# SIMD向量指令的非满载使用方法研究

徐金龙 赵荣彩 赵 博
(信息工程大学数学工程与先进计算国家重点实验室 郑州 450001)

**摘 要** 大规模SIMD体系结构提供了更强的向量并行硬件支持,但是,大量迭代次数不足的循环由于不能提供足够的并行性,难以用等价的向量方式实现。为了更有效地利用SIMD,提出了一种非满载地使用SIMD指令的向量化方法。研究了向量寄存器的使用方式,基于非满载的向量寄存器使用方式实现了非满载的向量操作和短循环的向量化,并将非满载的向量化方法用于一般循环的向量化。提供了收益分析方法来为本向量化方法作精确指导。实验结果表明了该方法的有效性,所选测试用例的目标循环被向量化,平均加速比达到1.2。

**关键词** 大规模SIMD,并行,向量化,非满载向量操作,收益分析
**中图法分类号** TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.7.049

## Research on Non-full Length Usage of SIMD Vector Instruction

XU Jin-long ZHAO Rong-cai ZHAO Bo
(State Key Laboratory of Mathematical Engineering and Advanced Computing, University of Information Engineering, Zhengzhou 450001, China)

**Abstract** Large-scale SIMD architecture provides stronger vector parallel support on hardware. However, a large number of loops which are short of iterations can not provide sufficient parallelism, and it is difficult to achieve them with the equivalent vector mode. In order to make full use of SIMD, this paper presented a vectorization method which can use non-full length of SIMD vector instruction. This paper studied the vector register usage, achieved a non-full vector operation based on non-full length usage of vector register, which can vectorize short loops. Finally, this method was used to vectorize the common loops. Moreover, This paper provided a benefit analysis method to guide the vectorization method. Experimental results show that the method is available, the target loops of the selected test programs are vectorized and the average speedup is about 1.2.

**Keywords** Large-scale SIMD, Parallel, Vectorization, Non-full vector operation, Benefit analysis

## 1 引言

随着社会进步和科技发展,人们对计算能力的需求越来越高,而处理器的主频不可能无休止提高,硬件流水线的级数也不可能无限增长。硬件设计者考虑通过并行手段来提高硬件计算能力[1],于是出现了多机结构、多核结构、SIMD(Single Instruction Multiple Data)结构。SIMD依靠同时在多个功能部件上处理一条指令来获得空间并行性,是向量处理的一种方式,适用于多媒体应用和大型科学计算等具有整齐操作的数据密集型运算程序。它作为一种低成本、高回报的硬件设计思路,已经成为微处理器必备的扩展部件。为了提高并行度,SIMD硬件提供了越来越长的向量宽度。以Intel系列处理器的SIMD扩展为例,从MMX开始,经历了SSE,SSE2,SSE3,SSE4,AVX[2,3],到现在的AVX512[4,5],向量长度也由64位增加到512位。而国产处理器的Matrix向量指令集[6]达到了1024位,可同时处理32个32位的整型数据。较长的向量长度能够带来较大的加速,但是也增加了可并行化循环的阈值,即一些迭代次数不足的循环无法提供足够的并行性,因而无法向量化。若*VF*(Vector Factor,向量因子,向量操作对应数据元素个数)=32,那么如例1所示的循环被识别为不可向量化,本文将这种迭代次数小于*VF*的短循环称为NEI_LOOP(Not Enough Iteration Loop)。

例1
```
For(i=0;i<31;i++){
    A[i]=B[i];
}
```

随着硬件向量长度的增加,程序中越来越多的循环转变为NEI_LOOP(大量存在于NPB、SPEC2006测试集中,部分由循环优化变换产生),因此,在向量因子较大时,如何实现短循环的向量化是亟待解决的问题。本文针对NEI_LOOP,提出了一种SIMD向量指令的非满载使用方法,旨在适应不断增加的硬件向量长度。

## 2 问题描述

### 2.1 SIMD的规模分类

SIMD最初属于一种多媒体扩展,提供基于寄存器的向量计算。Intel推出的多媒体扩展MMX使用了64位的

到稿日期:2014-08-09 返修日期:2014-12-13 本文受国家高技术研究发展计划(863)(2009AA01220),"核高基"重大专项(2009zx10036-001-001)资助。

徐金龙(1985—),男,博士生,主要研究领域为并行编译,E-mail:longkaizh@126.com;赵荣彩(1957—),男,教授,博士生导师,主要研究领域为体系结构、先进编译等;赵 博(1989—),男,硕士生,主要研究领域为并行编译。

MMX 寄存器，这一 SIMD 扩展能够同时实现两个 32 位整数、两个单精度浮点数计算[7]。随着体系结构的发展，SSE/SSE2/SSE3 采用了 128 位的向量寄存器，实现了向量规模的翻倍。后来，Intel-AVX 向量寄存器达到了 256 位，AVX512 更是达到了 512 位，国产飞腾处理器甚至达到了 1024 位。硬件向量寄存器位数（也即向量规模）的指数级增加，一方面极大地提高了硬件的并行程度，如果得到充分利用，显然可以获得更高的加速；另一方面，越来越多的 NEI_LOOP 无法向量化，阻碍了程序的加速。

按 SIMD 向量位数对 SIMD 规模进行分类：

（1）小规模（Small-scale）SIMD[8]，指向量寄存器位数在 128 以下（包含 128 位）的 SIMD 结构，对应较短的向量长度。

（2）大规模（Large-scale）SIMD，指向量寄存器位数在 256 以上（包含 256 位）的 SIMD 结构，对应较长的向量长度。规模越来越大是 SIMD 向量部件的发展趋势。

### 2.2 大规模 SIMD 引发的问题

程序的向量并行性大都存在于循环结构中，通常是将循环的多次不同迭代实例合并到一起来实现程序的向量化。合并到一起的迭代次数取决于循环核心计算的数据类型和 SIMD 部件的向量长度，若向量长度为 128 位，数据类型为整型，那么可同时处理的数据个数为 4（128/32，也称之为向量因子 VF）。目前的问题是：待处理的数据个数小于 4 的情况应怎样处理，当前所有的编译器都会忽略这种迭代次数不足以构成一次向量操作的情况，不对其做向量化。这是有一定道理的，因为统计发现，实际应用程序中很少存在迭代次数小于 4 的循环。而在大规模 SIMD 硬件环境中，向量长度达到或超过 256 位，以 1024 位为例，同样是整型计算，向量因子 $VF$ 达到了 32，编译器会忽略所有迭代次数小于 32 的循环，而应用程序中大量存在迭代次数小于 32 的循环。可以想象，当出现 2048 位甚至 4096 位的向量寄存器时，迭代次数达不到 64 或 128 的循环都属于 NEI_LOOP，将不能采用向量执行的方式来加速，这不但湮没了程序中大量并行性，更是浪费了性能良好的超长向量部件。怎样用大规模 SIMD 实现 NEI_LOOP 的向量化成为亟待解决的问题。

### 2.3 相关研究

大多数编译器为了提供对 SIMD 扩展结构的支持，都投入了大量的人力进行 SIMD 自动向量化编译器的开发，目前已经存在多种向量化方法。

Randy Allen 和 Ken Kennedy 提出了面向循环的传统的向量化算法[9]。算法基于依赖分析，首先构建依赖图，同过通用算法求解图中的强连通分量，若最终的强连通分量只存在一条语句，那么该语句可进行向量化。目前的大多数编译器都集成了该方法。

2000 年 Larsen 等提出了启发式的 SLP 算法[10]，此方法是针对基本块内的向量化发掘方法，因此适用于发掘循环迭代内的并行性。若配合循环展开，可同时发掘循环迭代内和迭代间的并行，因此，它是传统向量化方法的超集，越来越多的自动向量化编译器加入了对 SLP 算法的支持。Tenllado[11]在按照启发规则生成同构指令包之后增加了一个 PT（Pack Transposition）指令包调整阶段，用来解决内层循环携带数据依赖，外层循环可以向量化，但是数组引用相对于外层循环索引不连续的情况。Rajkishore[12]提出的 BS 算法除去了启发式 SLP 中按照定义-使用链和使用-定义链进行扩展的限制，采用动态规划的方法来寻求较优同构指令打包方式。Alexei Kudriavtsev[13]使用与 SLP 算法相同的指令打包启发式规则，但是打包过程存在差异，它根据的是数据流图和数据流树而不是 UD 和 DU 链，目的是通过向量重组提高向量化效率。

基于模式匹配的 SIMD 操作识别技术[14]主要是针对特殊 SIMD 扩展指令利用模式匹配技术识别串行程序中相同功能的代码段，并将其替代为 SIMD 指令。该方法回避了复杂的依赖关系分析，直接从利用模式匹配分析指令组成形式入手实现 SIMD 向量化。

上述向量化方法产生的向量代码都只能将 $VF$ 整数倍的同构语句打包执行，若同构语句数量不是 $VF$ 的整数倍，最终总会存在 $N$ 条语句（$N<VF$）无法向量执行。本文将用新的思路来挖掘这 $N$ 条语句的潜在并行性。

## 3 面向 SIMD 的非满载向量化方法

### 3.1 向量寄存器的使用方式

（1）向量寄存器的满载使用

向量寄存器可同时容纳大量数据，当前所有的编译器都将向量寄存器作为一个不可拆分的整体来使用（即满载使用方式）。整体使用是指，向量寄存器中的每个数据都是有效的，从向量装载到向量计算，再到向量存储，一系列的操作都要保证寄存器中每个数据的有效性。当程序中存在足够的数据并行性时这种整体使用寄存器的方式才适用。

（2）向量寄存器的非满载使用

若程序中的并行数据量不足以充满一个向量寄存器，当前自动向量化编译器则无能为力。本文设计一种部分使用向量寄存器的方法（即非满载使用方式）来解决此问题。部分使用是指向量寄存器中某些数据是有效的，其余数据为无效数据。

（3）向量寄存器的灵活使用

灵活的向量寄存器使用方式可归为图 1 所示的 4 种方式的组合：1）满载使用；2）部分使用——一端无效；3）部分使用——两端无效；4）部分使用——复杂。其中 2）—4）都属于非满载使用方式。

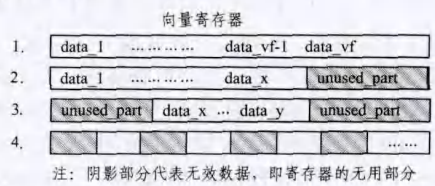


图 1 向量寄存器的使用方式

实际上，硬件只支持图 1 中的满载使用方式。所有的非满载使用方式，都需要采用间接方法实现——通过复杂数据重组或额外的访存操作，3.2 节将设计并实现这种间接方法。

### 3.2 非满载向量操作的实现

向量寄存器的灵活使用，要求硬件支持相应操作：本节假定向量因子为 8，图 2（a）属于满载向量操作，图 2（b）属于非满载向量操作。目前 SIMD 硬件并未支持图 2（b）

所示的向量操作方式，因此必须寻求其他方法来实现这种操作。


图 2　向量满载操作与非满载操作

### 3.2.1　实现方法分析

实际上，只要实现非满载的向量存储操作，就可以间接实现所有非满载的向量操作。通过死代码消除[9]的逆过程（添加死代码的方法）可以对此作出解释。其实现过程如图 3 所示。

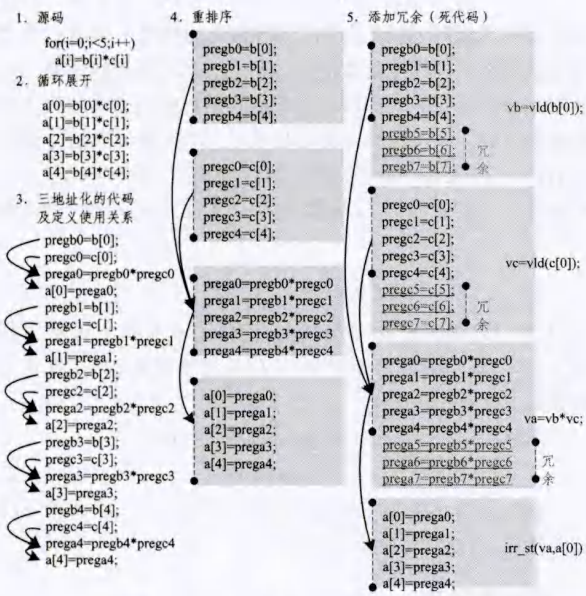


图 3　通过添加冗余实现短循环向量化

希望将图 3 中源码 1 做向量化，经过循环展开（见步骤 2）、三地址化（见步骤 3）、代码重排序（见步骤 4）后，其并行特征得以显现。图中箭头指示了语句间的定义使用关系，始端代表定义点，终端代表使用点。图 3 中步骤 4 是根据定义使用关系进行代码重排序的结果，共 4 组语句，第 1、2 组是对数组（$b$、$c$）的访问，第 3 组为计算，第 4 组将计算结果存储到数组 $a$ 中，数组 $a$ 的存储语句就是最终使用点。每组语句都是同构语句，可以打包在一起执行，语句条数为 5，由于 $VF=8$，难以用现有的向量操作实现。图 3 中步骤 5 添加了部分语句，其由于最终未被使用，因此属于冗余语句（死代码），不影响程序语义。添加冗余后，前 3 组语句条数都变为 8，显然可以用向量化操作实现，第一组语句被替换为 vb＝vld(b[0])，第二组语句为 vc＝vld(c[0])，第 3 组语句为 va＝vb＊vc。关键在于第 4 组存储语句，要实现 5 个数据的存储，显然这组语句不能直接被替换为向量指令，因此需要实现非满载的向量存储指令。非满载向量存储指令的目标是将第 4 组语句中的 5 个数据存入 5 个对应的内存单元，同时不改写其它内存单元的值。

### 3.2.2　非满载向量存储操作

图 4 描述了待实现向量存储指令的功能，将向量寄存器的 X0－X4 写入内存的 a[0]－a[4]位置，a[5]－a[7]保持不变，而当前的向量存储指令不能直接实现此功能。可以发现，若提前将向量寄存器的 X5－X7 替换为 a[5]－a[7]，那么可以达到预期效果，图 5 展示了采用上述思路来实现非满载向量存储指令的方法。

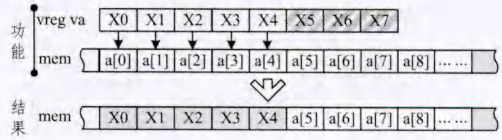


图 4　非满载向量存储的预期功能

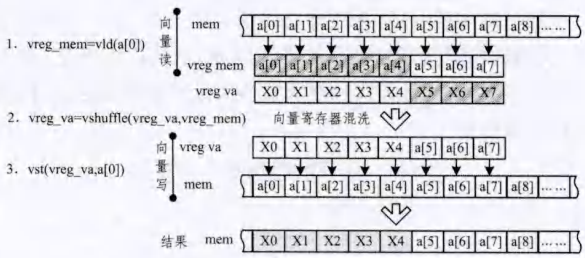


图 5　非满载向量存储的实现方法

用常规向量存储指令实现非满载向量存储的一般步骤：

（1）将向量存储指令要覆盖的内存数据读到向量寄存器 vreg_mem 中，对应图 5 所示的向量读。

（2）用向量重组指令将向量寄存器 vreg_va 中的无效数据替换为内存中将要被覆盖的数据，对应图 5 的向量寄存器混洗阶段。

（3）用普通的向量存储指令把经过处理的向量寄存器 vreg_va 写入对应内存中，对应图 5 的向量写阶段。

采用上述方法可以实现任何非满载的向量存储指令。由于每条非满载向量写等价于“常规向量读＋数据混洗＋常规向量写”，因此执行代价会随之上升，甚至会产生负收益，因此，本方法须有配套的收益分析，见后文。

## 3.3　非满载向量操作的应用

非满载的向量化方法主要针对 NEI_LOOP，本文将讨论如何将之应用于一般循环的向量化，也即发掘一般循环中的 NEI_LOOP。NEI_LOOP 来源于以下两种情况：1）源代码中原有的迭代次数不足的循环；2）索引集分裂产生的小循环，由于循环剥离是索引集分裂的主要形式，因此主要考虑循环剥离产生的头循环或尾循环。

### 3.3.1　循环剥离

循环剥离主要用于两方面的优化：1）通过访存对齐分析及循环剥离的方法来获得更多的对齐访存操作；2）将某几次迭代移出循环体来消除循环携带依赖，从而提升循环向量并行性。

（1）对齐相关剥离

由于主存的存取速度远低于 CPU，程序运行过程中，访存占用大量时间。对于向量访存而言，对齐特性也极大地影

响着访存效率。结果是，程序进行对齐优化后往往可以得到显著的性能提升。在自动并行化编译器中，一般通过循环剥离的方法来获得更多的对齐访存操作，本文称之为对齐剥离。

下面举例说明对齐剥离的作用，假设向量长度 512 位，对于 32 位整型，其向量因子为 16，访存一般要求 512 位对齐。$i=0$ 为循环首次迭代，显然，$a[3]$、$b[3]$为非对齐数组访存。对齐的访存包括($a[0]$,$a[8]$,…,$a[8*I]$)，$I$ 为整数。若首次迭代访问的数组为 $a[8]$、$b[8]$，那么访存将是对齐的，将前 5 次迭代分离到循环外，可形成如图 6 右侧所示的两个循环，loop1 为串行头循环，loop2 为对齐的循环。loop2 的迭代次数不是向量因子整数倍时，尾部将被继续剥离产生尾循环。

```
loop:  for(i=0; i<N; i++){                 loop1: for(i=0; i<5; i++){
         A[i+3]=B[i+3];          ⇨                   A[i+3]=B[i+3];}
       }                                   loop2: for(i=5; i<N; i++){
                                                   A[i+3]=B[i+3];}
```

图 6　简单的对齐剥离

经过剥离后，一般循环都会转变为图 7(b)所示的标准形式：头循环和尾循环都属于 NEI_LOOP，迭代次数不足，难以构成向量操作；主体循环迭代次数为向量因子整数倍，可构成整数向量操作。

```
for(i=0;i<align_begin;i++){
  loop_body
}
```

(a)剥离前

```
for(i=0;i<align_begin;i++){   //头循环
  loop_body
}
for(i=align_begin;i<align_end;i++){   //主体循环
  loop_body
}
for(i=align_end;i<N;i++){//尾循环
  loop_body
}
```

(b)剥离后

图 7　对齐剥离的标准形式

(2)依赖相关剥离

一种常出现的情况是，循环存在以开始几次(或最后几次)迭代为源点的携带依赖，而这种依赖可以通过剥离循环的前几次(后几次)迭代放到循环前(循环后)的方法将其转换化为从循环外来的循环无关依赖，从而消除相应的循环携带依赖。

通常这种用于消除某些依赖的循环剥离也将产生类似图 7(b)所示的形式，并且其头、尾循环可能没有足够迭代次数。

### 3.3.2　向量化方案

原有的向量化方案在实施向量化之前通常先检测循环迭代次数，迭代次数较少的循环被保持串行执行，迭代次数足够的循环首先会进行循环剥离，剥离后的主体循环可直接使用常规方法向量化，而头循环和尾循环由于迭代次数不足，也将保持串行执行。本文提供了针对 NEI_LOOP 的非满载的向量化方法，因此，相对于原有的向量化方案，迭代次数不足的循环包括头、尾循环都可被向量化。图 8 展示了本文的向量化方案。

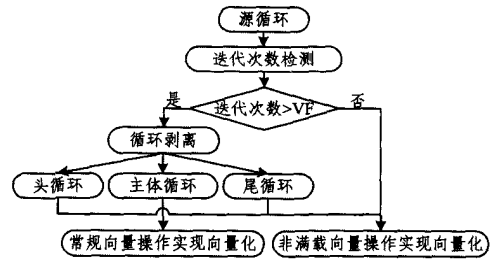


图 8　一般循环的向量化方案

## 4　收益分析

### 4.1　必要性分析

虽然非满载的向量操作可通过第 3 节提供的方法实现，但并不是所有的向量化都能带来正收益。由于采用了额外的向量读和向量重组，向量化带来并行收益的同时，也需要付出更多的访存及重组代价，综合收益往往为零或为负。因此必须通过精确的收益分析来指导向量化，若无收益或负收益，目标循环保持串行执行。

### 4.2　收益计算方法

针对 NEI_LOOP 的向量化，只需要一次向量操作。因此向量化后无循环(等同于迭代次数为 1 的循环)，向量化收益由每条向量语句产生的收益叠加而成。而每条向量语句的收益则用该条语句对应的所有标量语句的代价与该语句代价的差值来刻画。$B$ 代表循环向量化收益，$M$ 代表可向量化语句条数，$CV_i$ 代表第 $i$ 条向量语句的代价，$CS_i$ 代表与 $CV_i$ 对应的标量语句的代价，$ITER$ 代表 NEI_LOOP 的迭代次数，那么

$$B=\sum_{i=1}^{M}(CV_i-ITER*CS_i) \tag{1}$$

其中，$M$、$ITER$、$CS_i$ 都是已知的，代价 $CV_i$ 只有一种特殊情况，就是非规则向量存储指令，其代价需要单独计算，计算方法如式(2)所示。

$$C(abnormal\_vstore)=C(vload)+C(vshuffle)+C(vstore) \tag{2}$$

通过上述方法获得向量化收益 $B$，若 $B>0$，那么循环将被向量化，否则将保持串行执行。

## 5　测试及分析

### 5.1　测试实例及方案

(1)测试实例

本文提出的非满载向量化方法针对的是 NEI_LOOP。为了检测本方法的有效性，所选取的测试程序需要满足以下特征：1)测试程序本身含有 NEI_LOOP，或者可通过循环变换生成 NEI_LOOP；2)NEI_LOOP 运行时间在整个程序中占有较大比例(若 NEI_LOOP 运行时间可忽略，那么对其优化将毫无意义)。本文按上述要求选取了 SPEC2006 和 NPB 中的部分测试用例，包括 SPEC2006-435、SPEC2006-482、NPB-BT、NPB-IS。

(2)测试方案

本实验在国产神威服务器上进行，实验平台 CPU 主频为 2.0GHz，内存为 2GB，向量寄存器的宽度为 256 位，可以同时处理 4 个浮点型数据或者 8 个整形数据。

分别对选取的 4 个测试用例进行测试，统计以下信息：1)NEI_LOOP数量，包括程序中本身存在的 NEI_LOOP 个

数、循环变换可产生的 NEI_LOOP 个数；2)通过收益分析的 NEI_LOOP 个数；3)采用本文提供的非满载向量化方法前后的向量化的加速比。最后通过加速比的变化来证明本方法是否有效。

### 5.2 测试结果及分析

测试用例中的 NEI_LOOP 信息如表 1 所列，所有测试用例都存在 NEI_LOOP，其中 SPEC2006-435 收益分析未通过，因此不会实施向量化，其它用例都采用了非满载向量化方法。

表 1 测试用例中的 NEI_LOOP 信息

| 测试用例 | NEI_LOOP 数量 | | | 通过收益分析并向量化的 NEI_LOOP 个数 |
|---|---|---|---|---|
| | 源程序中 NEI_LOOP 个数 | 循环变换产生的 NEI_LOOP 个数 | 总数 | |
| SPEC2006-435 | 1 | 0 | 1 | 0 |
| SPEC2006-482 | 0 | 1 | 1 | 1 |
| NPB-BT | 79 | 3 | 82 | 39 |
| NPB-IS | 1 | 2 | 3 | 3 |

采用非满载向量化方法前后测试用例的加速比如图 9 所示，其中 SPEC2006-435 无加速，原因是未被向量化。SPEC2006-482、NPB-BT、NPB-IS 都存在加速，但 NPB-BT 的效果明显好于其他两个用例，原因在于 NPB-BT 程序原有的 NEI_LOOP 数量占比大，运行时间占比大，因而它的向量化对整个程序的加速贡献就大。而另外两个程序的 NEI_LOOP 主要是循环变换产生的尾循环，运行时间占比相对小，因此对程序的加速贡献相对较小。

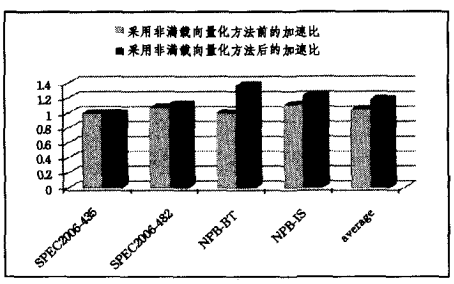


图 9 采用非满载向量化方法前后测试用例的加速比

总体来看，图 9 所示的结果表明本文提供的向量化方法能够带来较明显的加速效果，测试结果证实了本方法的有效性。

**结束语** 大规模 SIMD 是向量体系结构的发展趋势。向量因子的持续增加，使得 NEI_LOOP 数量大大增加，而通常情况下这种循环难以被向量化，NEI_LOOP 主要包括程序中原有的小循环和循环剥离产生的头、尾循环。本文提供了一种非满载向量操作的解决方案，其可以支持向量寄存器的多种形式的局部使用，从而实现 NEI_LOOP 的向量化。由于非满载的向量操作需要额外的代价，向量化并不总是有效的，因此本文同时提供了收益分析算法，仅仅对收益大于零的程序实施向量化。本文提供的方法不仅可以实现 NEI_LOOP 向量化，同时还可以解决向量化过程中的大部分非规则访存问题，如跨幅访存、普通的非连续访存等，基本思想就是向量寄存器的非规则使用。

本文在 3.3 节给出了一般循环的向量化方法，对齐剥离的目的是使主体循环获得更好的对齐特性，但剥离产生的头、尾循环访存通常是非对齐的，而本文提出的非规则向量化方法恰恰是针对头、尾循环的。因此，如何调整向量寄存器(非满载)的有效数据位置，使得向量寄存器对应到对齐数据，实现进一步优化，值得继续研究。

本文主要将非满载的向量化方法用于循环的向量化，未考虑并行性存在于基本块内部的情况。当基本块内存在并行性且并行语句不足时，结合 SLP[10] 超字并行算法来实现基本块内的非满载向量化也是要进一步展开的研究内容。

## 参 考 文 献


[1] 魏帅. 面向 SIMD 的向量化算法及重组技术研究[D]. 郑州：解放军信息工程大学，2012
Wei Shuai. Reaserch of SIMD Vectorization Algorithm and Optimization[D]. Zhengzhou: PLA Information Engineering University, 2012

[2] Peleg A, Weiser U. MMX Technology Extension to the Intel Architecture[J]. IEEE/ACM International Symposium on Microarchitecture, 1996, 16(4): 42-50

[3] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual[EB/OL]. http://www.intel.com/Assets/PDF/manual/252046.pdf, 2011

[4] Reinders J. AVX-512 instructions[EB/OL]. https://software.intel.com/en-us/blogs/2013/avx-512-instructions, 2013

[5] Reinders J. Additional AVX-512 instructions[EB/OL]. https://software.intel.com/en-us/blogs/additional-avx-512-instructions, 2014

[6] 辛乃军，陈旭灿，孙海燕，等. 基于 GCC 的高性能 DSP Matrix 向量指令集扩展[J]. 计算机工程与科学，2012，34(1)：58-63
Xin Nai-jun, Chen Xu-can, Sun Hai-yan, et al. Extending the Vector Instruction Set for High-Performance DSP Matrixes Based on GCC[J]. Computer Engineering and Science, 2012, 34(1): 58-63

[7] Intel Corporation. IA32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture[M]. Intel Press, 2004

[8] SIMD [EB/OL]. http://en.wikipedia.org/wiki/SIMD, 2014

[9] Allen R, Kennedyk. 现代体系结构的优化编译器[M]. 张兆庆，乔如良，冯晓兵，等译. 北京：机械工业出版社，2004
Allen R, Kennedy K. Optimizing compilers for modern architectures: a dependence-based approach[M]. Zhang zhao-qing, Qiao Ru-liang, Feng Xiao-bing. San Francisco: Morgan Kaufmann, 2002

[10] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[C]// Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2000: 145-156

[11] Prieto M, Piñuel L, Catthoor F, et al. Improving superword level parallelism support in modern compilers[C]// Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS'05). IEEE, 2005: 303-308

[12] Barik R, Zhao J, Sarkar V. Efficient selection of vector instructions using dynamic programming[C]// 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2010: 201-212

[13] Kudriavtsev A, Kogge P. Generation of permutations for SIMD processors[J]. ACM SIGPLAN Notices, ACM, 2005, 40(7): 147-156

[14] Manniesing R, Karkowski I, Corporaal H. Automatic SIMD parallelization of embedded applications based on pattern recognition [C]// Euro-Par 2000 Parallel Processing. Springer Berlin Heidelberg, 2000: 349-356