基于 LLVM-Polly 的自动向量化优化

黄驻峰[1]　商建东[1]　陈梦尧[1]　韩林[1]

[1]郑州大学河南省超级计算中心、信息工程学院，郑州 450000
(hzhufeng@outlook.com)

Automatic vectorization optimization based on LLVM-Polly

Huang Zhufeng[1]　Shang Jiandong[1]　Chen Mengyao[1]　Han Lin[1]

[1](Henan Supercomputing Center of Zhengzhou University, College of Information Engineering, Zhengzhou 450000)

Abstract In the compilation system, it is often difficult to give the optimal choice of how to implement the appropriate program transformation combination, which affects the effect of subsequent compilation optimization. In automatic vectorization compilation, the use of reasonable loop transformation will directly affect its vectorization ability. Polyhedron model is an effective method of loop transformation and code generation, but it has not been widely used in automatic vectorization compilation optimization. Based on the Polly polyhedron implementation in the open source compiler LLVM, a loop transformation method for optimizing automatic vectorization is proposed. Firstly, the loop selection is used to find the most suitable loop level for vectorization; secondly, the target platform Cache is used to calculate the score of the loop level Segment size; Finally, the loop level is segmented according to the segment size by loop section, so that the level loop can fully consider the improvement of the locality of the data while satisfying the automatic vectorization. Through the test of some PolyBench, the average speed of 15.8 times is achieved under the Large data scale.

Key words　automatic vectorization; polyhedral model; loop selection; loop section; data locality

摘要　编译系统中往往对如何实施恰当的程序变换组合难以给出最优的选择，从而影响其后续编译优化的效果，在自动向量化编译中，采用合理的循环变换将直接影响其向量化能力。多面体模型是一种有效的循环变换及代码生成方法，但目前尚未广泛应用于自动向量化编译优化中。基于开源编译器 LLVM 中的 Polly 多面体，提出一种用于优化自动向量化的循环变换方法，首先利用循环选择寻找最适合向量化的循环层；其次依赖于目标平台 Cache 计算该循环层的分段大小；最后对该循环层按照分段大小进行循环分段，使得该层循环在满足自动向量化的同时也能充分考虑到提升数据的局部性。通过对部分 PolyBench 的测试，在 Large 数据规模下取得平均 15.8 倍的加速效果。

关键词 自动向量化；多面体模型；循环选择；循环分段；数据局部性

中图法分类号 TP314

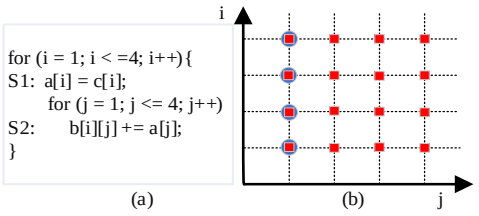
收稿日期：2020-07-03 修回日期：2020-08-28

1. 概述

SIMD 指令集(如 Altivec、Cell SPU 和 SSE)的自动向量化已经成为编译技术的一个研究热点。目前实现自动向量化的方式主要有三种：基于循环的传统向量化[1]、基于基本块的超字并行 SLP（Superword Level Parallelism）向量化[2]以及模式匹配向量化[3]。虽然编译技术在过去十年中取得了重大进展并对编译器产生深远的影响[4-12]。但是，现代 SIMD 体系结构在进行自动向量化编译时仍然会受到多种大型程序的影响，例如数据间的复杂依赖关系[8]、数据对齐问题[9]、归约变量的识别与处理，跨步访存问题[11]。此外，自动向量化也面临着处理复杂控制流中数据重组的难题，如 if-conversion[13]及其外层的循环向量化[14]。循环变换可以显著地提升自动向量化的收益，但由于循环变换方法多、组合复杂且相互干扰导致编译器很难选择如何实施恰当的循环变换及其组合策略。多面体编译具有表示一个或者多个循环变换的能力，为改进编译器中的循环变换策略提供了一种有效的手段。

多面体编译技术[15-19]是指在循环边界约束条件下将语句实例表示成空间多面体,并通过这些多面体上的几何操作来分析和优化程序的编译技术。这种模型称为多面体模型[20]。多面体模型使用 Scop[21-22]描述程序中的循环嵌套并抽象出四种数学结构表示程序，每一条语句都有一个迭代域（Iteration domain），每个内存引用（write and read）用访存映射关系（Access Map Relation）描述，数据依赖（Data dependences）用依赖多面体表示，最后循环变换用调度函数（Schedule）表示。迭代域是程序中每条语句在一次循环迭代下语句动态实例的集合，其动态实例用一组仿射不等式表示。访存映射关系用迭代向量的仿射函数形式表示语句访问数据的位置，用于表示语句实例与访存数据之间的映射关系。依赖关系是根据数据的引用定义的，如果两个语句的动态实例引用同一个内存数据单元并至少有一个是写操作时，称两个语句之间存在数据依赖关系。为了遵守程序的执行顺序，数据的生产实例一定要在数据的消费实例之前执行。Schedule 使用调度函数来表示在满足依赖关系的前提下语句实例之间的偏序或全序执行顺序，该函数对迭代域中点（即语句动态实例）进行重新排序，调度函数通常可以应用多个循环变换序列的内容。

如图 1(a)所示双层循环嵌套,其对应的多面体表示形式如图 1(b),蓝色圆形对应 S1 的语句动态实例,红色正方形对应 S2 的语句动态实例;图 1(c)中列出了该循环对应的迭代域、访存映射关系、依赖关系和调度。



$$Domain = \{S_1(i): 1 \le i \le 4; S_2(i,j): 1 \le i \le 4 \cap 1 \le j \le 4\}$$

$$Schedule = \{S_1(i) \to (i,0); S_2(i,j) \to (i,1,j)\}$$

$$Write = \{S_1(i) \to a(i): 1 \le i \le 4; S_2(i,j) \to b(i,j): 1 \le i \le 4 \cap 1 \le j \le 4\}$$

$$Read = \{S_2(i,j) \to a(i): 1 \le i \le 4 \cap 1 \le j \le 4\}$$

$$Dependence = \{S_1(i) \to S_2(i,j): 1 \le i \le 4 \cap 1 \le j \le 4\}$$

(c)

图 1 循环嵌套及其对应的多面体模型

Fig.1 A loop nest with its polyhedral representations

2. LLVM 中的多面体变换

LLVM[23]是近十年新兴的轻量级开源编译系统，它提供模块化和可重用的编译优化工具集合，用于程序的编译时、链接时、运行时以及空闲时的全时优化。

Polly[24]是 LLVM 面向循环和数据局部性的多面体编译工具。在 LLVM 编译系统中 Polly 架构如下图 2 所示。从 LLVM-IR （LLVM Intermediate Representation）开始，它检测并提取程序的循环内核，对于每个内核导出数学模型，该数学模型精确地描述了内核中的各个计算操作和访存操作。在 Polly 中，对该数学模型执行数据依赖分析、访存映射分析和调度转换。在应用新的调度转换之后，将重新生成优化后的抽象语法树（AST）转换成 LLVM-IR 并插入到 LLVM-IR 模块。

Polly 优化器内部主要分为三个阶段实现。第一阶段检测能够转换为多面体表示的循环，检测到的循环被称之为静态控制单元（Scop），在检测到 Scop 后可以将其转换为多面体模型。第二阶段在多面体表示中进行优化，调度变换构成多面体编译工具的中间优化部分，Polly 调度分为程序的原始调度、添加 ISL[25]优化之后的调度以及额外的优化调度。第三阶段生成新的 LLVM-IR，该阶段首先要生成抽象语法树，根据目标程序语言规范将 AST 转变成最终代码。在代码

收稿日期：2020-07-03 修回日期：2020-08-28

生成期间，Polly 检测并行循环生成 OpenMP 或 SIMD 代码。Polly 的分析和变换能力非常强大，章节 3 只使用其中的循环选择，循环分段，循环展开部分变换。

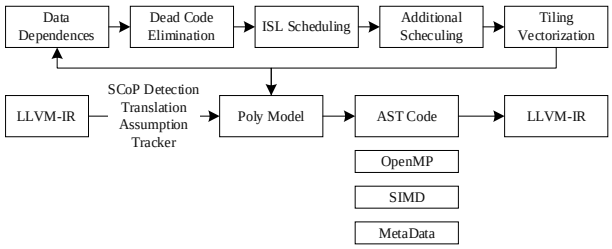


图 2 Polly 架构

Fig.2 Polly's architecture

3. 多面体变换与自动向量化

3.1 循环选择

程序中多层循环嵌套经过 Polly 识别之后会转换成多面体表示形式。该表示形式经过 Polly 默认 ISL 调度优化器之后多层循环嵌套大多会转换成完美循环嵌套，完美循环嵌套更有利于分析循环的特征和施加额外的优化。图 3 展示了 Polly 调度优化的整个过程，图 3（a）表示 PolyBench 中 correlation 内核的矩阵乘示例源码；图 3（b）表示经过 Polly ISL 默认调度优化后 correlation 内核矩阵乘代码；图 3（c）表示在图 3（b）基础上添加多维度默认分块后 correlation 内核矩阵乘代码；图 3（d）表示在图 3（b）基础上添加选择 j2 层循环分段后交换到最内层调度后 correlation 内核矩阵乘代码。观察图 3 示例不难发现图 3（d）的调度优化比图 3（c）的调度优化效果要好，图 3（d）j2 层分段交换到最内层后语句 S3 中访存操作变成连续访存，可以充分利用 SIMD 单元。

```
for (j1 = 0; j1 < _PB_M-1; j1++) {
S1:symmat[j1][j1] = 1.0;
  for (j2 = j1+1; j2 < _PB_M; j2++){
S2: symmat[j1][j2] = 0.0;
    for (i = 0; i < _PB_N; i++)
S3:  symmat[j1][j2] += (data[i][j1] *
data[i][j2]);
S4:  symmat[j2][j1] = symmat[j1][j2];
  }
}
```
(a)

```
for (int c0 = 0; c0 <= floord(p_0 - 2, 32); c0 += 1)
for (int c1 = 0; c1 <= -c0 + (p_0 - 2) / 32; c1 += 1)
for (int c2 = 0; c2 <= floord(p_1 - 1, 32); c2 += 1)
for (int c3 = 0; c3 <= min(31, p_0- 32 * c0 - 32 * c1 - 2); c3 += 1)
for (int c4 = 0; c4 <= min(31, p_0- 32 * c0 - 32 * c1 - c3 - 2); c4 += 1)
for (int c5 = 0; c5 <= min(31, p_1- 32 * c2 - 1); c5 += 1)
S3(32 * c0 + c3, 32 * c1 + c4, 32 * c2 + c5);
```
(c)

```
for (int c0 = 0; c0 < p_0 - 1; c0 += 1)
  for (int c1 = 0; c1 < p_0 - c0 - 1; c1 += 1)
    for (int c2 = 0; c2 < p_1; c2 += 1)
      S3(c0, c1, c2);
```
(b)

```
for (int c0 = 0; c0 < p_0 - 1; c0 += 1)
for (int c1 = 0; c1 <= (p_0 - c0 - 2) / 1024; c1 += 1)
for (int c2 = 0; c2 < p_1; c2 += 1)
for (int c4 = 0; c4 <= min(1023, p_0 - c0 - 1024 * c1 - 2); c4 += 1)
S3(c0, 1024 * c1 + c4, c2);
```
(d)

图 3 correlation 内核中的矩阵乘示例

Fig.3 Matrix multiplication example in the correlation kernel

在 ISL 调度优化的基础上计算最适合向量化的维度。图 3（a）S3 语句对应于多条中间表示指令，遍历 S3 中的每一条中间表示指令获取其对应的访存映射关系；根据每条指令的访存映射关系来对其所有维度按照最适合向量化的顺序进行排序；遍历结束从每条指令排序后的维度中选出综合最适合向量化的维度。Algorithm 1 展示了计算最适合向量化维度的整个过程。

```
Algorithm 1 计算最适合SIMD自动向量化的维度
calculateSIMDDim(Node) {
  PartialSchedule <- isl::manage(Node.get());
  map_n <- isl_union_map_n_map(PartialSchedule.get());
  if (map_n > 1) return; //现在只支持union_map =1的情况
  Stmt <- PartialSchedule;
  uint64_t ToPerIt = 0; //每次迭代Load数据总次数
  uint64_t MaxPerIt = 0; //迭代内最大数据类型
  //保存每层适合向量化的操作数量
  DenseMap <unsigned, unsigned> SIMDDims;
  Accesses = getAccessesInOrder(*Stmt);
  for Access in Accesses do
    if (!Access ->isLatestArrayKind()) // 当前操作对象不是数组
      continue;
    MaxPerIt = max(MaxPerIt, Access->getElementType());
    if (MemAccessPtr->isRead()) ToPerIt++;
    AccMap = Access ->getLatestAccessRelation();
    //收集每层循环适合向量化的操作数量到SIMDDims
    collectSIMDForDims(Stmt->getDomain(), AccMap, SIMDDims);
  endfor
  //反向遍历得到最大值对应的Dim
  bestDim <- max(SIMDDims);
}
```

3.2 循环分段

Polly 多面体模型的额外调度优化是采用多维度默认分块，分块大小为 32。研究发现沿着最适合向量化维度的分块带来的收益和多维度分块的收益相当[26]，甚至有些程序可以带来更好的加速效果，这主要得益于沿着最适合向量化维度分块暴露了外层（或内层）的并行性，充分利用了 LLVM 的自动向量化功能。但是随着分块大小的提升显然得到的收益是有限的，因为驻留在 L1 Cache 内的数据量是一定的，随着分块大小提升当 L1 Cache 利用率最高的时候数据会被换出，此时性能达到瓶颈。因此理想情况下，沿着最适合向量化维度的分块大小要小于等于 L1 Cache 的大小。此外，沿着最适合向量化维度分块后针对原来最内层不连续的访存现在对内层而言变成类似循环不变量的形式，在最内层向量化时原来从内存被多次加载的数据，分块后从 L1 Cache 中加载，大大节省了访存开销。计算循环中适应于目标架构一维分块大小，将 3.1 节中得到的最适合向量化的维度按照此分块大小进行循环分段。利用 Polly 将循环分段的新循环交换到最内层。面向目标体系结构的一维分块计算方法如下。

(1) 计算每次迭代语句动态实例加载数组数据的总量。ToPerIt 表示每次迭代加载数组数据的总量。MaxPerIt 表示每次迭代加载的最大数据类型大小。

(2) 从目标架构获取硬件信息

CL1：L1 Cache 大小（KB）

RegW：向量寄存器宽度（bit）

RegN：向量寄存器的数量

LatencyVectorLoad：发出两个向量 Load 操作指令的最小时钟周期。

ThroughputVectorLoad：每个时钟周期发出的 Load 操作指令的个数。

使用的 SIMD 指令集架构

(3) 根据 ToPerIt 计算最大驻留在 L1 Cache 的数据量 DataSize。

$$DataSize = C_{L1} * 1024/(ToPerIt * 8)$$

(4) 向量寄存器容纳的最大数据的数量 Nvec。

$$Nvec = RegW/(MaxPerIt * 8)$$

(5) 最终计算沿着最适合向量化维度的分块大小 Tblock 为：

$$T_{block} = \lfloor DatSize/Nvec \rfloor * Nvec \quad (1)$$

(6) 根据多面体编译对后期 LLVM 循环变换和自动向量化影响的研究，适当评估大于和小于最适合向量化维度的分块大小。此部分参考 3.3。Algorithm 2 展示了计算沿着最适合向量化维度分块大小的计算方法。

```
Algorithm 2 计算沿着最适合向量化维度分块大小
calculateTileSizeOfSIMDDim(TileSizes, bestDim) {
  // TTI:TargetTransformInfom目标平台信息
  FirstCacheLevelSize <- getTargetCacheParameters(TTI);
  RegisterBitwidth <- TTI->getRegisterBitWidth(true);
  Nvec = RegisterBitwidth / MaxPerIt;
  DataSize = FirstCacheLevelSize * 8 / (ToPerIt * MaxPerIt);
  Tblock = floor(DataSize/Nvec) * Nvec;
  TileSizes[bestDim] = Tblock;
  for Dim in {Dims < bestDim} do
    TileSizes[bestDim] = 32; //小于bestDim的循环层分块默认为32
  endfor
}
```

这里给出图 3（a）程序分块大小计算示例。循环中核心是语句 S3：symmat[j1][j2] += (data[i][j1] * data[i][j2])。不难发现最适合向量化的维度是 j2 层循环，当对 j2 层循环分块并交换到内层时，数据 symmat[i][j2]和 data[i][j2]是连续访存，对于分块后交换到最内层循环的 c4 来说 data[i][j1]是类似循环不变量的形式，在最内层循环执行时该数据都驻留在 L1 Cache 中。S3 语句在每次迭代动态实例中有 3 次加载操作，3 次加载操作的最长数据类型为 double、向量寄存器长度为 256bit、L1 Cache 大小为 32KB，代入公式（1）计算得分块大小为 1364。

3.3 循环展开

源程序中的循环嵌套经过 Polly 中 ISL 调度后大多转换成完美循环嵌套，这导致每个完美循环嵌套内的指令数目减少，为了暴露 ISL 调度后最内层循环更多的并行性，对适当的循环层进行循环展开。根据 3.2 节中读取的目标架构信息，当最适合向量化维度不等于最内层时。使用公式（2）计算最内层循环需要展开的次数 UnrollCount，以增加后期 LLVM 进行 SLP 自动向量化的机会，充分利用目标架构的指令级并行，同时减少最内层循环由于循环次数过多带来的分支开销。依赖于 LLVM 编译器目标平台对循环内寄存器使用情况的分析，从 LLVM 中获取循环内循环展开次数的阀值 RegThresholdOfLLVM。针对 ISL 调度后的最内层循环按照循环展开次数 $\min(RegThresholdOfLLVM, UnrollCount)$实施循环分段。在 Polly 生成 LLVM-IR 之后，借助 LLVM 编译器的循环展开对该分段循环进行展开。

$$UnrollCount = \lceil \sqrt{(Nvec * \text{LatencyVectorLoad} * \text{ThroughputVectorLoad})/Nvec} \rceil * Nvec \quad (2)$$

4. 实验与分析

4.1 测试环境

CPU：Intel(R) Xeon(R) CPU E5-2682 v4 @ 2.50GHz

L1 Cache（data）：32KB

SIMD Extension verion：AVX2

LLVM+clang+polly version：LLVM 10.0.0

4.2 测试用例

PolyBench/C 3.2 是 LLVM 多面体模型 Polly 提供的包含静态控制单元的基准性能测试集。测试用例选择 PolyBench/C 3.2 中 correlation、covariance、doitgen、gramschmidt 四个计算内核。correlation、covariance 计算内核核心是矩阵乘。Doitgen 计算核心是四层非完美循环嵌套。Gramschmidt 计算核心是三层非完美循环嵌套。计算内核详见本文结束图 6~9。

4.3 测试数据

针对 PolyBench/C 3.2 中 4 个计算内核的不同数据规模进行测试。测试基准 clang -O3 -mavx2 默认选项，Polly ISL 调度和默认分块调度 clang -O3 -polly -mavx2，Polly ISL 调度和本文提出的调度 clang -O3 -polly -tile-simd -mavx2。测试数据为 5 次测试舍弃最大值最小值的 3 次平均值。测试数据（数据类型：double；表格单位为秒）如表 1 所示。图 4 可以直观反映表 1 数据加速比对比情况。图 4 左 4 列为沿着最适合向量化层分块数据，右 4 列为 Polly 默认调度分块数据，clang -O3 -mavx2 基准测试数据加速比默认为 1，图 4 中不再展示。图 5 表示在数据规模为默认值（Standard）时，在沿着最适合向量化维度分块后开启向量化和不开启向量化的对比情况，从图 5 中可以看出在开启向量化是会带来一定的收益，收益不等于向量寄存器一次容纳多少个数据元素的原因是内核中有多个循环，且只有一部分循环满足向量化的条件。从图 4 和图 5 的数据中可以看出沿着最适合向量化维度分块不仅可以提升数据局部性，还可以充分发挥

LLVM 编译器的自动向量化功能，在最大规模下 correlation 和 covariance 加速比最高达到 25 倍。而 Polly 的默认分块在测试数据规模很大的情况下仅仅有大约 2 倍的收益。

表 1 PolyBench/C 3.2 部分内核测试数据

Table1 PolyBench/C 3.2 Partial kernel test data

	clang	Mini 32	Small 500	Standard 1000	Large 2000	Extralarge 4000
correlation	O3 avx2 def	0.000025	0.129952	0.856701	13.65008	179.711087
	polly avx2 def	0.000017	0.13045	1.034727	9.018632	74.911343
	polly avx2 tile-simd	0.000016	0.037679	0.097092	0.823301	7.167223
covariance	O3 avx2 def	0.000025	0.134699	0.963622	14.92978	182.083564
	polly avx2 def	0.000018	0.154175	1.036958	8.651079	75.231366
	polly avx2 tile-simd	0.000019	0.03618	0.096649	0.769856	7.231367
doitgen	O3 avx2 def	0.000016	0.000587	0.462329	10.35875	1228.334672
	polly avx2 def	0.000014	0.000339	0.206032	3.858961	735.851835
	polly avx2 tile-simd	0.000014	0.000343	0.037726	0.57068	158.182264
gramschmidt	O3 avx2 def	0.000051	0.008947	1.095135	30.75116	350.469272
	polly avx2 def	0.00003	0.005642	0.842577	17.6662	246.914824
	polly avx2 tile-simd	0.000043	0.001726	0.06487	3.375725	56.492523

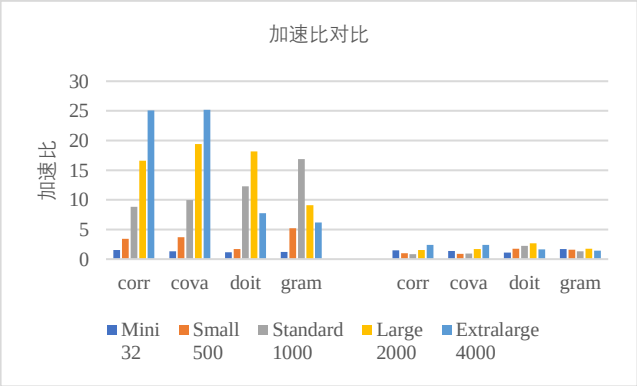


图 4 表 1 数据加速比

Fig.4 Table 1 Data acceleration ratio

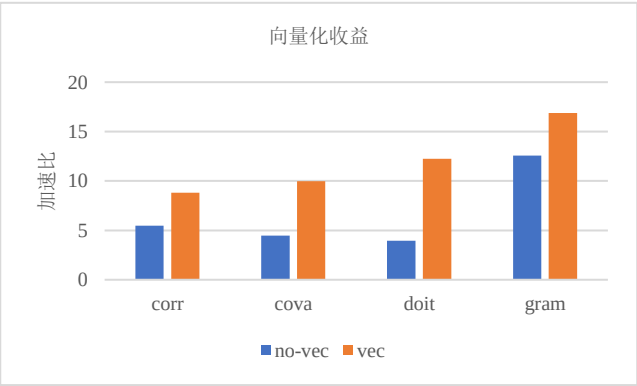


图 5 默认数据规模向量化收益

Fig.5 The default data size vectorization benefit

5. 结论

编译器对多种复杂情况下的自动向量化支持能力有限，本文基于 Polly 多面体模型提出一种用于优化自动向量化的循环变换方法。该方法首先计算循环每次迭代数据加载和存储时最适合做向量化的循环层 L，其次结合目标平台信息计算面向一维的分块大小 TBlock，最后依赖多面体模型的循环变换将 L 层按照 TBlock 大小进行分块并交换到内层循环。实验表明该方法相比 Polly 的默认分块获得更好的加速效果。对于 correlation 和 covariance 当数据规模大于 1000 时，此时 L1 Cache 不足以装下循环内所有迭代数据，但随着数据规模成倍数的增加，加速效果仍在不断增加，这与理论分析是一致的。值的注意的是对于 doitgen 和 gramschmidt 计算内核当数据规模较大时加速效果反而下降。未来工作一方面研究 doitgen 和 gramschmidt 计算内核在大数据规模下添加本文优化时加速到达瓶颈的原因。另一方面与 Openmp 结合

充分发挥多核的性能。

参 考 文 献

[1] Randy Allen, Ken Kennedy 著. 张兆庆,乔如良,冯晓兵等译. 现代体系结构的优化编译器[M]. 北京:机械工业出版社, 2004.

[2] Samuel Larsen, Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets[C]. In Proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2000, 145—156.

[3] Boekhold M, Karkowski I, Corparaal H. Transforming and parallelizing ANSI C programs using pattern recognition[R]. In: Lecture Notes in Computer Science 1593, 1999, 673.

[4] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao, “An integrated Simdization framework using virtual vectors,” in ICS, 2005.

[5] A. J. C. Bik, The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance. Intel Press, 2004.

[6] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, “Automatic intra-register vectorization for the Intel architecture,” IJPP, vol. 30, no. 2, pp. 65–98, 2002

[7] D. Nuzman and A. Zaks, “Autovectorization in GCC – two years later,” in the GCC Developer’s summit, June 2006.

[8] K. Kennedy and J. Allen. Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann, 2002.

[9] A. Eichenberger, P. Wu, and K. O’Brien. Vectorization for simd architectures with alignment constraints. In PLDI, 2004.

[10] S. Larsen and S. P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In PLDI, 2000.

[11] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In PLDI, 2006

[12] Porpodas V , Rocha R C O , Luís F. W. Góes. Look-ahead SLP: auto-vectorization in the presence of commutative operations[C]// International Symposium on Code Generation and Optimization. 2018.

[13] J. Shin, M. Hall, and J. Chame, “Superword-level parallelism in the presence of control flow,” in CGO, March 2005.

[14] D. Nuzman and A. Zaks, “Outer-loop vectorization - revisited for short SIMD architectures,” in PACT, October 2008.

[15] Bondhugula U , Hartono A , Ramanujam J , et al. A practical automatic polyhedral parallelizer and locality optimizer[J]. Acm Sigplan Notices, 2008, 43(6):101-113.

[16] Kong M , Veras R , Stock K , et al. When polyhedral transformations meet SIMD code generation[J]. acm sigplan notices, 2013.

[17] Uday Bondhugula, Aravind Acharya, Albert Cohen. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests[M]. ACM, 2016.

[18] Verdoolaege, Sven and Alexandre Isoard. “Extending Pluto-Style Polyhedral Scheduling with Consecutivity Sven Verdoolaege.” (2018).

[19] Jongen, M. H., Waeijen, L. J. W., Jordans, R., Jozwiak, L., & Corporaal, H. (2018). Optimization through recomputation in the polyhedral model. In Eighth International Workshop on Polyhedral Compilation Techniques: In conjunction with HiPEAC 2018

[20] 赵捷, 李颖颖, 赵荣彩. 基于多面体模型的编译"黑魔法"[J]. 软件学报, 2018, 029(008):2371-2396.

[21] 16. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. Intl. J. of Parallel Programming, 21(6):389–420, Dec. 1992

[22] 19. S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. International Journal of Parallel Programming, 34(3):261–317, June 2006.

[23] Lattner C , Adve V . LLVM: a compilation framework for lifelong program analysis & transformation[C]// International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004.

[24] Pouchet L N , Grlinger A , Andreas Simbürger, et al. Polly-polyhedral optimization in LLVM[C]// International Workshop on Polyhedral Compilation Techniques (IMPACT). 2011.

[25] Verdoolaege S . isl: An Integer Set Library for the Polyhedral Model[C]// Third International Congress Conference on Mathematical Software. Springer-Verlag, 2010.

[26] Feld D , Soddemann T , M Jünger, et al. Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation[C]// Impact. 2013.

Huang Zhufeng,borun in 1995. Master. The main research field is advanced compilation technology.

Shang Jiandong,born in 1968. Professor and PhD supervisor. Mainly engaged in research on 3D visualization technology of smart cities.

Chen Mengyao,born in 1995. Master. The main research field is advanced compilation technology.

Han Lin, born in 1978. Associate Professor, CCF member, the main research field is high-performance computing, advanced compilation technology.

```
/* Determine mean of column vectors of input data matrix */
 for (j = 0; j < _PB_M; j++)
   {
     mean[j] = 0.0;
     for (i = 0; i < _PB_N; i++)
       mean[j] += data[i][j];
     mean[j] /= float_n;
   }

 /* Determine standard deviations of column vectors of data matrix. */
 for (j = 0; j < _PB_M; j++)
   {
     stddev[j] = 0.0;
     for (i = 0; i < _PB_N; i++)
       stddev[j] += (data[i][j] - mean[j]) * (data[i][j] - mean[j]);
     stddev[j] /= float_n;
     stddev[j] = sqrt_of_array_cell(stddev, j);
     /* The following in an inelegant but usual way to handle
        near-zero std. dev. values, which below would cause a zero-
        divide. */
     stddev[j] = stddev[j] <= eps ? 1.0 : stddev[j];
   }

 /* Center and reduce the column vectors. */
 for (i = 0; i < _PB_N; i++)
   for (j = 0; j < _PB_M; j++)
     {
       data[i][j] -= mean[j];
       data[i][j] /= sqrt(float_n) * stddev[j];
     }

 /* Calculate the m * m correlation matrix. */
 for (j1 = 0; j1 < _PB_M-1; j1++)
   {
     symmat[j1][j1] = 1.0;
     for (j2 = j1+1; j2 < _PB_M; j2++)
       {
         symmat[j1][j2] = 0.0;
         for (i = 0; i < _PB_N; i++)
           symmat[j1][j2] += (data[i][j1] * data[i][j2]);
         symmat[j2][j1] = symmat[j1][j2];
       }
   }
 symmat[_PB_M-1][_PB_M-1] = 1.0;
```

图 6 kernel_correlation

Fig.6 kernel_correlation

```
/* Determine mean of column vectors of input data matrix */
 for (j = 0; j < _PB_M; j++)
   {
     mean[j] = 0.0;
     for (i = 0; i < _PB_N; i++)
       mean[j] += data[i][j];
     mean[j] /= float_n;
   }

 /* Center the column vectors. */
 for (i = 0; i < _PB_N; i++)
   for (j = 0; j < _PB_M; j++)
     data[i][j] -= mean[j];

 /* Calculate the m * m covariance matrix. */
 for (j1 = 0; j1 < _PB_M; j1++)
   for (j2 = j1; j2 < _PB_M; j2++)
     {
       symmat[j1][j2] = 0.0;
       for (i = 0; i < _PB_N; i++)
         symmat[j1][j2] += data[i][j1] * data[i][j2];
       symmat[j2][j1] = symmat[j1][j2];
     }
```

图 7 kernel_covariance

Fig.7 kernel_covariance

```
for (r = 0; r < _PB_NR; r++)
  for (q = 0; q < _PB_NQ; q++)  {
    for (p = 0; p < _PB_NP; p++)  {
      sum[r][q][p] = 0;
      for (s = 0; s < _PB_NP; s++)
        sum[r][q][p] = sum[r][q][p] + A[r][q][s] * C4[s][p];
    }
    for (p = 0; p < _PB_NR; p++)
      A[r][q][p] = sum[r][q][p];
  }
```

图 8 kernel_doitgen

Fig.8 kernel_doitgen

```
for (k = 0; k < _PB_NJ; k++)
  {
    nrm = 0;
    for (i = 0; i < _PB_NI; i++)
      nrm += A[i][k] * A[i][k];
    R[k][k] = sqrt(nrm);
    for (i = 0; i < _PB_NI; i++)
      Q[i][k] = A[i][k] / R[k][k];
    for (j = k + 1; j < _PB_NJ; j++)
      {
        R[k][j] = 0;
        for (i = 0; i < _PB_NI; i++)
          R[k][j] += Q[i][k] * A[i][j];
        for (i = 0; i < _PB_NI; i++)
          A[i][j] = A[i][j] - Q[i][k] * R[k][j];
      }
  }
```

图 9 kernel_ gramschmidt

Fig.9 kernel_ gramschmidt