Intel Knights Corner的结点级内存访问优化

林新华[1,2] 李 硕[3] 赵嘉明[1] 松岗聪[2]
(上海交通大学高性能计算中心 上海 200240)[1] (东京工业大学学术国际情报中心 东京 152-8550)[2]
(Intel公司软件与服务部门 波特兰 999039)[3]

摘 要 传统编程优化(Traditional Programming Optimization,TPO)在Intel Knights Corner (KNC)上收效甚微,因此提出内存访问优化(Memory Access Optimization,MAO)。将MAO应用到已经过TPO的程序Diffusion 3D上,发现其性能仍然提高了39.1%。主要有2个贡献:1)提出MAO,认为TPO+MAO有助于在KNC上获取最优化性能;2)发现对于stencil代码,基于intrinsic的MAO比基于编译器的MAO更高效。这些发现对于在KNC上优化大规模应用有启发意义。

关键词 传统编程优化,Intel Knights Corner,内存访问优化,最优化性能
中图法分类号 TP391 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.11.006

Node-level Memory Access Optimization on Intel Knights Corner

LIN Xin-hua[1,2] LI Shuo[3] ZHAO Jia-ming[1] MATSUOKA Satoshi[2]
(Center for High Performance Computing,Shanghai Jiao Tong University,Shanghai 200240,China)[1]
(Global Scientific Information and Computing Center,Tokyo Institute of Technology,Tokyo 152-8550,Japan)[2]
(Software and Services Group,Intel Corporation,Portland 999039,USA)[3]

Abstract Traditional programming optimization (TPO) has limited effects on Intel Knights Corner (KNC). Therefore, we proposed memory access optimization (MAO) for KNC. We applied MAO to TPO version of Diffusion 3D, and its performance is improved by 39.1%. We made two contributions in this paper: 1) MAO is indispensable to KNC and TPO+MAO is the path to Ninja Performance—the best optimized performance. 2) Intrinsic-based MAO is more efficient to stencil code than compiler-based MAO. Our findings on MAO will inspire optimizations of large-scale applications on KNC.

Keywords Traditional programming optimization (TPO), Intel Knights Corner (KNC), Memory access optimization (MAO), Ninja performance

1 简介

传统编程优化(TPO)[1]可分为2个层面:算法优化和最新编译器技术的应用,大致包含以下6种方法:

• AOS到SOA的转换。AOS即Array-Of-Structures,而SOA即Structures-Of-Array。访问AOS数据结构需要进行gather和scatter操作,而这通常会导致SIMD(Single Instruction Multiple Data)的低效。

• Cache Blocking。将具有空间局部性(Spatial Locality)的数据尽可能地放在cache中,可支持1D、2D和3D的数据结构。有时也能利用时间局部性(Temporal Locality),在多个迭代间进行blocking。

• 适合SIMD的算法。一些算法应用在迭代中存在相关性或是大量使用了gather/scatter操作,导致无法有效执行向量化代码。这些局限通常可以通过更适合SIMD的算法来解决。有时迭代的相关性也可以通过shuffle和maskmov等操作来消解。

• 并行化。线程级的并行通常使用OpenMP来实现。

• 向量化。充分利用SIMD部件。与较为简单的内层循环向量化不同,外层循环的向量化需要考虑更多的因素。

• EMU(Extended Math Unit)。对于数学操作密集的单精度程序,可稍微降低精度以提高性能。

Intel公司的Nadathur Sathish等人在ISCA12论文[1]中将上述TPO应用到Intel Knights Ferry(KNC的前一代产品,未公开发行)上,并达到最优化性能(Ninja Performance)[1]的80%左右。现在看来他们的结论似乎过于乐观了,比如发表在IPDPS14上的全球浅水气候模型在经过并行化、向量化、EMU、SOA转换和cache blocking等TPO之后,仍然仅达到KNC峰值性能的8%[2]。又如IPDPS13中分子动力学测试算例miniMD在经过SIMD等TPO之后,KNC

到稿日期:2014-11-03 返修日期:2015-01-12 本文受国家高技术研究发展计划(863):高性能计算环境应用服务优化关键技术研究,日本学术振兴会RONPAKU Fellowship资助。

林新华 男,讲师,主要研究方向为性能建模与优化;李 硕 男,工程师,主要研究方向为高性能计算在量化金融中的应用;松岗聪 男,教授,主要研究方向为高性能计算。

的性能仅为 CPU 的 1.42 倍[3]。类似结果不少，加上 TPO 已被广泛采用，不少工具甚至都已默认了这些优化，因此我们有理由认为如果只用 TPO，那么对 KNC 而言会收效甚微。

但其实在 KNC 上也是能获得好性能的。Intel 公司 Alexander Heinecke 等人在 IPDPS13 上发表的 KNC 上 LINPACK 优化的工作中，DGEMM 的效率接近 90%，LINPACK 的效率也接近 80%[4]。因此我们分析了 KNC 与 CPU 在体系架构上的 3 点不同，探究了 TPO 低效的原因，据此提出必须重视 MAO。我们将 MAO 应用到 Black-Sholes Formula 和 Diffusion 3D 这 2 个已经过 TPO 的程序上，随后发现性能仍然分别提高了 8.1%和 39.1%。本文主要有 2 个贡献：

1)提出 MAO，认为 TPO+MAO 有助于在 KNC 上获取最优化性能；

2)发现对于 stencil 代码，基于 intrinsic 的 MAO 比基于编译器的 MAO 更高效。

本文第 2 节介绍相关工作；第 3 节深入分析 KNC 体系架构的 3 个特点；第 4 节介绍 MAO 的 3 个方法；第 5 节介绍 2 种不同数据访问算例及所使用的 MAO；第 6 节是实验环境，包括如何采集硬件性能计数器；第 7 节是结果分析；最后是总结与下一步工作。

2 相关工作

以下 3 篇关于 Intel KNC 性能优化的论文颇值得一读。Intel 公司 Pakesh Krishnaiyer 等人在 IPDPSW2013 上发表的论文[5]讨论了基于编译器的数据预取(Data Prefetching)和 Streaming Non-temporal Store (NT-store)。此文对 KNC 体系架构的特点有比较深入的分析，并对数据预取和 NT-Store 的原理进行了详细介绍，但对于 MAO 仍局限于编译器技术的利用，而且没有指出有些测试用例对于基于编译器的优化不敏感是否与数据访问模式过于复杂有关。

德国 Erlangen-Nuremberg 大学的 George Hager 和 Gerhard Wellein 课题组在 ARCS14 上发表的论文[6]将不规则数据访问的医学图像算法 FDK 通过性能工程(Performance Engineering)的方法进行优化。此文对 KNC 的 cache 结构进行了深入的分析，并介绍了如何使用 intrinsic 指令进行 cache 预取。同时这很可能第一个分析了 IMCI-512 指令集中新增的 gather 操作的实现原理，并说明了 gather 操作成为性能瓶颈的原因。但该文结论仅对数据访问不规则的应用有参考价值，没有涉及数据访问规则的应用该如何优化。

英国 Warwick 大学的 Stephen Jarvis 课题组在 IPDPS13 上发表的论文[3]以美国 Sandia 国家实验室的 miniMD 测试程序为例，分析并优化了分子动力学应用在 128bit、256bit 和 512bit 这 3 种不同宽度 SIMD 上的性能，也认为 gather/scatter 操作是不规则数据访问应用的性能瓶颈。

当然，如需更全面了解 KNC 上性能优化的技巧，可以参阅 Intel 公司 James Reinders 等人编写的著作[7]以及 Intel 公司 Rezaur Rahman 编写的著作[8]。

3 KNC 体系架构(KNC Architecture)

与 CPU 相比，KNC 在体系架构上有 3 个特点。正是这 3 个特点导致了 TPO 低效，需要我们考虑 MAO。

3.1 宽度为 512bit 的向量处理单元

• 向量处理单元(Vector Processing Unit，VPU)对 KNC 性能至关重要

宽度为 512bit 的向量可容纳 8 个 64bit 的双精度浮点数或 16 个 32bit 的单精度浮点数。KNC 峰值性能的计算公式为 *Peak Performance*=*Clock Frequency* * *Cores* * *Lanes* * 2 *FMA*，其中乘加融合(Fused Multiply and Add，FMA)是指在 1 条指令中同时执行 1 个乘法浮点操作和 1 个加法浮点操作。以上海交通大学超级计算机 π 上配置的 KNC 5110p 为例，双精度峰值性能为 1.05GHz * 60cores * 8 Lanes * 2FMA=1008GFflops，单精度则为 2016GFlops。由此可见，要在 KNC 上达到高性能，必须充分利用 VPU。

• 数据不对齐会降低 VPU 利用率

KNC 每个线程有 32 个寄存器(register)，每个长度均为 512bit。因此读入对齐的数据只需 1 次操作，而不对齐的数据往往需要 2 次操作。同样，写回时亦是如此。

• gather/scatter 是 VPU 的性能瓶颈

KNC 使用 IMCI-512 指令集，直接支持 gather/scatter 操作，但这往往成为性能瓶颈[3]。究其原因[6]是 gather 操作的实现使用了循环的方式，将分散在各处的 Cache Line(CL)逐个读入。因此数据越分散，所需读入的 CL 就越多，延时也就越大。

综上所述，宽度为 512bit 的 VPU 对数据对齐要求更苛刻，而且应当尽可能避免 gather/scatter 操作。

3.2 硬件无法隐藏 cache latency

• 高 cache latency

以上海交通大学超级计算机 π 上配置的 KNC 结点为例，如图 1 所示，Intel Sandy Bridge E5-2670 的 L1、L2、L3 cache 的 latency 分别为 1.5ns、4.6ns、81ns，而 KNC 5110p 的 L1、L2 cache 和主存的 latency 分别为 2.9ns、22.9ns 和 295ns[9]。

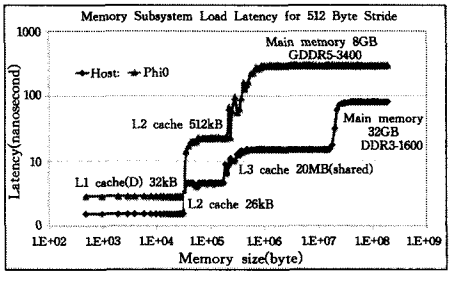

图 1 KNC 与 CPU 的 cache latency 对比[9]

• 低 ILP

根据经验公式 $Power=(Performance)^{1.73}$，使用乱序执行(Out-of-Order，OoO)的 P6 比使用顺序执行(in-order core)的 P54C 能耗提高 3 倍，但性能只提升了 1.9 倍。受限于能耗和工艺，KNC 采用了循序执行的 P54C，但因此损失了指令集并行(Instruction Level Parallelism，ILP)。

• 少硬件 cache prefetcher

受限于能耗与空间，与 Intel Xeon E5-2670 相比，KNC 不仅没有 L1 cache prefetcher，就连 L2 cache prefetcher 也打了对折，即只有 16 个 streamprefetcher，没有 stride prefetcher。

综上所述，CPU 可以通过乱序执行的 core 和硬件 cache prefetcher 等方法隐藏 cache latency，而 KNC 却不行，甚至配合编译器之后都收效有限[5]，因此程序员需要关注 cache 预取。

3.3 内存带宽有限、cache 大小有限

• 实测带宽不到理论带宽的一半

KNC 有 8 个内存控制器，每个能访问 2 条通道，每条通道的接口速度为 5.5TB/s，每个内存事务(Transaction)大小为 4B，因此 KNC 的理论带宽 8 * 2 * 5.5 * 4 = 352GB/s。如图 2 所示，使用 STREAM 实测的带宽为 140GB/s，还不到理论值的一半[9]。以 256 个线程计算，平摊下来每个线程所能占用的内存带宽仅为 0.5GB/s，远小于 CPU。

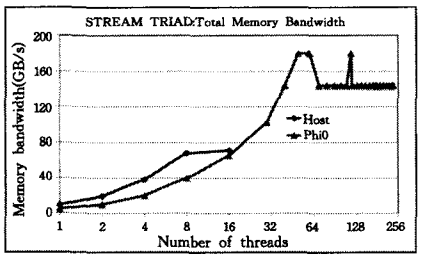

图 2 KNC 实测内存带宽[9]

• L2 cache 伪共享及远程访问很慢

KNC 5110p 每个核有 512KB 的 L2 cache，聚合起来有 30MB。但每个核不能直接访问其他核的 L2 cache 中的 CL，必须要把这条 CL 复制到自己的 L2 cache 中[10]才行。雪上加霜的是访问其他核的 L2 cache 仅比访问主存快 17%，且与双向环(Bi-direction ring)上两核之间的物理距离无关[11]。

• 没有 L3 cache

KNC 没有 L3 cache 是由于片上空间不够。根据经验公式，下一级 cache 大小应该为上级 cache 大小总和的 8 倍。例如 KNC 单核的 L1 cache 为 32kB(数据) + 32kB(指令) = 64kB，则 L2 cache 为 64kB * 8 = 512kB，以此类推，L3 cache 应为(64kB+512kB) * 8 = 4.5MB，聚合 L3 cache 将会达到 270MB。这么大的 cache，就算用上 Intel Knights Landing (KNC 的下一代产品)的 14nm 工艺，其面积也会超过 500mm²，而目前 Intel 最大的芯片面积不超过 200mm²。

综上所述，由于 KNC 内存带宽有限，cache 大小有限，所因此应当尽可能减少 cache 与主存间的通讯，需要采用 NT-Store 优化。

4 内存访问优化

针对 KNC 体系结构以上 3 个特点，我们使用以下 3 种方法进行 MAO。

4.1 数据对齐

• 前端对齐确保数组的起始地址对齐

1)堆内存分配(Heap Memory Allocation)

它共有 3 种方式。第 1 种方法最基本，可以将普通的内存分配和释放改写为对齐的 intrinsic 原语，即将 malloc() 改为_mm_malloc()，将 free()改为_mm_free()。由美国橡树林国家实验室开发的 KNC 版的 SHOC benchmark[12]就已经使用了这种方法。不过该方法由于采用了进程内单一全局锁，因此扩展性不佳。所以建议采用第 2 种方法，Intel Threading Building Blocks(TTB)提供的函数保证了线程间相互独立，因此更适合 KNC。方法是将 malloc() 改为 scalable_aligned_malloc()，将 free()改为 scalable_aligned_free()。通常其性能都会比第 1 种 intrinsic 原语更好，推荐使用这种方法。第 3 种方法是直接使用 SIMD 的 intrinsic 原语，例如使用_mm512_loadunpacklo_pd()和_mm512_loadunpackhi_pd()读入数值，并用_mm512_fmadd_pd()进行乘加融合操作(FMA)，这种方法性能最好，但编程相对困难。

2)静态分配变量(Statically Allocated Variables)

在 Linux 下使用_attribute_((aligned (64))) float temp (SIZE)、Windows 下使用_declspec_((aligned (64))) float temp(SIZE)分配静态变量。

• 后端对齐确保 2 维数组的每行第一个元素的地址对齐

在每行末尾分配一些多余的空间，以此保证 2 维数据的左端对齐，这称为 padding。

4.2 Cache 预取

硬件预取只支持 L2 cache 的 stream 模式，其余的就需要利用编译器或是手动预取。

• 编译器选项

-opt-prefetch=n，其中 n 可为 0～4。0 为关闭，4 最为激进。Intel 编译器优化级别选项为-O2 及以上就自动包含了-opt-prefetch=3。-opt-prefetch-distance=d1:d0 用来指定预取的距离(即迭代次数)，d1 为 L2 cache 的预取距离，d0 为 L1 cache 的预取距离。

• 编译器指导语句

在迭代前加入 # pragma prefetchvar:hint:distance，其中 var 是要预取的变量，hint 为 0 表示预取到 L1 cache，为 1 则表示预取到 L2 cache。

• 使用 Intrinsic 手动预取

使用 void _mm_prefetch(char const * address,int hint)函数预取 cache，其中 hint 为 0 表示预取到 L1 cache，为 1 则表示预取到 L2 cache。在手动预取时，需要设置-opt-prefetch=0 或 # pragmas noprefetch 以关闭编译器预取，避免发生不可知的冲突。

4.3 Streaming Non-temporal Store

通常更新某个数据时，如果该数据不在 cache 中，需要先把它(预)读取到 cache 中，在 cache 中更新后，再写回内存中，但这样既浪费内存带宽又浪费 cache 空间。更高效的做法是直接写到内存中，这称为 NT-Store。可以通过编译器选项-opt-streaming-stores来设置，有 3 个可选值 auto、always 和 never。always 会把所有对齐的向量写都当做 NT-Store。也可以直接在某个循环前添加编译器指导语句 # pragma vector nontemporal。当编译器知道存储地址是对齐的时，也可以使用编译器选项-opt-streaming-cache-evict=x 来指定。其中 0 表示不使用 CL 弹出，1 表示只弹出 L1 cache，2 表示只弹出 L2 cache，默认值 3 表示同时弹出 L1 和 L2。

5 数据访问模式及实例

5.1 stream 访问：Black-Scholes Formula

Black-Scholes Formula (BSF)是计算金融领域中常用的算法。它被用来求以下的偏微分(PDE)方程的近似解。

$$\frac{\partial f}{\partial t}+\frac{1}{2}\sigma^2 S^2\frac{\partial^2 f}{\partial S^2}+rS\frac{\partial f}{\partial S}-rf=0$$

$$c=S*N(d_1)-Ke^{-rT}N(d_2) \quad (1)$$

$$p=Ke^{-rT}N(-d_2)-S*N(-d_1)$$

其中：

$$d_1=\frac{\ln(S/K)+(r+\frac{\sigma^2}{2})T}{\sigma\sqrt{T}},$$

$$d_2=\frac{\ln(S/K)+(r-\frac{\sigma^2}{2})T}{\sigma\sqrt{T}}=d_1-\sigma\sqrt{T}。$$

根据表1中的代码,可以看出BSF的核心算法包含了5个stream读和1个stream写的连续数据访问。

表1 BSF核心算法的代码

```
for (int i=0;i< count;++i) {
float S=Sa[i],X=Xa[i];
float T=Ta[i],r=ra[i];
float v=va[i];
float d1=(logf(S/X)+(r+v * v *.5f) * T) / (v * sqrtf(T));
float d2=d1- v*sqrtf(T);
result[i]=S*CND(d1)-X * expf(-r * T) * CND(d2);
}
```

MAO的具体过程如表2所列。

表2 BSF的MAO过程

优化版本	优化方法
A:TPO基准	使用KNC Native模式。已经应用了AOS到SOA转换,cache blocking,并行化(使用OpenMP),向量化(使用#pragma simd)和EMU
B:A+数据对齐	设置对齐长度为4096,使用_mm_malloc()和_mm_free()函数分配和释放CallResult等5个数组,并在核心循环前添加#pragma vector aligned
C:A+数据对齐TBB	将_mm_malloc()和_mm_free()分配替换为scalable_aligned_malloc()和scalable_aligned_free(),其余与B相同
D:C+cache预取	添加编译器选项-opt-prefetch-distance=6,1
E:D+NT-Store	由于核心循环中的数组CallResult和PutResult没有读入就直接写回,因此在循环前添加#pragma vector nontemporal (CallResult,PutResult),并添加编译器选项-opt-streaming-cache-evict=0

5.2 stencil访问:Diffusion 3D

Diffusion 3D[Intel-book]用来模拟3D容器中溶质在溶液中的扩散过程,可以用式(2)中的微分(ODE)方程来表示:

$$\frac{\partial f}{\partial t}=\kappa\nabla^2 f=\kappa(\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}+\frac{\partial^2 f}{\partial z^2}) \quad (2)$$

离散化之后就得到1个3D 7点Stencil代码,如图3所示。

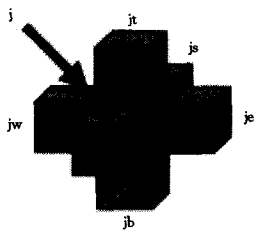

图3 Stencil示意图

表3 Stencil核心算法伪代码

```
fn[j]=cc * f[j]+
      ce * f[je]+cw * f[jw]+
      cn * f[jn]+cs * f[js]+
      ct * f[jt]+cb * f[jb]
```

表3的伪代码说明Diffusion 3D的核心算法包含了7个stencil读和1个stream写。MAO的具体过程如表4所列。

表4 Diffusion 3D的MAO过程

优化版本	优化方法
A:TPO基准	使用KNC Native模式,已经应用了AOS到SOA转换,cache blocking,并行化(使用OpenMP)和向量化(使用#pragma simd)
B:A+TBB数据对齐	设置对齐长度为64,使用scalable_aligned_malloc()和scalable_aligned_free()函数,分配和释放f和fn这个2数组,并在核心循环前添加#pragma vector aligned
C:B+cache预取	针对f[j]和f[jb]这7个值,使用编译器指导语句#pragma prefetch进行预取。L2 cache的distance为8,L1 cache的distance为1,同时添加编译器选项-opt-prefetch-distance=8,1
D:C+NT-Store	由于数组fn没有读入就直接写回,因此在核心循环前添加#pragma vector nontemporal (fn[j]),并添加编译器选项-opt-streaming-cache-evict=0
E:A+intrinsic数据对齐	使用SIMD的intrinsic原语_mm512_loadunpacklo_pd()和_mm512_loadunpackhi_pd()读入f[j]和f[jb]等,并使用_mm512_fmadd_pd()对它们进行FMA操作
F:E+intrinsic cache预取	针对f[j]到f[jb]这7个值,使用_mm_prefetch()原语进行预取。L2 cache的distance仍为8,L1 cache的distance仍为1
G:F+intrinsic NT-Store	使用_mm512_storenrngo_pd()原语存储fn数组

6 实验环境

6.1 硬件及系统软件

我们的实验在上海交通大学超级计算机π上的1个KNC结点(配置见表5)上进行。π是目前国内高校排名第1的超级算机,使用CPU+GPU+MIC的异构架构,峰值计算能力达到263TFlops,在2014年6月的TOP500中排名为251。

表5 KNC结点的硬件和系统软件配置

软硬件	配置
CPU	Intel SNB E5-2670 @2.6GHz
KNC	Intel KNC 5110p@1.05GHz
MPSS	MPSS version 3.2.3
编译器	Intel Composers XE 2013.3.163

6.2 硬件性能计数器

TPO通常看执行时间即可,但MAO却必须要把性能这个黑盒子打开,了解清楚各个部件的使用情况。这就需要借助存储于性能监视单元(Performance Monitoring Unit,PMU)中的HWPC。KNC共有10多个event,如VPU_ELEMENTS_ACTIVE。通过这些event可以推算出metric,如VPU usage。与MAO密切相关的有以下3个:

• VPU usage

可以通过计算公式*VPU_ELEMENTS_ACTIVE/VPU_INSTRUCTIONS_EXECUTED*获得。如果该值在双精度下小于8或在单精度下小于16,可能需要进行性能调优。

• L1 cache miss rate

可以通过计算公式(*DATA_READ_MISS_OR_WRITE_MISS+L1_DATA_HIT_INFLIGHT_PF1*)/ *DATA_READ_OR_WRITE*获得。如果该值大于5%,就一定要进行性能调优。

• Memory Bandwidth

测量内存带宽有2种方法:1)可以使用Core部件的events,即先计算读取带宽(*L2_DATA_READ_MISS_MEM_FILL+L2_DATA_WRITE_MISS_MEM_FILL+HWP_*

L2MISS) * 64 /*CPU_CLK_UNHALTED*，然后再计算写回带宽（*L2_VICTIM_REQ_WITH_DATA*+*SNP_HITM_L2*) * 64 / *CPU_CLK_UNHALTED*，最后再将两者相结合（*read*+*write*) * *freq*。但该方法有个缺点，即没有把 NT-Store 计算在内，因此若使用了 NT-Store，则测量出来的值会偏小。如果该值小于 80GB/s，则需要考虑进行性能调优。2）也可以使用 Uncore 部件的信息。KNC 有 8 个内存控制器，各自对应 2 条通道。每个内存控制器使用 4 个计数器（counter）来分别测量通道 0 的读、写和通道 1 的读、写。将所有 32 个计算器测量事件之和乘以 64 再除以执行时间，就能获得更加精确的内存带宽。当然某些算例中，也可以直接使用程序中对内存带宽的估算值。

6.3 Profiling 工具

如表 6 所列，我们在实验中使用了由德国 Erlangen-Nuremberg 大学开发的 Likwid[13] 和 Intel VtuneAmplifier XE 来测量这些 HWPC，当然也可以使用由美国田纳西大学开发的 PAPI[14]。

表 6 Profiling 工具

工具	优缺点
Likwid	开源免费。轻量级，方便易用。但功能不全，比如目前尚不支持 offload 模式
Intel VtuneAmplifier XE 2013	商业软件。重量级，功能完整，但操作相对复杂
PAPI	开源免费，函数库方式

7 结果分析

7.1 Black-Sholes Formula

运行时的环境参数为 *OMP_NUM_THREADS*=240、*KMP_AFINNITY*="compact，granularity=fine"。我们使用了程序内置的计数器测量 Wall Clock，使用 Likwid 测量 VPU usage 和 L1 cache miss rate，使用 Intel Amplifier XE 的 Uncore 方法测量 Memory Bandwidth，结果如表 7 所列。

表 7 BSF 的 MAO 性能结果

优化版本	Wall Clock(s) 越小越好	VPU usage 越大越好	L1 cache miss rate(%) 越小越好	Memory Bandwidth(GB/s) 越大越好
A：TPO 基准	6.15	9.5	7.7	0.80
B：A+数据对齐	6.09	9.7	6.0	0.72
C：A+TBB 数据对齐	5.96	9.7	6.0	0.50
D：C+cache 预取	5.88	9.7	4.2	0.49
E：D+NT-Store	5.69	9.7	0.5	0.48

总体来说，进行 MAO 之后，版本 E 的性能与版本 A 相比提高了 8.1%，尤其是 L1 cache miss rate 从之前的 7.7%降到了 0.5%。VPU usage 大于 8，属于可接受范围。Memory Bandwidth 特别低，是因为 BSF 属于计算密集型，输入输出的数据都比较小。考虑到这个数值偏小而容易受 profiling 工具误差的影响，这里就只做参考，不再展开讨论。使用 TBB 数据对齐的版本 C 的性能比版本 B 略有提升，考虑这个优化几乎是举手之劳，推荐使用。基于编译器选项的 cache 预取获得了较好的结果，将 L1 cache miss rate 降到了 5%的边界线以下。NT-Store 更是将该值降到了 0.5%。对比其他 2 种优化，NT-Store 的收效最为显著。综上所述，基于编译器的 MAO 对于数据连续访问的应用颇见成效。

7.2 Diffusion 3D

运行时的环境参数为 *OMP_NUM_THREADS*=180 和 *KMP_AFINNITY*="scatter"。之所以使用 180 线程而不是 240 线程，是因为 Diffusion 3D 属于内存带宽受限（Memory Bandwidth Bound）的应用，单核上跑满 4 个线程反而会导致性能下降。我们使用程序内置的计数器测量 Wall Clock，使用 Likwid 测量 VPU usage 和 L1 cache miss rate，使用程序内置的计数器测量 Memory Bandwidth，结果如表 8 所列。

表 8 Diffusion 3D 的 MAO 性能结果

优化版本	Wall Clock(s) 越小越好	VPU usage 越大越好	L1 cache miss rate(%) 越小越好	Memory Bandwidth (GB/s) 越大越好
A：TPO 基准	3.91	5.8	5.7	67.7
B：A+TBB 数据对齐	3.65	6.1	6.3	71.4
C：B+cache 预取	3.59	6.1	11.4	74.1
D：C+NT-Store	3.02	6.1	4.3	89.2
E：A+intrinsic 数据对齐	3.47	5.7	6.2	77.3
F：E+intrinsic cache 预取	3.47	5.7	11.1	74.7
G：F+intrinsic NT-Store	2.81	5.2	6.8	94.6

总体来说，进行 MAO 之后，虽然 VPU usage 和 L1 cache miss rate 变得更糟，但由于 Memory Bandwidth 利用率提高了 39.7%，因此版本 G 的性能与版本 A 相比还是提高了 39.1%。这说明 Memory Bandwidth 在 Diffusion 3D 的性能中起决定性作用。Cache 预取的优化并不成功。基于编译器的版本 B 虽然性能有所提升，但极大地增加了 L1 cache miss rate，而基于 SIMD 的版本 F 从 4 个指标来看几乎一无是处，因此后续还需要对 cache 预取的优化进行更进一步的研究。NT-Store 因为极大地增加了 Memory Bandwidth，与 BSF 中的 MAO 一样，对性能提升的贡献比其他 2 种方法都要大。基于 SIMD intrinsic 的版本的性能比对应基于编译器的版本要高出约 5%。综上所述，基于编译器的 MAO 对于单元跨步访问的应用有一定效果，为了追求最优化性能，可以使用基于 intrinsic 的 MAO，但编程会相对困难。

结束语 MAO 主要包括数据对齐、cache 预取和 NT-Store 这 3 种优化方法。我们将 MAO 应用到 BSF 和 Diffusion 3D 这 2 个已经过 TPO 的程序上，随后发现性能仍然分别提高了 8.1%和 39.1%。基于以上的工作，我们对 KNC 上的 MAO 有如下建议：

1）MAO 与 TPO 的优化效果可以叠加，因此 TPO+MAO 有助于在 KNC 上获取 Ninja Performance；

2）对于 stream 访问的应用，基于编译器的 MAO 就能达到不错的效果；

3）对于 stencil 代码，为了追求最优化性能，可以使用 intrinsic，但要考虑所收获的性能提升与所付出的编程努力之间的平衡；

4）尽量使用 TBB 的数据对齐；

5）如有可能，一定要使用 NT-Store 优化；

6）cache 预取的编程比其他 2 种方法更难，不过在数据连续访问的应用中通常能获得比较理想的效果。

下一步的工作主要包括以下两方面：

1)从 KNC cache 的体系结构[15]入手,深入研究 cache 预取的机制;

2)对 KNC 上的 MAO 进行性能建模[16]。

文中涉及的 BSF 和 Diffusion 3D 的 MAO 源代码及原始测试数据都已放在 Github[17]上,以供读者参考。

参 考 文 献

[1] Satish N, Kim C, Chhugani J, et al. Can traditional programming bridge the Ninja performance gap for parallel computing applications? [C]// 2012 39th Annual International Symposium on Computer Architecture (ISCA). 2012:440-451

[2] Xue W, Yang C, Fu H, et al. Enabling and Scaling a Global Shallow-Water Atmospheric Model on Tianhe-2[C]// Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. 2014

[3] Pennycook S J, Hughes C J, Smelyanskiy M, et al. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors[C]// Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. 2013:1085-1097

[4] Heinecke A, Vaidyanathan K, Smelyanskiy M, et al. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor[C]// Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. 2013:126-137

[5] Krishnaiyer R, Kultursay E, Chawla P, et al. Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor[C]// Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum. 2013:1575-1586

[6] Hofmann J, Treibig J, Hager G, et al. Performance Engineering for a Medical Imaging Application on the Intel Xeon Phi Accelerator[C]// 2014 27th International Conference on Presented at the Architecture of Computing Systems (ARCS). 2014:1-8

[7] Jeffers J, Reinders J. Intel Xeon Phi Coprocessor High Performance Programming (1st edition)[M]. Morgan Kaufmann Publishers Inc, 2013

[8] Rahman R. Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers[M]// Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers (1st edition). 2013

[9] Saini S, Jin H, Jespersen D, et al. An early performance evaluation of many integrated core architecture based SGI rackable computing system[C]// Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 2013

[10] Hofmann J. Performance Evaluation of the Intel ManyIntegrated Core Architecture for 3D Image Reconstruction in Computed Tomography (Master Thesis)[M]. Friedrich-Alexander-University Erlangen-Nuremberg, 2010

[11] Fang J, Sips H, Zhang L, et al. Test-driving Intel Xeon Phi[C]// Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. New York, USA, 2014:137-148

[12] SHOC-MIC benchmark[OL]. https://github.com/vetter/shoc-mic

[13] Likwid[OL]. https://code.google.com/p/likwid/

[14] PAPI[OL]. http://icl.cs.utk.edu/papi/

[15] Ramos S, Hoefler T. Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi[C]// Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing. New York, USA, 2013:97

[16] Hoefler T, Gropp W, Kramer W, et al. Performance modeling for systematic performance tuning[C]// 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2011:1-12

(上接第 21 页)

[70] Valafar M, Rejaie R, Willinger W. Beyond friendship graphs: a study of user interactions in Flickr[C]// Proceedings of the 2nd ACM workshop on Online social networks. ACM, 2009:25-30

[71] Yang J, Leskovec J. Modeling information diffusion in implicit networks[C]// 2010 IEEE 10th International Conference on Data Mining (ICDM). IEEE, 2010:599-608

[72] Bakshy E, Rosenn I, Marlow C, et al. The role of social networks in information diffusion[C]// Proceedings of the 21st international conference on World Wide Web. ACM, 2012:519-528

[73] Tang S, Yuan J, Mao X, et al. Relationship classification in large scale online social networks and its impact on information propagation[C]// 2011 Proceedings IEEE INFOCOM. IEEE, 2011:2291-2299

[74] Hansen D L, Johnson C. Veiled viral marketing: disseminating information on stigmatized illnesses via social networking sites [C]// Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium. ACM, 2012:247-254

[75] Alahakoon T, Tripathi R, Kourtellis N, et al. K-path centrality: A new centrality measure in social networks[C]// Proceedings of the 4th Workshop on Social Network Systems. ACM, 2011:1-6

[76] Burke M, Kraut R, Marlow C. Social capital on Facebook: Differentiating uses and users[C]// Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, 2011:571-580

[77] 张彦超,刘云,张海峰,等. 基于在线社交网络的信息传播模型[J]. 物理学报, 2011, 60(5):60-66
Zhang Y C, Liu Yun, Zhang H F, et al. Information diffusion Model based on Online Social Network[J]. Acta Phys. Sin, 2011, 60(5):60-66

[78] Mohaisen A, Yun A, Kim Y. Measuring the mixing time of social graphs[C]// Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement. ACM, 2010:383-389

[79] Squicciarini A C, Shehab M, Paci F. Collective privacy management in social networks[C]// Proceedings of the 18th International Conference on World Wide Web. ACM, 2009:521-530

[80] 徐婕,康慕宁,董谷音. 基于社交网络的实时搜索引擎的排序算法研究[J]. 科学技术与工程, 2011, 11(28):6879-6882
Xu Jie, Kang M N, Dong G Y. Sequencing Algorithm Based on the Social Network Real-time Search Engine[J]. Science Technology and Engineering, 2011, 11(28):6879-6882