第40卷 第4期 计 算 机 科 学 Vol. 40 No. 4
2013年4月 Computer Science Apr 2013

典型编译器自动向量化效果评估与分析

李春江 黄娟娟 徐 颖 杜云飞 陈 娟
(国防科学技术大学计算机学院计算机研究所 长沙 410073)

摘 要 SIMD(Single-Instruction-Multiple-Data)体系结构在现代处理器体系结构中扮演重要的角色。多种国产高性能通用处理器也大都实现了SIMD结构。SIMD体系结构提供了短向量数据并行处理能力,编译器自动向量化是应用程序获得性能提升的主要手段之一。使用成熟的支持SIMD的商用处理器平台评估典型编译器自动向量化的效果,对于处理器体系结构的设计以及编译器的分析和设计非常有益。采用SPECCPU2006和SPECOMPM2001基准测试程序,评估了典型编译器(包括Intel编译器、PGI编译器和GCC编译器)的自动向量化的效果。并且以产品级的开源编译器GCC为目标,用手工编写的程序片段(主要是多种类型的循环结构)评估了当前GCC编译器自动向量化的效果,并深入分析了GCC编译器中现有的自动向量化的能力和局限。此项工作为进一步研发高效的编译器自动向量化提供了有价值的参考。

关键词 自动向量化,典型编译器,GCC,评估与分析
中图法分类号 TP314 **文献标识码** A

Evaluation and Analysis of Effects of Auto-vectorization in Typical Compilers

LI Chun-jiang HUANG Juan-juan XU Ying DU Yun-fei CHEN Juan
(Institute of Computer, School of Computer Science, National University of Defense Technology, Changsha 410073, China)

Abstract SIMD(Single-Instruction-Multiple-Data) architecture plays an important role in the architecture of modern processors. Also, it is supported in multiple types of homemade high performance general purpose processors. For exploiting the data-parallel performance potentials of the short vector processing ability presented by the SIMD architecture, the auto-vectorization of compilers is one of the main means for improving the performance of applications. Evaluating the effects of auto-vectorization in typical compilers using mature commercial general purpose processor with SIMD support, is beneficial both to processor architecture design and to compiler analysis and design. We evaluated the effects of auto-vectorization in the typical compilers (including Intel compiler, PGI compiler and GCC compiler) with the standard benchmarks SPECCPU2006 and SPECOMPM2001. Then taking the open source product level GCC compiler as the target, we thoroughly evaluated the effects of auto-vectori-zation in GCC with hand-coded program segments (mainly multiple types of loops), and we analyzed the ability and limitations of the current implementations of the auto-vectorization in GCC. Our work provides a valuable contribution for the research and development of auto-vectorization in compilers.

Keywords Auto-vectorization, Typical compilers, GCC, Evaluation and analysis

1 引言

SIMD(单指令多数据)体系结构是非常普遍而高效的数据并行结构,它可以在不提升取指令、指令解码带宽的基础上提高执行吞吐率[1]。目前几乎所有的通用微处理器都实现了SIMD扩展,众多国产CPU也支持SIMD扩展。目前一个重要的发展趋势是处理器支持的SIMD数据宽度更宽、数据元素精度更高、SIMD指令集的指令数更多、功能更强大,如Intel在最新的基于32nm工艺实现的X86_64处理器(研发代号Sandy Bridge)中支持AVX[2](Advanced Vector eXtension,高级向量扩展),最大支持四路双精度SIMD计算。

SIMD指令可以看作是向量长度较短的向量指令,有较多的研究工作致力于将自动向量化引入面向SIMD的编译优化技术中。多种商用编译器和开源编译器已经支持面向SIMD指令的自动向量化。基于通用的支持SIMD的处理器平台,可评估典型编译器自动向量化的效果,并分析其实现的局限性,对编译器的研究开发和体系结构设计都有非常重要的意义。

到稿日期:2012-07-21 返修日期:2012-10-22 本文受国家自然科学基金项目(61170046,61103014)资助。

李春江(1974—),男,博士,副研究员,硕士生导师,CCF会员,主要研究方向为计算机体系结构、编译及优化技术,E-mail:chunjiangli@gmail.com;黄娟娟(1988—),女,硕士生,主要研究方向为编译及优化技术;徐 颖(1992—),女,硕士生,主要研究方向为编译及优化技术;杜云飞(1980—),男,博士,助理研究员,主要研究方向为计算机体系结构、编译及优化技术;陈 娟(1980—),女,博士,助理研究员,主要研究方向为计算机体系结构、编译及优化技术。

专门针对支持SIMD结构的通用高性能处理器，评估典型编译器的自动向量效果，并分析原因，这方面的系统性的工作国内外尚不充分。主要原因是商用编译器源代码封闭，编译器自动向量化的局限性对外并不详细公开。而产品级的开源编译器通常由开源社区开发和维护、自动向量化的设计和实现管理松散组成，对其效果的评估和分析也同样缺乏系统性。

本文在支持SSE4.2[3]的Intel至强处理器平台上，首先用标准的性能评测程序，对Intel编译器[4]、PGI编译器[5]、GCC编译器[6]的自动向量化效果进行测试评估；然后，以广泛使用的开源编译器GCC为目标，手工编写了多种模式的程序片断（主要是多种循环结构），根据编译器报出的向量化相关信息，也结合对目标代码的反汇编，分析了GCC编译器自动向量化的效果和局限性。

2 背景

2.1 SIMD体系结构

当前，并行体系结构已经几乎成为所有高性能处理器的普遍结构，在高性能通用处理器芯片内的并行有两方面最重要的发展趋势：1）多核甚至众核的并行；2）功能更强大的SIMD并行。

图1引自Intel公司在IDF2011报告中对AVX内核体系结构的介绍。

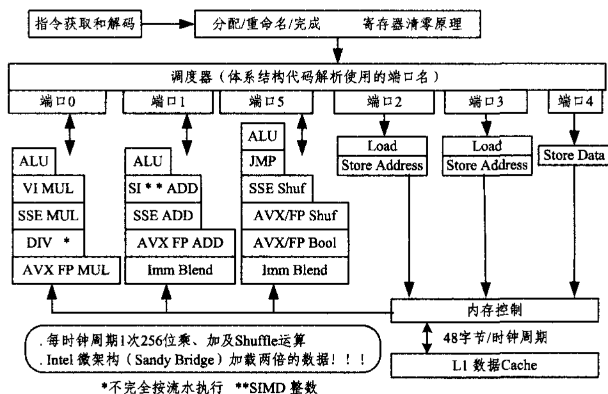

图1 Intel AVX内核架构（引自IDF2011）

通过对图1的研读，可以得到如下结论：

为了实现256位短向量操作，处理器内核新增了如下关键模块：AVX浮点乘、AVX浮点加、AVX/浮点数据的混洗、AVX/浮点布尔操作、两个向量重组操作单元（Imm Blend）、一个取单元和一个存单元。

对AVX的支持和原有对SSE的支持都是SIMD扩展。在Intel新的Sandy Bridge处理器内核架构中，其计算单元中相当多的计算和操作资源用于实现对SIMD处理的支持。可见，SIMD架构在处理器内核中有非常强大的生命力，紧耦合的SIMD结构是非常值得重视的处理器内核体系结构。

在支持AVX架构中，向量的存取没有旁路一级Cache，其一个取数单向存取链路的数据宽度为16个字节，两个取数单元同时工作刚好可以填满一个长256位的向量寄存器；这一访存能力对充分发挥向量处理能力至关重要。

而且，内核新增加了一个取、存单元，可以提升数据、指令的存取能力，也对带掩码的取、存提供了支持，这对发挥向量重组（数据重排）指令的功能也很关键。

目前，支持AVX的Intel商用处理器已经全面投入市场。并且Intel在AVX架构中采用了新的SIMD指令编码方式，为未来支持更长的向量（512位甚至1024位）预留了充分的设计空间。

2.2 编译器自动向量化

面向早期向量巨型计算机，针对科学计算类应用（主要用当时主流的Fortran语言编写），编译器的自动向量化获得了很大的成功。针对处理器内核中实现的SIMD结构，一直有大量的研究和工程实现工作以在编译器中实现面向SIMD指令的自动向量化为目标。当前，主流编译器如Intel编译器、PGI编译器、GCC编译器在O3优化级别，都已经开启了面向SIMD结构的自动向量化编译开关。

一方面，处理器内核中采用的SIMD体系结构不断发展；另一方面，更新、更高效的编译器自动向量化方法和技术不断被创造出来并投入应用。因此，面向SIMD结构的编译器自动向量化研究和工程实现工作一直是学术界和产业界的一个热点。

3 典型编译器对基准测试程序的自动向量化效果

3.1 实验环境

软件环境

本文评估的编译器包括：Intel编译器（版本11.1）、PGI编译器（版本11.8）和GCC编译器（版本4.6.1）。实验环境安装的操作系统为Redhat Linux Server 5.5。采用的标准性能评测程序是SpecCPU2006和SpecOMPM2001。

硬件环境

本文使用一台Intel X86_64至强工作站，该工作站采用双路XEON 5670处理器（主频2.93GHz）；每个处理器核L1级高速缓存（32kB指令、32kB数据），每个处理器核包含一个局部的L2级高速缓存（256kB）；每个芯片上的6个处理器核共享片上L3级高速缓存（6MB），在L3级高速缓存维护一致性；处理器间的L3级高速缓存间通过QPI（Quick Path Interconnect）[7]直连，使两个处理器构成SMP（共享主存多处理机）架构（并不是完全理想的SMP，有非常轻微的NUMA（非一致存储器访问）[8]效应），该实验平台处理器的架构如图2所示。该处理器支持超线程机制，但是本文在面向高计算性能的评估中，关闭了超线程功能。

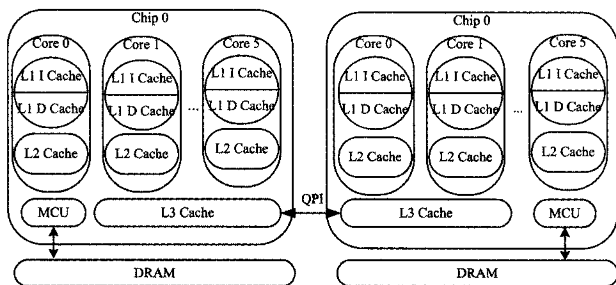

图2 实验环境的处理器架构

SIMD体系结构

该处理器支持SSE4.2，同时兼容Intel公司以前版本的SIMD指令。该SIMD结构支持的短向量长度为128位；实现了独立的8个128位短向量寄存器，其SIMD指令已经达到200多条[9]，提供了比较丰富的短向量处理能力。

3.2 实验结果及分析

3.2.1 SpecCPU2006

3种编译器对SpecCPU2006基准测试程序的自动向量

化加速比如图 3—图 8 所示。

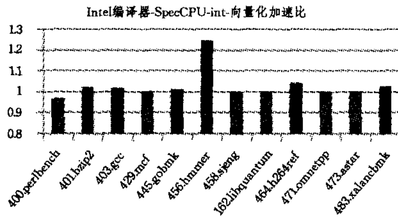

图 3

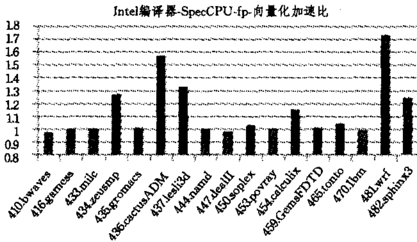

图 4

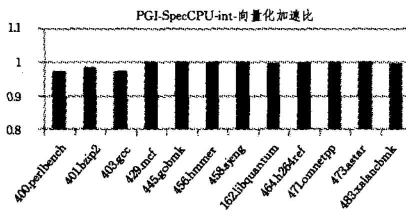

图 5

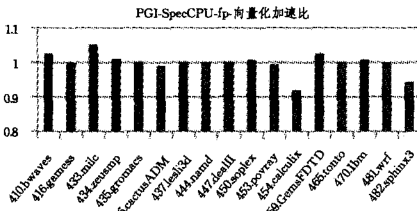

图 6

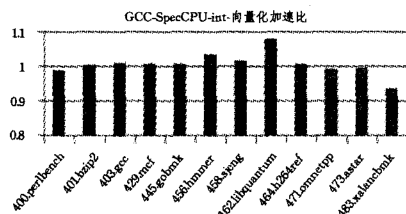

图 7

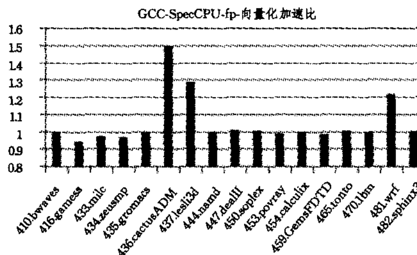

图 8

可见，对 SpecCPU2006 中所有串行应用（包括 12 道整型应用和 17 道浮点应用），3 种编译器的自动向量化仅对部分应用有明显的加速效果。

在 12 道整型应用中，Intel 编译器和 GCC 编译器的自动向量化加速效果明显好于 PGI 编译器（PGI 编译器对所有这些整型应用的自动向量化几乎没有加速）；而 Intel 编译器和 GCC 编译器相比，其仅对 456. hmmer 有超过 20％的加速效果，GCC 编译器对 462. libquantum 的自动向量化加速效果好于 Intel 编译器。GCC 编译器的自动向量化对 483. xalancbmk 却有 5％的性能降低，该应用为 C＋＋语言编写的将 XML 文档转换为 HTML 文档或纯文本文档或其他 XML 文档的应用程序，可见 GCC 编译器对此应用程序的自动向量化存在较大不足。

在 17 道浮点应用中，Intel 编译器对 434. zeusmp、436. cactusADM、437. leslie3d、454. calculix、481. wrf、482. sphinx 3 等 6 个应用的自动向量化都获得了超过 10％的性能提升；而 PGI 编译器对所有应用的自动向量化性能提升都未超过 6％；GCC 编译器仅对 436. cactusADM、437. leslie3d、481. wrf 的自动向量化性能有明显提升且都超过了 20％，其中前两道应用和 Intel 编译器相差不多。而对 481. wrf，GCC 编译器的自动向量化优化效果和 Intel 编译器的自动向量化优化效果差距非常大（达到了 50％），该应用是用 fortran90 编写的经典的用于天气预报的浮点应用，非常适合向量体系结构。

3.2.2 SpecOMPM2001

3 种编译器对 SpecOMPM2001 基准测试程序的自动向量化加速比，如图 9 至图 11 所示。在对 SpecOMPM2001 的测试中，采用了 Ref 模式（数据规模最大的模式），并且设置并行线程数为 12（使系统的所有处理器核都同时加载工作线程）。

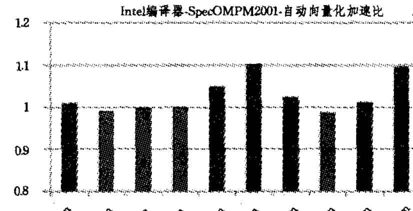

图 9

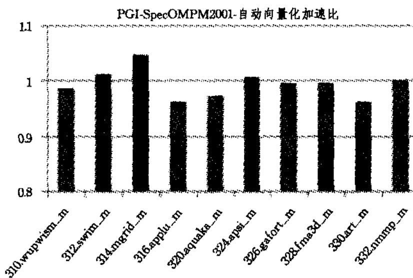

图 10

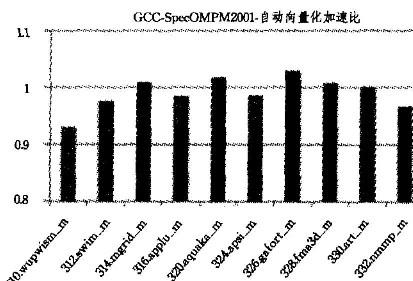

图 11

在本文评估的10道来自SpecOMPM2001的双精度浮点计算应用中，3种编译器的自动向量化加速效果都不是非常显著。Intel编译器仅对320. equake_m，324. apsi_m，332. ammp_m 3道应用的加速达到5%至10%，而PGI编译器仅对314. mgrid_m的加速效果接近5%，GCC编译器对所有应用的加速效果都小于5%。

Intel编译器的自动向量化仅使得2道应用的性能有1%左右的降低，但是PGI编译器和GCC编译器却使多达3道应用的性能降低超过2%。

在多线程并行，同时各个线程内部使用SIMD指令，并且系统满负荷工作（每个处理器核上都加载一个线程）的情况下，对这些科学计算浮点应用而言，其性能提升受限于存储器访问。

对有些题目，编译器自动向量化引发应用性能降低，是因编译器自动向量化的严重不足，这与编译器中使用的向量化代价模型有关。

PGI编译器和GCC编译器与该实验平台的自动向量化能力和Intel编译器相比有一定的差距。

3.2.3 基本结论

面向SIMD体系结构的编译器自动向量化经过多年的研究和开发，已经成为非常重要的编译优化之一。由于编译器自动向量化涉及到别名分析、相关性分析、循环变换等编译器对程序的分析和优化工作，也由于面向不同SIMD体系结构的编译器采用的向量化代价模型的差异，典型编译器的自动向量化优化效果差异显著。因此面向不断发展的SIMD体系结构，编译器自动向量化的研究与开发仍有非常大的发展空间。

4 GCC自动向量化的评估与分析

4.1 GCC自动向量化现状

GCC(Gnu Compiler Collection，Gnu编译器集合（下文简称GCC或GCC编译器））是广泛使用的开源编译器套件。近十余年来，随着Linux操作系统、GNU二进制工具的广泛使用，GCC编译器已经成为Linux平台上必备的编译器，发展非常迅速，已经发展为产品化水平的开源编译器。对GCC发展的贡献主要来自两个方面：其一是开源社区；其二是产业界重要的公司，如Redhat、IBM、Intel等等。所有基于GCC的开发工作都必须遵循GPL(Gnu Public Licensee)[10]规则，其中最重要的是必须保证对GCC所做的改动也开放源代码。

GNU编译器具有支持多种语言和多种目标处理器平台、文档及源代码开放等特点。基于GCC编译器面向新的语言、新的平台进行的编译器开发和编译优化工作非常活跃。

目前GCC的主分支中针对SIMD体系结构，实现了自动向量化的支持，包括两方面的工作：1)基于循环的自动向量化；2)基于SLP的自动向量化。

·基于循环的自动向量化

在GCC中，该部分工作由在以色列海法的IBM实验室的研究人员和工程师实现；他们在两次GCC峰会(GCCsummit2004和GCCsummit2006)上报告了自己的工作[11,12]。

在他们的工作最初被GCC接受到主分支(gcc4.0)时，仅可以对非常简单的循环进行自动向量化：带一个基本块的循环，并且循环跨步为1，数组边界已经按向量长度对齐，并且所有访问的数据类型相同。

此后他们对GCC中基于循环的自动向量化持续进行增强，2006年在他们的报告中介绍了如下增强：归约操作的自动向量化支持(GCC4.1)、归约模式识别(GCC4.2)、对非单位跨步以及多种数据类型的支持(GCC4.2)，以及对多种目标平台的支持[13]。

·基于SLP的自动向量化

SLP(Superword Level Parallelization)的思想由Samuel Larsen在其硕士论文中提出[14]，在完成硕士论文的6年之后(2006年)，他在博士论文[15]中又对SIMD优化中的数据对准问题进行了深入研究，并在SLP思想的基础上提出了选择性向量化方法。

SLP的基本思想是：在基本块中寻找操作模式相同（操作相同、操作的数据类型相同）但操作数不同的指令，将若干条这样的指令打包成SIMD指令来利用处理器提供的SIMD处理能力。这与以往在循环级的向量化方法有本质的区别。

基于SLP的思想和方法，GCC中实现了基于SLP的向量化，同样由以色列海法的IBM实验室完成，于2007年加入GCC的主分支。

4.2 GCC自动向量化能力分析

采用测试的方法，用多种类型的程序片段（主要是多种循环结构）测试GCC自动向量化的能力（包括循环级向量化和SLP向量化两个方面）。由于篇幅的限制，本文仅列出部分测试例子，指出GCC编译器自动向量化存在的问题。

4.2.1 GCC自动向量化现有能力

1)基于循环的自动向量化方面

GCC支持简单嵌套循环的自动向量化（循环体内无分支，循环内对数组引用模式简单）；

GCC支持简单的指针运算的自动向量化；

GCC支持循环内简单归约操作的自动向量化。

2)基于SLP的自动向量化方面

GCC基本实现了SLP支持，可以在一个基本块内寻找同型语句进行向量化。

4.2.2 GCC自动向量化存在的主要不足

1)基于循环的自动向量化方面

·GCC对非计数循环不能进行向量化

例如图12所示的程序，GCC不能对这样一个简单的非计数循环向量化。像这种循环情况，可以结合其它编译优化措施进行向量化，例如可以采用运行时的动态向量化方法，或者利用值剖视信息[16]来实现向量化，目前GCC可基于对程序静态分析的自动向量化，尚不能对这种运行时方知循环次数的循环进行自动向量化。

```
while (*p ! = NULL) {
  *q++=*p++;
}
```

图12 非计数循环举例

·GCC不能对结构内数组元素的顺序处理向量化

如图13所示的程序，对于结构内的数组的顺序访问，GCC目前不能自动向量化，同样对于C++程序中类的数组成员的顺序处理也不能自动向量化，这就是对于SpecCPU2006中的483. xalancbmk程序，GCC编译器自动向量化效果特别差的主要原因。

```
int N;
struct strut1 {
  int * a;
};
struct strut1 s;
int foo(struct strut1 s){
  int i;
  for (i = 0;I < N;i++) {
    s.a[i] = 5;
  }
  return 0;
}
```

图 13 结构内数组访问

• GCC 对稍做变形的归约操作就不能识别，也不能自动向量化

图 14 所示程序其实是一种归约的变形，GCC 不能进行自动向量化。通过分析编译过程中程序的中间表示（见图 15），我们发现原因是在树一级中间表示中，为了实现 SSA[17]，对同一个 sum 变量命名了不同的名字，而程序的高层次信息没有传递到该中间表示中，因此编译器分析认为是两个不同的归约目标，并且两个归约间存在数据相关，所以没有向量化。这暴露了 GCC 在中间表示层面，面向程序的高级优化，程序原始特征信息有部分缺失，限制了自动向量化的实现。

```
  int sum;
int a[128];
void foo(void)
{
  int i;
  for(i = 0;i < 64;i++)
    {
      sum += a[2 * i];
      sum += a[2 * i + 1];
    }
}
```

图 14 变形归约

```
i=0;
goto<D.1591>;
<D.1591>;
D.2691=i * 2;
D.2692=a[D.2691];
sum.0=sum;
sum.1=D.2692+sum.0;
sum=sum.1;
D.2691=i * 2;
D.2695=D.2691+1;
D.2696=a[D.2695];
sum.0=sum;
sum.2=D.2696+sum.0;
sum=sum.2;
i=i+1;
<D.1591>;
if(i<=63)goto<D.1590>;else goto <D.1592>;
<D.1592>;
D.2698=sum;
return D.2698;
```

图 15 变形归约程序的中间表示

• GCC 的循环变换能力弱限制了自动向量化的实现

如图 16 所示的多重循环 C 语言程序，GCC 编译器不能自动向量化，事实上最内两重循环是可交换的，手工交换 i 和 j 两重循环，GCC 就能够进行自动向量化。这说明，GCC 编译器的循环变换能力不足限制了自动向量化的实现。

```
oo(int M,int N,int K){
  int i,j,k;
  int sum;
  int in[N+K][M],coeff[N][M],out[K];
  for(k=0;k<K;k++){
    sum = 0;
    for (i = 0;i < N;i++)
      for (j = 0;j < M;j++)
        sum += in[i+k][j] * coeff[i][j];
        out[k] = sum;
  }
  return 0;
  }
```

图 16 多重嵌套循环

• GCC 现有的向量化实现使相邻的无关循存在相互影响

图 17 所示的程序段中，本来 3 个循环各不相关且都是可以向量化的，但是，当将 while 循环处在最前面时，后面两个 for 循环就不能自动向量化；当 while 循环放在 3 个循环的最后一个时，3 个循环都可以自动向量化；当 while 循环处于两个 for 循环之间时，while 循环前的 for 循环和 while 循环可以自动向量化，而 while 循环之后的循环却不能自动向量化。可见，GCC 在面向循环结构的自动向量化方面还存在循环间相互干扰的问题。

```
typedef int aint __attribute__ ((__aligned__(4)));
int a[256],b[256],c[256];
foo (int n,aint * __restrict__ p,aint * __restrict__ q)
{
int i;
while (n--){
*p++ = *q++ + 5;
}
for (i=0;i<n;i++){
a[i] = b[i+1] + c[i+3];
}
for (i=0;i<n;i++){
j = a[i];
b[i] = (j > MAX ? MAX:0);
}
}
```

图 17 多个相邻无关循环

• GCC 对循环跨步的支持较弱

GCC 仅支持数组访问跨步均匀且跨步不大于 2 的循环结构的自动向量化。

• GCC 对引入指针别名的相关性分析能力很差

仅对非常简单的指针顺序引用能进行自动向量化。

2)基于 SLP 的自动向量化方面

• GCC 不能将循环变换和 SLP 结合

例如对图 18 所示的程序，GCC 仅能采用 SLP 方式进行

向量化，但是由于目标平台的向量寄存器的宽度为128位，因此对此程序循环内的基本块仍不能采用SLP方式自动向量化。而事实上，只要将循环展开4次，扩大基本块内的同型语句数目，是可以采用SLP方式向量化的。可见，GCC在SLP向量化以及循环展开优化之间没有很好地结合。

```
#define N 256
int * pOutput;
foo(int * pInput){
    int i;
    int a,b,c;
    int M00,M01,M02,M10,M11,M12,M20,M21,M22=10;
    for (i=0;i<N;i++)
    {
      a=*pInput++;
      b=*pInput++;
      c=*pInput++;
      *pOutput++=M00*a+M01*b+M02*c;
      *pOutput++=M10*a+M11*b+M12*c;
      *pOutput++=M20*a+M21*b+M22*c;
    }
}
```

图18 采用SLP方式的向量化

4.3 GCC自动向量化的发展趋势

在GCC编译器中实现自动向量化的努力已经持续了近10年。除了本节前面评估的采用经典相关性分析方法对循环和基本块进行自动向量化的实现外，从2009年4月开始，GCC中的GRAPHITE分支[18]开始采用多面体模型[19]来提升GCC编译器的循环分析和变换能力，并以自动并行化、自动向量化、访存优化为目标。在GCC中使用多面体模型需要使用到独立的Cloog库（该库采用多面体模型进行代码生成）[20]和PPL库（该库是意大利帕尔马大学开发的，用于计算可表示为n维向量空间中点的数值信息）[21]，在编译GCC的过程中要首先配置上面两个库。GRAPHITE分支首先将GCC中SSA形式的树中间表示（称为Gimple）转换为用多面体模型来表示，然后利用多面体模型求解库进行分析和变换，实现循环变换来达到自动并行化或自动向量化的目的，然后再将多面体表示形式回写为Gimple表示。目前，GRAPHITE分支还在研发中，因此本文未对该分支的自动向量化能力进行评估。

自动向量化仍是GCC编译器中非常重要的优化方向，很多学术机构和个人在此方面进行了积极的探索，不断有新的创造性的工作出现。但是，自动并行化和自动向量化本来就是编译技术中的难题，短期内不能期望有大的突破性的进展。另外，新的并行编程范型和新的编程语言不断出现，而且SIMD体系结构不断发展变化，这些都为编译器的自动向量化提出了很多新的挑战。

结束语 本文首先用基准测试程序对典型编译器（Intel编译器、GCC编译器和PGI编译器）的自动向量化效果进行了测试、评估和分析；然后，面向开源编译器GCC，采用手工编写的特征程序段较为详尽地分析了GCC自动向量化的能力和不足。

在下一步的工作中，一方面将针对现有GCC自动向量化方面的不足开展研究和实现；另一方面，将紧密结合GCC中新的优化架构（如GRAPHITE）开展研究。此外，将研究探索新的自动向量化方法。

参 考 文 献

[1] Maruyama T, Motokurumada T, Morita K, et al. Past, Present and Futures of SPARC64 Processors[J]. FUJITSU Sci. Tech. J., 2011, 47(2): 130-135

[2] Firasta N, Buxton M, Jinbo P, et al. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency[M]. Intel white paper, 2008

[3] Intel. SSE4 Programming Reference [M]. Intel Corporation, 2009

[4] Intel compilers [OL]. http://software.intel.com/en-us/articles/intel-compilers/

[5] PGI Compilers[OL]. http://www.pgroup.com/

[6] Gnu Compiler Collection[OL]. http://gcc.gnu.org

[7] An Introduction to the Intel QuickPath Interconnect[M]. Intel Corporation, January 2009

[8] Manchanda N, Anand K. Non-Uniform Memory Access (NUMA) [OL]. http://cs.nyu.edu/～lerner/spring10/projects/NUMA.pdf, 2012

[9] Intel64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 3A and 3B[M]. Intel Corporation, May 2011

[10] The GNU General Public License[OL]. http://www.gnu.org/licenses/ licenses.html#GPL

[11] Naishlos D. Autovectorization in GCC[C]//Proceedings of the GCC Developers' Summit. Ottawa, Ontario Cana-da, June 2004

[12] Nuzman D, Zaks A. Autovectorization in GCC—two years later [C]//Proceedings of the GCC Developers' Summit. Ottawa, Ontario, Canada, June 2006

[13] Nuzman D, Henderson R. Multi-platform Auto-vectorization [C]//Proc. of the 4th Annual International Symposium on Code Generation and Optimization (CGO). March 2006

[14] Larsen S. Exploiting Superword Level Parallelism with Multimedia Instruction Sets[D]. Massachusetts Institute of Technology, May 2000

[15] Larsen S. Compilation Techniques for Short-Vector Instructions [D]. Massachusetts Institute of Technology, April 2006

[16] Calder B, Feller P, Eustace A. Value profiling[C]//30th International Symposium on Microarchitecture. December 1997

[17] Sassa M. Static Single Assignment Form. Tokyo Institute of Technology[OL]. http://www.is.titech.ac.jp/～sassa/coins-www-ssa/english/ssa-lecture-80.pdf

[18] GRAPHITE: Polyhedral Analyses and Optimizations for GCC [C]//Proc. of GCC Summit 2006. Ottawa, Ontario, Canada, June 2006

[19] Benabderrahmane M-W, Pouchet L-N, Cohen A, et al. The Polyhedral Model Is More Widely Applicable Than You Think[C]//Compiler Construction. Lecture Notes in Computer Science, Volume 6011, 2010: 283-303

[20] Cloog[OL]. http://www.cloog.org/

[21] Bagnara R, Hill P M, Zaffanella E, et al. The Parma Polyhedra Library User's Manual, version 0.12.1[M]. April 2012