doi:10.3772/j.issn.1002-0470.2020.02.005

提升高性能计算程序性能可移植性的领域特定语言①

李 韦②*** 文渊博*** 孙广中③* 陈云霁**

(*中国科学技术大学计算机科学与技术学院 合肥 230026)

(**计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)

(***上海寒武纪信息科技有限公司 上海 201203)

摘 要 高性能计算(HPC)应用程序大多基于标准函数库和编译制导语句进行编写,这种做法可以有效提升高性能计算应用的可编程性和可移植性。相比传统优化方法中针对单个函数库进行优化,本文的研究将优化注意力放到不同函数库调用之间,提出了一种用于高性能函数库的领域特定语言及编译器,实现了对原始C代码的源代码到源代码优化,解决了因为胶水代码而产生的高性能计算程序性能可移植性欠佳的问题。实验结果表明,在真实应用中,使用支持该领域特定语言的编译器,在通用处理器硬件架构上,可以取得相比原始版本最高4.89倍的优化加速;而在实验性的异构高峰值加速器架构上,可以取得最高8.21倍的优化加速。

关键词 高性能计算(HPC);可移植性;胶水代码;领域特定语言;编译器

0 引 言

过去的几十年,在高性能计算(high-performance computing,HPC)社区的软件开发工作中,诞生了许多广泛使用的高性能函数库,这些函数库或针对计算进行优化,如基本线性代数例程(basic linear algebra subprograms,BLAS)[1-3]、高扩展线性代数库(scalable LAPACK,ScaLAPACK)[4]、西方最快傅立叶变换(fastest Fourier transforms in the West,FFTW)[5];或提供等特定场景中的基本原语,如信息传递接口(message passing interface,MPI)。同时,使用编译制导语句进行编程,提高程序并行度也已成为高性能应用开发中的常用技术,例如开放并行处理(open multi-processing,OpenMP)[6]和开源加速器(open accelerators,OpenACC)[7]。使用标准化的库函数以及编译制导语句的一个重要原因是:使用它们进行编程,可以提高所编写应用程序的性能可移植性。使用库函数和编译制导语句,程序员不必过分纠结底层实现,而相同的代码也可以更为轻松移植到新硬件架构上。同时,当标准库的开发人员为所需移植的目标架构提供优化版本时,应用程序在硬件架构上的性能也会同步得到提高。

尽管在许多应用程序中广泛使用了高度优化的函数库,但是将所有的库"链接"到一个应用程序中的过程仍然存在问题。其原因在于不同函数调用之间通常需要使用手写代码(即胶水代码),在将应用程序的输入或者函数调用的输出作为输入传递给另一个函数调用之前,进行一些类似于更改数据布局格式或分配存储空间的额外操作,以实现数据预处

① 国家重点研发计划(2017YFA0700900,2017YFA0700902,2017YFA0700901,2017YFB1003101),国家自然科学基金(61432016,61532016,61672491,61602441,61602446,61732002,61702478,61732020),北京市自然科学基金(JQ18013),973计划(2015CB358800),"核心电子器件、高端通用芯片及基础软件产品"科技重大专项(2018ZX01031102),中国科学院青年创新促进会优秀会员项目(CX2150110004),中国科学院科技成果转移转化重点专项(KFJ-HGZX-013)和中国科学院战略性先导科技专项(B类)(XDB32050200)资助项目。

② 男,1995年生,硕士生;研究方向:高性能计算,计算机软件;E-mail:wei123@mail.ustc.edu.cn

③ 通信作者,E-mail:gzsun@ustc.edu.cn

(收稿日期:2019-03-21)

理。胶水代码常会阻碍将应用程序代码移植到新的架构上,原因在于它们大多是针对硬件架构进行编写的,也同样需要针对新的硬件架构和应用程序进行特定的优化,并且胶水代码中可能使用了硬件架构特定的指令。例如,如果使用矩阵向量乘法实现定制化矩阵乘法,因为循环的存在,应用程序有了更多的访存和指令跳转,当其被移植到对访存带宽更为敏感的硬件平台上时,性能瓶颈会被进一步放大。再例如,在支持高级向量扩展指令集(advanced vector extensions,AVX)指令的处理器上,会使用单指令流多数据流(single instruction multiple data,SIMD)指令进行向量化加速,而这样的程序无法运行到不支持 AVX 指令的处理器上。因此,胶水代码的存在使得高性能计算程序性能可移植性降低。

有许多关注于特定领域内程序性能的优化工作。SPIRAL[8]作为面向数字信号处理领域的调优系统,提供了从高层次的数学算法描述到低层次的代码实现的映射,能够自动生成给定硬件平台的算法的高性能实现。Halide[9]是专为图像处理领域设计的编程语言,它的核心思路是将算法描述和计算过程分离,程序员可以灵活的使用循环展开、向量化等方法手动尝试不同的调度算法来得到最好的性能。Chen 等人[10]针对于深度学习应用,提出 TVM 优化框架,通过 AutoTVM 的方式用机器学习的方法去优化程序空间的代价估计函数,在庞大的参数空间之中预测最优的优化方式,从而将深度学习灵活、高效地部署在不同的硬件平台之上。

本文针对于高性能计算应用,描述了与编译器优化[11,12]相关的工作,对高性能库函数调用中存在的语义信息进行了分析,在编译时对应用程序进行优化。对使用了大量胶水代码的高性能应用程序,本文尝试用更为高效的库函数替代胶水代码与原始的库函数调用,并增加更多的编译制导语句,使整个应用在不同平台上的整体性能得到提升。

本文描述了一种抽象的关于高性能函数库的领域特定语言,以及支持它的原型编译器实现。这种源代码到源代码的编译器,首先将库函数调用转换为关于库函数的领域特定语言,之后针对生成的领域特定语言进行分析和优化,最后将优化后的领域特定语言重新翻译为更为高效的 C 代码。这种设计,实现了对原始 C 代码的性能优化。针对需要移植的应用和目标硬件,优化领域特定语言到 C 代码的翻译过程,就可以使高性能计算的应用程序具有更好的性能可移植性。本文以简单的矩阵乘算法以及复杂的空时自适应处理(space-time adaptive processing,STAP)[13-15]算法为应用示例,展示了移植到不同的硬件平台的具体实现。实验结果表明,该领域特定语言以及编译器可以显著提高应用程序在不同硬件平台上的性能表现。

1 研究动机

在调用高性能函数库的过程中,需要增加许多胶水代码,其作用多数功能为数据重排、数据补齐等。图 1 以一段基于 FFTW 函数库编写的应用程序为例,展示了需要实际解决的问题和需要完成的优化。

图 1 中的代码片段取自一个真实应用,其描述了多次调用 FFTW 库函数执行多个一维快速傅里叶变换(FFT),接着用一段手写代码将多个 FFT 的输出转换为不同的布局格式。这里的胶水代码具体是

```
1.  //planning
2.  fftw_plan plan_fft = fftw_plan_dft_1d(N_DOP,
3.    &datacube_pulse_major_padded[0][0][0],
4.      &datacube_pulse_major_padded[0][0][0],
5.      FFTW_FORWARD,
6.      FFTW_PATIENT);
7.
8.  // multiple 1-D FFTs
9.  for (chan = 0; chan < N_CHAN; ++chan)
10.   for (range = 0; range < N_RANGE; ++range)
11.     fftw_execute_dft(plan_fft,
12.       &datacube_pulse_major_padded[chan][range][0],
13.       &datacube_pulse_major_padded[chan][range][0]);
14.
15.  // data layout transform
16.  for (chan = 0; chan < N_CHAN; ++chan)
17.    for (range = 0; range < N_RANGE; ++range)
18.      for (dop = 0; dop < N_DOP; ++dop)
19.        doppler_data_cube[chan][dop][range] =
20.    datacube_pulse_major_padded[[chan][range][dop]);
```

图 1 快速傅里叶变换原始代码和胶水代码

指外围调用 FFT 的循环(第 2 个代码块)以及执行数据重排的循环(第 3 个代码块)。

首先,针对第 1 个代码块中的函数 fftw _ plan _ dft _ 1d 进行分析。该函数的作用是计算大小为 *N _ DOP* 的一维 FFT。FFT 本身是一个原位操作,因为输入指针和输出指针都指向相同的内存地址(由第 2 个和第 3 个参数可得)。

图 1 中的第 2 个代码块,按照代码块 1 得到的 plan _ fft 进行运算。函数 fftw _ execute _ dft 表示,在不同的输入/输出数组(array)上执行多个 FFT。输入/输出地址使用包围 fftw _ execute _ dft 的两层循环进行索引,而这些循环代码即上文所述胶水代码。对开发以及维护程序员来说,明晰这些胶水代码实际进行的操作,是能否进行后续开发维护的关键点。

最后一个代码块是胶水代码的另一个例子。在库函数调用后,需要执行数据重排。更具体的,图中第 3 个代码块中,循环包围着一个简单数据拷贝,即输入基地址为 datacube _ pulse _ major _ padded,输出基地址为 doppler _ data _ cube 的数据重排。由于不同硬件架构的基础内存排列方式可能不同,导致这些循环必须以高效且不同的方式进行实现(硬件的特点,例如内存交织,对齐方式等),以在不同平台上实现良好性能。

针对图 1 中的代码逻辑,实际上可以使用 FFTW 函数库中更为高效的 fftwf _ plan _ guru _ dft 函数对其进行优化和改写。如果将循环和数据重排融合到单一的库函数调用中,如图 2 所示,首先,胶水代码大量减少,将计算执行和访存处理重叠起来隐藏了部分时间开销;最后,同样因为胶水代码的减少,程序访存压力得到缓解,提升程序运行性能、降低应用程序运行能耗(如图 3 所示)。图 3 为在 Haswell 架构处理器(Intel Xeon E5-4667V3,40 MB L3 Cache,16 核,32 线程)的服务器上的测试性能对比,从图中可以看出,相对于图 1 中的代码,融合优化后的图 2 中的代码,在性能、能耗与能耗延迟积 3 个方面都有显著的性能优化。更重要的是,由于胶水代码被替换为了对标准库(此情况下为 FFTW guru)的单一库函数调用,使得优化重构后的代码具有更好的性能可移植性,即程序员不用再根据硬件平台基础架构做类似于优化循环以及内存拷贝的手动优化。

```
1.  //planning
2.  fftwf_ plan plan_ fft;
3.  const fftwf_ iodim dims[1] = {{N_DOP, 1, 1}};
4.  const fftwf_ iodim howmany_ dims[2] =
5.        {{N_RANGE, N_DOP, N_DOP},
6.        {N_CHAN, N_RANGE * N_DOP, N_RANGE * N_DOP}};
7.
8.  // FFT operation
9.  plan_ fft = fftwf_ plan_ guru_ dft(1, dims,
10.       2, howmany_ dims,
11.       datacube_ pulse_ major_ padded,
12.       doppler_ data_ cube,
13.       FFTW_FORWARD, FFTW_PATIENT);
14.
15. // execute plan
16. fftw_execute(plan_ fft);
```

图 2　对胶水代码融合优化后的版本

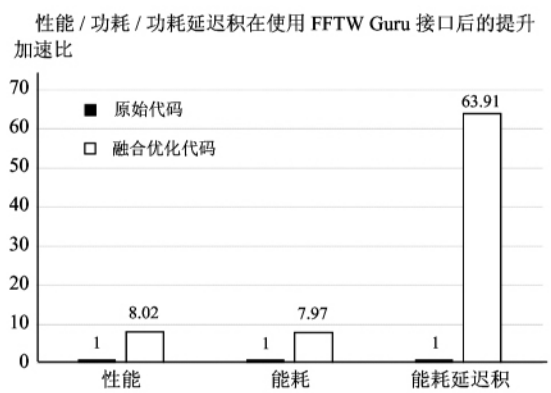


图 3　融合胶水代码和库函数调用,性能提高、能耗降低、能耗延迟积降低

综上分析,高性能计算程序中调用的函数通常来自高性能函数库,例如 BLAS 和 FFTW。在调用多个库函数编写高性能计算应用时,需要编写大量胶水代码以保证应用程序功能与性能。而胶水代码的存在,导致应用程序维护困难,程序访存量增加,最终导致应用程序性能可移植性变差。针对胶水代码进行优化,减少不必要的循环和内存拷贝,使用更为

高效的函数调用替换原始函数调用以及胶水代码以优化程序,可以使得应用程序更充分利用高性能函数库所提供的优化,使应用程序具有更好的性能可移植性。

2 DSL 语言和编译器

针对高性能计算程序在调用高性能计算函数库的过程中,需要插入大量进行循环和数据重排的胶水代码,以至应用程序性能可移植性差的问题,本文构建了一个原型 Source-To-Source 编译器。它能将基于高性能函数库的程序解析为一种作为中间表示的领域特定语言,然后对中间表示进行分析优化,最终得到使用更高效的库函数替代循环、数据拷贝和原始的库函数调用的 C 代码。对于无法通过该方法进行优化的循环,该编译器尝试使用 OpenMP 或 OpenACC(取决于目标架构)来插入编译制导语句。图 4 为该编译器框架图。

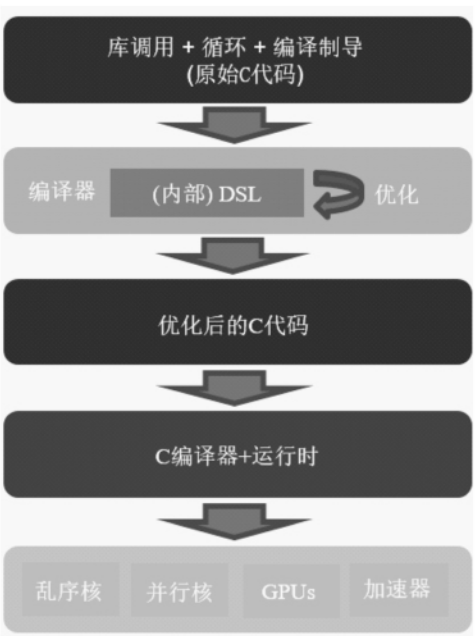


图 4 编译器架构示意图

在 Source-To-Source 编译器实现中,本文先定义了一个改编自 The Spiral Language[16] 的通用的领域特定语言(domain-specific language, DSL)表示。在原始代码重构和移植的过程中,原有的 C 代码被编译器前端翻译为关于库的领域特定语言。随后使用了一系列在 DSL 之上的循环展开、循环交换、循环自动向量化等的优化,从而得到更好的访存局部性和并行化加速。同时制定了可扩展的模板匹配规则,用更高效的高性能库函数对应的 DSL 原语替换中间 DSL 表示。在编译器后端代码生成模块中,将优化后的 DSL 翻译成优化后的 C 代码。对于不能用高性能计算库函数优化,但可以进行并行化加速的代码,插入编译制导语句,通过 OpenMP 等并行加速库优化用户的原始程序。

2.1 DSL 语言表示和编译器前端

DSL 语言的设计目标是尽可能充分的表达原始 C 代码中的数据分布描述与数学运算表示,同时能够满足后续优化的需求。图 5 提供了用巴科斯-诺尔范式描述的 DSL 语言的形式化语法。

<DSL> ::= <MATH> | <SYM> | <TPL>
<DSL> + <DSL> |
<DSL> ⊗ <DSL> |
…
<MATH> ::= *number* | *math func* | $vector(a_0, \ldots, a_{n-1})$ | …
<SYM> ::= I_n | L_n | …
<TPL> ::= FFT | FFTW_guru | gemv | gemm …

图 5 DSL 形式化语义描述

DSL 有最基本的数学运算表示功能。它能表达一些常数,比如 2、3/7、2.33、π,也能表达一些简单的数学函数运算,如 exp(2)、sin(π/2)等。特殊的,对于在高性能计算领域中比较常见的向量或者矩阵运算,DSL 拓展了一些如 $vector(a_0, \cdots, a_{n-1})$ 之类的原语,从而高效地表示数学运算。

DSL 还包括一些高性能计算库函数原语 <TPL>,它们与以 FFTW 及 BLAS 为代表的常用高性能库中的函数调用存在对应关系,从而方便编译器后端代码生成模块的一对一转换。例如,图 1 中的函数 fftw _ plan _ dft _ 1d 由如下原语表示:

$\text{fftw\_plan\_dft\_1d}^{\text{inplace}}_{N\_DOP}$

此外,本文抽象了一些符号与运算符,用来精确表示 DSL 语言中数学计算的行为。在如下的符号运算符例子中:

L^N_m, $(I_n \otimes A)$, $(A^{\circ}B)$

其中，L_m^N 是 N 个元素的排列，m 跨距的元素被排列为连续存储地址，以 I_n 表示数据的 n 个不相交子集，$(I_n \otimes A)$ 表示库函数 A 应用于数据的 n 个不相交子集的每一个子集上，而 $(A \circ B)$ 表示调用函数 B 之后，调用函数 A。简单而言，这些运算符可以被理解为：L_m^N 为一个数据重排操作，或数据拷贝操作，$(I_n \otimes A)$ 是一个围绕库函数的循环，$(A \circ B)$ 对应连续的库函数调用。

基于上述原语介绍与扩展，实现了基于 GAP 4[17] 的编译器前端，完成从原始 C 代码到 DSL 中间表示的转换工作。GAP 4 是在离散代数领域中使用广泛的语言系统，它提供了基本的描述复杂代数符号、函数的数据结构，且方便扩展。通过上述编译器前端，能够将图 1 中的代码片段翻译为如下的 DSL 表示：

$$(I_{N\_CHAN} \otimes L_{N\_RANGE}^{N\_RANGE \cdot N\_DOP}) \circ (I_{N\_CHAN} \otimes (I_{N\_RANGE} \otimes \mathrm{fftw\_plan\_dft\_1d}_{N\_DOP}^{\mathrm{inplace}})) \tag{1}$$

2.2　优化及代码生成

基于 DSL 的代数表达能力，应用集合基本知识对上面的表达式进行化简和重排，可以得到下面的表达式：

$$(I_{N\_CHAN} \otimes (I_{N\_RANGE} \otimes (L_{N\_RANGE}^{N\_RANGE \cdot N\_DOP} \circ \mathrm{fftw\_plan\_dft\_1d}_{N\_DOP}))) \tag{2}$$

对于重点优化路径，制定了可灵活扩展的模板匹配规则，完成库函数调用之间的优化。

例如，通过 FFTW guru 接口（guru-interface）的模板匹配搜索，式(2)所表示的计算可以被等效替换。这样能够在代码生成时使用单个 FFTW guru 接口调用优化图 1 中代码片段的胶水代码和 FFT 函数调用。

对比图 1 和图 3 可以看出，化简和重排后的方案将局部数据重排紧接在调用 FFT 计算之后，不仅可以将计算执行和访存处理重叠起来隐藏部分时间开销，还可以利用数据的局部性来提升访存效率。

从理论上可以大致获知，原先算法时间复杂度为 $O(n^3 + n^2)$，即 $O(n^3)$，经过表达式优化后，算法时间复杂度可以被降低到 $O(n)$。假设原先的运算中，计算时间为 tc，访存时间为 tm。在经过计算、访存优化后，访存时间为 tc'，且有 $tc' <= tc$，计算访存总时间为 $\max(tc', tm)$。可以得到的理论加速比为 $(tm + tc)/(\max(tc', tm))$。当 $tm = n \cdot tc$ 时，将其代入式(2)，可以获得的加速比至少为 $(n + 1)/n$。

3　应用优化示例

为了证明本文所提出解决方法的可行性，本文分别以较为简单的矩阵乘算法以及一种重要的雷达系统处理算法，space time adaptive processing（STAP）算法作为应用示例来介绍。

3.1　矩阵乘算法优化示例

简单的矩阵乘是科学计算中最为常用的计算。矩阵乘算法有 3 种可行的实现方式：矢量-矢量运算（Level 1）、矩阵-矢量运算（Level 2）、矩阵-矩阵运算（Level 3）。这 3 种实现方式会分别调用 BLAS、ScaLAPACK 中的 axpy、gemv 以及 gemm 函数。使用 axpy 和 gemv 运算拼接矩阵运算时，需要使用多重循环以拼接完整的矩阵乘。图 6 展示了最为简单和典型的算法优化思想。

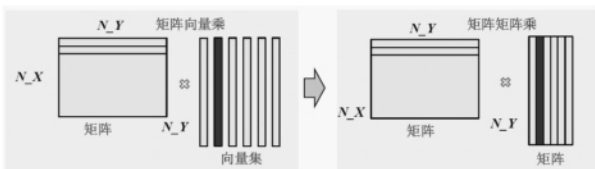


图 6　使用 gemm 替换用户编写的 gemv

用户所编写的应用程序中，为了实现较为灵活的功能，使用了矩阵-向量乘法，而非矩阵-矩阵乘法。在编译器将原始 C 代码翻译成中间表示后，会对中间表示进行依赖分析，尽可能使用更为高效的矩阵-矩阵乘法代替原始的矩阵-向量乘法。

3.2　STAP 算法优化示例

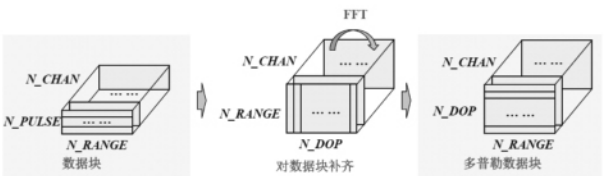
STAP 算法由 4 个阶段组成：多普勒处理（Doppler processing，DP）、协方差矩阵构造（covariance matrix construction，CMC）、计算自适应权重（computing adaptive weights，CAW），以及应用自适应加权（applying adaptive weighting，AAW）。在不同阶段之间，当进行 FFT 相关运算，或者 BLAS、Sca-

LAPACK 中的线性代数运算,则需要进行数据重排、对齐或数据拷贝。

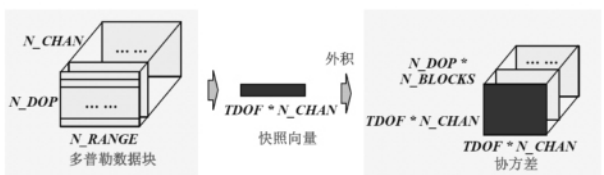
接下来对优化过程进行展开描述。图 7 以图示的形式重点表现该编译器的关键优化内容和核心思想。整个计算过程主要的 3 个优化点如下。

(1)将数据布局与库函数调用合并。DP 阶段如图 7(a)所示,其对应原始代码如图 1 所示。在该阶段的原始运算过程中,中间的 FFT 函数前后存在大量的胶水代码(用于数据重排和对齐的循环)。尝试合并一系列一维 FFT 后的胶水代码,并替换为图 3 中所示的代码。

(2)用矩阵乘法取代外积。在 CMC 阶段,每个协方差矩阵是由累加多个外积(outer-products)得到的,如图 7(b)所示,而每个外积均由多个从数据立方(datacube)中提出的快照向量(snapshot vectors)计算得到。编译器将提取快照向量的过程用一个 rank 为 0 的 FFTW-guru 接口替代。除此之外,多个外积也可以被一个单独的矩阵乘法替代(rank 为 k),如图 8 所示。



(a) DP阶段



(b) CMC阶段

图 7　FFT 和 BLAS 运算后/前存在胶水代码

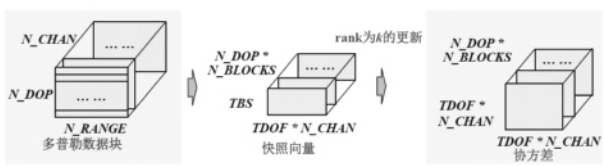


图 8　CMC 阶段优化

(3)跨阶段优化。在 AAW 阶段,该编译器识别提取了相同的快照矢量。由此,上述快照矢量可被复用,不再进行数据提取。

通过上述优化,将原先的胶水代码所执行的数据重排以及循环调用等过程进行了一个更高层次的抽象,变为了一个高级函数调用。

4　实验平台与性能分析

针对第 3 节提到的矩阵乘算法和 STAP 算法,本文在 3 种不同平台上进行了实现。这些平台包括:

(1) Haswell 平台:通用多核处理器(Intel Haswell),Intel Xeon E5-4667V3,40 MB L3 Cache,16 核,32 线程。

(2) Xeon Phi 平台:多核协处理器(Intel Xeon Phi),Intel Xeon Phi Processor 7210,32 MB L2 Cache,64 核,64 线程。

(3) 异构高峰值加速器:一种集成了更多运算器,同时存储端使用 3D 堆叠 DRAM[18] 的研究架构,其结构类似于 FFT 加速器[19,20]、数据重塑单元[21]、Dot Product 硬件。相比 Haswell 平台和 Xeon Phi 平台,其具有更高的硬件峰值算力。

针对 Haswell 平台和 Xeon Phi 平台,算法实现直接调用了公开实现的高性能函数库,例如 FFTW、Intel MKL[22]。而在异构高峰值加速器上,实现了可能调用的原始以及优化接口以供性能对比,例如 FFTW-guru、gemv、gemm 等。进行性能对比的 3 种实现分别为:(1)未优化版本,具有显式循环、数据拷贝和库函数调用的原始 C 代码;(2)编译优化版本,使用更高效的库函数优化循环、内存拷贝后的 C 代码;(3)编译优化 + 并行优化版本,在编译优化版本的基础上添加供 OpenMP/OpenACC 识别的编译制导语句以利用并行加速库进行优化的版本。

4.1　矩阵乘算法性能分析

矩阵乘的原始代码为一份使用 gemv 函数加 for 循环的 C 代码。经过编译器优化后,自动将原始代码转为中间表示,最后再将中间表示翻译为直接使用 gemm 函数实现矩阵乘的 C 代码。

图 9 展示了在 3 种硬件平台上,分别仅进行编译优化性能对比,横坐标为优化方式,纵坐标为优化后相比原始版本的加速比。

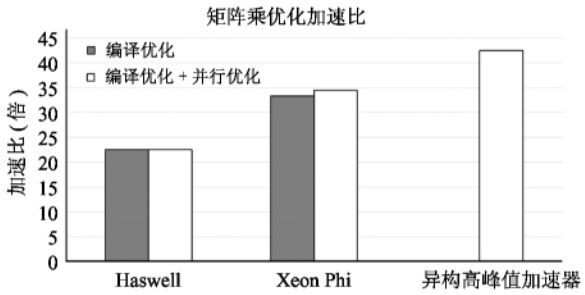


图 9　矩阵乘应用程序在 3 个不同平台上的性能比较

首先,可以看出,3 种硬件平台上,经过编译优化的 C 代码执行效率相比原始 C 代码都有超过 20 倍的提升。其次,在 Xeon Phi 平台和异构高峰值加速器上,使用编译优化后的加速比,相比 Haswell 要更好,这种现象的原因是 3 者的访存带宽几乎一致,但后两者有更高的硬件计算峰值,当减少访存后,硬件效率提升更为明显。最后,在 Haswell 和 Xeon Phi 平台上,是否开启编译器中的并行优化,没有明显的性能提升,这种现象的原因是矩阵乘是一个较为简单的例子,编译优化所作的工作与并行优化所做工作基本类似,所以增加并行优化效果不佳。

4.2　STAP 算法性能分析

针对 STAP 算法的一种原始实现,使用 3 组来自 PNNL PERFECT Benchmark 集合[23]的输入数据,即大(large)、中(medium)、小(small),分别测试原始代码,进行编译优化的代码以及同时进行编译优化、并行优化的代码。

图 10 显示了不同平台上的性能,横坐标为 3 种硬件上不同的数据集,纵坐标为优化后的性能与未优化版本的加速比。首先,在 Haswell 和 Xeon Phi 硬件平台上测试 3 种数据集,仅仅进行编译优化后,优化版本相比原始版本有加速,但不够明显,最高可以取得的加速比是在 Xeon Phi 硬件上对小数据集进行测试得到的 1.39 倍加速。这种现象的原因是整个应用中所调用的函数,可以被编译器识别优化的较少。其次,同时使用编译优化和并行优化后,加速效果较为明显。在 Xeon Phi 硬件和小数据集上可以取得 4.89 倍加速,而在异构高峰值加速器上,可以取得 8.21 倍加速。这种现象的出现是因为原始实现中存在较多可以并行的循环代码。

综合在简单矩阵乘和 STAP 算法两种应用上原型编译器的表现,可以得到的结论是:在简单应用中,本文所实现的原型编译器,可以通过优化代码实现的方式,显著提升应用性能。而在复杂的应用中,该原型编译器,可以通过优化代码实现和插入编译制导语句,使复杂应用不再需要进行繁杂的手工优化。因此,本文提出的中间表示和编译器,可以解决因为胶水代码而产生的高性能计算程序性能可移植性欠佳的问题。

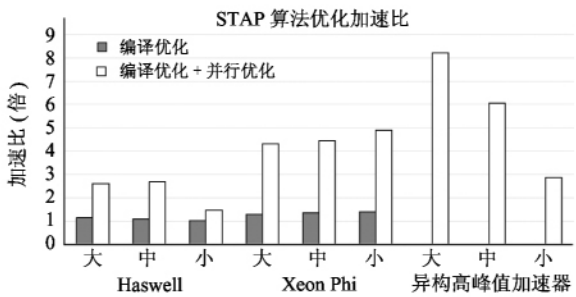


图 10　STAP 应用程序在 3 个不同平台上的性能比较

5　结　论

许多 HPC 应用程序是直接基于标准化库和编译制导语句实现的。由于将多个不同的库函数调用与应用进行链接的胶水代码的存在,高性能计算应用程序性能可移植性(即在不同平台上保持高性能)依然是一个不小的挑战。通常,围绕库函数调用的胶水代码会提供可以进行优化的上下文信息。利用这些胶水代码提供的上下文信息,本文定义和引入了一种作为中间表示的领域特定语言,最终使用更高效的函数调用替代胶水代码和原始的库函数调用,或根据并行硬件架构本身,自动插入供 OpenMP 或 OpenACC 的识别编译制导语句,完成对应用程序的优化。这种方法使得程序员不再需要针对不同硬件平台,对应用程序代码进行复杂的手工优化。本文以简单的矩阵乘算法和较为复杂的 STAP 算法优化过程展示了实际的优化效果。实验结果表明,根据应用程序原始代码的编写以及硬件架构的不同,原型编译器可以取得不同的优化效果。而对于真实应用,在通用处理器硬件架构上,经过优化的代码相比原始代码可以取得最高 4.89 倍的加速比;在类似异构高峰值加速器的实验性硬件架构上,本文所描述的工作可以取得最高加速 8.21 倍。

参考文献

[1] Lawson C L, Hanson R J, Kincaid D R, et al. Basic linear algebra subprograms for fortran usage [J]. *ACM Transactions on Mathematical Software*, 2002, 5(3): 308-323

[2] Dongarra J J, Du Croz J, Hammarling S, et al. An extended set of FORTRAN basic linear algebra subprograms [J]. *ACM Transactions on Mathematical Software*, 1988, 14(1): 1-17

[3] Dongarra J J, Du Croz J, Hammarling S, et al. A set of level 3 basic linear algebra subprograms[J]. *ACM Transactions on Mathematical Software*, 1990, 16(1): 1-17

[4] Choi J, Dongarra J J, Pozo R, et al. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers[C]//The 4th Symposium on the Frontiers of Massively Parallel Computation, McLean, USA, 2003: 120-127

[5] Frigo M, Johnson S G. The design and implementation of FFTW3[J]. *Proceedings of the IEEE*, 2005, 93(2): 216-231

[6] OpenMP A. OpenMP Application Program Interface, v5.0[EB/OL]. https://www.openmp.org/: OpenMP Architecture Review Board, 2018

[7] Openacc. Directives for accelerators. [EB/OL]. https://developer.nvidia.com/openacc/overview: Nvidia, 2016

[8] Puschel M, Moura J M F, Johnson J R, et al. SPIRAL: code generation for DSP transforms[J]. *Proceedings of the IEEE*, 2005, 93(2): 232-275

[9] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. *ACM SIGPLAN Notices*, 2013, 48(6): 519-530

[10] Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning[C]// Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Berkeley, USA, 2018: 579-594

[11] Guyer S Z, Lin C. An annotation language for optimizing software libraries[J]. *ACM SIGPLAN Notices*, 2000, 35(1): 39-52

[12] Chauhan A, McCosh C, Kennedy K, et al. Automatic type-driven library generation for telescoping languages [J]. *Proceedings of the* 2003 *ACM/IEEE Conference on Supercomputing*, 2003, 1(1): 51

[13] Ward J. Space-time adaptive processing for airborne radar [C]// International Conference on Acoustics, Speech, and Signal Processing, Detroit, USA, 1995: 2809-2812

[14] Wicks M C, Rangaswamy M, Adve R, et al. Space-time adaptive processing[J]. *IEEE Signal Processing Magazine*, 2006, 23(1): 51-65

[15] 谢文冲, 段克清, 王永良. 机载雷达空时自适应处理技术研究综述[J]. 雷达学报, 2017, 1(6): 575-586

[16] Franchetti F, Low T M, Popovici D T, et al. SPIRAL: extreme performance portability[J]. *Proceedings of the IEEE*, 2018, 106(11): 1935-1968

[17] The GAP-Group. GAP-Groups, Algorithms, and Programming, Version 4.10.1 [EB/OL]. https://www.gap-system.org/, GAP Groups, 2019

[18] Guo Q, Alachiotis N, Akin B, et al. 3D-stacked memory-side acceleration: accelerator and system design[C]// In the Workshop on Near-Data Processing (WoNDP) (Held in Conjunction with MICRO-47.), Cambridge, UK, 2014: 1-6

[19] Milder P, Franchetti F, Hoe J C, et al. Computer generation of hardware for linear digital signal processing transforms[J]. *ACM Transactions on Design Automation of Electronic Systems*, 2012, 17(2): 1-33

[20] Pu J, Bell S, Yang X, et al. Programming heterogeneous systems from an image processing DSL[J]. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017, 14(3): 26

[21] Akin B, Franchetti F, Hoe J C. Data reorganization in memory using 3D-stacked DRAM[C]// ACM SIGARCH Computer Architecture News, Phoenix, USA, 2015: 131-143

[22] Intel. Math kernel library[EB/OL]. https://software.intel.com/en-us/intel-mkl, Intel, 2015

[23] Barker K, Benson T, Campbell D, et al. Test, assessment and verification effort (TAV) for the DARPA PERFECT program [EB/OL]. http://hpc.pnnl.gov/projects/PERFECT/: Pacific Northwest National Laboratory, 2013

Domain-specific language for improving the performance portability of high-performance computing programs

Li Wei[*,***], Wen Yuanbo[*,***], Sun Guangzhong[*], Chen Yunji[**]

([*] University of Science and Technology of China, School of Computer Science and Technology, Hefei 230026)

([**] State Key Laboratory of Computer Architecture of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([***] Cambricon Tech. Ltd, Shanghai 201203)

Abstract

High-performance computing (HPC) applications are mostly written based on standard function libraries and compiler pragmas, which can improve the programmability and portability of high-performance computing applications. Compared to the traditional optimization method for the optimization of a single function library, the research in this work puts the optimization attention between different function library calls, and proposes a domain-specific language and compiler. The source to source code optimization of original C code solves the problem of poor performance portability of high performance computing programs due to glue code. Experiment results show that in real-world applications, using a compiler that supports the domain-specific language in the field, on the general-purpose processor hardware architecture, an optimization acceleration of up to 4.89 times compared to the original version can be achieved; while in an experimental heterogeneous high-peak accelerator architecture on top, it can be achieved an optimization acceleration of up to 8.21 times.

Key words: high-performance computing (HPC), portability, glue code, domain specific language, compiler