数是需要传递的一个数组的指针，在 Fortran 中，则是一个 integer 类型的变量。这里需要注意两点，其中一点是在不同语言中参数的区别，另一点是 signal 中能且只能使用一个参数，即使在 C/C++中，无论同时传输多少数组，只需要其中任意 1 个作为 signal 的参数即可。

在使用数据传输异步优化时，最主要的是需要注意对卡上内存空间的占用。由于卡上要同时拥有传输和计算两部分的空间，即内存占用成倍增长，因此如果分块过大，很有可能造成卡上内存空间不够用，导致程序运行失败。因此，即使是传输时间较长的应用，也不一定适宜使用过多的数组空间进行异步。

2．计算与计算异步

计算异步多用于 CPU 与 MIC 协同计算的情况，让 CPU 与 MIC 各分担一部分任务，以充分利用节点内的计算资源，达到加速程序运行的目的。通常意义上的协同计算，是指 MIC 计算函数与 CPU 计算函数分属不同线程，甚至不同进程当中，二者之间是相对较严格的并行方式。这种情况也可以算是计算异步的一种方式，但在本节所述的计算异步，则更多地是指另一种方式，即二者处于同一线程，启动函数是串行的，但函数的执行是并行的。

传统 offload 方式的 MIC 程序中，当调用 offload 语句，即 MIC 函数后，MIC 函数即接管控制权，CPU 线程处于等待状态，直到 MIC 函数执行完成返回时，才将控制权交回给 CPU 线程，此时 CPU 线程才能继续向下执行。在计算异步的 MIC 程序中，调用 offload 语句后，代码段在 MIC 卡上启动以后，驱动即刻将控制权交还给 CPU 线程，CPU 线程继续下面的工作，当 MIC 函数执行完毕返回时，会给 CPU 线程发送一个信号。可以很显然地发现，异步模式下，CPU 线程和 MIC 函数在一部分时间内是并行执行的，这样自然节省了运行时间。

这种计算异步方式相对前文所述的协同计算，在代码上比较简单，也不需要启动多个线程，虽然并发性可能略有不如，但灵活性却有很大提高。MIC 端的计算函数不需要很大改变，而 CPU 端的函数也未必与 MIC 计算函数完成同样的任务（也许规模不同），CPU 端的函数也可以做一些数据准备，或是其他的工作，以便为下次 MIC 计算做好准备。这是狭义的协同计算方式所无法做到的。

计算异步方式用到的除了传统的 offload 语句以外，也会使用数据传输异步中介绍的 offload_wait 语句等待计算完成的信号。以下是一个简单示例：

```
1    int counter;
2    float *in1;
3    counter = 10000;
4    __attributes__((target(mic))) mic_compute;
5    while(counter>0)
6    {
7    #pragma offload target(mic:0) signal(in1)
8          {
```

```
9                mic_compute();
10           }
11           cpu_compute() //此时本函数与上面的 MIC 函数并行执行
12   #pragma offload_wait target(mic:0) wait(in)
13           counter--;
14   }
```

```
1    integer signal_var
2    integer counter
3    counter = 10000
4    !DIR$ ATTRIBUTES OFFLOAD:MIC :: mic_compute
5    do while (counter .gt. 0)
6            !DIR$ OFFLOAD TARGET(MIC:0) SIGNAL(signal_var)
7                call mic_compute()
8            call cpu_compute()！此时本函数是与上面的 MIC 函数并行执行的
9            !DIR$ OFFLOAD_WAIT TARGET(MIC:0) WAIT (signal_var)
10           counter = counter - 1
11   end do
12   end
```

如上例所示，在调用 mic_compute 后，程序马上返回，将控制权交还给 CPU，CPU 可以继续执行 cpu_compute，并在 offload_wait 处等待，直到 MIC 函数计算完成后，offload_wait 才返回，继续执行下面的代码。

9.2.3.3 SCIF 传输优化

我们在前面章节已经知道 CPU 与 MIC 之间是通过 PCI-E 总线进行数据通信，而通过 PCI-E 的数据传输速度较慢，因此，我们需要尽量减少 CPU 与 MIC 之间的数据通信，但有时候 CPU 与 MIC 之间进行频繁的小数据通信或者在内核计算完后，MIC 与 MIC 或 MIC 与 CPU 之间需要小数据通信等情况是避免不了的，遇到这些情况该如何解决呢？可以考虑 SCIF 这种数据传输方式，这种数据传输方式可以有效地提高在频繁地进行小数据通信的情况下数据传输的性能。下面我们通过一个简单的例子来看一下这种通信是如何进行的。

在 host 端运行的程序（作为请求端）：

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <stdint.h>
4    #include <unistd.h>
5    #include <fcntl.h>
6    #include <string.h>
7    #include <sys/ioctl.h>
8    #include <scif.h>
```

```
#include <sys/time.h>
int main(int argc, char **argv)
{
        scif_epd_t epd;
        int err = 0;
        int req_pn = 11;
        int con_pn;
        struct scif_portID portID;
        int i, num_loops = 0, total_loop = 10;
        char *senddata;
        char *recvdata;
        int msg_size;
        int block;
        int node;
        struct timeval tv;
        struct timeval tv1;
        if (argc != 4) {
                printf("Usage ./scif_connect_send_recv* <msg_size><0/1 for noblock/block><node>\n");
        exit(1);
        }
        msg_size = atoi(argv[1]);
        block = atoi(argv[2]);
        node = atoi(argv[3]);
        portID.node = node;
        portID.port = 10;
        printf("Open the scif driver\n");
        if ((epd = scif_open()) < 0) {//在请求端建立一个新的 endpoint。
                printf("scif_open failed with error %d\n", (int)epd);
        exit(1);
        }
        printf("scif_bind to port 11\n");
        if ((con_pn = scif_bind(epd, req_pn)) < 0) {//请求端：绑定该 endpoint 到该 endpoint 所在的端口上。
                printf("scif_bind failed with error %d\n", con_pn);
        exit(2);
        }
        printf("req_pn=%d",req_pn);
        retry:
        if ((scif_connect(epd, &portID)) != 0) {//此处使用 retry 去试着和监听端建立通信，如果监听端
        已经做好连接准备，此处的连接则会直接连通，否则会进行几次 retry 连接操作。如果已经连
        通则可以进行下面的数据交换等操作。
                if (ECONNREFUSED == errno) {
                        printf("scif_connect failed with error %d retrying\n", errno);
```

```
49                      goto retry;
50              }
51              printf("scif_connect failed with error %d\n", errno);
52              exit(3);
53      }
54      printf("scif_connect success\n");
55      printf("node=%d,port=%d\n",portID.node,portID.port);
56      while (num_loops < total_loop) {
57              senddata = (char *)malloc(msg_size);
58              if (!senddata) {
59                      perror("malloc failed");
60                      err = ENOMEM;
61              }
62              memset(senddata, 0x25, msg_size);
63              err = 0;
64              gettimeofday(&tv,0);
65              while ((err = scif_send(epd, senddata, msg_size, block))<= 0) {//此处两个 endpoints 之间
                已经建立连通，请求端向监听端发送数据。
66                      if (err < 0) {
67                              printf("scif_send failed with err %d\n", errno);
68                              fflush(stdout);
69                              free(senddata);
70                              goto close;
71                      }
72              }
73              gettimeofday(&tv1,0);
74              printf("err=%d",err);
75              printf(" total = %f\n", (tv1.tv_sec-tv.tv_sec)*1e6+(tv1.tv_usec-tv.tv_usec));
76              err = 0;
77              free(senddata);
78              num_loops++;
79      }
80  close:
81      printf("Close the scif driver\n");
82      close(epd);
83      if (!err) {
84              printf("Test success\n");
85      }
86      else
87              printf("Test failed\n");
88      return (err);
89  }
```

在 MIC 端运行的程序（作为监听端）：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdint.h>
4   #include <unistd.h>
5   #include <fcntl.h>
6   #include <string.h>
7   #include <sys/ioctl.h>
8   #include <scif.h>
9   int main(int argc, char **argv)
10  {
11          int epd;
12          int newepd;
13          int err = 0;
14          int req_pn = 10;
15          int con_pn;
16          int backlog = 16;
17          struct scif_portID portID;
18          int i, num_loops = 0, total_loop = 10;
19          char *senddata;
20          char *recvdata;
21          int msg_size;
22          int block;
23          if (argc != 3) {
24                  printf("Usage ./scif_accept_send_recv* <msg_size><0/1 for noblock/block>\n");
25                  exit(1);
26          }
27          msg_size = atoi(argv[1]);
28          block = atoi(argv[2]);
29          portID.node = 2;
30          portID.port = 11;
31          if ((epd = scif_open())< 0) {//监听端建立新的 endpoint。
32                  printf("scif_open failed with error %d\n", errno);
33                  exit(1);
34          }
35          printf("scif_bind to port 10\n");
36          if ((con_pn = scif_bind(epd, req_pn))< 0) {//监听端：绑定已建立的 endpoint 到该 endpoint 所
            在的端口上。
37                  printf("scif_bind failed with error %d\n", errno);
38                  exit(2);
39          }
40          printf("scif_listen with backlog of 16\n");
```

```
41          if ((scif_listen(epd, backlog))< 0) {//监听端已经建立好了监听的准备，并且设置了可以在监
            听端最大的等待连接请求。
42                  printf("scif_listen failed with error %d\n", errno);
43                  exit(3);
44          }
45          printf("scif_accept in syncronous mode\n");
46          if (((scif_accept(epd, &portID, &newepd, SCIF_ACCEPT_SYNC))< 0) && (errno != EAGAIN)) {
            //监听端接受请求端的连接请求，并且此处设置了接受方式，是同步还是异步的方式。
47                  printf("scif_accept failed with errno %d\n", errno);
48                  exit(4);
49          }
50          printf("scif_accept complete\n");
51          printf("node=%d,port=%d",portID.node,portID.port);
52          while (num_loops < total_loop) {
53                  recvdata = (char *)malloc(msg_size);
54                  if (!recvdata) {
55                          free(senddata);
56                          perror("malloc failed");
57                          err = ENOMEM;
58                  }
59                  memset(recvdata, 0x00, msg_size);
60                  err = 0;
61                  while ((err = scif_recv(newepd, recvdata, msg_size, block))<= 0) {//接受请求端发送的数据。
62                          if (err < 0) {
63                          printf("scif_recv failed with err %d\n", errno);
64                          fflush(stdout);
65                          free(senddata);
66                          free(recvdata);
67                          goto close;
68                          }
69                  }
70                  printf("err=%d",err);
71                  err = 0;
72                  free(recvdata);
73                  num_loops++;
74                  }
75  close:
76          printf("Connection is complete\n");
77          fflush(stdout);
78          scif_close(newepd);
79          fflush(stdout);
80          if (!err) {
```

```
81                printf("Test success\n");
82          }
83          else
84                printf("Test failed\n");
85          scif_close(epd);
86          return (err);
87    }
```

在上面这个例子中，在 CPU 上运行的是请求端而在 MIC 卡上运行的是监听端，当监听端监听到与之相连接的端口发送的请求的时候就会与请求端建立连接进行数据传输。不同的 MIC 卡之间也可以通过这种传输方式进行数据通信。在介绍 SCIF 基础章节当中，我们已经知道 SCIF 这种传输方式比较擅长于小数据之间的通信，基于上面的例子我们进行了一系列的实验，实验结果如图 9-6 所示。

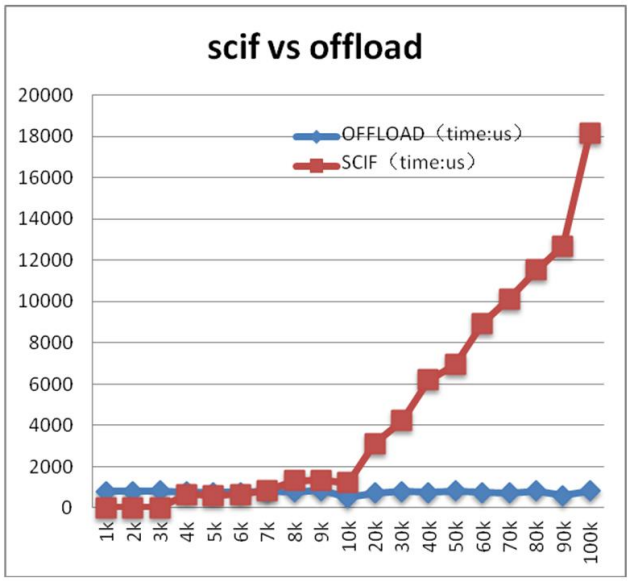

图 9-6　SCIF 与 offload 两种模式的性能对比

当数据为小于 4K 的时候，SCIF 传输模式的性能要好于 offload 这种模式的性能。当数据量为 1K、2K、3K 的时候，SCIF 的性能是 offload 性能的 80 倍。当数据量为 4K～6K 的时候，SCIF 的性能基本和 offload 的性能差不多，SCIF 的性能仍然好于 offload 的性能，但仅仅为一倍多点儿；当数据量大于 7K 的时候，SCIF 的性能开始呈下降趋势；在数据量大于 10K 的时候，SCIF 的性能与 offload 的性能之间的差距十分明显，offload 的性能要完全好于 SCIF 的性能。通过上面的实验结果，相信读者已经很清楚在单次传输数据量为多少的情况下使用 SCIF 这种传输方式会比较好，从上图我们也可以看出 offload 这种传输方式所表现出的性能十分稳定，而使用 offload 这种传输方式在代码的书写上要比 SCIF 这种传输方式简单得多，这两种传

输方式各有优点，请读者根据项目的实际情况选择合适的通信模型，便于进行数据通信优化。

注意：在运行这两个程序的时候，先运行在 MIC 端的程序，先让一个 endpoint 建立监听，当 host 端的程序运行时意味着向这个 endpoint 发送连接请求。当然，模拟这样一个过程需要把这两段程序分别运行在服务器端和客户端，有时候对程序的控制不是很方便，比如我们还是以 CPU 端和 MIC 端之间的通信为例来简单说明一下如何让这两个程序写在一起并且只在客户端运行就可以模拟这样一个过程。这样做的好处是 CPU 和 MIC 的程序可以写在一起，使整体代码的书写比较整齐，而且符合常规编程思想。以下是单程序使用 SCIF 传输模式的基本方法和流程：

（1）把上面两个建立通信的整体过程分别封装为两个独立的函数。

（2）在 CPU 端开两个独立的线程分别控制在 CPU 端运行的部分和在 MIC 端运行的部分。

（3）在 MIC 端运行的部分先执行（先有监听才能接受发送端的连接请求）。

（4）在 CPU 端运行的部分后执行（当有监听后发送请求的请求才能被接受）。

9.2.4 存储器访问优化

存储墙问题一直是计算机系统发展的瓶颈。现代处理器速度的快速发展和存储器速度的慢速发展导致处理器要花费大量的时间等待存储器数据的返回，这就是存储墙问题。在过去的十几年中，处理器速度以每年 50%～100%的速度平稳增长，而存储器的速度却只以每年 7%左右的速度增长。我们可预计处理器与存储器之间的速度差异将会越来越大，所以存储系统仍将是影响整个计算机系统性能的一个关键瓶颈。在并行计算体系结构中，计算速度更快，存储墙问题更加突出，因此，MIC 众核处理器需要开启数百个线程，存储器的访问优化对性能的影响极为突出。

9.2.4.1 MIC 层次存储结构

MIC 卡不仅核架构与 CPU 类似——基于 x86 架构，存储器结构也和 CPU 类似。MIC 采用了两级 Cache 结构，KNC 芯片框图如图 9-7 所示，MIC 层次存储结构如图 9-8 所示。

KNC 卡包含 8 个双通道 GDDR5 内存控制器，内存传输带宽为 5.5GT/s。

KNC 包含两级 Cache 结构：L1 Cache 和 L2 Cache。

KNC 每个核包含 32KB L1 指令 Cache 和 32KB L1 数据 Cache，L1 Cache line 为 64B，采用 8 路关联，8 个 bank，L1 Cache 为每个核私有，访问速度快。

KNC 拥有共享的 L2 Cache，每个核上的 L2 Cache 包括 L1 的数据缓存和指令缓存。不仔细分析可能不清楚那些核之间是怎么组织成一个大的共享的 L2 Cache（达到 31MB）。因为每一个核包含 512 KB L2 Cache，62 个核的 L2 Cache 即为 31MB，看起来好像 31MB L2 Cache 都是可用的。然而，如果两核或多核之间共享数据，这些共享数据在不同核的 L2 Cache 中是重复的。如果核之间没有共享任何数据或代码，那么片上 L2 的全部的大小为 31MB，相反如

果每一个核同时共享相同的代码或数据，那么片上 L2 的全部的大小仅仅为 512KB（每个核上 L2 Cache 存放相同的 512KB 数据）。

图 9-7　KNC 芯片框图

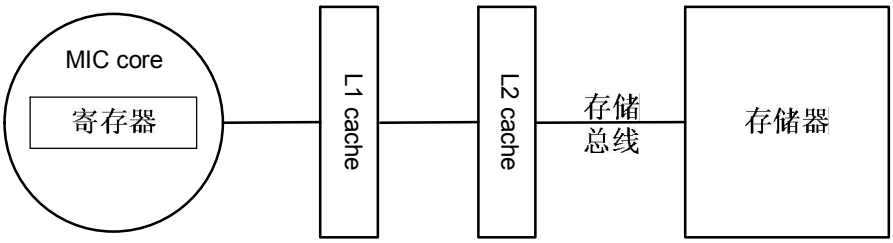

图 9-8　MIC 层次存储结构

9.2.4.2　MIC 存储器访问优化策略

在 MIC 平台上提高存储器访问性能主要通过两种方法：

1．隐藏存储器访问延迟

隐藏存储器访问延迟的基本思想是在处理器进行计算时，如果出现访问存储器时发生延迟，则可以通过预先的存储器操作或者另外的计算将这些延迟和处理器的计算重叠起来，使处理器不至于因为等待存储器操作的结果而停顿。下面给出 MIC 中两种常用的隐藏访存延迟的方法：

（1）多线程：多线程基于暴露线程级并行的思想来隐藏访存延迟。比较典型的就是同时多线程技术，其基本思想是在一个线程的指令发生访存延迟的时候，从另一个线程中选择适当

的指令执行，这样不至于让处理器发生停顿。MIC 卡每个 core 支持最多 4 个线程，采用了硬件多线程技术，该技术就是通过多线程隐藏访存延迟，因此，在 MIC 程序设计时要尽量提高并行度，使每个核上运行 3～4 个线程比较理想。

（2）预取：预取技术指在处理器需要数据或者指令之前将其从存储器中取出，以备需要时使用。目前的预取技术可以分为硬件预取（扩展存储器管理子系统的体系结构）、软件预取（利用现代处理器的非阻塞预取指令）和混合预取三种。MIC 支持硬件预取，由 MIC 硬件自动完成。

2．利用 Cache 优化

通常情况，程序访存时绝大部分时间集中在少量的区域，大量实验表明，程序执行时 90%的访存集中在 10%的区域，这就是程序的局部性原理。程序的局部性又可以分为两种，一种是时间局部性（Temporal Locality），指某一区域如果被访问，那么它很快被再次访问的几率较大；另一种是空间局部性（Spatial Locality），指某一区域如果被访问，那么它相邻的区域很快被访问的几率较大。

MIC 包含 L1 和 L2 两级 Cache，充分利用程序的局部性原理，可以提高 Cache 命中率，也就可以提高访存效率，下面小节详细介绍 MIC 上 Cache 优化方法。

9.2.4.3 Cache 优化方法

Cache 优化主要利用程序的局部性原理，代码级别的 Cache 优化主要有两个方法：

1．代码变换

代码变换是指针对程序指令进行的程序变换，绝大部分的编译优化技术都属于这种。通过代码变换，不但能够改变指令之间的关系，优化指令自身的局部性，提高指令 Cache 的性能，而且还能够通过改变指令的执行顺序来优化程序数据的局部性，提高数据 Cache 的性能。代码变换的主要方法是循环的变换，包括以下几种变换方法：

（1）循环融合

循环融合是一种将多个循环合并成一个循环的代码变换方法。在进行融合的过程中，同一数据在多个循环中的多次使用能够变成在一个循环的一次迭代中的多次使用，这样提高了这些数据的时间局部性，从而提高数据 Cache 性能。通过循环融合，还能够扩大循环体，从而有利于进行指令调度。在循环融合中，变换前的多个循环之间也许存在依赖关系，融合变换不能违反这些依赖关系，这是循环融合的难点。

程序循环融合示例：

```
//原始循环
for(i=0;i<n;i++)
        a[i]=b[i]+1;
for(i=0;i<n;i++)
        c[i]=a[i]/2;
```

```
//循环融合
for(i=0;i<n;i++)
{
        a[i]=b[i]+1;
        c[i]=a[i]/2;
}
```

（2）循环分割

循环分割是一种和循环融合相逆的代码变换技术。通过循环分割能够将一个循环变换成多个循环，提高循环体中访问数据的空间局部性。如果一个循环的循环体中的数据之间存在依赖关系，循环分割将这些数据分到不同的循环体中，从而消除这些依赖关系，以便进行其他的循环变换。

程序循环分割示例：

```
//原始循环
for(i=0;i<n;i++)
{
    a[i]=a[i]+b[i-1];
    b[i]=c[i-1]*x*y;
    c[i]=1/b[i];
    d[i]=sqrt(c[i]);
}
```

```
//循环分割
for(i=0;i<n;i++)
{
    b[i]=c[i-1]*x*y;
    c[i]=1/b[i];
}
for(i=0;i<n;i++)
    a[i]=a[i]+b[i-1];
for(i=0;i<n;i++)
    d[i]=sqrt(c[i]);
```

（3）循环分块

循环分块是一种按照循环访问的数据的特性将一个循环分成多个嵌套的循环的代码变换技术。通过分块，循环尽量完成某一个数据集的处理，才开始下一个数据集的数据处理，这样提高循环访问数据的时间局部性。循环分块可以通过 Cache 大小确定分块的大小，充分利用 Cache 优化程序，提高性能，第 9 章中的矩阵乘示例展示了分块对性能的影响，分块需要特别注意的是“尾巴”的处理，如下面示例中：for(i=it; i<min(it+nb, n); i++)。

程序循环分块示例：

```
//原始循环
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        x[i][j]=y[i]+z[j];
```

```
//循环分块
for(it=0;it<n;it+=nb)
    for(jt=0;jt<m;jt+=mb)
        for(i=it;i<min(it+nb,n);i++)
            for(j=jt;j<min(jt+mb,m);j++)
                x[i][j]=y[i]+z[j];
```

（4）循环交换

循环交换指在嵌套循环中，改变循环嵌套的顺序，以此来改变数据的访问方式，从而提高循环访问数据的空间局部性。

程序循环交换示例：

```
//原始循环
for(j=0;j<m;j++)
    for(i=0;i<n;i++)
        c[i][j]=a[i][j]+b[j][i];
```

```
//循环交换
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        c[i][j]=a[i][j]+b[j][i];
```

通过代码变换来提高 Cache 性能的基本思想是改变指令执行的顺序，从而改变 Cache 命中率。基于循环的代码变换能否有效地提高 Cache 性能取决于循环访问数据的局部性特性。这样每种代码变换都需要基于其访问的数据建立相应的代价模型，这是代码变换的一个难点。在进行代码变换的时候，不但需要确保变换后程序的正确性，而且还需要确保变换前后程序可执行语义的等价性，这些都是代码变换的难点。代码变换也有一些优点，例如代码变换通过改变程序自身的局部性（包括时间局部性和空间局部性）来提高 Cache 性能，与平台无关，无须特定硬件的支持，也就是说 MIC 版本的升级无需重新设计代码变换。

2．数据变换

相比于代码变换改变程序指令的执行顺序，数据变换主要是改变程序中数据的布局，依据空间局部性原理提高数据 Cache 性能。数据变换的基本思想是：当程序访问数据时，将经常一起访问的数据组织在一起，使其在内存中的位置邻近。这样当发生 Cache 失效的时候，每次调入相应的 Cache 块，紧接着访问的数据也就由于在同一 Cache 块而被一起调入，在一定程度上减少了 Cache 失效次数。下面给出了两种常见的数据变换方法：

（1）数据放置

程序变量的地址在编译时或者运行时才能决定，这些地址决定了程序数据在内存中的位置，也就决定了这些数据在 Cache 中的位置。例如在虚拟索引的 Cache 中，变量的地址与数据 Cache 大小取模后就能得到变量所在 Cache 的相应地址。因此，可以通过将变量放置在适当的位置来决定其对应的 Cache 的位置。如果将经常一起访问的变量通过重新放置使其位于同一 Cache 块上，则能有效提高数据的空间局部性。在实际的数据放置实现中，需要先判断数据之间的关系来选择放置在一起的数据，常见的方法有 Clustering 和 Coloring 等，这种方法对一些基于指针的数据结构效果特别明显。

（2）数组重组

在科学计算程序中，大量的数据是以数组的形式出现，对这些数组进行重组能够有效地提高 Cache 性能。在这样的程序中经常出现一些循环，这些循环以循环变量为下标访问多个数组，循环的每次迭代都需要到各个数组所在的内存区域读取数据，如果将这些数组重组成以结构体为元素的数组，则循环的每次迭代用到的数据都在一个结构体中，读取这些数据时就只需访问内存中一个连续的区域，提高了数据的空间局部性，相应的 Cache 失效次数就会减少。

9.2.5 向量化优化

9.2.5.1 什么是向量化

Intel 的编译器支持向量化（Vectorization），向量化是使用向量处理单元进行批量计算的方法，可以把循环计算部分使用 MMX、SSE、SSE2、SSE3、SSSE3、AVX、Knights Corner Instructions 等扩展指令集进行向量化，从而大大提高计算速度。

MIC 处理器支持 512 位宽的 Knights Corner 指令，支持 16*32bit 或 8*64bit 处理模式，即向量化宽度为 8 或 16。512 位相当于 16 个单精度浮点型数据的长度，单精度浮点数据向量化的操作过程如图 9-9 所示，例如向量加操作 C[0～15]=A[0～15]+B[0～15]（A、B、C 均为 float 型数据），没有使用向量化时这个操作需要 16 次加运算，而向量化之后，把 A、B、C 放到向量寄存器中，进行一次向量加操作即可完成原来的 16 次加操作，因此，向量化可以大大提高计算速度。

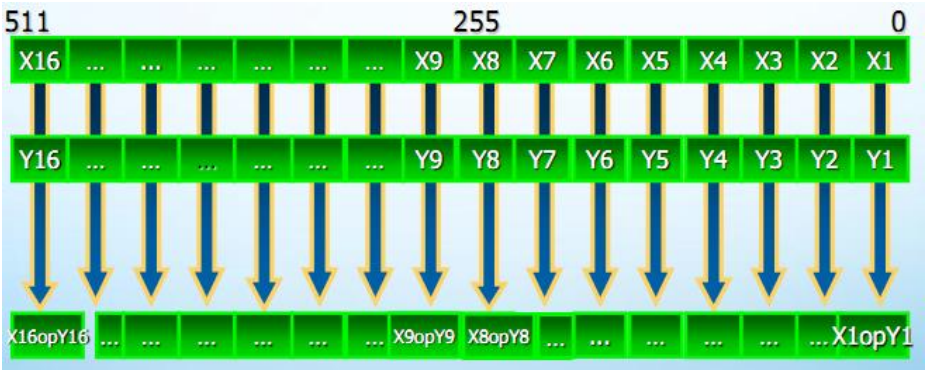

图 9-9　单精度浮点数据向量化示意图

9.2.5.2　MIC 向量化优化策略

MIC 向量化优化主要有两种方式：自动向量化和 SIMD 指令优化，向量化优化步骤一般为：

（1）插入引语自动向量化：不改变原程序结构，只需要插入预编译指令（引语）即可自动向量化。

（2）调整程序循环结构并插入引语自动向量化：对原程序做一些结构调整，如嵌套循环交换次序等，然后插入引语可以自动向量化。

（3）编写 SIMD 指令：SIMD 指令可以比自动向量化获得更好的性能，但针对不同的硬件平台编写的 SIMD 指令也不同，并且 SIMD 指令易读性较差，所以 SIMD 指令可以选择性的使用。

9.2.5.3　自动向量化

自动向量化是英特尔编译器提供的一个可以自动地使用 SIMD 指令的功能。在处理数据时，编译器自动选择 MMX、Intel Streaming SIMD 扩展（Intel SSE、SSE2、SSE3、SSE4、AVX 和 Knights Corner Instructions）等指令集，对数据进行并行的处理。使用编译器提供的自动向量化功能是提高程序性能的一个非常有效的手段。自动向量化在 IA-32 和 x86-64 的平台上均提供很好的支持。自动向量化能最大程度地解放程序员，既为程序员屏蔽底层 CPU/MIC 的细节，又能通过底层 CPU/MIC SIMD 并行获得有效的性能提升，MIC 处理器拥有 512 位宽的向

量化处理器，因此自动向量化对 MIC 程序优化意义更为重大。

1．自动向量化的好处

（1）提高性能：向量化处理，实现了单指令周期同时处理多批数据。

（2）编写单一版本的代码，减少使用汇编使编码工作简化：较少的汇编意味着会大大减少为特定的系统编程的工作，程序将很容易升级并使用于最新的主流系统而不必重新编写那些汇编代码。

2．什么样的循环可以向量化

（1）对于一个循环，如果编译器认为循环内的每一个语句都没有依赖于另一个语句并且没有循环的依赖关系，那么这个循环就是可向量化的。换句话说，每一个语句必须能独立执行，读写数据的操作必须中立于循环的每次迭代。

例如：

```
for (int i=0; i<1000; i++)
{
      a[i] = b[i] * T + d[i] ;
      b[i] = (a[i] + b[i])/2;
      c = c + b[i];
}
```

等价于下面的操作：

```
for (int i=0; i<1000; i++) a[i] = b[i] * T + d[i] ;
for (int i=0; i<1000; i++) b[i] = (a[i] + b[i])/2;
for (int i=0; i<1000; i++) c = c + b[i];
```

因此，这个循环是可以被向量化的。

再看一个例子：

```
for (int i=1; i<1000; i++)
{
      a[i] = a[i-1] * b[i];
}
```

无论如何，这个循环是不能被向量化的，因为 a[i]在每次迭代中都读取了前一次迭代中的 a[i-1]。我们称这是一个交叉迭代的数据依赖或者“flow dependence”，这样的循环不能被编译器向量化。

（2）向量化只能作用在最内层的循环：在一个嵌套的循环中，向量器只能尝试向量化最内层的循环，查看向量器的输出信息可以知道循环是否能被向量化以及原因，如果影响性能的关键循环没有向量化，你可能需要做一些更深层的打算，比如调整嵌套循环的顺序。

（3）向量化处理的数据类型尽量一致：需要向量化处理的语句，其包含的变量数据类型尽可能一致。如尽量避免在同一表达式中同时出现单精度和双精度变量。

3．编译器自动向量化方法

（1）编译器向量化选项：对于 MIC 程序，默认向量化编译选项为-vec，即默认情况下向量化是打开的，若关闭向量化可以在编译选项中添加-no-vec。向量化编译器可以生成自己的向量化报告，通过-vec-report 开关开启这一功能，具体选项功能如表 9-1 所示。

表 9-1　向量化报告

-vec-report[n]	含义
n=0	不显示诊断信息
n=1	只显示已向量化的循环（默认值）
n=2	显示已向量化和未向量化的循环
n=3	显示已向量化和未向量化的循环以及数据依赖信息
n=4	只显示未向量化的循环
n=5	显示未向量化的循环以及数据依赖信息

（2）#pragma ivdep 和 restrict 的使用

为了向量化一个包含或可能包含依赖关系的循环，加上#pragma ivdep（ivdep，ignore vector dependencies）。

例如：

```
1   void foo(int k)
2   {
3   #pragma ivdep
4         for(int j=0; j<1000; j++)
5         {
6               a[j] = b[j+k] * c[j];
7               b[j] = (a[j] + b[j])/2;
8               b[j] = b[j] * c[j];
9         }
10  }
```

当向量化这个循环时，编译器会认为数组 b 依赖了交叉迭代，原因就是使用了变量 k，如果我们知道 k 不会造成数据依赖，加上#pragma ivdep 编译指导语句忽略数据的依赖关系并尝试进行向量化。程序员必须知道这个依赖是怎么产生的，并确信它们没有数据依赖性。

使用“#pragma vector always”编译指导语句，指定循环向量化的方式可以避免一些没有内存对齐的操作没有被向量化。甚至可以使用“#pragma simd”语句，同时在编译选项中加入-simd，强制向量化最内层循环，如果这么做的话，需要程序员保证程序的正确性。最后一条语句与前面两条不同的是，前面两条对编译器来说只是建议，最终是不是被向量化，是由编译器决定的，但“#pragma simd”指令对编译器来说是强制的，如果编译器坚持认为这段代码无法向量化，将会产生一个编译器错误。

在 MIC 平台上，可以采用“#pragma vector aligned”进行向量化对齐，但必须保证内存分配以 64B 对齐，即以 align(64)声明变量。

使用指针的循环可能造成依赖性，如果为了向量化这样的循环，可以使用 restrict 关键字。

```
1    void foo(float*restrict a, float*restrict b, float*restrict c)
2    {
3        for(int j=0; j<1000; j++)
4        {
5            a[j] = b[j] * c[j];
6            b[j] = (a[j] + b[j])/2;
7            b[j] = b[j] * c[j];
8        }
9    }
```

注意，使用了 restrict 关键字需要使用-restrict 编译选项。如果不使用 restrict 关键字，编译器会认为数组的引用可能有交叉迭代的依赖性。这是因为指针在循环中用来访问数据，编译器无法知道指针是否指向了相同的地址（一般就是别名），为了安全必须阻止这样程序向量化，restrict 关键字告诉编译器指针指向的地址是受限的，只能通过这个指针访问，换句话说，这里没有别名。

4．自动向量化优化方法

（1）调整嵌套循环的顺序

自动向量化只能对嵌套中的最内层的循环进行向量化，然而内层循环向量化效果未必最好，我们可以通过调整嵌套循环的顺序达到更好的向量化效果，如调整之后的向量化可以满足更好的连续访问，如下面的代码所示，右面代码的向量化效果比左面的更好。

示例：

```
for(j=0; j<N; j++)
#pragma ivdep
for( i=0; i<M; i++)
{
    C[i][j] = A[i][j]+B[i][j];
}
```

```
for( i=0; i<M; i++)
#pragma ivdep
for(j=0; j<N; j++)
{
    C[i][j] = A[i][j]+B[i][j];
}
```

（2）拆分循环

在某些情况下，除了最内层的循环比较耗时外，其他不在最内层循环的代码也比较耗时，而这部分代码是无法自动向量化的，为此，我们可以采取拆分循环的方法实现更多的自动向量化，下面通过一段伪代码说明其使用方法。

示例一：

```
1    for(i=0; i<N; i++)
2    {
3        rand();
```

```
4          …
5          …
6    }
```

假设上面的代码中循环无数据依赖，由于 rand 函数无法向量化，从而导致整个循环无法向量化，我们可以通过把一个循环拆成两个循环的方法达到第二个 for 循环（主要耗时的）实现向量化的目的，代码如下：

```
1    for(i=0; i<N; i++)
2    {
3          rand();
4    }
5    for(i=0; i<N; i++) //自动向量化
6    {
7          …
8          …
9    }
```

示例二：

```
1    float s;
2    for(i=0; i<N; i++)
3    {
4          …
5          s=…;
6          for(j=0; j<M; j++)//自动向量化
7          {
8                if(s>0)
9                {
10                     …
11               }
12         }
13   }
```

假设上面的代码中两层循环均无数据依赖，除了内层循环 for(j=0; j<M; j++)比较耗时，对于 s 的求解也很耗时，然而求解 s 的部分是无法自动向量化的。我们可以通过拆分外层的循环做到更好的自动向量化效果，修改后的伪代码如下：

```
1    float s[16];
2    for(i=0; i<N; i+=16)
3    {
4          T=min(N-1,16);
5          for(k=0; k<T; k++) //自动向量化
6          {
7                …
```

```
8                s[k]=…;
9          }
10         for(k=0; k<T; k++)
11         {
12               for(j=0; j<M; j++) //自动向量化
13               {
14                     if(s[k]>0)
15                     {
16                           …
17                     }
18               }
19         }
20   }
```

通过对循环的拆分，我们可以使更多的代码自动向量化，获取更好的向量化性能。

（3）并行度与向量化

由于在 MIC 内核中我们可以开启数百个线程，所以我们要保证足够多的线程数，然而，在多数应用中，我们一般采取并行外层循环，内层循环采用向量化优化，如果外层循环次数较少将会影响到并行度，因此，我们需要考虑到并行度和向量化的权衡。

下面通过示例讲解这个方面的知识。

```
1    for(i=0;i<100;i++)
2    {
3          for(j=0;j<1024;j++)
4          {
5                …
6          }
7    }
```

假设上面的代码两层 for 均无数据依赖， MIC 并行的最佳线程数为 200，上面的代码内层 for 可以采用自动向量化的方法，外层 for 作为 MIC 并行时仅有 100 次循环，并行度较低，不利于发挥 MIC 的最佳性能，对于这种情况，我们可以采用拆分内层 for 的方式让并行程序既可以满足并行度也可以满足自动向量化，一种拆分方法如下：

```
1    for(i1=0;i1<200;i1++)
2    {
3          for(j1=0; j1<512; j1++)
4          {
5                i = i1/2;
6                j = (i1%2)*512+j1;
7                …
8          }
9    }
```

上面的方法可以满足 MIC 的并行度，同时满足自动向量化，上面的拆分也可以把外层 for 循环次数变得更大，但要保证内层 for 循环次数>16 次以保证向量化的效果。

向量化性能提升明显，但在某些应用的情景下，会出现精度的损失。因为向量化使用的是向量单元，比如以前一个浮点数计算的时候，处理器给它分配了 1 个校验位，现在 16 个浮点数同时操作，但校验位仍然只有 1 个（硬件限制），导致精度会出现误差。当然，在绝大多数情况下，这点误差都是在可以接受的范围内的。

9.2.5.4 SIMD 指令优化

SIMD（Single Instruction Multiple Data）指令可以在程序执行中复制多个操作数，并把它们直接打包在向量寄存器中。显而易见，SIMD 指令在性能上有较大的优势，可以以同步方式，在同一时间内对多个数据执行同一条指令。

向量化的层次如图 9-10 所示，越往上的级别，使用的语言越低级，编程越复杂，但可以控制的部分也越多，理论上性能也越高。相反的，越往下的级别，编程越容易，但性能可能未必如此理想。

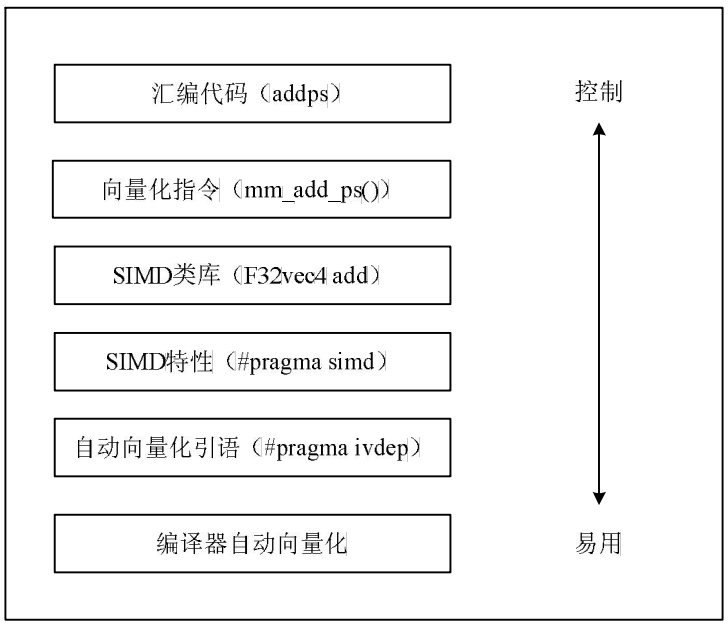

图 9-10 向量化层次

第一代 Intel MIC 产品为 KNC（Knights Corner），Knights Corner Instructions 是 KNC 支持的 SIMD 指令的总称。可以看作是类似于 SSE、AVX 等的指令集。通过使用 Knights Corner 指令，可以细粒度地控制向量化运算。

Knights Corner Instructions 分类：

（1）Knights Corner 指令（Knights Corner Instruction）是指具体的 SIMD 指令，是汇编指令集中关于 SIMD 操作的子集。

（2）内建 Knights Corner（Intrinsics of Knights Corner）是对 Knights Corner 指令的封装（几乎涉及到所有指令），可以认为这些函数和数据类型是 C/C++的内建类型。

（3）Knights Corner 类库（Knights Corner Class Libraries）是为了方便使用 Knights Corner 指令而做的封装，可以让程序员尽量简单地使用 SIMD 指令，介于引语方式和 SIMD 代码之间。其支持整型和浮点型数据。

下面通过单精度浮点向量加的例子说明三者的区别，见表 9-2。

表 9-2　Knights Corner 实现向量加

Knights Corner 指令	内建 Knights Corner	Knights Corner 类库
__m512 a,b,c; __asm{ vloadd v0,b vloadd v1,c vaddps v0,v1 vstored a, v0 }	#include <immintrin.h> ... __M512 a,b,c; a = _mm512_add_ps(b,c); ...	#include <micvec.h> ... F32vec16 a,b,c; a = b + c; ...

通过上面的例子可以看出使用类库的方式非常简单，能够以最类似于标量的方式（把数组看成变量），进行向量化改造。而直接使用内建 Knights Corner 则更接近常规的思维方式，将两个数组通过向量化函数进行运算，当然，其代码要比使用类库方式复杂一些，但由于减少了封装和调用，因此性能也会略有提高。而内联汇编则是最难阅读的，由于最贴近底层，因而执行效率也最高，只是编程的成本也是最高的。在实际的 SIMD 指令编写中，我们一般采用内建 Knights Corner 的方式。

下面我们通过一个向量加的示例说明 SIMD 指令的使用方法。

```
1   #include <immintrin.h>
2   void foo(float *A, float *B, float *C, int N)
3   {
4   #ifdef __MIC__
5           __M512 _A,_B,_C;
6           for(int i=0; i<N; i+=16)
7           {
8                   _A =_mm512_loadunpacklo_ps (_A, (void*)(&A[i]) );
9                   _A = _mm512_loadunpackhi_ps (_A, (void*)(&A[i +16]) );
10                  _B = _mm512_loadunpacklo_ps (_B, (void*)(&B[i]) );
11                  _B = _mm512_loadunpackhi_ps (_B, (void*)(&B[i +16]) );
12                  _C = _mm512_add_ps(_A,_B);
13                  _mm512_packstorelo_ps ((void*)(&C[i]) , _C );
```

```
14                _mm512_packstorehi_ps ((void*)(&C[i +16]), _C );
15          }
16    #endif
17    }
```

SIMD 指令与汇编指令类似，可读性较差，并且严重依赖于硬件，可移植性较差。SIMD 指令可以选择性使用，如代码量较少，计算却十分密集的地方。

9.2.6 负载均衡优化

9.2.6.1 什么是负载均衡

负载是指多个任务之间的工作量分布情况，负载均衡是指各任务之间的工作量平均分配。负载均衡在并行计算里指的是将任务平均分配到并行执行系统中各个计算资源上，使之充分发挥计算能力，没有空闲或等待，也不存在负载过度。好的并行方法可以发挥好的负载均衡效果，负载不均衡将会导致计算效率的下降以及糟糕的扩展性。因此，实现负载均衡是并行计算中的重要方面，尤其针对 MIC，其核数众多，负载均衡对其性能的影响更为明显。

通常情况下，实现负载均衡有两种方案：静态负载均衡和动态负载均衡。静态负载均衡需要人工将工作区域分割成多个可并行的部分，并保证分割成的各个部分（工作量）能够均衡地分布到各个处理器上运行，也就是说工作量在多个任务之间均衡地进行分配，使并行程序的加速性能最高；动态负载均衡是在程序运行过程中进行任务的动态分配以达到负载平衡的目的。实际情况中存在着很多静态负载均衡解决不了的问题，比如，在一个循环中，每次循环的计算量均不同，且不能事先预知。一般来说，动态负载均衡的系统总体性能比静态负载均衡要好，但代码实现上更复杂。

9.2.6.2 CPU/MIC 协同计算负载均衡优化方法

CPU/MIC 协同计算应用程序中包含 3 个层次的负载均衡：

（1）计算设备（CPU 或 MIC）内部各线程/进程之间的负载均衡。

（2）CPU/MIC 协同计算时，一个节点内 CPU 设备与 MIC 设备之间的负载均衡。

（3）集群计算时，节点之间的负载均衡。

图 9-11 展示了 CPU/MIC 协同计算的负载均衡层次结构。

1．设备内负载均衡

设备内的负载均衡可以采用 OpenMP 中的三种负载均衡策略：

（1）schedule(static [,chunk])：静态调度，线程每次获得 chunk 个迭代次数，并以轮询的方式进行。如果不指明 chunk，则以平均分配的方式进行，这是默认的调度方式。

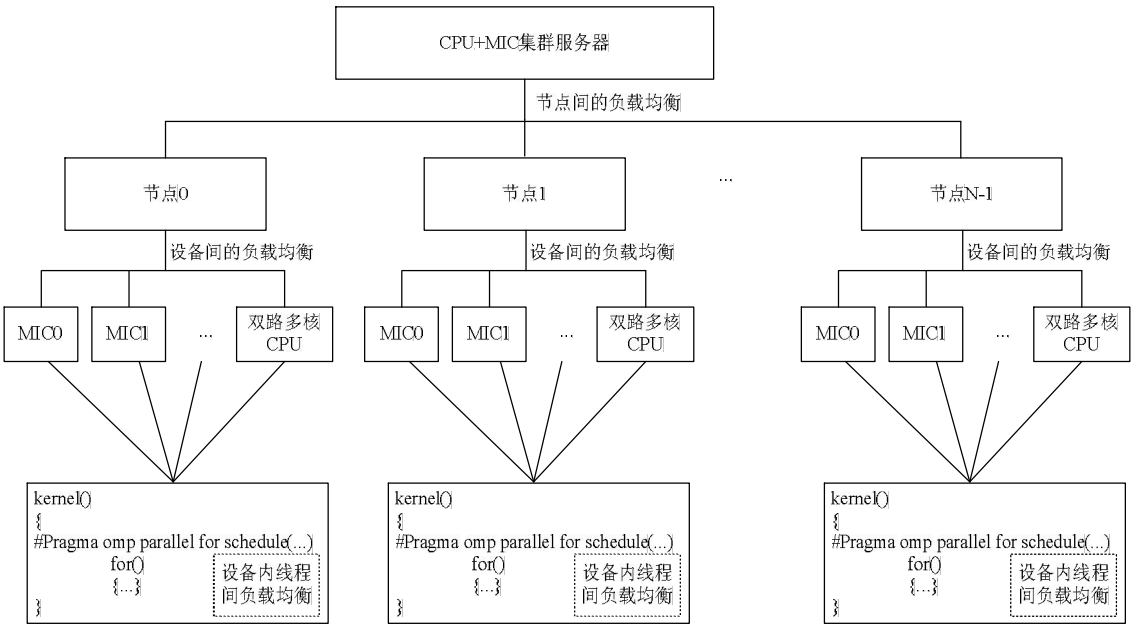

图 9-11　CPU/MIC 协同计算负载均衡

（2）schedule(dynamic [,chunk])：动态调度，动态地将迭代分配到各个线程，不使用 chunk 参数时将迭代逐个地分配到各个线程，使用 chunk 参数时，每次分配给线程的迭代次数为指定的 chunk 次。

（3）schedule(guided [,chunk])：导引调度是一种采用指导性的启发式自调度方法。开始时每个线程分配到较大的迭代块，之后分配到的迭代块会逐渐递减。迭代块的大小会按指数下降到指定的 chunk 大小，如果没有指定 chunk 参数，那么迭代块大小最小降到 1。

OpenMP 的调度策略使用范围如表 9-3 所示。

表 9-3　OpenMP 中不同调度算法的使用范围

调度算法	使用范围
Static	固定任务量，并且每次迭代任务量相同
Dynamic	任务量不固定，每次迭代任务量均不同
Guided	这是 dynamic 调度算法的特殊情况，导引调度算法可以减少调度的开销

2．CPU/MIC 设备间的负载均衡

由于 CPU 与 MIC 的计算能力不等，因此 CPU 与 MIC 之间分配的计算量也不能相同，CPU 与 MIC 之间的负载均衡最好的方式是采用动态负载均衡的方法。下面我们分别对任务划分和数据划分的情况下设备间负载均衡应该采用的优化方法进行介绍。

（1）任务划分：对于任务划分的应用程序，在 CPU 与 MIC 之间的负载均衡可以采用动

态负载均衡的优化方法，例如有 N 个任务，一个节点内有 2 个 MIC 卡，即三个设备（CPU 和 2 个 MIC），动态负载均衡的方法是每个设备先获取一个任务进行计算，计算之后立即获取下一个任务，不需要等待其他设备，直到 N 个任务计算完成。这种方式只需要设定一个主进程，负责给各个计算进程分配任务。

（2）数据划分：由于我们需要一次性在设备上分配内存空间，因此，对于数据划分的应用程序，我们无法采用动态负载均衡，需要采用静态数据划分方法，然而静态数据划分方法使异构设备间的负载均衡变得困难，有时甚至无法实现。对于一些迭代应用程序，我们可以采用学习型的数据划分方法，如让 CPU 和 MIC 分别做一次迭代相同计算量的计算，然后通过各自的运行时间计算出 CPU 与 MIC 的计算能力比例，然后再对数据进行划分。

3．节点间的负载均衡

节点间的负载均衡与传统 CPU 集群上的负载均衡一致，即可以采用静态负载均衡和动态负载均衡的方法。

9.2.6.3 线程亲和性优化

前面讲到的负载均衡，是指将任务分配到线程上的方式，使得每个线程的运行时间尽量相等。但是线程并不是凭空存在的，线程的执行需要运行在处理器的核心上。本小节讨论的问题，是如何将线程分配在不同的处理器核心上，使得每个线程的运行时间尽量相等，达到负载均衡的目的。

线程亲和性的负载均衡优化涉及到几个情况：

（1）线程只认识逻辑核，因此使用超线程技术的 1 个物理核也会被视为 2 个计算核心。

（2）这种优化思路，主要解决 cache 利用和物理核心负载均衡的问题，但二者互相制约。

（3）这方面的优化，通常用于不能利用全部物理核心或任务区域性很明显的情况。

综合以上情况，可以发现，线程会根据亲和性设置的不同，被分配到不同的逻辑核心上。如果线程间有数据相关，当逻辑核心处于同一物理核心时，线程间可以利用同一物理核心的 cache，提高运行速度。但此时计算集中于同一物理核心，在这个物理核心负载较高的同时，其他核心却处于低负载的状态，仍然影响了性能。而这种情况，在没有利用全部逻辑核心时，体现得更为明显。

对线程亲和性的负载均衡优化，需要设置环境变量：KMP_AFFINITY，其语法及解释请参见第 5 章中，OpenMP 环境变量条目中的相关解释。

通过第 5 章的学习，我们知道线程亲和性可以设置 scatter、compact 和 balanced（MIC 专属）等 3 种方式。

scatter 模式将线程优先分配到负载较轻的物理核心上，这种方式可以较好地达到负载均衡，但由于相邻线程不处于同一物理核心当中，因此如果相邻线程间有数据共享的话，则不能利用 cache 进行加速。而在 MIC 中，虽然可以读取其他核心的 L2 Cache，但会导致可用 cache 容量减少，而且从其他核心读取的速度也慢于从本地 cache 中读取数据。

compact 模式将线程按顺序分配至逻辑核心上，这种方式可以使相邻线程尽量处于同一物理核心当中，如果相邻线程间有数据共享的话，这种方式可以尽可能地利用 cache。如果相邻线程间计算量差距不大，则很有可能造成严重的负载不均衡。但是，如果是一种比较特殊的任务分配方式，例如奇数线程负载较轻，偶数线程负载较重，则这种方式反而可以达到较好的负载均衡。

balanced 模式是 MIC 上特有的模式，虽然与 scatter 模式类似，也是尽量将线程分配到负载较轻的物理核心，但与之不同的是，balanced 模式在兼顾均衡性的同时，也会尽量将相邻线程分配在同一物理核心当中。这种方式在负载均衡和 cache 利用上做到了一定的平衡。

需要注意的是，这三种方式都是静态划分的方式，因此需要根据程序运行的实际负载情况，有针对性地进行设置，才能达到比较好的效果。

9.2.7 MIC 线程扩展性优化

9.2.7.1 什么是线程扩展性

线程扩展性是指在同样算例、同样程序的前提下，将线程数扩展到更多时，程序性能提高的程度。举例来说，如果原来程序是在 4 核 CPU 上，4 个线程并行的程序，现在由于 CPU 升级为 8 核，所以程序也随之改变为 8 个线程并行，而这时的性能理论上应该提高到原来并行程序的 2 倍。

本例中有几点需要注意：

（1）扩展性是在计算硬件资源充足的情况下考量的。即线程数应该小于或等于计算核心数。这里计算核心数通常指逻辑核心，即包括超线程以后的核心，但在某些 CPU 上，超线程的性能未必最佳，因此做扩展性测试时，有时也会以物理核心数为准。甚至有时，在每个线程不能充分利用计算硬件的情况下，每核心运行 2 个甚至 3 个线程都能达到较高的性能。但这种情况在计算密集型的 MIC 程序中几乎不可能出现。实际应用中，由于硬件所限，如果脱离计算核心数单独考察扩展性的话，并没有太多意义。

（2）扩展性通常是指线程数提高以后，程序性能提升的情况。

（3）性能提高倍数的上限，是线程数增加的倍数。即线程数增加 1 倍，性能也随之增加 1 倍。这种情况被称为线性增长。但通常受其他条件制约，很难达到理想情况，尤其在线程数不断增多的情况下。这时需要注意拐点出现的地方，即性能增长率显著下降的地方，优化程序应将拐点尽量延后到充分利用计算核心以后。例如假设 MIC 卡为 200 个线程，应尽量让程序在 200 个线程以内的性能接近线性增长。当然，偶尔也会出现超线性增长的情况，即性能增长倍数超过线程增长倍数。这种情况一般出现在算法有更改的时候，但也有可能是 cache 引起的。

9.2.7.2 如何提高线程扩展性

正如上文所述，我们的目的，是希望程序能够达到，或接近线性加速比。而制约我们达成这一目的的，主要有三方面因素：

（1）算法。热点部分有依赖，不能完全并行，或热点部分所占比例不够高。

（2）并行度。如果程序本身并行度不高，那么增加线程也没有作用。

（3）硬件瓶颈制约。由于我们以计算核心数为线程数上限，因此计算核心并不可能成为瓶颈。此时瓶颈通常为内存大小、内存带宽、网络、存储带宽等。

提高线程扩展性，是一门综合过程，也是 MIC 程序优化的终极目标之一。因此，对提高线程扩展性来说，并没有什么独门秘籍，而是应该综合利用其他小节介绍的各种方法，借以达成目的。

（1）算法：改变算法通常是最根本的解决方法，对线程扩展性方面也不例外。提高线程扩展性的算法，需要注意尽量减少线程间依赖，使用并行性尽可能高的算法，即在理论上，应该尽量能够达到线性加速比。如果理论上都与线性加速相去甚远，又如何在现实中受到诸多限制时，达到较好的性能呢？另外一点是，需要注意热点所占比例，不仅仅是在串行程序中的比例，当热点被并行化以后，其运行时间缩短，热点所占比例也会缩短，一旦缩短至一定程度——在 CPU 中也许只是几分之一，在 MIC 上可能就是几十分之一了——虽然可能热点本身的扩展性仍然很好，但对整个程序来说，扩展性仍会变得很差。这也是在 CPU 并行优化中不常注意的问题。

（2）并行度：对于并行度的优化，请参见 9.2.1 节“并行度优化”。这里补充一点，如果原始程序为 MPI 版本，并可以在集群上运行，这时如果并行度不够高，可以尝试在一个节点运行一个或多个 MPI 进程，并在一个 MPI 进程内部，使用 OpenMP 并行子循环，即采用 MPI+OpenMP 方式，如果代码确实适合使用这种方式（即 MPI 进程内部并行性较好），则可以大幅提升线程扩展性。

（3）硬件瓶颈制约：硬件瓶颈通常是指内存带宽、内存容量等瓶颈。由于内存容量限制，导致程序无法扩展到更多核心的问题，请参见 9.2.2 节“内存管理优化”。由于内存带宽限制，导致程序扩展性不佳的问题，请参见 9.2.4 节“存储器访问优化”。