doi:10.3772/j.issn.1002-0470.2014.10.005

面向软硬件协同设计的性能优化框架①

骆裕龙② 谭光明③ 孙凝晖

(中国科学院计算技术研究所 北京 10086)

摘 要 面对高性能计算机系统随着性能的提高其复杂性成倍增大的挑战,研究了复杂科学计算应用的优化,提出了一种面向软硬件特性设计的性能优化框架 CPTF。该框架根据应用在运行时的剖析结果,结合应用的软件特性和平台的硬件特性,全局性地分析系统性能瓶颈及种类,并给出源码级的优化建议,并针对优化循环一类常见的问题,提出一种改进的循环合并算法。使用 CPTF 优化了一个物质点法粒子模拟应用,取得了近 20% 的性能提升。

关键词 高性能计算,优化,软硬件协同设计,循环合并,静态分析

0 引 言

随着计算机科学的飞速发展,高性能计算系统在未来十年将逐渐步入 E 级时代,其规模会越来越大,其复杂性会越来越高,这对于计算机使用者和开发者来说,都是一个巨大的冲击和挑战。使用者需要综合考虑其性能、功耗和费用,开发者需要考虑其运行系统的算法和体系结构的特征。不同算法的应用引起的瓶颈的原因是不同的,例如:访存密集算法的应用,如松弛线性代数算法,影响其性能的关键因素是带宽能力和通信的延迟;计算密集型算法的应用,如稠密线性代数算法,影响其性能的关键因素是计算能力。另一方面,算法的最优实现也依赖于体系机构的特征,如 Stencil 计算中的 Tiling 优化算法[1,2],其最优分块的大小取决于高速缓存的大小;而并行算法,任务的划分更要考虑系统中的计算核心数及计算核心的组织结构。单纯地优化硬件或者单纯地优化软件已不能满足未来高性能计算系统的需求,面对这种情况,通过软硬件协同设计,对应用和实现平台进行综合调优来达到既定的性能、功耗及费用目标是未来高性能计算系统设计和优化的趋势[1]。这方面的研究已取得一定成果,主要有:硬件模拟工具 RAMP Gold[4] 和离散事件模拟器工具如 SST[5],它们能够提供精确的性能数据用于系统分析调优,但是计算开销大,分析周期长;Vtune 性能分析器[6],独立于编译器及语言,且性能开销低,但只支持 INTEL 架构处理器的性能分析;性能应用编程接口(performance application programming interface,PAPI)[7],它可以收集程序在运行过程中的性能数据,但只能手动地在源代码中需要采集的位置添加采集代码,不能全局地分析应用瓶颈;基于采样的性能剖析工具 hpctoolkit[8],它能够快速、全局地分析应用的瓶颈,但没有基于源码方面的分析,无法对软件的优化给出直接指导;Pbound[9],它使用静态编译分析的技术来评估 C/C++ 应用 kernel 的性能表现,能够在较短的时间内完成对应用性能的分析,但没有考虑到输入数据对应用造成的影响,也没有结合体系结构的参数进行综合分析。在以上工作的基础上,本文提出了一种面向软硬件特性设计的性能优化框架(software/hardware co-design performance tuning framework,CPTF),它根据运行时的剖析结果,确定在实际输入情况下的应用热点,结合应用的软件特性和平台的硬件特性,全局地分析系统性能瓶颈及种类,并给出源码级的优化建议。

① 973 计划(2011CB302502)资助项目。

② 男,1988 年生,博士生;研究方向:计算机体系结构;E-mail: luoyulong@ncicac.cn

③ 通讯作者,E-mail: tgm@ict.ac.cn

(收稿日期:2014-04-24)

1 性能衡量指标

受 Roofline 模型[10]的启发,本文借助以下两个指标来衡量高性能计算系统软件特性和硬件特性:

(1) 计算访存比:应用子模块平均完成一个浮点操作所需访存的比特数。若浮点操作数为 f, 算法访存数为 m, 记该算法的计算访存比为 f/m;

(2) 最大计算能力访存能力比:计算机系统所能提供的计算峰值能力和访存峰值能力的比值。若计算机系统的最大每秒执行浮点运算次数(floating-point operations per second, FLOPS)数为 F, 其存储系统能够提供最大的带宽为 B, 那么记该计算机系统的最大计算能力最大访存能力比为 F/B。

另外,根据软硬件特性分析系统的瓶颈:当应用子模块的计算访存比大于计算机系统提供的最大计算能力访存能力比时,我们则认为这部分模块在运行时为计算受限型:模块对计算敏感,运行时计算能力成为短板,此时要通过增加计算核心的频率、数量和算法的并行度来提升该模块的运行性能;当应用子模块的计算访存比小于计算机系统提供的最大计算能力访存能力比时,我们认为这部分模块在运行时为访存受限型:模块对访存敏感,运行时访存成为性能的短板,此时要在硬件上提升存储器带宽、访存延迟,在软件上要通过优化算法的访存来提升该模块的运行性能。

2 方法

考虑到输入对控制流、基本代码块性能的影响,首先对目标系统进行运行时剖析,收集到系统运行的时间开销分布和调用关系;结合剖析信息得到瓶颈并分离出瓶颈部分源码,通过编译分析得到软件特性;把软件特性与硬件平台给出的硬件特性一同输入性能模型,评估得到系统瓶颈的种类;最后根据瓶颈的种类给出优化建议,整个工作流程如图 1 所示。其中,软件特性是瓶颈部分所涉及的浮点操作数及读写访存操作数,硬件特性是硬件平台主要的体系结构参数,包括计算核心数,主频及主存带宽。下面,分别对这 4 个步骤进行详细的介绍。

(1)运行时剖析:首先针对运行的硬件平台对应用进行编译得到可执行程序,结合应用输入,运行系统。出于精度和灵活的考虑,我们选用 google 性能工具 gperftools(google performance tools)[11]作为剖析器,在应用的链接阶段链接 gperftools 动态库 profiler,并在系统运行时通过环境变量控制剖析过程。系统完成运行后,gperftools 输出性能剖析文件,该输出记录了系统运行时函数的调用信息及运行开销,得到核心函数。

(2)编译分析:得到系统的核心函数后,对核心函数的代码进行静态分析。首先对整个应用项目的源代码进行词法、语法分析,转化成抽象语法树表示(SageIII IR),之后读入带有核心函数信息的文件,根据核心函数名及所在的源文件这一、二元组信息搜索应用项目的抽象语法树,得到表示核心函数的语法子树结点,向语法树结点的综合属性中添加浮点操作计数器、访存操作计数器等属性并重载语法树结点综合属性的计算函数:当遇到结点为变量引用表达式时,评估出变量的大小 x, 综合属性中读操作计数器加 x; 当遇到结点为赋值表达式时,评估出其左操作数的大小 x, 综合属性中写操作计数器加 x; 当遇到结点为一元、二元的算术操作表达式,且其左右操作数至少有一个为浮点变量时,综合属性中浮点操作计数器加一,而且对核心函数子树进行自下而上的遍历。在遍历的同时,通过对综合属性的计算,得到核心函数子树关于浮点操作数、访存操作数的统计信息。最后,遍历核心函数子树,得到核心函数下主要循环语句子树。循环语句子树由 4 个子树构成,分别是初始条件子树、边界条件子树、增量子树和循环体子树。前 3 个子树决定了循环执行的次数,即循环的权重,最后一个子树即循环体子树则表示实际执行的语句,其综合属性决定了循环的软件特性。综合循环的软件特性和循环的权重,我们得到了应用的软件特性,其中静态分析工具基于 ROSE 编译框架[12]开发。

(3)瓶颈分析:把影响系统性能的软件特性与硬件平台体系结构参数组成的硬件特性,一同输入到性能模型。性能模型首先根据硬件特性,计算出硬件平台的浮点计算峰值 p_c 和带宽峰值 p_b, 再把浮点峰值作为最大计算能力,带宽峰值作为最大访存能力,得到系统的最大计算能力访存能力比:

$$PMAX_{\text{ratio}} = p_c/p_b \tag{1}$$

之后处理系统的软件特性:根据浮点操作数 fop (floating-point operations),读操作数 ld, 写操作数 st, 计算出每个循环的计算访存比:

$$LOOP_{\text{ratio}} = fop/(ld + st) \tag{2}$$

最后,依次把每个循环的计算访存比与系统的最大计算能力访存能力比做比较,得到性能瓶颈类型,给

出优化建议:对于计算访存比小于最大计算能力访存能力比的情况,判断为计算受限型瓶颈,建议提高硬件平台的计算能力和算法的并行性,减少计算开销;对于计算访存比大于系统最大计算能力访存能力比的情况,判断为访存受限型瓶颈,建议提高硬件平台的访存能力,利用局部性原理减少访存操作和通过预取降低访存带来的延迟。

(4)优化:根据优化建议,在应用的瓶颈处选择符合可行性的方法进行优化。优化完成后迭代整个过程,直至算法需求和硬件供给的性能达到平衡,完成优化。

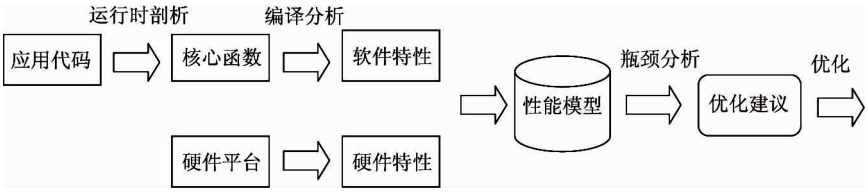

图 1　软硬件协同设计的性能优化框架

3　应用优化

3.1　瓶颈的分析

本文选取了物质点法粒子模拟和两个应用进行实际分析优化。我们运行的平台为英特尔至强 E5-2670,主频 2.6GHz,8 核心 16 线程,计算峰值为 166.4Gflops,内存规格为 DDR3-1333M,8 通道,内存的峰值带宽为170.6GB/s,所能提供最大计算能力访存能力比为0.975,见表 1。应用使用 Linux 平台的 MPICC 编译,最大迭代步为 100 步,网格规模 160 ×160 ×160。

表 1　体系结构参数

处理器	E5-2670
核心	8
线程	16
全频	2.6GHz
缓存	20MB
指令集扩展	AVX
计算峰值	166.4 Gflops
内存规格	DDR3 – 1333
内存峰值带宽	170.6GB/s
最大计算能力访存能力比	0.975

在剖析运行过程中,我们设核心函数的阈值为 10%,即核心函数运行时间开销大于整个系统运行时间的 10%。

经过剖析,在物质点模拟应用中共有 5 个函数运行时间的比重超过 10%:UpdateStressCell(42.2s/227s)、CellMomentum(32.2s/227s)、ComputingCell(47s/227s)、EdgeMomentum(29s/227s)、UpdateMPV(40s/227s)。这 5 个核心函数所占运行时间比重的总和为整个应用耗时的 84%(191s/227s),具体的比重分布见图 2。可以看出,这 5 个核心函数占用了系统绝大部分的运行时间,剩余的代码运行时间只占总时间的 16%;而 Himeno 应用中的核心函数为 Jacobi,其运行时间占到了整个应用运行时间的 94.5%(8.65s/9.15s)。

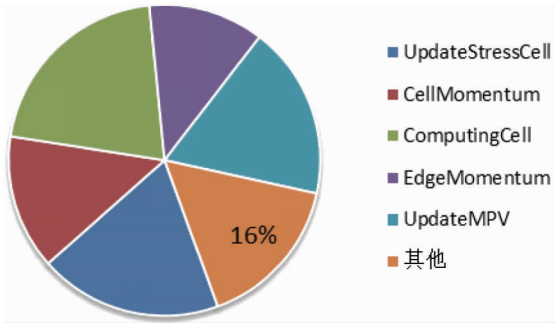

图 2　时间分布

把核心函数名及所在文件的文件名作为输入,使用编译分析工具计算核心函数的软件特性,结果如表 2 所示。把软件特性与硬件特性输入性能模型,性能模型首先根据硬件特性计算出平台能够提供的最大计算能力访存能力比:$PMAX_{\text{radio}} = p_c/p_b = 0.975$,其中 $p_c = 166.4\text{Gflops}$,$p_b = 170.6\text{GB/s}$。之后,分析物质点法粒子模拟与 Himeno 应用的软件特性,求解每个核心函数循环的计算访存比:$LOOP_{\text{radio}} = fop/(ld + st)$,再把核心函数循环的计算访存比与系统最大计算能力访存能力比做比较,判断瓶颈类型,结果见表 3。

表 2 核心函数的软件特性

函数信息	循环信息		浮点操作	访存操作数		
	循环次数	嵌套层次		读(次)	写(次)	总计
MPM::CellMomentum	8	1	7	8	3	88
	变量	0	56	64	24	704
MPM::UpdateStressCell	8	1	39	24	18	336
	变量	0	331	208	156	2912
MPM::ComputingCell	8	1	136	96	48	1152
	变量	0	276	240	107	2776
MPM::UpdateMPV	8	1	96	48	48	768
	变量	0	116	70	54	992
	变量	0	2	7	4	88
MPM::EdgeMomentum	变量	0	0	0	3	24
Himeno::Jacobi	4	1	0	0	1	0
	变量	0	33	35	4	312

表 3 瓶颈类型

函数信息	循环信息		
	循环	计算访存比	瓶颈
CellMomentum	1	0.079545	访存
	2	0.079545	访存
UpdateStressCell	1	0.116071	访存
	2	0.113668	访存
ComputingCell	1	0.118056	访存
	2	0.099424	访存
UpdateMPV	1	0.125	访存
	2	0.116935	访存
	3	0.022727	访存
EdgeMomentum	1	0	访存
Jacobi	1	0	访存
	2	0.10	访存

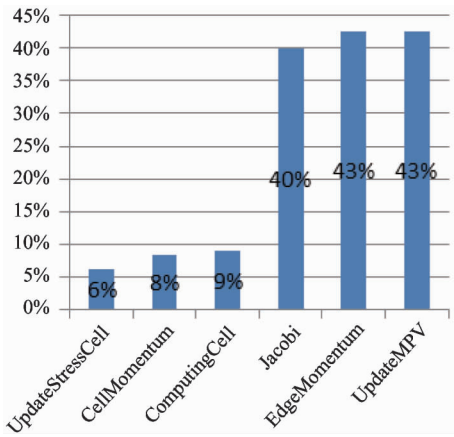

图 3 性能提升

3.2 优化

通过上述分析,我们获得了整个系统所有瓶颈的位置及种类,借助这些信息并结合优化经验,我们应用了循环边界替换、循环展开、循环合并、非局部写操作的高速缓存旁路等优化手段来缓解瓶颈部分的访存开销,最后重新进行编译运行。图 3 是核心函数在针对性优化后收获的性能提升。通过针对性的优化,函数 UpdateStressCell、CellMomentum、ComputingCell,Jacobi 的性能分别上涨了 6.3%,8.4%,9% 和 40%,而 EdgeMomentum 和 UpdateMPV 由于应用了一种改进的循环合并技术,性能提升达到了 42.5%,通过对这小部分核心代码的改造,整体应用性能也有了显著的提升。函数 EdgeMomentum 和 UpdateMPV 的优化技术,将在第 4 节中进行介绍。

4 一种改进的循环合并技术 s

函数 EdgeMomentum 和 UpdateMPV 顺序执行,为了进一步优化,首先把两者合并到一个函数 EdgeMomentum _ UpdateMPV 内。在合并后的 EdgeMomentum _ UpdateMPV 中有三个循环:循环 A,循环 B,循环 C,如图 4 所示,图 5 是函数 UpdatePosAndVel 的展开。其中循环 A,C 具有相同的循环结构:顺序遍历 grid 的 Node 集合,每次都涉及 Node 集合集合项的读写。根据 PBAT 分析结果,这两个循环都是访存受限型,需要对访存进行优化,那么使用循环合并的方法合并两个循环,可以使两者的工作集合并,减小访存操作。但是,这里有一个问题:循环 A,B,C 之间存在数据依赖,且循环 B 的循环结构不同于循环 A、C,三者之间不能进行简单的循环合并。这在里,我们应用一种优化算法,专门针对这类情况

循环合并。下面的内容分为两小节,第一小节描述算法,第二小节应用该算法优化函数 EdgeMomentum _ UpdateMPV。

```
    循环 A
1:  for(unsigned int i = 0; i <Grid.NumNode; i ++) {
2:  if(Grid.Nodes[i] != NULL)
3:  Grid.Nodes[i]->UpdateMomentum();
4:          }
    循环 B
5:      UpdatePosAndVel();
    循环 C
6:  for(unsigned int i = 0; i <Grid.NumNode; i ++) {
7:  if(Grid.Nodes[i] != NULL)
8:  Grid.Nodes[i]->InitializeMomemtum();
9:          }
```

图 4　EdgeMomentum _ UpdateMPV 源码

```
1:  for(size_t i = 0; i < p->GetSize(); i ++) {
        /*此处不相关的代码被省略*/
2:    MPM::CCell * cell = (MPM::CCell *)p->CellPointer[i];
3:    ICTCORE::CVector3<double> v, a;
4:  for(unsigned char n = 0; n < 8; n ++) {
5:        MPM::CnodeProperty*node = (*cell)[n]->GetNode(part);
6:  if(node->Mass <Grid.CutOff)continue;
7:        v += node->Momentum*shape[n] / node->Mass;
8:        a += node->Force*shape[n]/node->Mass;
9:      }
        /*此处不相关的代码被省略*/
10: }
```

图 5　UpdatePosAndVel 源码

4.1　算法的描述

假设有三个循环,即循环 1、2、3。三个循环都涉及集合 E 中的集合项的读写,其中循环 1 和循环 3 是顺序对集合 E 进行遍历,循环 2 是非顺序对集合 E 进行访问,如图 6 所示。对于集合 E,它在三个循环间存在时间局部性,即集合 E 中的每个项都会

```
1:  循环1
2:  {
3:      B1
4:  }
5:  循环2
6:  {
7:      B2
8:  }
9:  循环3
10: {
11:     B3
12: }
```

图 6　循环 A、B、C

在循环 1、2、3 中被访问到。现在要通过循环合并减少因为访问集合 E 带来的访存开销。设循环 1 中的执行语句的集合为 B_1(body1),循环 2 的为 B_2,循环 3 的为 B_3,且循环 2 对所要访问的集合项只访问一次。

通过分析,集合 E 中的集合项可以分为两部分:一部分会出现在循环 2 中,即依次在 B_1、B_2、B_3 中被读写,记为 E_1;另一部分只会被 B_1、B_3 读写,记 E_2。通过从集合 E 中提取集合 E_1、E_2 并区别计算,合并三个循环:在初始化阶段,向集合 E 中的每个集合项附加一个值 stage,表示集合项所处的计算阶段;在循环 2 中,判断 stage 的值,若为 0,则执行 B_1、B_2、B_3,并修改 stage 的值为 1,这部分集合项即为 E_1;在循环 3 中,根据 stage 的值,筛选出集合 E 出去 E_1 的剩余部分,即 E_2,执行 B_1、B_3,如图 7 所示。相比较之前 E_1 中的集合项需要访存三次才能完成 B_1、B_2、B_3 的计算,区分计算后只需一次的访存开销就可完成;同样 E_2 中的集合项之前需要访存二次才能完成 B_1,B_3 的计算,区分计算后只需要访存 1 次。相比优化前,集合 E 中的所有集合项都利用其时间局部性,以最小的访存开销完成相关计算。

```
1:  循环2
2:  {
3:    if(stage==0)
4:    {
5:      B1,B2,B3
6:      stage=1
7:    }
8:  }
9:  循环3
10: {
11:   if(stage==0)
12:   {
13:     B1,B3
14:   }
15: }
```

图 7　优化算法 1

下面建立访存开销的模型,分析优化算法对其影响。设每个集合项的大小为 S_e 字节,集合 E 中共有 N_e 个集合项,集合 E_1 中共有 N'_e,在优化前,三个循环涉及集合 E 的访存开销:

$$M_{\text{memory\_traffic}} = 2S_eN_e + S_eN'_e = S_e(2N_e + N'_e) \tag{3}$$

优化后的访存开销为

$$M'_{\text{memory\_traffic}} = S_eN'_e + (N_e - N'_e)S_e + 1 \times N'_e + 1 \times N_e$$

$$= S_e N_e + N'_e + N_e \quad (4)$$

其中 stage 的大小为 1 字节,增加的访存开销为 $1 \times N'_e + 1 \times N_e$。

优化算法总体减少的访存开销为

$$\begin{aligned}\Delta M &= M_{\text{memory\_traffic}} - M_{\text{memory\_traffic}} \\ &= S_e(2N_e + N'_e) - (S_e N_e + N'_e + N_e) \\ &= (N_e + N'_e) \times (S_e - 1) \quad (5)\end{aligned}$$

但是在函数 EdgeMomentum _ UpdateMPV 中,当循环 B 遍历到多个粒子处于相同网格的情况时,该网格相邻的 node 会被多次访问到,即不满足 B_2 只访问每个集合项一次的限制条件。如果按照图 8 的方式进行优化,将会多次执行语句 B_1,语句 B_3,在有些情况下会产生应用的逻辑错误。

```
1:  循环2
2:  {
3:    if(stage==0)
4:    {
5:     B1
6:     stage=1
7:    }
8:  B2
9:  }
10: 循环3
11: {
12:   if(stage==0)
13:   {
14:    B1
15:   }
16: B3
17: }
```

图 8 优化算法 2

对算法做改进,需去除循环 2 对要访问的集合项只访问一次的限制:在循环 2 中,若结合 E_1 被第一次访问,那么执行语句 B_1、B_2,否则只执行语句 B_2;在循环 3 中,对于根据 stage 的值区分集合 E_1、E_2,对 E_1 只执行语句 B_3,对 E_2 执行语句 B_1,B_3。这样做即避免了 B_1 在循环 2 中的重复执行。算法如图 8 所示。

此时减少的访存开销为

$$\begin{aligned}\Delta M &= M_{\text{memory\_traffic}} - M'_{\text{memory\_traffic}} \\ &= N_e S_e - (N_e + N'_e) \quad (6)\end{aligned}$$

4.2 应用优化

现在我们继续之前的优化工作。根据提出算法的描述,集合 Nodes 在三个循环中具有时间局部性,且根据进入的循环可以分为两个集合,即 $Nodes_1$ 和 $Nodes_2$,其中集合 $Nodes_1$ 在循环 A 和循环 C 执行时被访问,集合 $Nodes_2$ 在循环 A、循环 B、循环 C 执行时都被访问。对于集合 $Nodes_1$ 中集合项涉及的所有操作,都放到新循环 C 中执行,对于集合 $Nodes_2$ 中集合项涉及的循环 A、循环 B 语句放入新循环 B 中执行,涉及的循环 C 语句依旧在循环 C 中执行。转换后的三个循环如图 9、图 10 所示。

```
1:  循环 B
2:  UpdatePosAndVel();
3:  循环 C
4:  for(unsigned int i = 0; i <Grid.NumNode; i ++) {
5:  if(Grid.Nodes[i] != NULL)
6:          {
7:  if(Grid.Nodes[i]->stage==0)
8:              {
9:  Grid.Nodes[i]->UpdateMomentum();
10: Grid.Nodes[i]->stage=1;
11:             }
12: Grid.Nodes[i]->InitializeMomemtum();
13:         }
        }
15: }
```

图 9 优化后的 EdgeMomentum _ UpdateMPV

```
1:  for(size_t i = 0; i < p->GetSize(); i ++) {
2:        /*此处不相关的代码被省略*/
3:    MPM::CCell * cell = (MPM::CCell *)p->CellPointer[i];
4:    ICTCORE::CVector3<double> v, a;
5:  for(unsigned char n = 0; n < 8; n ++) {
6:        MPM::CnodeProperty*node = (*cell)[n]->GetNode(part);
7:  if(cnode->stage==0)
8:          {
9:  cnode->UpdateMomentum();
10: cnode->stage=1;
11:         }
12: if(node->Mass <Grid.CutOff)continue;
13:       v += node->Momentum*shape[n] / node->Mass;
14:       a += node->Force*shape[n]/node->Mass;
15:     }
16:     /*此处不相关的代码被省略*/
17:     }
```

图 10 优化后的 UpdatePosAndVel

4.3 性能测试及分析

优化后,对应用进行重新编译、运行,函数 EdgeMomentum 和 UpdateMPV 所占用的时间大大下降,以绝对时间为参考,相比优化前性能提升了 32%。对此,我们根据之前设立的访存模型对其进行理论验证:

集合 Nodes 中每个集合项由两个 3 维 double 型

向量 Momentum、Force 组成,大小分别为 24 字节,即每个集合项 S_e 的大小为 48 字节。由于合并循环数量为三个,stage 为 bool 即可区分 Nodes 不同子集,故 stage 大小也为 1 字节。由于在应用中,Node 的个数、粒子的个数和位置会随着时间步的增加发生改变,所以我们记录了每个时间步 Nodes 集合的总数和在循环 B 中访问的集合 $Nodes_2$ 总数,再使用公式

$$\begin{aligned}\Delta M &= M_{\text{memory\_traffic}} - M'_{\text{memory\_traffic}} \\ &= N_e S_e - (N_e + N'_e) \end{aligned} \quad (7)$$

分别建模求得每步减少的访存开销 ΔM,并求和除以总的步数,得到应用的总访存收益。与未优化前相比,集合 Nodes 带来的访存操作减少了 32.18%,基本与实测性能提升一致,这说明在函数 EdgeMomentum 和 UpdateMPV 执行中,对集合 Nodes 访存的操作时间基本上占了两个函数 100% 的执行时间,计算开销因为访存的延迟被隐藏,这个实验结果与我们之前性能模型的结论相一致:综合函数 EdgeMomentum 和 UpdateMPV 的软件特性与系统最大计算能力访存能力比的硬件特性,两个函数的性能瓶颈为访存瓶颈型,单纯的提高计算能力对性能没有帮助,只有缓解访存瓶颈才能带来性能的提升。

5 结 论

本文认为要优化日益复杂的科学计算应用,不能单纯地从软件优化或者硬件优化下手,而要结合整个系统的软件特性和硬件特性进行综合分析,找出系统性能瓶颈的位置和种类,才能有针对性地有效地选择提升系统性能的手段,从而满足既定性能、功耗和费用目标。为此,本文提出了面向软硬件特性设计的性能优化框架(CPTF):首先通过运行收集应用的运行时信息,得到最耗时的核心函数;然后通过编译分析对应用编译、获取核心函数的软件特性;最后结合系统的硬件特性,确定系统瓶颈的位置及类型,选择相适应的优化手段。为了实践该方法,本文对物质点类的粒子模拟应用进行实际优化,并取得了预期的优化结果,验证了 CPTF 的可行性。在面对循环优化中常见的一类情况,本文又提出一种改进的循环合并方法,克服了循环合并时结构不同所引起的合并问题。

但是,单纯的通过分析计算操作和访存操作并不能完全反映所有类型的应用的性能表现,在后续的工作中,我们还将添加系统其他方面的特性,如依赖步长、主存占用率、高速缓存失效比、工作集变化情况等,对系统进行更全面、更细致的剖析,再使用数据挖掘的方法对剖析结果进行分类,建立特征应用系统与优化方案的隐射关系,使得对这类复杂应用系统的优化更加准确,更具针对性。

参考文献

[1] Rivera G, Tseng C. Tiling optimizations for 3D scientific computations. In: Proceedings of ACM/IEEE Conference on Supercomputing, 2000. doi: 10.1109/SC.2000.10015

[2] Lim A, Liao S, Lam M. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In: Proceedings of the ACM SIGPLAN Symposiumon on Principles and Practice of Parallel Programming, Snowbird, USA, 2001

[3] Mohiyuddin M, Murphy M, Oliker L, et al. A design methodology for domain-optimized power-efficient supercomputing. ACM/IEEE Conference on Supercomputing, Portland, USA, 2009. 1-12

[4] Tan Z X, Waterman Z, Avizienis P, et al. RAMP Gold: an FPGA-based architecture simulator for multiprocessors. In: Proceedings of the 47th Design Automation Conference, Anaheim, USA, 2010. 463-468

[5] Curtis L. Janssen, et al. A simulator for large-scale parallel computerarchitectures. *International Journal of Distributed Systems and Technologies*, 2010, 1(2):57-73

[6] Intel. IntelVtune guide. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-2011-documentation, 2013

[7] utk. PAPIguide. http://icl.cs.utk.edu/papi/, 2013

[8] Adhianto L. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*. 2010, 22(6): 685-701

[9] Krishna Narayanan S H, Norris B, Hovland P D. Generating performance bounds from source code. In: Proceeding of the 39th International Conference on Parallel Processing Workshops (ICPPW), San Diego, USA, 2010: 197-206

[10] Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65-76, 2009

[11] google. gperftools web page. https://code.google.com/p/gperftools/

[12] Quinlan D. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 2000, 10(02, 03): 215-226

A performance optimization framework for hardware/software co-design

Luo Yulong, Tan Guangming, Sun Ninghui
(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 10086)

Abstract

The optimization of the application of complex scientific computation was studied to face the challenge that the complexity of high performance computers grows quickly with their performance increasing, and the CPTF, a hardware/software co-design performance tuning framework, was proposed. The CPTF can give an overall analysis of a computer system's performance bottlenecks and their types according to the profile of the application at run-time combined with the application's software characters and the platform's hardware characters, and finally give optimization suggestions on source-level. And also it can propose an advanced loop fusion algorithm to solve the common problems in loop optimizing. The CPTF was used to optimize the particles simulation application of MPM and achieved about 20% of performance improvement.

Key words: high performance computing, optimization, software/hardware co-design, loop fusion, static analysis