面向异构众核从核的数学函数库访存优化方法

许瑾晨 郭绍忠 黄永忠 王 磊
(解放军信息工程大学数学工程与先进计算国家重点实验室 郑州 450002)

摘 要 数学库函数算法的特性致使函数存在大量的访存,而当前异构众核的从核结构采用共享主存的方式实现数据访问,从而严重影响了从核的访存速度,因此异构众核结构中数学库函数的性能无法满足高性能计算的要求。为了有效解决此问题,提出了一种基于访存指令的调度策略,亦即将访存延迟有效地隐藏于计算延迟中,以提高基于汇编实现的数学函数库的函数性能;结合动态调用方式,利用从核本地局部数据存储空间 LDM(local data memory),提出了一种提高访存速度的 ldm_call 算法。两种优化技术在共享存储结构下具有普遍适用性,并能够有效减少函数访存开销,提高访存速度。实验表明,两种技术分别能够平均提高函数性能 16.08%和 37.32%。

关键词 异构众核,数学函数库,访存优化,指令调度,局部数据存储空间
中图法分类号 TP311 **文献标识码** A

Access Optimization Technique for Mathematical Library of Slave Processors on Heterogeneous Many-core Architectures

XU Jin-chen GUO Shao-zhong HUANG Yong-zhong WANG Lei
(State Key Laboratory of Mathematical Engineering and Advanced Computing,
PLA Information Engineering University, Zhengzhou 450002, China)

Abstract Due to the nature of mathematical function's algorithms, there are a great deal of access operations remaining in reality. In the heterogeneous many-core architectures, which is becoming ubiquitous recently, the slave processors are equipped with shared memory to access data, thereby impacting the accessing rate heavily. Therefore, the performance of the mathematical library's functions is not able to meet requirements of high performance computing. To efficiently solve this problem, this study proposesd a novel accessing instructions based scheduling strategy to cover the access delay with the necessary computation. With the help of the dynamic calling mode, an algorithm called ldm_call was introduced based on the LDM (local data memory) of the slave processors, which can speed up the accessing rate significantly. These two optimizing technologies both possess general applicability in the shared memory. At the same time, they can efficiently reduce the accessing frequency and speed up the accessing rate. The experimental results show that they can improve the functions' performance 16.08% and 37.32% on average respectively.

Keywords Heterogeneous many-core, Mathematical library, Access optimization, Instruction-scheduling, Local data memory

1 引言

数学函数库是处理器配套软件的重要组成部分之一,是高性能计算机平台上科学计算、工程数值计算和专用软件开发所必备的最基础和最核心的软件之一,具有高可靠、高精度和高性能的特点。

数学函数库的算法已趋于成熟,在函数算法实现过程中,通常通过大量的读表(访存)操作维系高性能需求。在此背景下,优化工作则成为数学函数库在不同体系结构下满足高性能需求的研究重点之一。这类研究大致可分为两类:

A. 隐藏访存延迟

当前主流的处理器中,主要有以下几种常用的隐藏访存延迟技术:增加指令窗口(Larger Instruction Window)[1]、乱序执行(Out-of-order Execution)[2]、多线程(Multi-Threading)[3]、软件流水(Software Pipelining)[4]、投机执行(Speculative Precomputation)[2]和数据预取(Prefetching)[5]。其中数据预取技术发展最为迅速。

预取技术是指通过计算和访存时间的重叠来实现隐藏因 cache[6]失效而引起的访存延时,该技术被认为是解决容量失效和强制性失效的一种有效手段。传统的预取技术主要分为软件预取和硬件预取。两者之间的不同在于软件预取主要依靠指令来指定预取的地址,而硬件预取则是通过专用的硬件机制来预测可能会发生的失效。

然而无论是硬件预取还是软件预取都存在着缺陷。硬件

到稿日期:2013-07-31 返修日期:2013-10-26

许瑾晨(1987—),男,博士生,主要研究方向为高性能计算,E-mail:atao728208@126.com;郭绍忠(1964—),女,副教授,硕士生导师,主要研究方向为高性能计算、分布式系统;黄永忠(1968—),男,教授,博士生导师,主要研究方向为大数据分析、分布与并行处理;王 磊(1977—),男,博士生,副教授,主要研究方向为高性能计算。

预取往往过于盲目，预取可能会存在大量的无用数据块，造成cache污染和可能重用的数据块被提前替换。软件预取虽然能够准确预取，但是对于预取的时机把握不够精确，其插入预取指令、计算有效地址等操作，都会造成大量的处理器开销。

B. 利用 cache 层次

硬件由于能够自动实现 cache 管理，因此逐渐得到了发展和利用，利用其缩小存储器和处理器之间速度的差距，是缓解“存储墙”问题的一种重要方法[7-9]。

相对于数据 cache 而言，指令 cache 的性能优化方法更为完善，对整个系统的影响也相对较小。因此对 cache 性能优化的重点依旧是对数据 cache 的优化，如何提高数据 cache 性能已经成为当前研究的一个热点问题。

随着高性能计算机的蓬勃发展，其体系结构也已经从多核逐步发展为现阶段的众核异构结构，这种结构由通用计算主核和精简的计算从核组成，具有超高的定点和浮点峰值性能和较高的实际应用性能，主要应用在高难密码破译领域、信号分析和部分国家科学/工程计算领域等。

为进一步提高异构结构中上层数值应用的性能，增强数学库函数在异构结构下的适用性，发挥众核异构结构的计算性能和增强系统可用性，实现异构结构下的高性能数学函数库显得尤为重要。而在此新型体系结构下，从核数学函数库的优化面临着新的挑战，以使用频繁且所占比重较大的三角及指数对数函数类(此类函数占数学函数库中函数的比例高达 80%)中的部分典型函数为例进行性能测试，在从核平台下的测试结果如表 1 所列。

表 1　从核函数性能

函数	运行节拍数	计算部分	访存部分	访存所占比例
sin	1685	107	1578	93.65%
exp	1316	189	1127	85.64%
cosf	853	72	781	91.56%
log	1028	231	797	77.53%
tanf	1413	163	1250	88.46%
powf	1422	196	1226	86.22%

由此可见，在从核平台上，这些高频次访问的函数性能中平均 87.18%的消耗集中在访存操作，远远超过计算消耗的 12.82%。而在数学函数库中这种现象并非特例而普遍存在，因为三角类函数及指数对数类函数采用的算法均与典型函数类似，都通过大量的访存维系着高性能需求，而此类函数占数学函数库中函数的比例高达 80%。因此，要满足高性能需求，就必须抓住性能消耗的热点，对访存进行重点深入优化。

一方面，从面向异构众核的从核数学函数库这样一个应用软件角度出发，并不适合研究并应用硬件预取；同时，从核处理器并不支持预取指令，也无法进行软件预取。

另一方面，从核不存在通常意义下的 cache 结构，只有一个大小为几 k 的 LDM，并且系统不允许底层函数库对其直接进行写操作，因为这种操作虽然能够获得最佳的性能提升，但在程序运行结束前会一直占用 LDM 空间，从而导致别的程序甚至系统程序都无法利用 LDM 加速。这种方式在实际应用过程中存在的弊端是显而易见的，这是因为：假设现有一个外围应用程序占用 LDM 空间，同时进来一个急需加速的系统程序因没有 LDM 空间而无法加速，导致机器整体性能下降，得不偿失。因此，没有一种有效的方案能够判断当前 LDM 应该给哪类应用程序使用。上述问题促使数学函数库函数的访存操作都以访主存方式存在，严重影响了函数的执行效率(见表 1)。同时，数学函数库函数也无法简单直接地利用 LDM 对访存进行有效优化。

鉴于此，目前上述主流的有效优化方法并不适用于从核数学函数库，需要针对从核结构寻找新的优化方法，以提高函数运行性能，达到高性能需求。

针对数学函数库性能瓶颈及现阶段优化方法不适用于当前结构的问题，本文研究实现了利用指令调度思想将访存延迟隐藏到计算过程中的方法，此方法有别于传统指令调度。传统指令调度以全局最优为目标，而本文提出的方法是一种以访存操作为重点、局部最优为目标的指令调度技术。在当前体系结构下，访存延迟是数学库函数性能提升的瓶颈，围绕访存实现局部最优既有效实现了访存延迟的隐藏，也降低了调度的复杂度。优化工作往往需要步步推进，并与体系结构紧密结合，在深入分析研究从核处理器结构的基础上，抓住从核特有的 LDM，提出了利用从核 LDM 提高访存速度的 ldm_call 算法，将访主存转化为访局存，大幅提升了访存速度，提高了函数性能。

两种从核优化方法相互关联，彼此配合才能最大限度地提高函数性能。一方面，函数中的计算过程是有限的，是否能够完全或尽可能多地实现访存延迟的隐藏，取决于访存操作的多少以及访存操作与计算过程的依赖程度。因此，单靠指令调度并不能实现访存延迟的完全隐藏。另一方面，从核 LDM 存在空间小、使用受限等局限性，从处理器全局出发，数学库函数对其的使用并不是无限制的，这就限制了大范围对函数进行访主存转访局存操作的可能。因此，有必要在进行这一步优化前，采用上一步方法对函数进行优化，以缓解 LDM 的使用压力。

2　基于访存操作的局部指令调度

指令调度[10,11]可以有效地将访存延迟隐藏到计算过程中，它是目前编译优化研究领域的一个热点，同时也是从核数学函数库函数优化的重点之一。

目前，一方面基本上所有的优化编译器中的调度方案对汇编程序都无效；另一方面全局指令调度方法不存在调度重点，实现复杂，造成较高的寄存器使用压力(特别是从核结构下只有 32 个整数和浮点共用的寄存器的情况)。因此如何更好地利用体系结构的特性，提高程序执行效率，实现以访存指令为主的指令调度是本节研究的重点。

执行指令调度实际上是一种对程序指令的重排序变换，而实施重排序变换首先需要检测是否会违反依赖，这就需要考察程序的数据依赖关系，而数据依赖图是依赖关系[12]的直观体现，图能够更准确、更直观地体现依赖关系的信息。

在确定函数数据依赖图的基础上，需要选择适合的基础调度算法，以提高指令调度的效率，降低其实现难度。本文主要借鉴经典的表调度算法思想实现具体的调度过程，具体的表调度算法详见文献[13]。

2.1　调度策略

指令调度问题是 NP-Complete 问题[14]，涉及很多方面的内容，在此并不对其进行全面的研究，而是利用其调度思想，结合函数库函数特点，将调度重点放在以计算为主的基本块内的访存操作上，将一个较复杂的全局优化问题转化为相对简单的局部优化问题。同时，将调度重点集中到程序核心代

码段，形成调度基本块，在函数性能取得有效提升的基础上，大大降低调度工作的复杂性，大幅提高此技术在从核数学函数库中的特殊应用及其实际应用能力。在此，提供一个启发式调度策略 MAISS，具体步骤如下：

1. 首先确定程序核心代码段，定位核心代码段中的访存操作，以访存操作为核心，确定一个用于指令调度的以计算为主的调度基本块（可以由程序中的多个跳转分支构成），并分析此基本块的数据依赖关系。

2. 构造第 1 步中基本块的有向数据依赖图，在依赖图中标记出访存指令对应的节点。

3. 分析访存节点的依赖关系，利用寄存器重命名技术尽可能消除访存指令数据依赖关系。当基本块由程序中单跳转分支构成时，消除前面节点对后继访存节点的数据依赖关系，使得访存节点成为依赖图中的某个可调度根节点；当基本块由程序中多个跳转分支构成时，消除上述依赖的同时，还需尽可能消除跳转分支之间的数据依赖关系。

4. 重新确定数据依赖关系，形成新的数据依赖图。

5. 利用表调度算法遍历数据依赖图中的根节点，实现以访存节点为主的调度。在调度过程中遵循以下的调度原则：

• 当出现多个可选调度指令时，如果在之前的指令序列中没有出现访存指令而在当前可选调度中出现访存指令，则对此访存指令进行调度。如果之前已经存在访存指令，则选择一条非访存指令进行调度。

• 在调度过程中要兼顾指令的执行延迟，将指令调度到适当位置即可。由于寄存器资源的限制，当访存操作和数据实际使用操作相隔较远时，会长时间占用寄存器资源，降低寄存器的使用率，严重情况下将导致寄存器资源不够。基于此点，并非将访存操作调度得越远越好，而是以能恰当地隐藏访存执行延迟为最佳。

6. 重复第 5 步直到数据依赖图为空，即所有指令执行序列都被生成。

与当前广泛应用于编译器的指令调度方法相比，MAISS 具有更强的现实意义，特别是对于解决从核数学函数库的访存瓶颈问题。该方法有效弥补了编译器无法对基于汇编语言实现的数学函数库进行优化的局限，通过对存在访存指令的基本块的指令调度，获得了函数整体性能的提升。

2.2 调度实例

函数库函数中为指令调度确定的基本块可以分为两类：一类是以程序中单个跳转分支为基础的基本块；另一类是以程序中多个跳转分支为基础的基本块。

2.2.1 单分支调度

图 1 所示是 atan 函数主体计算的核心代码段，以此形成一个调度基本块，此基本块的确定即完成了调度策略的第 1 步，在此基础上进行调度策略的第 2 步，确定此基本块的数据依赖图，如图 2 所示。

从图 2 中可以很直观地分析出节点之间的数据依赖关系，其中灰色节点为标记好的访存操作节点。此基本块包括 14 个节点，相互之间存在真依赖 δ 有 16 个，反依赖 δ^{-1} 有 5 个，输出依赖 δ^{0} 有 4 个。以节点 S11、S12、S13 和 S14 为例，S11 与 S12 存在真依赖关系，S12 与 S14 存在真依赖关系，S13 与 S11 存在输出依赖关系，S13 与 S12 存在反依赖关系，S13 与 S14 存在真依赖关系，即这 4 个节点的执行顺序只能是 S11、S12、S13、S14，无法进行调度。

```
S1:vshuffle     a5,a5,0x44,a5
S2:vmuld        a5,a5,t5
S3:vmuld        t5,t5,a5
S4:vldd         t0,80(t2)
S5:vldd         t1,64(t2)
S6:vmad         t0,a5,t1,t0
S7:vldd         t3,48(t2)
S8:vldd         t4,32(t2)
S9:vmad         t3,a5,t4,t3
S10:vshuffle    t0,t3,0xee,t7
S11:addd        t3,t7,t8
S12:addd        t8,t8,a4
S13:vldd        t8,96(t2)
S14:fmad        a4,t8,t7,a2
```

图 1　atan 函数核心代码段

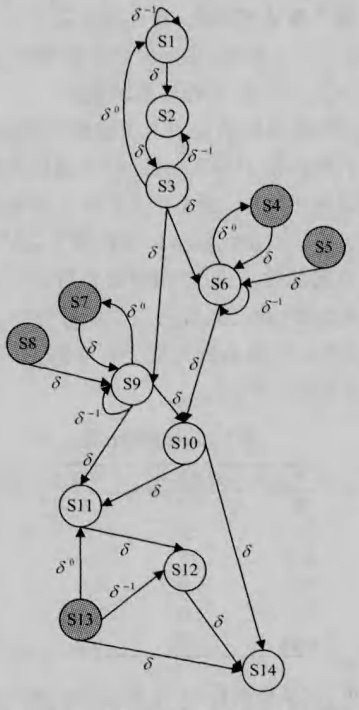

图 2　图 1 核心程序段对应的数据依赖图

通过对依赖图的分析，可以确定访存节点的可调度性，并进入调度策略的第 3 步。利用寄存器重命名技术消除存在较多依赖关系的访存节点 S13 的依赖关系，以达到调度要求，实现寄存器重命名之后的基本块及依赖图如图 3 和图 4 所示。

```
...
S10:vshuffle    t0,t3,0xee,t7
S11:addd        t3,t7,t9
S12:addd        t9,t9,a4
S13:vldd        t8,96(t2)
S14:fmad        a4,t8,t7,a2
```

图 3　图 1 经过寄存器重命名后的程序段

经过寄存器重命名之后访存操作节点 S13 的依赖关系明显减少，只存在一个与 S14 节点的真依赖关系，有利于下一步的调度。

在新的依赖图的基础上，利用表调度思想，实现基本块的指令调度，调度以访存指令为主。

其中，δ 表示真依赖，δ^{-1} 表示反依赖，δ^{0} 表示输出依赖。

经过调度后的指令执行顺序为：S4、S5、S7、S8、S1、S2、

S3、S6、S9、S13、S10、S11、S12、S14。

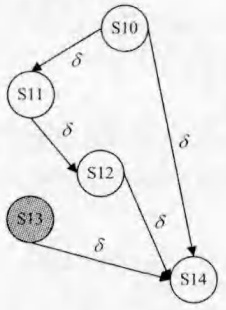

图 4　图 3 程序段对应的数据依赖图

2.2.2　多分支调度

图 5 是多分支调度基本块示例，此基本块包含 3 个分支。

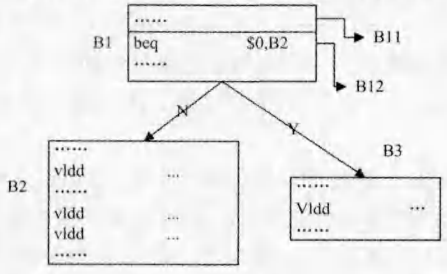

图 5　多分支调度示例

在对此类基本块进行调度时，利用 2.1 节中提出的调度策略，将分支 B1、B2 和 B3 看成一个整体进行数据依赖分析，形成数据依赖图并消除依赖关系，经过访存指令调度，最终可以得到较理想的运行结果。但这样的处理大大增加了数据依赖关系的分析和处理，不利于调度的实现。鉴于此，在进行数据依赖分析前，对访存指令的最终调度位置进行预判断，根据预判断结果，进行分类处理，判断结果以访存指令最终位置为准。

情况 1　当 B2 中的访存指令(B2v)在 B2 中，B3 中的访存指令(B3v)在 B3 或 B12 中时，分别对分支 B2 和分支 B12、B3 进行数据依赖分析，当作单分支调度处理。

情况 2　当 B2v 在 B11 中，B3v 不在 B11 中时，分别对 B11、B2 和 B12、B3 进行数据依赖分析，并进行调度。

情况 3　当 B3v 在 B11 中，B2v 不在 B11 中时，分别对 B2 和 B1、B3 进行数据依赖分析，并进行调度。

情况 4　当 B2v 和 B3v 都在 B11 中时，对 B1、B2 和 B3 进行整体数据依赖分析，并直接调度。

针对多分支调度，为了能够有效降低调度的实现难度，同时达到应有的调度效果，预判断结果的准确性尤为重要。根据指令的延迟，以及计算部分和访存指令的相对位置，对调度的最终位置进行预判断，此判断依赖于编程经验。

多分支调度根据预判断结果确定数据依赖分析方式后，其调度处理及实现方式与单分支调度处理方式相同。

从上述调度实例的分析可知，本文的指令调度为局部指令调度，以包含访存指令的基本块为调度单位，数据依赖关系相对简单，在获得较大函数性能提升的同时更易于实现，与全局指令调度方法相比，以小代价可获得高回报。小代价主要体现在以下 3 个方面：指令间依赖关系的刻画更简单，参与调度的指令数大大减少；明确了被调度指令为访存指令(这是由从核数学函数库访存慢的特点决定的)，更易于实现；将增加寄存器使用压力的可能降到了最低，在提高函数性能的同时缓解了寄存器使用压力。高回报则主要体现在函数性能的提升。

3　基于阈值 N 和申请空间的 ldm_call 算法

通过访存指令调度优化后，我们总希望所有的访存延迟都隐藏在计算过程中，但在实际过程中还是会有相当部分的访存操作无法通过上述方法进行优化：一方面，无法有效地消除所有访存指令的反依赖关系，不利于调度；另一方面，函数计算有限，无法有效隐藏访存延迟。对于这类情况，从代码层次出发已经没有较好的解决方法，而从核处理器特有的高速局部存储 LDM 却带来了进一步优化的可能。

用户程序对 LDM 的使用主要以动态调用方式为主，本文结合数学库函数访存操作多为读取数据表数据触发的特点，研究实现了基于申请阈值 N 和申请空间大小相结合的 ldm_call 算法。算法基本思想是根据数据表特征，确定申请空间大小，并以 N 为最大申请次数，申请 LDM 空间，若申请成功则发起 DMA，实现数据表数据的传送，最后释放 LDM 空间；否则调整申请空间大小，重复上述申请过程。算法具体过程如下：

```
//功能:实现对 LDM 空间的动态申请及数据批量传输
//输入:待交换数据表大小 size_mem,申请次数 N
//输出:数学库函数运行结果
1. size_ldm=0,add_ldm =0;//初始化
2. add_mem==get_add();//获取数据表首地址
3. while ((size_mem!==size_ldm)&&(N>0)) do
4. N--;
5.    slave_ldm_malloc(size_mem);// 申请 LDM 空间,并返回首地址
      及空间大小
6.    if(add_ldm==NULL) goto 4
7. end
8. if(N==0) DEC(size_mem);//减小申请空间
9.    else if (return(DEC())) goto 3
10.       else goto 16
11. end
12. if(size_mem==size_ldm)
13.    DMA(size_men add_ldm add_mem)//实现数据批量传输
14.    if(dma_reply==0) goto 13//判 DMA 回答字
15. end
16. 函数调用
17. slave_ldm_free()
18. end
```

其中，DEC()函数的功能是选取将要进行数据交换的数据表空间大小，存在可选空间大小，则返回空间大小的值，否则返回 0。根据各函数算法及数据表特征，我们分析确定了各函数的可选数据表大小，以供 DEC()函数调用，如 sin 函数存在的可选数据表大小为(4k，2k，512B 和 128B)。数据表大小确定的原则是：优先考虑整体数据表(首先选择 4k)，其次根据数据被访问频次逐渐降低选取范围(从 2k 到访问最为频繁数据 128B)。

该算法与传统动态调用算法的最大区别在于空间申请阈值 N(每次调用可申请 ldm 空间的最大次数)和 DEC()函数的引入，它们的存在使得 ldm_call 算法能够在当前状态下尽最大可能申请到最大的可用空间，能够有效保证当前的优化效果。空间申请的成功与否、函数性能提升的高低与申请空间大小 *size_mem* 及空间申请阈值 N 都有着密切的关系。*size_mem* 越大，空间申请成功率越低，但申请成功后函数性

能提升越大；N 越大，空间申请成功率越高，但函数性能提升越小。因此，需要在 *size_mem* 和 N 之间找到一个平衡点，以最大限度地发挥函数性能。

假设 $size\_mem=x$，$N=K$，函数性能为 P，数据传输消耗的性能为 $D(x)$，每次成功申请到 LDM 空间的概率是 $P_i(x)$，每次申请所消耗的性能是 $A_i(x)$，申请成功后（不包括申请过程中的性能消耗）能够提升函数性能 $f(x)\%$，则此方法有效的条件为：$P*f(x)\%>A_1(x)+\cdots+A_j(x)+D(x)$，其中 j 的含义为第 j 次申请成功。

此状态下，函数性能可能获得的最大提高为 $P*f(x)\%-\mathrm{Max}(P_1(x),(1-P_1(x))P_2(x),(1-P_1(x))(1-P_2(x))P_3(x)\cdots(((1-P_1(x))\cdots(1-P_{(k-1)}(x)))P_k(x))(A_1(x)+\cdots+A_m(x))$，其中 m 的含义为 Max() 函数中最大值所对应的 $P(x)$ 函数的最大下标。

4 实验结果与分析

主要针对访存指令调度策略 MAISS 对函数性能的提升效果及 ldm_call 算法相关参数设置等进行了详细的测试分析，以验证本文方法的有效性及实用性。测试平台为异构众核的从核结构，其处理器主频为 1200MHz，内存主频为 800MHz，内存容量为 1.8GB。

4.1 调度测试

利用访存指令调度对函数库函数进行优化后，测试结果如图 6 所示。

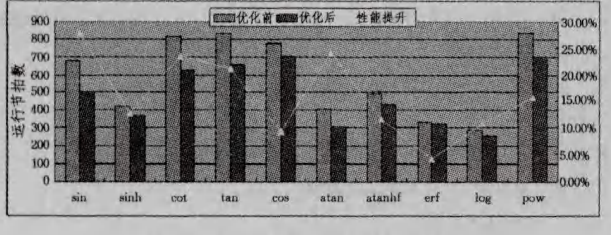

图 6 访存指令调度性能对比测试

测试结果表明，利用 MAISS 对函数库进行优化，函数性能平均提升 16.08%。在测试中包括通过单分支调度和多分支调度两种方式实现的函数，其中通过单分支调度实现的函数的性能平均提高 25.12%，通过多分支调度实现的函数的性能平均提高 12.21%。两种情况下具体性能提升的分布情况如图 7 所示。

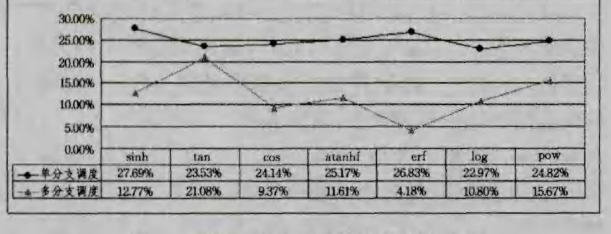

图 7 两种调度方式下性能提升的对比

不难发现，利用多分支调度实现的函数的性能提升总体低于利用单分支调度方式实现的。其中函数 tan 的性能提升水平接近利用单分支调度方式，这主要是因为在 tan 函数中，虽然调度方式属于多分支调度范畴，但在实际应用过程中访存的调度还是处在单分支中，属于多分支调度的第 1 种情况；函数 erf 性能提升较低，这是由调度方式决定的，它属于多分支调度的第 4 种情况，此种情况涉及到复杂的分支间依赖处理，限制了性能提升。所以，在函数优化过程中，总是以处理单分支调度为主，尽可能将多分支调度转化为单分支调度，以获得尽可能多的性能提升。

4.2 ldm_call 测试

利用 ldm_call 算法对函数进行优化后，能够有效提高访存速度，提高函数运行性能，经过测试可以发现函数性能提升明显，如表 2 所列。

表 2 优化前后性能对比测试

函数	Mem_size	N	Clocks (Before)	Clocks (after)	Improvements
sin	4k	3	491	262	46.64%
sinh	2k	1	369	247	33.06%
cot	3k	2	624	382	38.78%
tan	1k	1	655	423	35.42%
cos	4k	3	786	529	32.70%

测试表明，此方法对函数性能提升帮助巨大。同时要取得这样的优化效果，还需要更多地关注 *mem_size* 和 N 的大小。

mem_size 和 N 的变化与函数本身的性质是分不开的，要在每个函数上取得优化效果的最大化，就需要根据不同的函数确定相应的 *mem_size* 和 N 值，以 sin 函数为例，其相互关系如图 8 所示。

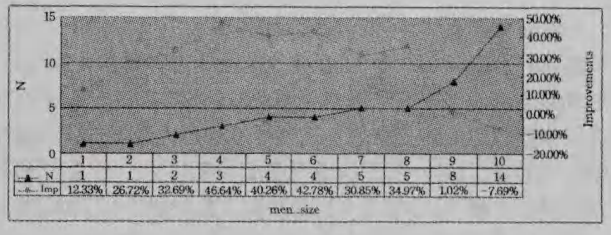

图 8 *mem_size* 与 N 的变化关系及性能提升图

图中横坐标为 *mem_size*，纵坐标为 N，函数性能的变化随着 *mem_size* 和 N 的变化而变化。当 *mem_size* 为 4k，N 为 3 时，函数取得最大的性能提升；当 *mem_size* 变大时，则需要更多次的申请才可能在申请成功的同时获得函数性能提升，在 SIN 函数中当 *mem_size* 大于等于 10k 后，对 LDM 空间的申请基本不会获得成功，即使成功，函数性能也会出现下降，因为多次申请所消耗的性能已经大于数据加载到 LDM 空间中获得的性能提升。

通过对函数库函数的整体分析和对实验结果的整理，可以得到这样的结论，即 *mem_size* 最大不宜超过 7.5k，N 最大不宜大于 5，否则函数加速效果不明显，甚至取得相反的加速效果。

进一步，与传统的直接调用（没有 N 和 DEC() 的介入）方法相比，ldm_call 算法具有更强的优化效果，同时从数学库整体来看，具有更强的稳定性，测试结果如表 3 所列。

表 3 ldm_call 算法与传统调用方法对比测试

函数	ldm_call			传统调用		
	Mem_size	N	Improvements	Mem_size	N	Improvements
sin	4k	3	46.64%	4k	2	47.24%
sinh	2k	1	33.06%	3k	>10	—
cot	3k	2	38.78%	4k	5	24.68%
tan	1k	1	35.42%	5k	>10	—
cos	4k	3	32.70%	6k	9	−7.52%

注：—表示申请失败。

测试结果表明，传统调用方法对函数性能的影响更多地受 *mem_size* 的影响，且较容易出现申请失败的情况，这主要是由于当前运行状态下 *mem_size* 过大。通过测试可知，如直接使用传统的调用方法，将有 37.25%的函数无法成功申请到 LDM 空间。同时，传统方法也容易出现不稳定性，如 sin 函数基本保持一致；cot 函数则通过更多的申请次数申请到了更大的 LDM 空间，但函数性能并没有获得最大的提升；cos 函数由于申请次数增大，数据传输时间和申请消耗时间的总和大于 LDM 空间给函数带来的性能提升，最终导致函数性能反而有所下降。

结束语 基础函数库在数值计算中发挥着巨大的作用，为大型数值应用提供有效的保障，其性能的高低往往直接或间接地影响着大型应用的实际运行效率。为了提高对基础函数库高性能的要求，本文深入了解体系结构特点，针对体系结构对函数性能的限制，实现了两种有效的访存优化方法，取得了较显著的成效，并将之实际应用于异构从核系统中。

目前，本文主要利用静态程序分析及大量的实验测试实现对核心代码段（计算密集基本块）和访存指令调度位置的判断。而对于高性能的追求是无止境的，同时对于进一步提升函数性能的技术方法也还有很多。但沿着本文的研究方法和思路，下一步的研究重点将主要集中在这两个方面，以实现更精确有效的分析方法，使访存调度工作更精细化，进一步提高函数性能。另外，结合系统实际运行情况，从理论角度实现对 *mem_size* 和 N 关系的分析验证，也是一个新的研究方向，它将为异构众核结构下从核函数库的优化方法提供理论保证。

参 考 文 献

[1] Zhou Hui-yang, Conte T M. Performance modeling of memory latency hiding techniques[R]. Technical report, ECE Department, N. C. State University, January 2003

[2] Lebeck A R, Koppanalil J, Li T, et al. A large, fast instruction window for tolerating cache misses[C]//Proceedings of the 29th International Symposium on Computer Architecture(ISCA'02). Anchorage, Alaska, USA, IEEE Computer Society, 2002: 59-70

[3] Wang P H, Wang H, Collins J D, et al. Memory latency-tolerance approaches for itanium processors: out-of-order execution-vs. speculative precomputation[C]//Proceedings of the 8th International Symposium on High Performance Computer Architecture(HPCA'02). Boston, Massachusettes, USA: IEEE Computer Society, 2002: 187-196

[4] Beyls K, D'Hollander E. Compiler generated multithreading to alleviate memory latency[J]. Journal of Universal Computer Science, 2000, 6(10): 968-993

[5] 贺红，朱大铭，马绍汉. 用神经网络求解时间依赖网络最短路径问题的新算法[J]. 复旦学报：自然科学版，2004，43(5)：714-716

[6] Raman E, Hundt R, Mannarswamy S. Structure layout optimization for multithreaded programs[C]//Proceedings of the International Symposium on Code Generation and Optimization (CGO'07). San Jose: IEEE Computer Society, 2007: 271-282

[7] Lattner C, Adve V. Auto-matic pool allocation: improving performance by controlling data structure layout in the heap[C]//Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05). Chicago, IL, USA: ACM Press, 2005: 129-142

[8] 黄安文，高军，张民选. 多核处理器非一致 Cache 体系结构延迟优化技术研究综述[J]. 计算机研究与发展，2012，49(S1)：118-124

[9] 李浩，谢伦国. 片上多处理器末级 Cache 优化技术研究综述[J]. 计算机研究与发展，2012，49(Suppl)：172-179

[10] 余磊，刘志勇，宋风龙. LU 分解在众核结构仿真器上的指令级调度研究[J]. 系统仿真学报，2011，23(12)：2603-2610

[11] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures, A Dependence-Based Approach[M]//Elsevier Science, 2004: 47-374

[12] Zhao Jie, Zhao Rong-cai, Han Lin. A Nonlinear Array Subscripts Dependence Test[C]//Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communications(HPCC'12). Liverpool, IEEE Computer Society, 2012: 764-771

[13] Rau B R, Fisher J A. Instruction level parallel-processing: history, overview and perspective[J]. The Journal of Supercomputing, 1993, 7(1): 950

[14] Garey M R, Johnson D S. Computers and Intractability: A Guide to the Theory of NP-Completeness[M]. Freeman W H. Co, San Francisco, 1979

(上接第 11 页)

[17] Yee Ka-ping, Fisher D, Dhamija R, et al. Animated exploration of dynamic graphs with radial layout[C]//Proceedings of the IEEE Symposium on Information Visualization. 2001: 43-50

[18] Book G, Keshary N. Radial Tree graph drawing algorithm for representing large hierarchies[D]. University of Connecticut, December 2001

[19] Sheth N, Cai Q. Visualizaling mesh dataset using radial tree layout [R/OL]. Spring 2003 Information Visualization Class Project, Indiana University, 2003

[20] Douma M, Ligierko G, Ancuta O, et al. SpicyNodes: Radial Layout Authoring for the General Public[J]. IEEE Transactions on Visualization and Computer Graphics, 2009, 15(6): 1089-1096

[21] Card S K, Suh B, Pendleton B A, et al. Time Tree: Exploring time changing hierarchies[C]//IEEE Symposium on Visual Analytics Science and Technology (VAST'06). 2006: 3-10

[22] Tu Y, Shen H-W. Visualizing changes of hierarchy information using treemaps[J]. IEEE Transactions on Visualization and Computer Graphics, 2007, 13(6): 1286-1293

[23] Burch M, Beck F, Diehl S. Timeline Trees: Visualizing sequences of transactions in information hierarchies[C]// International Working Conference on Advanced Visual Interfaces (AVI'08). 2008: 75-82

[24] Burch M, Diehl S. TimeRadarTrees: Visualizing dynamic compound digraphs[J]. Computer Graphics Forum, 2008, 27(3): 823-830

[25] http://prefuse.org

[26] Heer J, Card S K, Landay J A. Prefuse: a toolkit for interactive informa-tion visualization[C]//Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. New York: ACM Press, 2005: 421-430

[27] 孙宁伟，刘海峰，赵瑜，等. TVBPS：一种基于 Parallel Sets 的具有度量属性的多变元时态数据可视化方法[J]. 计算机应用研究，2014，31(5)：1591-1596