# 基于 Intel MIC 架构的 3D 有限差分算法优化

郝 鑫 郭绍忠
(数学工程与先进计算国家重点实验室 郑州 450002)

**摘 要** 有限差分算法是一种基于偏微分方程的数值离散方法,被广泛应用于弹性波传播问题的数值模拟中。该算法访存跨度大、计算密度高、CPU 利用率低,这在实际应用中成为了性能瓶颈。针对上述问题,在详析 3D 有限差分算法(3DFD)的基础上,基于 Intel MIC 架构,采用三步递进法对其进行优化:首先,通过分支消除、循环展开、不变量外提等基本优化法削减计算强度并为向量化扫除障碍;然后,通过分析数据依赖及循环分块,使用向量指令集改写核心算法等并行优化法,充分利用 MIC 协处理器多线程、长向量的机制;最后,在异构众核平台(CPU+MIC:Many Integrated Cores)下通过数据传输最小化、负载均衡等异构协同优化法实现 CPU 和 MIC 的并行计算。实验验证,与原有算法相比,优化后的算法在异构平台上获得了 50～120 倍的加速。

**关键词** 有限差分算法,MIC 架构,向量化,异构协同,并行计算
**中图法分类号** TP301 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.05.005

## Optimization of 3D Finite Difference Algorithm on Intel MIC

HAO Xin GUO Shao-zhong
(State Key Laboratory of Mathematics Engineering and Advanced Computing, Zhengzhou 450002, China)

**Abstract** Finite difference algorithm is a numerical discrete method based on the partial differential equation which is widely applied in elastic wave propagation simulation. Because of the high computation density, long distance memory access pattern and low CPU utilization, it becomes the performance bottleneck in practical applications. Aiming at solving above problems, this paper deliberated the key points of 3D finite difference(3DFD) algorithm and then proposed the three-step progressive method to optimize 3DFD algorithm based on Intel MIC. Firstly, the basic optimization methods, such as branch elimination, loop unroll, and invariant extraction, were proposed to reduce calculation strength and remove the obstacle of SIMD(Single Instruction Multiple Data). Secondly, by leveraging the parallel optimization methods such as data dependence analysis, loop tiling, and intrinsic SIMD instructions, it took full advantage of the mechanism of MIC coprocessor with multithreads and long vector. At last, the heterogeneous cooperative optimization methods, such as data transformation minimization and load balancing, were applied to the platform of CPU+MIC (Many Integrated Cores) which parallelizes the algorithm execution in both CPU and MIC. Experimental results show that the optimized 3DFD algorithm gains 50～120 speedup compared with original algorithm.

**Keywords** Finite difference algorithm, MIC architecture, SIMD, Heterogeneous cooperation, Parallel computation

## 1 引言

近年来,随着大规模并行体系结构的发展,异构众核架构在超级计算领域获得了广泛应用,其中以 CPU(Intel Xeon)配置 MIC(Intel Xeon Phi)协处理器的超级计算机在 HPC Top500[1]中占据绝对优势:在 2015 年 11 月公布的榜单中,排名第一的天河 2 号与排名第二的 Titan 均采用了该体系架构。一方面,Intel Xeon Phi 具有数十个精简的 x86 核心以提供更多的硬件线程,采用更宽的向量单元以满足高度并行化应用的需要[2],与 Intel Xeon 相比,其具有更快的浮点峰值速度;另一方面,Intel Xeon Phi 与 Intel Xeon 使用了相同的编程语言和硬件模型,运行于 Intel Xeon 上的程序可以直接运行于 Intel Xeon Phi,但是大多数科学计算领域中的算法必须经过有效的优化才能最大程度地发挥 Intel Xeon Phi 的性能。

作为科学计算中一种重要的求解偏微分方程的数值算法,3DFD 具有通用性强、易于程序实现等优点。但是 3DFD 存在访存跨度大、计算密度高、算法性能差等问题,因此在模拟弹性波传播时受限于模拟区域以及模拟时域的规模[3-4]。许多研究围绕如何利用异构众核体系结构的并行特性提升算法性能展开:L. M. Itu 等[5]分析了 2D 有限差分算法在 GPU 异构系统上的性能瓶颈,通过对全局内存级联访问以及精心安排访存指令以减少对全局内存的访问等优化手段,使算法在 GPU 上运行时最高获得了 12.64 倍的加速;Wang 等[6]针对 GPU 不同存储层次的带宽差异,通过合理安排数据访问

到稿日期:2016-04-30 返修日期:2016-08-27
郝 鑫(1989-),男,硕士生,主要研究方向为并行优化、高性能计算,E-mail:elvis.hao@qq.com;郭绍忠(1964-),女,硕士,教授,主要研究方向为计算机体系结构、高性能计算。

模式、调整线程粒度,尽可能减少对低带宽存储层的访问,同时采用预取机制减少了数据访存延迟,使 3D 有限差分算法获得了 34.3 倍的加速;Mario Hernandez 等[7]分析了 3 种使用 3D 有限差分算法的科学应用在 Intel MIC 架构上的运行情况,从扩展性、亲缘性、分块规模和网格形状等方面考察了影响算法性能的限制因素,为算法优化提供了指导,但是并没有从向量化以及 CPU+MIC 异构协同等方面对算法进行优化。

因此,本文基于 Intel MIC 架构,使用三步递进法对应用于模拟各向同性弹性波传播[8]的 3D 有限差分算法进行优化,所用方法包括:分支判断消出、循环展开、循环不变量外提、循环分块、AVX-512 向量指令改写循环核心以及 CPU+MIC 协同计算,从算法和体系结构两个层面进行了并行优化。实验表明,与未经优化的算法相比,优化后算法平均获得了 90 倍的加速。根据应用实际情况设量最大误差阈值(1e−4),经测试,算法优化前后结果误差小于阈值,从而确保了优化后算法的正确性。

## 2　3D 有限差分算法

在各向同性介质中,质点间由相互作用的弹性力而处于平衡位置。如果质点受到扰动或外力作用,质点就会偏离平衡位置,即发生应变。此时发生应变的质点在弹性力的作用下产生振动,从而引起周围质点的应变和振动,进而形成弹性波[10-12]。三维各向同性弹性波方程如下[13]:

$$\frac{\partial p^2}{\partial t^2}=v^2\left(\frac{\partial p^2}{\partial x^2}+\frac{\partial p^2}{\partial y^2}+\frac{\partial p^2}{\partial z^2}\right) \tag{1}$$

其中,$p$ 是时间变量 $t$ 的压力场函数,$v$ 是压力的速度场函数,与 $t$ 无关。由泰勒公式得:

$$p_{t+1}=p_t+\Delta t\frac{\mathrm{d}p_t}{\mathrm{d}t}+\frac{\Delta t^2}{2!}\frac{\mathrm{d}^2p_t}{\mathrm{d}t^2}+\frac{\Delta t^3}{3!}\frac{\mathrm{d}^3p_t}{\mathrm{d}t^3}+R_4 \tag{2}$$

$$p_{t-1}=p_t-\Delta t\frac{\mathrm{d}p_t}{\mathrm{d}t}+\frac{\Delta t^2}{2!}\frac{\mathrm{d}^2p_t}{\mathrm{d}t^2}-\frac{\Delta t^3}{3!}\frac{\mathrm{d}^3p_t}{\mathrm{d}t^3}+R_4 \tag{3}$$

使用有限差分法,由式(1)−式(3)可得三维各向同性弹性波差分方程:

$$p_{t+1}[x,y,z]=2p_t[x,y,z]-p_{t-1}[x,y,z]+v^2\Delta t^2(FD_x+FD_y+FD_z) \tag{4}$$

其中,$p_{t+1}$,$p_t$,$p_{t-1}$分别是 $t+1$,$t$,$t-1$ 时刻的压力场函数;$FD_x$,$FD_y$,$FD_z$ 分别是 $x$,$y$,$z$ 方向导数的有限差分格式:

$$FD_x=\frac{1}{\Delta x^2}\left(c_0p_t[x,y,z]+\sum_{i=1}^{n}c_i(p_t[x+i,y,z]+p_t[x-i,y,z])\right) \tag{5}$$

$$FD_y=\frac{1}{\Delta y^2}\left(c_0p_t[x,y,z]+\sum_{i=1}^{n}c_i(p_t[x,y+i,z]+p_t[x,y-i,z])\right) \tag{6}$$

$$FD_z=\frac{1}{\Delta z^2}\left(c_0p_t[x,y,z]+\sum_{i=1}^{n}c_i(p_t[x,y,z+i]+p_t[x,y,z-i])\right) \tag{7}$$

其中,$c_i$ 是有限差分格式的系数,在算法实现时 $n$ 取 8。

式(4)表示已知空间点$(x,y,z)$处 $t-1$ 和 $t$ 时刻的压力场值 $p_{t-1}[x,y,z]$和 $p_t[x,y,z]$,则可计算 $t+1$ 时刻$(x,y,z)$处的压力场值 $p_{t+1}[x,y,z]$。

在算法实现时,使用一维数组存储三维空间上的压力场和速度场。由于 $p_{t-1}$仅在当次运算中被使用,并且只使用一次,因此在计算出的 $p_{t+1}$后可以直接覆盖 $p_{t-1}$,故只需要两个数组来存储 $t+1$,$t$,$t-1$ 时刻的压力场数值。另外,通过增加数组存储空间来模拟弹性波传播的边界,边界区域不在计算范围内(算法 1 第 4−5 行)。对 $FD_x$,$FD_y$,$FD_z$ 的计算是算法最核心的部分(算法 1 第 6−9 行)。

**算法 1**　三维有限差分核心算法 3dfd_kenel

输入:数组 prev;数组 next;数组 vel;空间维度 n1,n2,n3;有限差分格式的系数 coeff,n 取 8

输出:数组 next

```
1. for (z=0;z<n3;z++) do
2.   for (y=0;y<n2;y++) do
3.     for (x=0;x<n1;x++) do
4.       if (x>=8 && x<(n1-8) && y>=8 && y<(n2-8) && z>=8 && z<(n3-8))
5.         Initialize FDx,FDy,FDz with coeff[0] * prev[x+y * n1+z * n1 * n2]
6.         for (i=1; i<=8; i++) do
7.           Calculate ith component and accumulate to FDx, FDy, FDz;
8.         end do
9.         next=2prev-next+vel(FDx+FDy+FDz);
10.      end if
11     end for
12.  end for
13. end for
```

算法 2 表示在某一时间段内对弹性波传播进行有限差分数值模拟。在第一次循环调用 3dfd_kernel 后,$t=1$ 时刻的压力场数值存储在 *next* 中,$t=0$ 时刻的压力场数值存储在 *prev* 中,交换两个数组指针,如算法 2 第 3 行所示;在第二次循环调用 3dfd_kernel 后,得到 $t=2$ 时刻的压力场数值,并将它存储在 *next* 中,$t=1$ 时刻的压力场数值存储在 *prev* 中。依次进行循环迭代,最终得到 *period* 时间段的模拟数值。

**算法 2**　时间段 period 有限差分数值模拟算法

输入:数组 prev;数组 next;数组 vel;空间维度 n1,n2,n3;有限差分格式的系数 coeff;模拟时间段 period

输出:period 后压力场模拟结果:next(period 为偶数)或 prev(period 为奇数)

```
1. for (t=0;t<period;t++)
2.   3dfd_kernel (next,prev,vel,coeff,n1,n2,n3);
3.   swap next and prev;
4. end for
```

综上,3DFD 算法由 5 层循环组成:最外层循环由于存在数据依赖(算法 2 第 3 行)而无法并行执行,因此优化工作主要针对算法 1 展开;内四层循环虽然没有数据依赖,可以进行循环间并行化与循环内向量化优化,但是在计算 $FD_y$ 和 $FD_z$ 时需要跨越访问内存单元,导致缓存脱靶,从而造成计算单元因等待数据传输而空闲,严重影响算法效率[14]。针对以上问题,本文提出三步递进优化法对 3DFD 算法进行优化。

## 3　三步递进优化法

在第 2 节中分析了 3DFD 算法,该算法具有可并行性,并

且适合进行向量化，因此首先对3DFD算法进行基本优化。提取循环不变量以削减计算强度，消除循环分支以利于向量化。然后从算法层面进行并行优化：1)循环分块后使用OpenMP并行模型，使多线程执行算法时具有更好的空间局部性；2)使用内建向量指令并插入数据预取指令，在充分利用Intel Xeon Phi的512位向量处理能力的同时，有效缩短计算单元等待数据传输的时间。最后从体系结构层面上进行并行优化：对数据传输和负载均进行优化，通过数据划分，使算法在CPU和MIC上实现并行。基于此，优化工作从以下3个方面展开。

### 3.1 基本优化法

在算法1中，可以通过变换循环变量初值及退出条件来消除分支判断，因为处理器在处理条件分支时，分支预测逻辑单元在计算结果可用之前就会采用基于统计的方法对该计算结果进行预测，一旦分支预测失误，指令流水线将重新回到该分支位置，产生流水线气泡，造成时钟周期的浪费。此外，分支预测失败后，编译器也就不能继续进行循环展开或SIMD向量化等后续优化，影响了程序性能[9]。

算法1使用了一个8次循环计算 $FD_x$，$FD_y$ 和 $FD_z$，将此循环展开，以利于向量化。由于算法实现时采用一维数组，在计算每一点的压力场 $p$ 时需要计算三维坐标对应的数组下标，因此每次循环都要进行冗余的计算，例如与 $p[x_o,y_o,z_o]$ 对应的 $next[z_o*n2*n1+y_o*n1+x_o]$ 中就包含循环无关子表达式 $n2*n1$，将这些子表达式在循环开始前计算出来，可以有效地减小循环计算强度，如图1所示。经过基础优化法优化后的算法取得了1.37倍的加速。

```
n2n1=n2*n1;
y维度不变量(y1,y2,…,y8)=(n1,2*n1,…,8*n1);
z维度不变量(z1,z2,…,z8)=(n2n1,2*n2n1,…,8*n2n1);
for (z=8; z<n3-8; z++) do
  for (y=8; y<n2-8; y++) do
    for (x=8; x<n1-8; x++) do
      p=z*n2n1+y*n1+x;
      …
      FDx+=coeff[1…8]*(prev[p+1…8]+prev[p-1…8]);
      FDy+=coeff[1…8]*(prev[p+y1…y8]+prev[p-y1…
            y8]);
      FDz+=coeff[1…8]*(prev[p+z1…z8]+prev[p-z1…
            z8]);
      …
    end for
  end for
end for
```

图1 基本优化法的伪代码

### 3.2 并行优化法

在3.1节基本优化的基础上，采用OpenMP并行模型，通过在核心循环前加入编译指示，实现线程级并行。作为基于共享内存的并行编程模型[15]，OpenMP程序会被编译成主线程和若干个并发子线程，并把并发部分编译成一个由主线程和子线程并发调用的单独函数，在程序启动后，主线程立即运行，当遇到并行区域时，所有线程根据编译指示并行执行。在两个并行区域之间，除了主线程执行代码外，其他任何线程将不执行，即fork-join模式[16]。

#### 3.2.1 循环分块

Intel MIC架构提供了更多的核和硬件线程，例如Intel Xeon Phi 5110P提供60个核以及每核4个硬件线程共240个线程的配置，因此需要算法具有很高的并行性。使用分块技术可以减小并行粒度，进而增加可并行的线程数；由于MIC架构的主频大约只有CPU主频的一半，因此除了需要足够的并行线程，还需要使每个线程的计算任务尽量简单[17]。

采用循环分块的方法对算法的 $x,y,z$ 3层循环进行分块，使算法核心中的三重循环局部于数据块内[18]，块大小分别为 $XBF,YBF,ZBF$；同时使用OpenMP编译指示collapse子句压缩分块后的循环，划分出 $\frac{n2*n1*n3}{XBF*YBF*ZBF}$ 个并行任务以供所有线程调度计算，经测试，采用动态(dynamic)调度策略时效果最好，如图2所示。

```
#pragma omp parallel for schedule(dynamic) collapse(3)
for (xx=8; xx<n1-8; xx+=XBF)
for (zz=8; zz<n3-8; zz+=ZBF)
for (yy=8; yy<n2-8; yy+=YBF) {
  zmax=MIN(n3-8, zz+ZBF);
  ymax=MIN(n2-8, yy+YBF);
  xmax=MIN(n1-8, xx+XBF);
  for (z=zz; z<zmax; z++)
  for (y=yy; y<ymax; y++)
  for(x=xx; x<xmax; x++) {
    …
  }
}
```

图2 循环分块示意图

分块值 $XBF,YBF,ZBF$ 决定着每个线程的访存跨度以及计算规模，确定其大小对程序性能有重要的影响。本文在实验中实现了分块探索工具，并使用枚举法找出最优分块值。由式(4)—式(7)可知：3DFD算法中 $x,y,z$ 的计算顺序可以相互转换，对于任意输入规模 $(n1,n2,n3)$ 下坐标系O-XYZ中的点 $(x_o,y_o,z_o)$，其所对应的一维数组下标为 $z_o*n2*n1+y_o*n1+x_o$；将输入按升序排列为 $(n1',n2',n3')$，其中 $n1'\leqslant n2'\leqslant n3'$，原坐标系中的点 $(x_o,y_o,z_o)$ 在新坐标系O-X'Y'Z'中则变为 $(x_o',y_o',z_o')$，数组下标为 $z_o'*n2*n1+y_o'*n1+x_o'$，转换规则如表1所列。因此可以将输入规模升序排列后再探索最优分块，从而大幅提高了分块探索工具的使用效率。使用循环分块优化后的并行算法获得了35.67倍的加速。

表1 坐标转换规则表

| | $x_o'$ | $y_o'$ | $z_o'$ |
|---|---|---|---|
| $n1\leqslant n2\leqslant n3$ | $x_o$ | $y_o$ | $z_o$ |
| $n1\leqslant n3\leqslant n2$ | $x_o$ | $z_o$ | $y_o$ |
| $n2\leqslant n1\leqslant n3$ | $y_o$ | $x_o$ | $z_o$ |
| $n2\leqslant n3\leqslant n1$ | $y_o$ | $z_o$ | $x_o$ |
| $n3\leqslant n1\leqslant n2$ | $z_o$ | $x_o$ | $y_o$ |
| $n3\leqslant n2\leqslant n1$ | $z_o$ | $y_o$ | $x_o$ |

#### 3.2.2 向量指令

在程序优化过程中，编译器自动向量化后的代码并没有

取得较好的优化效果。为充分利用Intel MIC架构多达512位的向量长度,采用Intel内建向量指令改写算法核心循环。具体实现分为3步:1)将$X$维循环for($x=xx;x<xmax;x++$)改写为for($x=xx;x<xmax;x+=16$),以16个单精度浮点数为一个向量,每循环一次得到16个计算结果,循环量降为原来的1/16(算法3第1行);2)将16个连续数据prev[$x+i$]…prev[$x+i+15$]读取至512位向量寄存器xVec中以计算$FD_x$的第$i$个分量(算法3第8—12行),并将结果累加至向量寄存器sumVec中,将16个连续数据prev[$x+i*n1$]…prev[$x+i*n1+15$]读取至512位向量寄存器yVec中,计算$FD_y$的第$i$个分量(算法3第13—21行)并将结果累加至向量寄存器sumVec中;同理将16个连续数据prev[$x+i*n1n2$]…prev[$x+i*n1n2+15$]读取至512位向量寄存器zVec中,计算$FD_z$的第$i$个分量(算法3第22—30行)并将结果累加至向量寄存器sumVec中;3)使用向量融合加/减指令计算next,并将计算结果写回next[$x$]…next[$x+15$](算法3第32—38行)。

在向量指令改写核心算法的过程中,对于$FD_x$的计算采用了向量右移指令_mm512_alignr_epi32,以减少向量读取指令_mm512_load_ps的使用,如图3所示,将prev[$x-16$]…prev[$x-1$]读入向量寄存器backVec,将prev[$x$]…prev[$x+15$]读入向量寄存器currentVec,将prev[$x+16$]…prev[$x+31$]读入向量寄存器afterVec,再通过向量右移指令得到计算$FD_x$所需要的16个值:prev[$x+i$]…prev[$x+i+15$],$i=\pm1,\pm2,\cdots,\pm8$,仅用3次向量读取指令与16次向量移位操作便可完成计算,共省去13次向量读取操作,从而进一步优化了算法的访存,获得了53.64倍的加速。

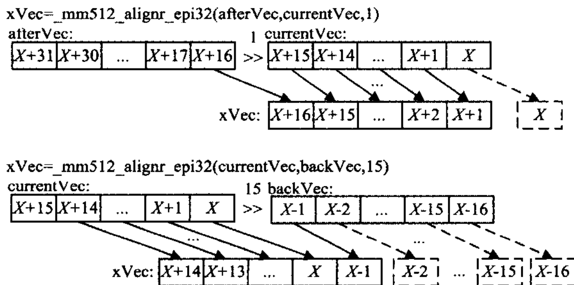


图3　向量移位法计算$FD_x$的原理图

此外,由于程序需要对$Y$和$Z$维度的数据进行多次访存,存在访存不连续现象,因此加入缓存预取指令_mm_prefetch以降低由于缓存脱靶所产生的延迟:在每次读取一个维度的向量数据之后,调用_mm_prefetch将下一次循环所需要的同维度向量数据预取至L0 Cache,同时将距离DIST的向量数据预取至L1 Cache(算法3第15—16,19—20,24—25,28—29行),其中DIST需要通过测试获得,但必须是16的整数倍,因为MIC的cache是以64字节对齐的,16个单精度浮点数刚好占用64字节。实验证实,相较于向量指令优化后的并行算法,加入预取指令后算法又取得了1.8倍的加速。

**算法3**　内建向量指令实现3dfd_kernel最内层$X$维循环

```
1. for (x=xx; x<xmax; x+=16)
2.    currentVec=_mm512_load_ps(&prev[x]);
3.    sumVec=_mm512_mul_ps(currentVec, coeffVec[0]);
4.    backVec=_mm512_load_ps(&prev[x-16]);
5.    afterVec=_mm512_load_ps(&prev[x+16]);
6.    twoVec=_mm512_set1_ps(2.0f);
7.    for (i=1; i<=8; i++)
8.      /* Calculate ith component of FDx */
9.      xVex=_mm512_alignr_epi32(currentVec,backVec,16-i);
10.     sumVec=_mm512_fmadd_ps(xVex, coeffVec[i], sumVec);
11.     xVex=_mm512_alignr_epi32(afterVec, currentVec, i);
12.     sumVec=_mm512_fmadd_ps(xVex, coeffVec[i], sumVec);
13.     /* Calculate ith component of FDy */
14.     yVec=_mm512_load_ps(&prev[x+i*n1]);
15.     _mm_prefetch((char const*)(&prev[x+i*n1]+16), 0);
16.     _mm_prefetch((char const*)(&prev[x+i*n1]+32), 1);
17.     sumVec=_mm512_fmadd_ps(yVec, coeffVec[i], sumVec);
18.     yVec=_mm512_load_ps(&prev[x-i*n1]);
19.     _mm_prefetch((char const*)(&prev[x-i*n1]+16), 0);
20.     _mm_prefetch((char const*)(&prev[x-i*n1]+32), 1);
21.     sumVec=_mm512_fmadd_ps(yVec, coeffVec[i], sumVec);
22.     /* Calculate ith component of FDz */
23.     zVec=_mm512_load_ps(&prev[x+i*n1n2]);
24.     _mm_prefetch((char const*)(&prev[x+i*n1n2]+16),
        0);
25.     _mm_prefetch((char const*)(&prev[x+i*n1n2]+80),
        1);
26.     sumVec=_mm512_fmadd_ps(zVec, coeffVec[i], sumVec);
27.     zVec=_mm512_load_ps(&prev[x-i*n1n2]);
28.     _mm_prefetch((char const*)(&prev[x-i*n1n2]+16),
        0);
29.     _mm_prefetch((char const*)(&prev[x-i*n1n2]+80),
        1);
30.     sumVec=_mm512_fmadd_ps(zVec, coeffVec[i], sumVec);
31.   end for
32.   /* Pt+1[x,y,z]=2Pt[x,y,z]-Pt-1[x,y,z]+v2△t2(FDx+
      FDy+FDz) */
33.   /* nextVec=2*currentVec-nextVec+velVec*sumVec*/
34.   nextVec=_mm512_load_ps(&next[x])
35.   tmpVec=_mm512_fmsub_ps(twoVec, currentVec, nextVec)
36.   velVec=_mm512_load_ps(&vel[x]);
37.   nextVec=_mm512_fmadd_ps(sumVec, velVec, tmpVec)
38.   _mm512_store_ps(&next[x], nextVec)
39. end for
```

### 3.3　异构协同优化

在完成基本优化和并行优化后,通过CPU与MIC协同并行以进一步提高算法效率。使用offload分载模式可以将部分计算从CPU端分载至MIC端,由于CPU和MIC不共享物理内存,对3DFD算法进行异构协同优化时分载至MIC端的计算所依赖的数据也要从CPU的内存空间传输至MIC的内存空间,因此会产生额外的通信开销;另一方面,由于算法并行运行在CPU和MIC上,硬件结构的差异势必造成算法在两端运行时性能上的差异。本节将从数据传输和负载均衡两方面入手,通过数据依赖分析、分端性能测试等优化手段,得使程序获得最佳的整体执行效率。

3.3.1 数据传输优化

CPU与MIC之间通过PCI-E通信,PCI-E传输相对共享存储器访存较慢,实验实测传输速度为6.53GB/s,因此需要尽量减少CPU与MIC之间的数据传输。分析3DFD算法可知,共有3个数组需要进行数据传输:next,prev和vel。3个数组的传输量均与分载给MIC的负载有关:设在$X,Y,Z$维度下的输入规模为$N1$,$N2$和$N3$,CPU与MIC的负载比例为$L_c:L_m$,数组vel作为只读数组,仅需首次由CPU传向MIC并由MIC保留在其内存中,传输数据量为$\frac{L_m*N1*N2*N3}{L_m+L_c}*sizeof(float)$字节,之后的迭代不再需要将vel传给MIC,可以使用分载模式中的nocopy技术实现;数组next和prev均为读写数组,在时间步迭代中依次轮流担任写目的地数组,因此除了需要首次将数据传输至MIC外,在迭代过程中还要相互交换依赖部分。首次传输数组next和prev的数据量均为$(\frac{N1*N2*N3*L_m}{L_m+L_c}+N1*N2*8)*sizeof(float)$字节;每次迭代结束时,CPU端与MIC端交换依赖数据:CPU端传输$N1*N2*8*sizeof(float)$字节的数据至MIC,同时MIC端传输$N1*N2*8*sizeof(float)$字节的数据至CPU。图4以数组next为例展示了该过程,将数组next分给CPU和MIC进行并行计算,阴影部分数据不在计算范围,在当次迭代结束后由另一端传输过来:数组$next_{mic}$最后的$N1*N2*8$个数据由CPU端对应部分的计算结果传入,$next_{cpu}$最前的$N1*N2*8$个数据由MIC端对应部分的计算结果传入。数据交换结束后即可开始下次迭代,直至程序结束。使用offload传输数据时,仅需指定待传数据的起始位置和传输长度以及数据流方向in或out,即可完成传输。

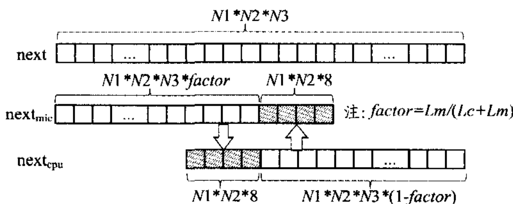


图4 迭代间数据传输示意图

虽然该数据传输优化中对于next和prev分别存储了$N1*N2*8$个冗余数据,但是大大减少了每次迭代结束后需要传输的数据量,为基于PCI-E进行通信的异构程序传输优化提供了借鉴思路。

3.3.2 负载均衡优化

由于CPU和MIC在硬件结构上的差异,经过上述优化技术后两者计算效率是不同的,因此需要按照CPU和MIC的计算能力合理地分配负载。在负载均衡优化中,需首先根据输入规模分别在CPU端和MIC端运行优化后的程序,使用分块探索工具得到最优的分块大小;然后使用在CPU和MIC上测得的程序吞吐量(单位时间内处理浮点数的数据量)$T_{cpu}$和$T_{mic}$计算得到CPU和MIC的负载$L_{cpu}$和$L_{mic}$,如(8)式所示:

$$L_{cpu}:L_{mic}=T_{cpu}:T_{mic} \tag{8}$$

最后根据负载比例,将数据从CPU端分载至MIC端,使用offload分载模式中的异步计算模式在CPU和MIC端并行计算,以此实现负载均衡。如图5所示,数据从CPU分载至MIC后,CPU和MIC分别执行3DFD_kernel,除了最后一次时间步迭代结束时不需要交换边界值,其余迭代结束时均需要交换依赖边界,最终计算结果由MIC端传回CPU端,由CPU完成结果验证,保证了异构协同计算的正确性。经数据传输、负载均衡优化后,算法获得了大约1.2倍的加速。

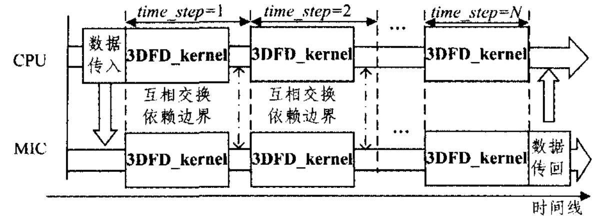


图5 3DFD算法异构协同计算原理图

## 4 实验结果及分析

### 4.1 实验环境

实验使用一颗Intel CPU与一颗Intel MIC组成计算节点,具体配置如下。

1)CPU计算节点:Intel Xeon E5-2680 V3(2.5GHz)处理器,128GB DDR4内存,12个核心,每核2个硬件线程,且每个核配置一个256位宽的向量处理单元,单/双精度浮点峰值分别是480GFLOPS和240GFLOPS。

2)MIC计算节点:Intel Xeon Phi 5110P(1.053GHz)协处理器,8GB GDDR5内存,60个核心,每核4个硬件线程,且每个核配置一个512位宽的向量处理单元,核间采用片上高速双向方式互连。每核带宽峰值8.6GB/s,理论总峰值带宽504GB/s,实际带宽200GB/s~250GB/s。单/双精度浮点峰值分别是2.021TFLOPS和1.010TFLOPS。

CPU与MIC通过PCI-E连接,支持对Xeon Phi的Offload,Symmetric,Native模式的调用。实验使用CentOS 6.4操作系统,Intel ICC编译器,Intel(R) VTune(TM) Amplifier XE 2015性能分析工具,OpenMP 4.0并行库以及MPSS(Manycore Platform Software Stack),针对协处理器的特定运行软件,包括开发工具所用到的中间件接口、通信和控制相关的设备驱动程序、协处理器管理工具、协处理器的本地Linux操作系统。

### 4.2 结果与分析

本文通过测量吞吐量来衡量算法性能,对比原有算法与三步递进法优化后的算法的计算结果是否在误差范围内(0.0001f)来保证优化的正确性。原有算法和优化后算法均使用编译器最高级别优化编译选项-O3进行编译。实验选取了3组输入规模:528*528*1056,800*800*800和400*1100*2464,分别进行4,8,12,16次时间步迭代。需要说明的是,3组输入规模依次增大,分别模拟等长宽长方体、立方体、一般长方体3种空间结构,直到完全占用MIC内存。进一步增大输入规模时则需使用计算机集群完成计算:根据集群中节点的计算能力划分迭代空间,采用3.3.1节得到的依

赖数据传递模式,使用 MPI(Massage Passing Interface)消息传递模型,在节点之间传输必要的依赖数据即可。

运行优化后的算法前需要确定分块大小和负载比例,如表 2 所列。首先针对每组输入使用分块探索工具找出当前输入下 CPU 和 MIC 的最优分块大小;其次使用式(8)确定 CPU 和 MIC 的负载比例;最后将得到的分块大小、负载比例作为参数代入优化后的算法测得吞吐量,并根据吞吐量计算加速比。

表 2　不同输入规模下的程序参数值

| 输入规模 ($X*Y*Z$) | 吞吐量 $T$(MPoints/s) | | 分块参数($X/Y/Z$ 块大小) | |
|---|---|---|---|---|
| | CPU | MIC | CPU | MIC |
| 528 * 528 * 1056 | 3635.47 | 3275.95 | 528/39/79 | 256/1/155 |
| 800 * 800 * 800 | 2795.45 | 3162.46 | 800/8/205 | 784/1/17 |
| 400 * 1100 * 2464 | 3493.80 | 3923.05 | 400/8/400 | 400/1/80 |

在实验过程中,CPU 端和 MIC 端的最优性能均是在使用最大线程配置时获得的,分别为 24 个线程和 240 个线程。实验测得了优化后的算法与原有算法在 4 组时间步 3 种输入规模下的吞吐量,如图 6 所示。从图中可以看出:原有算法的吞吐量基本不受输入规模和时间步数的影响,保持在 32x～39x MPoints/s 之间,优化后的算法的吞吐量随时间步数增加而增大,在单位时间内处理更多的浮点数,因此性能更优。在图 7 中,输入规模为 800 * 800 * 800 时算法的性能略低于其他两组规模下的性能,最大差距为 13.3%,原因在于该输入规模下的循环分块数($\frac{X*Y*Z}{XBF*YBF*ZBF}$)大于其他两组,即此时的 CPU+MIC 最大线程配置仍无法最大程度地并行的算法,算法仍存在进一步并行潜力。

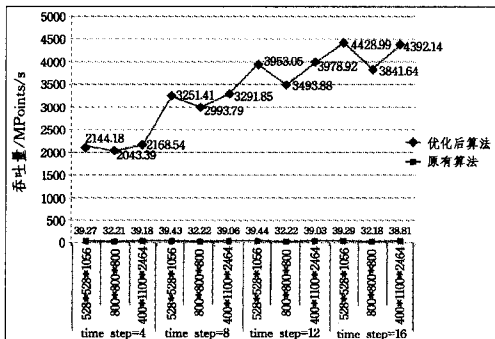


图 6　优化前后算法的性能对比图

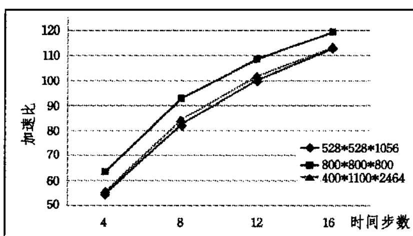


图 7　异构平台下 3DFD 加速比随迭代的变化示意图

通过计算优化后的算法与原有算法的吞吐量之比得到加速比。优化后算法的运行时间包括初始 CPU 传输数据至 MIC 的传输时间、迭代内 3DFD 的计算时间、迭代间交换依赖数据的传输时间和结束时将 MIC 运算结果传回 CPU 的传输时间 4 部分,其中传输时间由传输数据量和传输速率决定,实验中测得 PCI-E 传输速率为 6.53GB/s。以输入规模 400 * 1100 * 2464 为例,当时间步迭代次数为 4 时,传输时间占运行时间的比例 $R_{trans}$ 为 68.47%,随着迭代次数增加,$R_{trans}$ 逐渐降低,当时间步迭代次数为 16 时,$R_{trans}$ 为 34.67%,因此加速比随 $R_{trans}$ 的减小而增大,如图 7 所示。考虑到实际应用中时间步迭代次数一般比较大,所以该情况的优化效果会更好。

从图 7 中可以看出,输入规模 800 * 800 * 800 所对应的加速比略优于其他两组规模对应的加速比,经过分析原有算法的性能后发现:输入规模为 800 * 800 * 800 时,原有算法的吞吐量(32x MPoint/s)低于其他两组规模下原有算法的吞吐量(39x Mpoints/s),并且由图 6 可知,输入规模为 800 * 800 * 800 且 time step=16 时优化后的算法的吞吐量(3800x Mpoints/s)也低于其他两组(4400x Mpoints/s),但是加速比是通过优化后的算法与原有算法吞吐量之比计算的,因此算法在该输入规模下反而可以在异构平台上获得更高的加速比。

此外,相对于单 CPU 端和单 MIC 端,异构协同优化后的算法可以获得随时间步数增加而增大的加速比,为算法在不同时间步数下选取最适合算法运行的平台提供了依据,如图 8 所示,当时间步数小于或等于 8 时选择单 MIC 平台,当时间步大于或等于 12 时选择 CPU+MIC 平台。

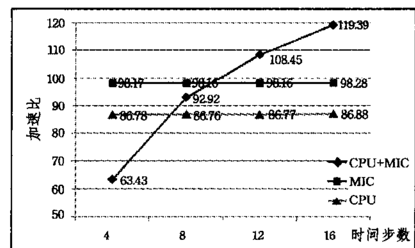


图 8　输入规模 800 * 800 * 800 在不同平台下的加速比

最后给出使用三步递进优化法中每一步优化的效果,其中第二步分为 3 个优化结果:循环分块+OpenMP、内建向量指令和内建向量指令+Prefetch。结果如图 9 所示。

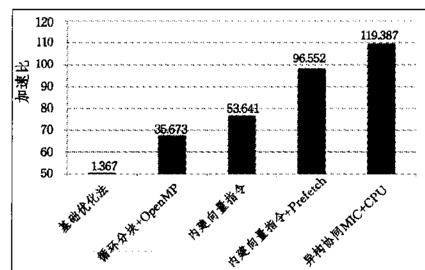


图 9　时间步为 16 时三步递进优化法分步优化的效果图

**结束语**　本文基于 Intel MIC 架构对 3D 有限差分算法进行了实现,并从分支消除、循环分块、向量指令以及异构协同 4 个角度对原有算法进行并行优化,并通过实验方法获得了在不同规模下程序运行的最佳配置,为平台选择提供了可靠依据。相对于原有算法,优化后的算法在 Intel MIC 平台上取得了 50～120 倍的加速;实验结果显示,程序加速比与时间步迭代次数成正比,取得了理想的加速效果。优化过程中所使用的技术,特别是针对异构协同所做的优化,为其他运行于异构平台的程序优化提供了可借鉴的思路。

## 参 考 文 献


[1] Top500. org. Top 500 supercomputer site [EB/OL]. [2015-12-09]. http://www. top500. org.

[2] 王恩东,张清,沈铂,等. MIC 高性能计算编程指南[M]. 北京:中国水利水电出版社,2012:15-21.

[3] ZHUKOV V T,KRANSON M M,NOVIKOVA ND,et al. Multigrid effectiveness on modern computing architectures[J]. Programming and Computer Software,2015,42(1):14-22.

[4] REINDERS J,JEFFERS J. High Performance Parallelism Pearls, Multicore and Many-core Programming [M]. Waltham, MA, USA:Morgan Kaufmann,2014:377-396.

[5] ITU L M,SUCIU C,MOLDOVEANU F,et al. GPU Optimized Computation of Stencil Based Algorithms[C]//2011 RoEduNet International Conference 10th Edition:Networking in Education and Reasearch. 2011:1-6.

[6] WANG G B,YANG X J,ZHANG Y,et al. Program Optimization of Stencil Based Application on the GPU-accelerated System [C]//2009 IEEE International Symposium on Parallel and Distributed Processing with Applications. 2009:219-225.

[7] HERNANDEZ M,CEBRIAN J M,CECILIA J M,et al. Evaluating 3-D Stencil codes on Intel Xeon Phi: Limitations and Tradeoffs[C]//XXVI Jornadas De Paralelismo. 2015:441-444.

[8] KOMATITSCH D,ERLEBACHER G,GODDEKE C,et al. High-order finite element seismic wave propagation modeling with MPI on a large GPU cluster[J]. Journal of Computational physics,2010,229(20):7692-7714.

[9] FANG J B,SIPS H,ZHANG L L,et al. Test- driving Intel Xeon Phi[C]//5th ACM/SPEC International Conference on Performance Engineering. 2014:137-148.

[10] LIANG X,WANG Z Y,LIU G H,et al. Numerical simulation of elastic wave based on the staggered grid finite difference method [C]//2011 International Conference on Consumer Electronics, Communications and Networks. 2011:3283-3286.

[11] WANG T,LI L G,ZHANGY,et al. Pseudo-spectral method for modeling elastic wave propagation in isotropic medium[C]// 12th International Conference on Signal Processing. 2014:58-62.

[12] WOSKOBOYNIKOVA G M. Seismic wave propagation in fractured media[C]//Third International Forum on Strategic Technologies. 2008:307-309.

[13] LIU H W,LI B,LIU H,et al. The Algorithm of high order finite difference pre-stack reverse time migration and GPU implementation [J]. Chinese Journal of Geophysics, 2010, 53(7): 1725-1733. (in Chinese)
刘红伟,李博,刘洪,等. 地震叠前逆时偏移高阶有限差分算法及 GPU 实现[J]. 地球物理学报,2010,53(7):1725-1733.

[14] PERAZA J,TIWARI A,LAURENZANO M,et al. Understanding the performance of stencil computations on Intel's Xeon Phi [C]// IEEE International Conference on Cluster Computing. 2013:1-5.

[15] SHAN Y,WU J P,WANG Z H. Hierarchial parallel programming model and parallelization and optimization techniques based on SMP cluster[J]. Appolication Research of Computers, 2006,23(10):254-256. (in Chinese)
单莹,吴建平,王正华. 基于 SMP 集群的多层次并行编程模型与并行优化技术[J]. 计算机应用研究,2006,23(10):254-256.

[16] SATO M. OpenMP:Parallel Programming API for shared memory multiprocessors and on-chip multiprocessors[C]//15th International Symposium on System Synthesis. 2002:109-111.

[17] JEFFERS J,REINDERS J. Intel Xeon Phi coprocessor high-performance programming [M]. 陈建,周珊,李慧等,译. 北京:人民邮电出版社,2014:92-96.

[18] STROUT M M,LUPORINI F,KRIEGER C D,et al. Generalizing Run-time Tiling with the Loop Chain Abstraction[C]// IEEE 28th International Parallel and Distributed Processing Symposium. 2014:1136-1145.


(上接第 25 页)


[14] XIAO C,WANG W,LIN X,et al. Efficient similarity joins for near-duplicate detection[J]. ACM Trans. Database Syst. ,2011, 36(3):1-41.

[15] XIAO C,WANG W,LIN X. Ed-join:an efficient algorithm for similarity joins with edit distance constraints[J]. Proceedings of the Vldb Endowment,2008,1(1):933-944.

[16] WANG W,QIN J,XIAO C,et al. Vchunkjoin:An efficient algorithm for edit similarity joins[J]. IEEE Trans. Knowl. Data Eng. ,2013,25(8):1916-1929.

[17] ARASU A,GANTI V,KAUSHI K R. Efficient exact set-similarity joins[C]//Proceedings of the 32nd International Conference on Very Large Data Bases. Seoul,Korea:VLDB Endowment,2006:918-929.

[18] LI G,DONG D,WANG J,et al. PASS-JOIN:A Partition-based Method for Similarity Joins[J]. Proceedings of the Vldb Endowment,2011,5(3):253-264.

[19] VERNICA R,CAREY M J,LI C. Efficient parallel set-similarity joins using MapReduce[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. Indianapolis,IN,USA:ACM,2010:495-506.

[20] METWALLY A,FALOUTSOS C. V-SMART-Join:A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors[J]. Proceedings of the Vldb Endowment, 2012,5(8):704-715.

[21] AFRATI F N,SARMA A D,MENESTRINA D,et al. Fuzzy Joins Using MapReduce[C]//Proceedings of IEEE 28th International Conference on Data Engineering. Washington, DC: IEEE,2012:498-509.

[22] LIN C,YU H,WENG W,et al. Large-Scale Similarity Join with Edit-Distance Constraints[C]//Proceedings of 19th International Conference on Database Systems for Advanced Applications (DASFAA 2014). Bali,Indonesia:Springer International Publishing,2014:328-342.