Б. Я. Штейнберг, О. Б. Штейнберг, А. А. Василенко

Слияние циклов для локализации данных

Аннотация. Для улучшения локализации данных используется слияние циклов. Слияние циклов, имеющих общие переменные, может ускорить исполнение за счёт уменьшения количества кэш-промахов. Это преобразование известно давно, но компиляторы выполняют его лишь для простейших случаев.

Наши улучшенные алгоритмы используют предварительные преобразования для корректного слияния циклов, имеющих разное количество итераций и информационные зависимости.

Ключевые слова и фразы: слияние циклов, оптимизирующий компилятор, преобразования программ, локальность данных, оптимизация обращений к памяти.

Введение

В течение нескольких десятилетий наблюдается рост производительности процессоров (выполнение операций умножения) примерно на 30% в год, а рост пропускной способности памяти (скорость чтения/записи данных из оперативной памяти) растёт только на 9% [1]. Таким образом, если ранее основное время работы вычислительных систем уходило на вычислительные операции, то сейчас – на доступ к данным.

Актуальность оптимизации параллельных вычислений подчеркивается тем, что параллельные программы часто показывают низкую эффективность [2]. Для многих приложений наблюдается, что при повышении уровня параллелизма с некоторого момента скорость выполнения программ даже замедляется. Проблема отставания реальной производительности программного обеспечения от пиковой производительности вычислительных систем отмечается как отечественными, так и зарубежными исследователями [3], [4].

Наиболее типичная последовательность прохождения данных в компьютере с общей памятью, иерархией кэш-памяти и многоядерным процессором выглядит примерно следующим образом:

$$RAM \rightarrow L3 \rightarrow L2 \rightarrow L1 \rightarrow R \rightarrow ALU$$

Распараллеливание происходит после $L3$ – у каждого процессорного ядра свой $L2$ -кэш. Векторизация происходит после $L1$ через векторные регистры. Каждая стрелка означает передачу данных: время передачи уменьшается в порядке слева направо.

У разных программ узкое место производительности может быть в разных местах. При этом используются различные приемы оптимизации. В частности, если в программе самые глубоко вложенные циклы имеют в теле много операций с небольшим объемом данных, то вполне интересна оптимизация регистров. Но большинство используемых современных программ имеет много данных, с которыми выполняется относительно малое число операций. Для таких программ узким местом является доступ к памяти.

В данной работе рассматривается преобразование программ «слияние циклов». Это преобразование может способствовать локализации данных для оптимизации использования кэш-памяти. В современных оптимизирующих компиляторах такое преобразование используется, но для очень простых случаев.

1. Условия корректности и эффективности преобразования «слияние циклов»

Преобразование «слияние циклов» является обратным к преобразованию программ «разбиение цикла». Поэтому условие эквивалентности слияния циклов может быть выражено через условие эквивалентности разбиения циклов: если после слияния циклов результирующий цикл можно разбить так, что в результате получатся исходные циклы (которые были до слияния) – значит, слияние эквивалентно. Для разбиения циклов условие эквивалентности состоит в том, чтобы в исходном цикле между разбиваемыми частями не было дуг графа информационных связей, направленных снизу-вверх (см., например, [5]).

Эффект полезности слияния циклов известен, опишем его для полноты изложения. Пусть имеются два последовательно записанных (в конкатенации) цикла, обрабатывающих общие массивы b , c :

C++ —

```

1 for (i = 1; i < N; i++)
2     LoopBody1(a[i], b[i], c[i], ...);
3
4 for (i = 1; i < N; i++)
5     LoopBody2(b[i], c[i], d[i], ...);

```

(1)

(2)

Предположим, что данные, которые обрабатывает (1), не помещаются в кэш-памяти (уровень памяти не имеет значения). Тогда, на последних итерациях (1) первые элементы массивов b , c будут вытеснены из кэш-памяти. Поэтому, при выполнении (2) массивы b , c придется подкачивать в кэш-память заново.

Предположим, что (1) и (2) можно слить в один цикл:

C++ —

```

1 for (i = 1; i < N; i++){
2     a[i] = LoopBody1(b[i], c[i], ...);
3     b[i] = LoopBody2(c[i], d[i], ...);
4 }

```

Тогда элементы массивов b , c считываются в кэш-память один раз и используются для выполнения каждого исходного тела цикла.

2. Слияние циклов с использованием вспомогательных преобразований

Использование слияния циклов для локализации данных отмечено в работах [6], [7], [8]. Однако, в этих работах либо не рассмотрены случаи, при которых сливаемые циклы связаны информационными зависимостями относительно вышестоящих циклов, либо лишь упоминается о том, что основную сложность при слиянии циклов вызывает анализ возникших зависимостей, при этом сам анализ остается нерассмотренным. Слияние циклов – известное преобразование, поэтому входит в список оптимизаций большинства современных компиляторов. Например, компилятор компании Oracle проводит анализ информационных зависимостей при слиянии циклов¹, однако циклы, содержащие такие зависимости, признаются компилятором небезопасными для слияния. В документации приводится следующий пример:

¹<https://docs.oracle.com/cd/E19205-01/820-7598/bjaew/index.html>

C++ —

```

1 for (i=0; i < 100; i++){
2     a[i] = a[i] + b[i];
3 }
4 for (i=0; i < 100; i++){
5     a[i+1] = a[i] * d[i];
6 }

```

Безусловно, слияние циклов при некорректном применении (в случае возникновения дополнительной информационной зависимости) может привести к неэквивалентной программе. Однако к элементам класса циклов, содержащих зависимости, подобные рассмотренным в документации Oracle, может быть применено слияние после дополнительных преобразований, например, расщепления цикла. Не рассматривается и использование цепочки вспомогательных преобразований (таких как «введение временных массивов», «растягивание скаляров», «перестановка операторов») для слияния. Распараллеливающая система Pluto² и система анализа и преобразований пространства итераций гнезда циклов Polly³ изучают гнезда циклов с точки зрения распараллеливания и локализации данных. В работе [6] рассматриваются как тайлинг, так и слияние циклов.

Алгоритм заключается в поиске пространства итераций, для которого возможно произвести слияние циклов с сохранением эквивалентности программы. Для этого для каждой информационной зависимости (кроме входной) определяются множества значений индексных выражений, далее вычисляется пересечение этих множеств. Затем для каждого цикла находится подмножество пространства итераций, для которого множество значений индексных выражений совпадает с этим пересечением. Итоговое пространство итераций для каждого цикла является пересечением всех найденных для этого цикла подмножеств.

Пример 1. Следующие два цикла нельзя слить непосредственно (нарушится корректность, результирующий цикл не будет эквивалентен исходному). Нарушение корректности связано с характером информационной зависимости между вхождением переменной *b*: после слияния циклов возникает дуга графа информационных связей, ведущая «снизу вверх».

²<http://pluto-compiler.sourceforge.net>

³<https://www.sites.google.com/site/parallelizationforllvm/why-not-polly>

```

1 for (i = 10; i < 20; i++)
2   a[i] = b[2 * i - 9];
3 for (i = 10; i < 20; i++)
4   b[2 * i - 7] = d[i];

```

Но если от первого цикла отщепить начальные итерации (в данном случае одну), а во втором цикле отщепить последние итерации (в данном случае тоже только одну), затем все полученные циклы привести к каноническому виду (левая граница и шаг цикла равны единице), то слияние циклов будет корректно.

Информационную зависимость образуют вхождения переменной b , обозначим множество значений индексных выражений массива b в первом цикле как B_1 , во втором цикле как B_2 . I_1 и I_2 — пространства итераций, для которых возможно провести слияние.

$$B_1 = \{11, 13, 15, \dots, 29\}, \quad B_2 = \{13, 15, 17, \dots, 31\}$$

$$B_1 \cap B_2 = \{13, 15, \dots, 29\}$$

$$I_1 = [11, 19], \quad I_2 = [10, 18]$$

Расцелим циклы, опираясь на найденные множества:

```

1 for (i = 10; i < 11; i++)
2   a[i] = b[2 * i - 9];
3 for (i = 11; i < 20; i++)
4   a[i] = b[2 * i - 9];
5 for (i = 10; i < 19; i++)
6   b[2 * i - 7] = d[i];
7 for (i = 19; i < 20; i++)
8   b[2 * i - 7] = d[i];

```

Проведем канонизацию второго и третьего циклов:

```

1 for (i = 10; i < 11; i++)
2   a[i] = b[2 * i - 9];
3 for (i = 0; i < 9; i++)
4   a[i + 11] = b[2 * i + 13];
5 for (i = 0; i < 9; i++)
6   b[2 * i + 13] = d[i + 10];
7 for (i = 19; i < 20; i++)
8   b[2 * i - 7] = d[i];

```

После этого можно провести слияние второго и третьего циклов:

```

1  for (i = 10; i < 11; i++)
2      a[i] = b[2 * i - 9];
3  for (i = 0; i < 9; i++){
4      a[i + 11] = b[2 * i + 13];
5      b[2 * i + 13] = d[i + 10];
6  }
7  for (i = 19; i < 20; i++)
8      b[2 * i - 7] = d[i];

```

Далее для упрощения кода можно сделать раскрутки коротких первого и последнего циклов.

3. О выборе вспомогательных преобразований, позволяющих выполнить слияние циклов, имеющих информационные зависимости относительно внешнего цикла

Иногда, если сливаемые циклы находятся внутри некоторого объемлющего цикла, то можно к внешнему циклу применить преобразование циклического сдвига [9], после чего слияние может стать возможным (эквивалентным).

Рассмотрим псевдокод, в котором блоки B1, B2 – это циклы, слияние которых целесообразно.

```

1  for (i = 1; i < N; i++) {
2      B1(i);
3      B2(i);
4  }

```

Для слияния циклов (соответствующих блокам B1, B2) необходимо устранить ведущие «снизу-вверх» информационные зависимости между ними. Рассмотрим более задачу устранения зависимостей («ведущих снизу-вверх») между блоками, которые в данном конкретном случае являются циклами. В общем случае эта задача решения не имеет, поскольку, например, рассматриваемые блоки могут быть связаны рекуррентной зависимостью. Известен ряд приемов (вспомогательных преобразований), которые позволяют выполнить устранение таких зависимостей в некоторых ситуациях. Эти приемы описаны для задач

автоматической векторизации циклов и для разбиения циклов (loop distribution), в которых также требуется устранение дуг информационной зависимости, ведущих «снизу-вверх». Такими преобразованиями программ являются «круговой сдвиг», «перестановка операторов», «введение временного массива», «растягивание скаляров» [9], [5]. Таким образом, алгоритм выполнения слияния циклов, имеющих зависимости относительно внешнего цикла, подобны алгоритмам разбиения цикла, описанным в [5] и [9].

Пример 2. Следующие два цикла нельзя слить непосредственно (нарушится корректность, результирующий цикл не будет эквивалентен исходному).

Исходная программа:

C++ —

```

1 for (i = 0; i < 1000; i++){
2     for (j = 2; j < 1000; j++){
3         a[i][j] = b[i][j];
4         c[i][j] = b[i][j - 2];
5     }
6     for (j = 2; j < 1000; j++){
7         b[i][j + 2] = d[i][j] + e[i][j];
8     }

```

После кругового сдвига:

C++ —

```

1 for (j = 2; j < 1000; j++){
2     b[0][j + 2] = d[0][j] + e[0][j];
3 for (i = 0; i < 999; i++){
4     for (j = 2; j < 1000; j++){
5         b[i + 1][j + 2] = d[i + 1][j] + e[i + 1][j];
6     for (j = 2; j < 1000; j++){
7         a[i][j] = b[i][j];
8         c[i][j] = b[i][j - 2];
9     }
10 }
11 for (j = 2; j < 1000; j++){
12     a[999][j] = b[999][j];
13     c[999][j] = b[999][j - 2];
14 }

```

Пример 3. Следующие два внутренних цикла нельзя слить как непосредственно, так и после их перестановки. Для слияния используется введение временного массива. Следует отметить, что дополнительно введенный массив одномерный, а слиянию будут подвергаться циклы, содержащие вхождения двумерного массива. Т.е., расходы на обработку одномерного массива могут окупиться уменьшением передвижений двумерного. Несложно построить аналогичные примеры и для трехмерных массивов.

Исходная программа:

C++ —

```
1 for (i = 0; i < 1000; i++){
2     for (j = 2; j < 1000; j++)
3         a[i][j] = b[i] * c[i][j] - b[i + 2];
4     for (j = 2; j < 1000; j++)
5         b[i + 1] = b[i + 1] * d[i][j] + e[i][j];
6 }
```

Шаг 1 (введение временных массивов):

C++ —

```
1 temp_b[0] = b[1];
2 for (i = 0; i < 1000; i++){
3     temp_b[i + 1] = b[i + 2];
4     for (j = 2; j < 1000; j++)
5         a[i][j] = b[i] * c[i][j] - temp_b[i + 1];
6     for (j = 2; j < 1000; j++)
7         b[i + 1] = temp_b[i] * d[i][j] + e[i][j];
8 }
```

Шаг 2 (перестановка операторов):

C++ —

```
1 temp_b[0] = b[1];
2 for (i = 0; i < 1000; i++){
3     temp_b[i + 1] = b[i + 2];
4     for (j = 2; j < 1000; j++)
5         b[i + 1] = temp_b[i] * d[i][j] + e[i][j];
6     for (j = 2; j < 1000; j++)
7         a[i][j] = b[i] * c[i][j] - temp_b[i + 1];
8 }
```

Шаг 3 (слияние циклов):

```
1 temp_b[0] = b[1];
2 for (i = 0; i < 1000; i++){
3     temp_b[i + 1] = b[i + 2];
4     for (j = 2; j < 1000; j++){
5         b[i + 1] = temp_b[i] * d[i][j] + e[i][j];
6         a[i][j] = b[i] * c[i][j] - temp_b[i + 1];
7     }
8 }
```

Шаг 4 (разбиение цикла):

```
1 temp_b[0] = b[1];
2 for (i = 0; i < 1000; i++)
3     temp_b[i + 1] = b[i + 2];
4 for (i = 0; i < 1000; i++)
5     for (j = 2; j < 1000; j++){
6         b[i + 1] = temp_b[i] * d[i][j] + e[i][j];
7         a[i][j] = b[i] * c[i][j] - temp_b[i + 1];
8     }
```

***Пример 4.** Слияние циклов со скалярной переменной с использованием кругового сдвига.*

Исходная программа:

```
1 for (i = 0; i < 1000; i++){
2     for (j = 0; j < 1000; j++)
3         a[i][j] = c[i][j] - b;
4     for (j = 0; j < 1000; j++){
5         d[i][j] = e[i]*b + f[i][j];
6         b = g[i]*h[i][j] + k[i][j];
7     }
8 }
```

Шаг 1 (круговой сдвиг, отщепление итерации):

C++ —

```

1 for (i = 0; i < 1000; i++){
2     for (j = 0; j < 999; j++)
3         a[i][j] = c[i][j] - b;
4     a[i][999] = c[i][999] - b;
5     d[i][0] = e[i] * b + f[i][0];
6     for (j = 0; j < 999; j++){
7         b = g[i] * h[i][j] + k[i][j];
8         d[i][j + 1] = e[i] * b + f[i][j + 1];
9     }
10    b = g[i] * h[i][999] + k[i][999];
11 }

```

Шаг 2 (перестановка операторов):

C++ —

```

1 for (i = 0; i < 1000; i++){
2     a[i][999] = c[i][999] - b;
3     d[i][0] = e[i] * b + f[i][0];
4     for (j = 0; j < 999; j++)
5         a[i][j] = c[i][j] - b;
6     for (j = 0; j < 999; j++){
7         b = g[i] * h[i][j] + k[i][j];
8         d[i][j + 1] = e[i] * b + f[i][j + 1];
9     }
10    b = g[i] * h[i][999] + k[i][999];
11 }

```

Шаг 3 (переименование переменной):

C++ —

```

1 for (i = 0; i < 1000; i++){
2     a[i][999] = c[i][999] - b;
3     d[i][0] = e[i] * b + f[i][0];
4     for (j = 0; j < 999; j++)
5         a[i][j] = c[i][j] - b;
6     for (j = 0; j < 999; j++){
7         b1 = g[i] * h[i][j] + k[i][j];
8         d[i][j + 1] = e[i] * b1 + f[i][j + 1];
9     }
10    b = g[i] * h[i][999] + k[i][999];
11 }

```

Шаг 4 (слияние циклов):

```
1 for (i = 0; i < 1000; i++){
2     a[i][999] = c[i][999] - b;
3     d[i][0] = e[i] * b + f[i][0];
4     for (j = 0; j < 999; j++){
5         a[i][j] = c[i][j] - b;
6         b1 = g[i] * h[i][j] + k[i][j];
7         d[i][j + 1] = e[i] * b1 + f[i][j + 1];
8     }
9     b = g[i] * h[i][999] + k[i][999];
10 }
```

Шаг 5 (разбиение цикла):

```
1 for (i = 0; i < 1000; i++){
2     a[i][999] = c[i][999] - b;
3     d[i][0] = e[i] * b + f[i][0];
4 }
5 for (i = 0; i < 1000; i++){
6     for (j = 0; j < 999; j++){
7         a[i][j] = c[i][j] - b;
8         b1 = g[i] * h[i][j] + k[i][j];
9         d[i][j + 1] = e[i] * b1 + f[i][j + 1];
10     }
11 }
```

Заключение

В представленной статье предлагаются новые алгоритмы слияния программных циклов. Эти алгоритмы существенно расширяют множество пар циклов, допускающих слияние. Представленные алгоритмы позволяют выполнять слияние циклов благодаря вспомогательным преобразованиям. Корректность выполняемых преобразований основана на анализе информационных зависимостей.

В работе рассмотрено два типа информационных зависимостей между сливаемыми циклами: зависимости, которые возникают относительно результирующего (после слияния) цикла; зависимости относительно цикла, объемлющего сливаемые циклы. Для каждого из этих двух случаев предлагается свой набор вспомогательных преобразований.

Слияние циклов может давать ускорение программного кода за счет улучшения локализации данных, уменьшая количество обменов данных между кэш-памятью процессора и оперативной памятью.

Полученные результаты вносят вклад в теорию преобразования программ и могут быть использованы при разработке оптимизирующих компиляторов.

Список литературы

- [1] S.L. Graham, M. Snir, C.A. Patterson. *Getting up to speed. The future of supercomputing*, Consensus Study Report, National Academies Press, 2005, 289 pp. [↑]₁₇
- [2] Л. К. Эйсымонт, В. С. Горбунов, Г. С. Елизаров. «На пути к экзафлопсному суперкомпьютеру: результаты, направления, тенденции», МСКФ 2012 (Москва, 1 ноября 2012), с. 5–7. [↑]₁₇
- [3] A. Mycroft. “Programming language design and analysis motivated by hardware evolution (invited presentation)”, *SAS 2007: Static Analysis*, International Static Analysis Symposium (22–24 August, 2007, Kongens Lyngby, Denmark), Springer, Berlin–Heidelberg, 2007, ISBN 978-3-540-74060-5, pp. 18–33. [↑]₁₇
- [4] А. И. Галушкин. «Стратегия развития современных супернейрокомпьютеров на пути к экзафлопсным вычислениям», *Приложение к журналу «Информационные технологии»*, 2012, №3. [↑]₁₇
- [5] О. Б. Штейнберг. «Минимизация количества временных массивов в задаче разбиения циклов», *Известия ВУЗов. Северо-Кавказский регион. Естественные науки*, 2011, №5(165), с. 31–35. [↑]_{18, 23}
- [6] J. Xue, Q. Huang, M. Guo. “Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences”, 2005 International Conference on Parallel Processing (ICPP’05) (14–17 June 2005, Oslo, Norway), 2005, pp. 107–115. [↑]_{19, 20}
- [7] K. Barton, J. Doerfert, H. Finkel, M. Kruse. “Loop fusion, loop distribution and their place in the loop optimization pipeline”, EuroLLVM 2019 Final (April 9, 2019). [↑]₁₉
- [8] E. Wang, Q. Zhang, B. Shen, X. Lu, Q. Wu, Y. Wang. *High-Performance Computing on the Intel® Xeon Phi™. How to Fully Exploit MIC Architectures*, Springer, 2014, ISBN 978-3-319-06485-7, xxiii+338 pp. [↑]₁₉
- [9] О. Б. Штейнберг. «Круговой сдвиг тела цикла — преобразование программ, способствующее распараллеливанию», *Вестн. ЮУрГУ. Сер. Матем. моделирование и программирование*, 10:3 (2017), с. 120–132 (на англ.). [↑]_{22, 23}

Поступила в редакцию 29.04.2020
 Переработана 02.07.2020
 Опубликована 27.07.2020

Рекомендовал к публикации

к.ф.-м.н. А. В. Климов

Пример ссылки на эту публикацию:

Б. Я. Штейнберг, О. Б. Штейнберг, А. А. Василенко. «Слияние циклов для локализации данных». *Программные системы: теория и приложения*, 2020, 11:3(46), с. 17–31. 10.25209/2079-3316-2020-11-3-17-31

http://psta.psiras.ru/read/psta2020_3_17-31.pdf

Об авторах:

Борис Яковлевич Штейнберг

д.т.н, зав. каф., с.н.с. Института математики, механики и компьютерных наук Южного федерального университета. Научные интересы: распараллеливание программ, дискретная оптимизация.

0000-0001-8146-0479

e-mail: borsteinb@mail.ru

Олег Борисович Штейнберг

к.ф.-м.н., доцент ИММиКН ЮФУ. Научные интересы: преобразования программ, алгоритмы на графах.

0000-0003-4152-0988

e-mail: olegsteinb@gmail.com

Александр Александрович Василенко

аспирант ИММиКН ЮФУ. Научные интересы: преобразования программ, распараллеливание программ.

0000-0002-3412-8809

e-mail: avas@sfnedu.ru

CSCSTI 50.07.05

UDC 519.681.3:004.415.3

Boris Ya. Steinberg, Oleg B. Steinberg, Alexandr A. Vasilenko. *The loop fusion for data localization.*

ABSTRACT. The loop merging can be used to improve data localization. Merging loops that share variables can speed up execution by reducing the number of cache misses. This transformation has been known for a long time, but compilers perform it only for the simplest cases.

Our improved algorithms use pre-transformations to correctly merge loops with different numbers of iterations and informational dependencies.

Key words and phrases: loop fusion, optimizing compiler, program transformations, data locality, memory access optimization.

2020 *Mathematics Subject Classification:* 97P30; 97P20, 97R40

References

- [1] S.L. Graham, M. Snir, C.A. Patterson. *Getting up to speed. The future of supercomputing*, Consensus Study Report, National Academies Press, 2005, 289 pp. [↑]₁₇
- [2] L. K. Eysymont, V. S. Gorbunov, G. S. Yelizarov. “Towards an exascale supercomputer: results, directions, trends”, MSKF 2012 (Moskva, 1 noyabrya 2012), pp. 5–7 (in Russian). [↑]₁₇
- [3] A. Mycroft. “Programming language design and analysis motivated by hardware evolution (invited presentation)”, *SAS 2007: Static Analysis*, International Static Analysis Symposium (22–24 August, 2007, Kongens Lyngby, Denmark), Springer, Berlin–Heidelberg, 2007, ISBN 978-3-540-74060-5, pp. 18–33. [↑]₁₇
- [4] A. I. Galushkin. “The development strategy of modern super-neurocomputers on the way to exaflop computing”, *Prilozheniye k zhurnalu «Informatsionnyye tekhnologii»*, 2012, no. 3 (in Russian). [↑]₁₇
- [5] O. B. Shteynberg. “Minimizing the number of temporary arrays in the problem of splitting loops”, *Izvestiya VUZov. Severo-Kavkazskiy region. Yestestvennyye nauki*, 2011, no. 5(165), pp. 31–35 (in Russian). [↑]_{18, 23}
- [6] J. Xue, Q. Huang, M. Guo. “Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences”, 2005 International Conference on Parallel Processing (ICPP’05) (14–17 June 2005, Oslo, Norway), 2005, pp. 107–115. [↑]_{19, 20}
- [7] K. Barton, J. Doerfert, H. Finkel, M. Kruse. “Loop fusion, loop distribution and their place in the loop optimization pipeline”, EuroLLVM 2019 Final (April 9, 2019). [↑]₁₉

- [8] E. Wang, Q. Zhang, B. Shen, X. Lu, Q. Wu, Y. Wang. *High-Performance Computing on the Intel® Xeon Phi™. How to Fully Exploit MIC Architectures*, Springer, 2014, ISBN 978-3-319-06485-7, xxiii+338 pp. [doi](#)[↑]₁₉
- [9] O. B. Steynberg. “Circular shift of loop body — programme transformation, promoting parallelism”, *Bulletin of the South Ural State University «Mathematical Modelling, Programming & Computer Software»*, **10:3** (2017), pp. 120–132. [doi](#)[↑]_{22,23}

Sample citation of this publication:

Boris Ya. Steinberg, Oleg B. Steinberg, Alexandr A. Vasilenko. “The loop fusion for data localization”. *Program Systems: Theory and Applications*, 2020, **11:3**(46), pp. 17–31. (*In Russian*). [doi](#) 10.25209/2079-3316-2020-11-3-17-31

http://psta.pspiras.ru/read/psta2020_3_17-31.pdf