ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ

РАСЧЕТНО-ГРАФИЧЕСКОЕ ЗАДАНИЕ

по дисциплине “Параллельное программирование”

на тему

Векторизация циклов средствами OpenMP

Выполнил студент Тимошкин Владислав Николаевич
Ф.И.О.

Группы МГ-173

Работу принял _____ доцент д.т.н. М.Г. Курносов
подпись

Защищена _____ Оценка _____

Новосибирск – 2018

Оглавление

ВВЕДЕНИЕ 3

Intel Advanced Vector Extensions 7

Инструменты векторизации 10

ПОСАТНОВКА ЗАДАЧИ 13

ДИРЕКТИВА OMP SIMD 14

КОНФИГУРАЦИЯ ТЕСТОВОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ 16

РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ 17

Перемножение матриц 17

Векторизация работы с массивом частиц 19

ЗАКЛЮЧЕНИЕ 20

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 21

ПРИЛОЖЕНИЕ 1 22

Умножение матриц (linear) 22

Умножение матриц (#pragma omp) 23

Умножение матриц (#pragma omp simd) 23

Работа с массивом частиц (#pragma omp simd) 24

ВВЕДЕНИЕ

Потребность в решении сложных прикладных задач с большим объемом вычислений и принципиальная ограниченность ресурсов и максимальной производительности «классических» – по схеме фон Неймана – привели к появлению многопроцессорных вычислительных систем (МВС). Широкое распространение параллельные вычисления приобрели с переходом компьютерной индустрии на массовый выпуск многоядерных процессоров с векторными расширениями. В настоящие время практически все устройства оснащены многоядерными процессорами, поддерживающими векторные расширения.

Векторизация — вид распараллеливания, при котором однопоточные приложения, выполняющие одну операцию в каждый момент времени, модифицируются для выполнения нескольких однотипных операций одновременно, векторных операций. Одна и та же операция применяется к целому набору данных, например, к массиву или его части.

По одной инструкции векторный процессор обрабатывает сразу некоторый массив(вектор) значений полностью, а не единственное значение, в отличие от скалярного процессора. Для наглядной демонстрации возьмем три массива Arr1, Arr2 и Res, имеющие одинаковую размерность и одинаковую длину, и имеется оператор Res = Arr1 * Arr2.

Векторный процессор за один цикл выполнения команды выполнит попарное умножение элементов массивов Arr1 и Arr2, и присвоит полученные значения соответствующим элементам массива Res. Каждый операнд при этом хранится в векторном регистре. Векторный процессор выполняет лишь одну команду, в отличие от скалярного процессора, которому пришлось бы циклично складывать элементы по очереди.

За счет векторизации возможно получить высокую производительность. Кроме того, векторным ЭВМ присущи и другие интересные особенности. Количество команд, необходимых для выполнения одной и той же программы, использующей векторные инструкции, намного

меньше в случае векторного процессора, чем обычного, скалярного. Другой момент заключается в том, что при соответствующей организации оперативной памяти данные в процессор будут передаваться на каждом такте, что дает значительный выигрыш в производительности компьютера. На рисунке 1 приведен пример перемножения двух векторов.

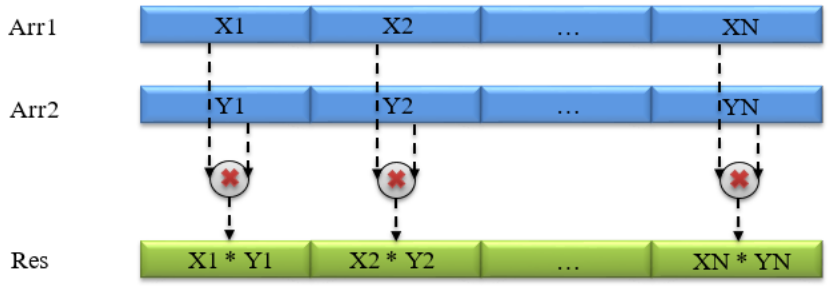

Рисунок 1 – Пример перемножения двух векторов

Максимальное ускорение (Speedup) линейно зависит от числа элементов находящихся в векторном регистре. Использование векторных инструкций может привести к сокращению количества команд в программе, а это может обеспечить более эффективное использование кеш-памяти.

Рассмотрим в листинге 3.1 фрагмент программы, который поэлементно перемножает два массива.

Листинг 1 – Псевдокод последовательного перемножения двух векторов

```
for (i = 0; i < 1024; i++)
    Res[i] = Arr1[i] * Arr2[i];
```

Данный цикл может быть векторизован, его SIMD версия представлена в листинге 2.

```
for (i = 0; i < 1024; i += 4)
    Res[i:i+3] = Arr1[i:i+3] * Arr2[i:i+3];
```

Листинг 2 – Псевдокод перемножения двух векторов с помощью векторных инструкций

Запись Res[i:i + 3] означает вектор из 4 элементов — от Res[i] до Res[i + 3] включительно, а под * понимается операция поэлементного умножения векторов. Векторный процессор в данном примере сможет выполнить 4 скалярные операции при помощи одной векторной инструкции за время, близкое к выполнению скалярной операции. Таким образом, векторных операций потребуется в 4 раза меньше, и программа исполнится быстрее.

Преимущество векторизации в том, что за время выполнения одной векторной инструкции операция проводится над массивом данных, но времени затрачивается столько же сколько и на одну скалярную инструкцию. Одна векторная команда распознаётся, декодируется и выполняется быстрее нескольких скалярных, выполняющих тот же набора инструкций. Векторизация позволяет не только добиться значительного ускорения выполнения кода, но также и уменьшить его объем.

Существует ряд вопросов, который возникает при использовании векторизации и требует решения:

1. Переносимость реализованного кода. Какие из существующих векторных расширений использовать и рассмотреть возможность создания кроссплатформенного кода.
2. Выравнивание доступа к данным, это означает, что адрес первого элемента должен нацело делится на заданное число, например, тогда загрузка данных из памяти будет происходить быстрее.
3. Доказательство независимости операций, обязательно необходимо чтобы операции могли выполняться параллельно.
4. Поиск однотипных операций в программе над различными данными или приведение к однотипным операциям.
5. Оценка затрат на сборку и разборку векторов. Рассмотрение зависимости выполнения векторной инструкции и затрат на сборку и разборку векторов.

Одной важной особенностью, которая в значительной мере может облегчить написание кода является возможность компиляторов автоматически векторизовывать последовательный код.

Почти все процессоры, на данный момент, позволяют выполнять код с использованием векторных инструкций.

Еще в начале 70х годов были применены решения, позволяющие архитектурно решить задачу ускорения вычислений с помощью векторной обработки данных. Первые векторные процессоры поддерживали набор инструкций, которые выполнялись над векторами данных и располагались прямо в памяти. Например, в процессор ЭВМ Cray-1 были добавлены векторные регистры, что сильно ускорило выполнение инструкций. Процессор 8086, выпускавшийся компанией Intel с 1976 года, работал вместе с математическим сопроцессором, задачей которого были вычисления для чисел с плавающей точкой. Этот сопроцессор работал с регистровым стеком и поддерживал набор команд x87. С появлением процессора Intel 486DX математический сопроцессор был интегрирован в процессор. Это позволило добавить восемь новых 80-битных регистров данных для хранения чисел с плавающей точкой, но они никак не использовались при целочисленных вычислениях. Тогда помощью данного модуля, для ускорения вычислений, стали обрабатывать целочисленные массивы данных, векторы. В процессоре Intel Pentium MMX впервые дебютировало векторное расширение под названием MMX. Одним из нововведений этого расширения стало добавление новых регистров (8 64-битных регистров MM0 и MM7), которые адресовались к регистрам устройства обработки чисел с плавающей точкой. Добавлен новый набор инструкций, работающих с этими регистрами и выполняющими над ними целочисленные операции. Эти инструкции еще называются SIMD-инструкциями (сокращение от Single Instruction, Multiple Data). Каждый из этих регистров мог хранить 2 32-битных целых числа, 4 16-битных или 8 8-битных чисел. SIMD-инструкции позволяли над этими наборами чисел проводить операции одновременно. Но работать с векторами и

вещественными числами одновременно было невозможно. Для этого приходилось переключать процессор в специальный режим, что занимало большое количество времени.

Расширение SSE было логическим продолжением MMX, оно впервые было реализовано в Pentium 3. Важной особенностью расширения стал увеличенный размер векторных регистров до 128 бит. Это позволило решить проблему одновременной работы с упакованными целыми и вещественными данными. Упакованные целые числа стали обрабатываться с помощью MMX инструкций, в то же время вещественные вычисления проводились с помощью SSE инструкций и векторных регистров. Помимо векторных регистров, SSE добавил в вычислительную систему 32-битный регистр флагов и операции с этим регистром, а также дополнительно расширил набор SIMD-операций над целыми. Добавлены инструкции явной предвыборки данных, контроля кэширования данных и контроля порядка операции сохранений. Расширение SSE2 стало обновлением SSE, которое добавило новые инструкции в SSE с целью полного вытеснения технологии MMX, и обработку упакованных данных с плавающей точкой с двойной точностью. Далее последовало усовершенствование SSE инструкций, SSE3, SSE4, ну и SSE4.1, где основными нововведениями стало добавление инструкции для взаимодействия с векторными регистрами.

Intel Advanced Vector Extensions

Новой ступенью векторных технологий стало расширение Intel Advanced Vector Extensions. Оно предоставляет возможности обработки большего количества данных за одну инструкцию. Эти инструкции расширяют возможности расширений MMX и SSE, добавляя следующие новые функции:

1. 128-битные SIMD-регистры были расширены до 256 бит. Intel AVX предназначен для расширения поддержки до 512 или 1024 бит в будущем.

2. Добавлена возможность использования трех операндов, если раньше приходилось использовать два операнда A = A + B, в результате чего перезаписывался исходный операнд; Теперь новые операнды могут выполнять такие операции, как A = B + C, оставляя исходные операнды исходного текста неизменными.
3. В нескольких инструкциях используются операнды с четырьмя регистрами, позволяющие выполнять меньший и быстрый код, удаляя ненужные инструкции
4. Требования к выравниванию памяти для операндов были ослаблены.
5. Новая схема кодирования расширения (VEX) была разработана для облегчения будущих дополнений, а также для того, чтобы кодирование инструкций было меньше и быстрее выполнялось.

Тесно связанные с этими достижениями новые инструкции Fused-Multiply-Add (VMA), которые позволяют выполнять более быстрые и более точные специализированные операции, такие как одна команда A = A * B + C. Команды FMA доступны во втором поколении Intel Core. Другие функции включают в себя новые инструкции для работы с шифрованием и расшифровкой Advanced Encryption Standard, операцию умножения на переносимую операцию переноса, полезную для определенных примитивов шифрования, и некоторые зарезервированные слоты для будущих инструкций, таких как генератор случайных чисел оборудования.

Новые инструкции кодируются с использованием того, что Intel называет префикс VEX, который представляет собой двух- или трехбайтовый префикс, предназначенный для устранения сложности текущей и будущей кодировки команд x86 / Intel 64. Два новых префикса VEX формируются из двух устаревших 32-битных инструкций – Load Pointer Using DS и Load Pointer с использованием ES, которые загружают регистры сегментов DS и ES в 32-битном режиме. В 64-битном режиме коды операций LDS и LES генерируют исключение недействительного кода операции, но в Intel AVX эти коды операций перераспределяются для кодирования новых инструктивных

префиксов. В результате инструкции VEX могут использоваться только при работе в 64-битном режиме. Префиксы позволяют кодировать больше регистров, чем предыдущие инструкции x86, и необходимы для доступа к новым 256-битным SIMD-регистрам или использования синтаксиса трех и четырех операндов.

Аппаратное обеспечение, поддерживающее Intel AVX (и FMA), состоит из 16 256-битных регистров YMM YMM0- YMM15 и 32-битного регистра управления / состояния, называемого MXCSR. Регистры YMM сглажены в более старых 128-битных регистрах XMM, используемых для Intel SSE, обрабатывая регистры XMM как нижнюю половину соответствующего регистра YMM, как показано на рисунке 2 и рисунке 3.

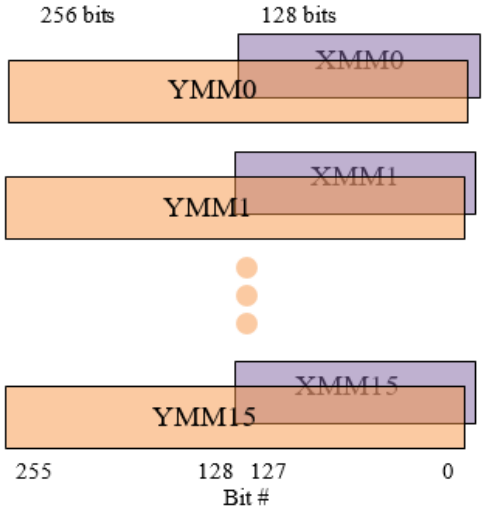

Рисунок 2 – Структура YMM и XMM регистров

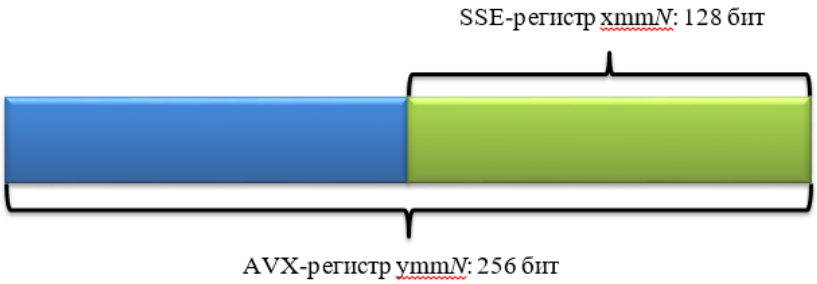

Рисунок 3 – Структура векторного AVX регистра

На рисунке 4 показаны типы данных, используемые в инструкциях Intel SSE и Intel AVX. Для Intel AVX допускается использование кратных 32-битных или 64-битных типов с плавающей запятой, добавляющих до 128 или

256 бит, а также кратные любому целочисленному типу, который добавляет 128 бит.

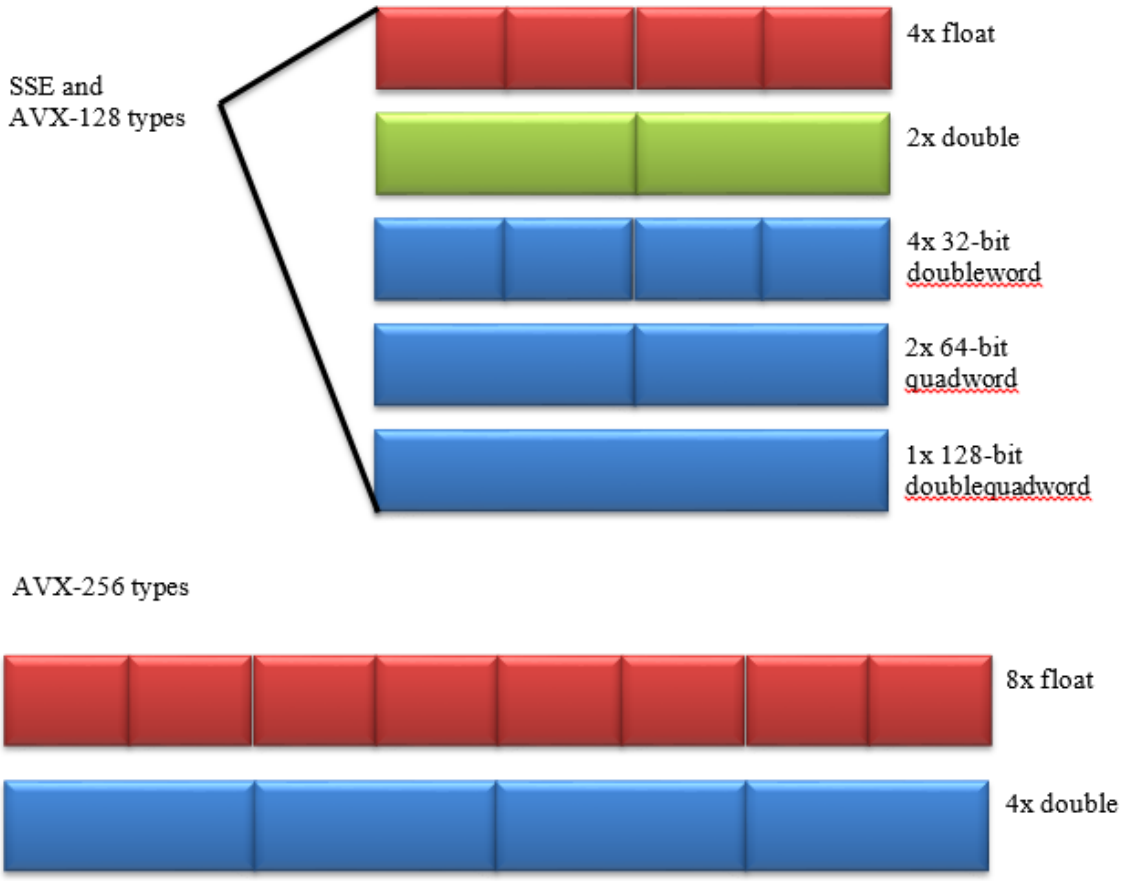

Рисунок 4 – Типы данных Intel AVX и Intel SSE

Инструменты векторизации

Для написания векторного кода существует множество способов, начиная от ассемблера и до автоматического векторизатора. На рисунке 5 показаны возможные варианты получения векторизованного кода.

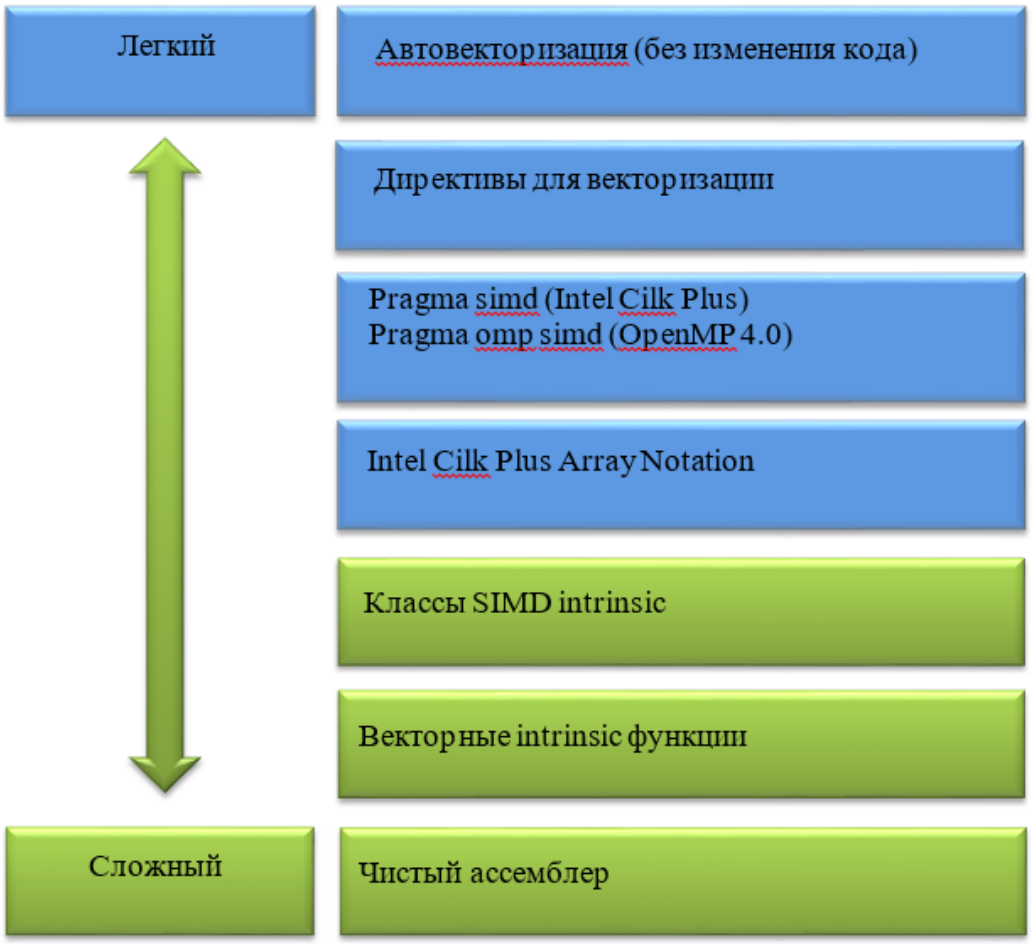

Рисунок 5 – Методы векторизации кода в порядке возрастания сложности

У ассемблера есть несколько преимуществ, код можно писать с использованием нужных инструкций и использовать все возможности процессора. Но данный метод содержит один серьёзный недостаток, это платформозависимость, и для каждой новой версии аппаратного обеспечения потребуется адаптация или, что еще хуже, полностью новая реализация.

Далее идут функции-интринсики, более удобный аналог ассемблера, но тоже не лишенный вышеупомянутого недостатка, времени для переписывания кода под новую платформу потребуется немало. В листинге 3 приведен пример реализации перемножения двух массивов.

Листинг 3 – Перемножение векторов с помощью функций-интринсиков на языке Си.

```
#include <immintrin.h>

double Arr1[200], Arr2[200], Res[200];
for (int j = 0; j < 200; j = j + 4) {
          __m256d m1 = _mm256_load_pd(&Arr1[j]);
          __m256d m2 = _mm256_load_pd(&Arr1[j]);
          __m256d r = _mm256_mul_pd(m1, m2);
          _mm256_store_pd(&Res[i], r);
}
```

Ограничение на зависимость кода от используемой платформы будет всегда, пока необходимые инструкции явно используются в коде, и не важно будет это ассемблер или функции-интринсики. Данная зависимость сохраняется даже при использовании SIMD intrinsic классов, которые являются следующим уровнем абстракции, хотя в этом случае разработчику не нужно знать какие функции использовать, а просто создать данные нужного класса.

Наиболее простым способом является возможность векторизации кода автоматически с помощью компилятора, т.е. компилятор должен уметь сам осуществлять преобразование скалярного кода в векторный. Автоматический векторизатор имеет также большую самостоятельную ценность, как важный компонент оптимизирующего компилятора, позволяющий сравнительно легко улучшать производительность вычислительных программ. Достоинство автоматического векторизатора в том, что достаточно добавить опции при компиляции, и компилятор сам выполнит векторизацию тех циклов, которые выгодно векторизовать.

ПОСАТНОВКА ЗАДАЧИ

В рамках расчетно-графического задания, в соответствии с вариантом 4, предполагается реализовать несколько примеров техники векторизации циклов средствами OpenMP, сравнить полученные результаты с последовательными версиями алгоритмов и оценить последовательное ускорение. Описать основные возможности данной техники оптимизации.

ДИРЕКТИВА OMP SIMD

Директива omp simd применяется к циклу, чтобы указать, что несколько итераций цикла могут выполняться одновременно с помощью инструкций SIMD.

Параметры директивы OMP SIMD:

aligned(list[:alignment])

Объявляет, что объект в списке выровнен в заданным количеством байт. Выравнивание должно быть постоянным положительным целочисленным выражением. Если выравнивание не указано, выравнивание определяется целевыми платформами.

collapse(n)

Указывает количество циклов, к которым применяется директива omp simd. Выражение, представленное n, должно быть константным положительным целочисленным выражением. Если параметр collapse не указан, то директива omp simd применяется только к одному следующему циклу.

linear(list[:linear-step])

Объявляет переменные в списке приватными для каждой SIMD-полосы и создает линейную связь с итерационным пространством цикла. Значение элемента списка на каждой итерации связанных циклов является результатом сложения следующих значений:

• Значение исходного элемента списка перед входом в конструкцию

• Произведение логического числа итерации и линейного шага

Конечное значение каждой переменной в списке имеет значение, присвоенное этой переменной в последовательной последней итерации.

reduction(reduction-identifier:list)

Выполняет операцию редукции по каждой переменной в списке. Создается приватная копия для переменных в списке для каждой SIMD-полосы, инициализирует частные копии значением инициализатора и

обновляет исходный элемент списка после конца региона со значениями личных копий.

safelen(length)

Указывает, что никакие две итерации, которые выполняются одновременно с использованием инструкций SIMD, не могут иметь расстояние в пространстве логических итерации 2, большее, чем значение, заданное длиной. Длина должна быть постоянным положительным целочисленным выражением.

simdlen(length)

Указывает предпочтительное количество итераций, которые должны выполняться одновременно. Каждая параллельная итерация выполняется с помощью другой SIMD-полосы. Длина должна быть постоянным положительным целочисленным выражением.

private, lastprivate, reduction, collapse - выполняют то же, что и в директиве omp for.

КОНФИГУРАЦИЯ ТЕСТОВОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

Тестирование разработанных программ проводилось с помощью системы Acer Aspire E5-575G-57X6, с использованием виртуальной машины VMware® Workstation 12 Pro версии 12.5.6 build-5528349, гостевой ОС выступала elementary OS 0.4.1 Loki. Виртуальной машине все 4 потока процессора Quad-Core Intel® Core™ i5-7200U CPU @ 2.50GHz и 8.7Gb оперативной памяти. Полные характеристики системы представлены в таблице 1.

Таблица 1 – Конфигурация тестового стенда.

Системная плата	Acer Aspire E5-575G
Процессор	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 МГц
Оперативная память	12Gb (8Gb + 4Gb DDR4-2133 (1066 МГц))
Жесткий диск	SSD KINGSTON RBUSNS8280S3128GH2 (119 ГБ)
Жесткий диск	HDD WDC WD5000LPCX-21VHAT0 (465 ГБ)

РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Перемножение матриц

В данной задаче исследовалось возможность получения ускорения при умножении квадратных матриц и их распараллеливании с помощью OpenMP и директивы OMP SIMD. При реализации матриц был выбран оптимальный вариант прохода матриц по столбцам, что позволило существенно уменьшить количество промахов по кэшу и увеличить быстродействие. Сравнивалось время выполнения трех алгоритмов: последовательного (листинг 4), распараллеленного с помощью директивы #pragma omp for (листинг 5) и распараллеленного алгоритма, векторизованного с помощью директивы #pragma omp for simd (листинг 6).

Листинг 4. Обход матрицы A по строкам, B по столбцам

```
for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++)
        for (int k = 0; k < size; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Листинг 5. Использование директивы #pragma omp for

```
#pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < size; i++)
            for (int k = 0; k < size; k++)
                for(int j = 0; j < size; j++ )
                C[i][j] += A[i][k] * B[k][j];
    }
```

Листинг 6. Использование директивы #pragma omp for simd

```
#pragma omp parallel
    {
        #pragma omp for simd
        for (int i = 0; i < size; i++)
            for (int k = 0; k < size; k++)
            {
                for(int j = 0; j < size; j++ )
                C[i][j] += A[i][k] * B[k][j];
            }
    }
```

Тестирование проводилось на псевдослучайно генерируемых матрицах размером от 256 * 256 элементов типа float, до размеров матриц 2048 * 2048 элементов, с шагом в 256 элементов. Результаты тестирования представлены в таблице 2.

Таблица 2 – Результаты экспериментов распараллеливания матриц.

Size\Time	**Linear**	**#pragma omp for**	**#pragma omp for simd**
256	0,099426	0,058261	0,061960
512	0,767040	0,434940	0,409296
768	2,353862	1,480747	1,349723
1024	5,611004	3,489120	3,174398
1280	10,760349	6,712340	6,196973
1536	18,397385	12,285569	10,681954
1792	30,190600	18,355736	16,961207
2048	44,967499	26,812834	24,063034

В среднем ускорение версии программы, векторизованной с помощью директивы omp simd, по отношению к скалярной версии составило 1.76 раза. Для версии программы с #pragma omp for ускорение составило 1.63.

Векторизация работы с массивом частиц

В данном задании было необходимо поработать с массивом частиц. Векторизовать функцию distance с помощью директивы omp simd, оценить полученное ускорение. Реализация итогового задания представлена в листинге 7.

Листинг 7. Использование директивы #pragma omp for simd

```
#pragma omp for simd
      for (int iter = 0; iter < 100; iter++) {
          distance(x, y, z, d, r);
      }
```

При векторизации данного алгоритма удалось достигнуть небольшого ускорения в 1.11. Результаты эксперимента представлены в таблице 3.

Таблица 3 – Результаты векторизации массива частиц.

	#particles	**#particles omp**
$\mathbf{10^2}$	0,000116	0,000097
$\mathbf{10^3}$	0,001140	0,000932
$\mathbf{10^4}$	0,010655	0,009622
$\mathbf{10^5}$	0,108697	0,100079
$\mathbf{10^6}$	1,097738	1,021450
$\mathbf{10^7}$	10,196754	10,061926

ЗАКЛЮЧЕНИЕ

В результате выполнения работы были исследованы возможности векторизации циклов с помощью OpenMP. На входных данных различных размеров и типов данных (double, float) автоматическая векторизация показала ускорение выполнения циклических операций, близкое к максимальному теоретическому ускорению.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. IBM Knowledge Center / pragma omp simd [Электронный ресурс] URL:https://www.ibm.com/support/knowledgecenter/en/SSXVZZ_13.1.6/com.ibm.xlcpp1316.lelinux.doc/compiler_ref/prag_omp_simd.html#prag_omp_simd
2. OpenMP Loop Collapse Directive [Электронный ресурс] URL: https://software.intel.com/en-us/articles/openmp-loop-collapse-directive
3. Explicit Vector Programming with OpenMP 4.0 SIMD Extensions. By Xinmin Tian and Bronis R. de Supinski, November 19, 2014 [Электронный ресурс] URL: http://www.hpctoday.com/hpc-labs/explicit-vector-programming-with-openmp-4-0-simd-extensions/

ПРИЛОЖЕНИЕ 1

Умножение матриц (linear)

gemm1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

double wtime() {
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

void work(int size, int flag) {
    float **A = (float**)malloc(size * sizeof(float*));
    float **B = (float**)malloc(size * sizeof(float*));
    float **C = (float**)malloc(size * sizeof(float*));
    for ( int i = 0; i < size; i++ ) {
        A[i] = (float*)malloc(size * sizeof(float));
        B[i] = (float*)malloc(size * sizeof(float));
        C[i] = (float*)malloc(size * sizeof(float));
    }

    for (int i = 0; i < size; i++)
      for(int j = 0; j < size; j++) {
        A[i][j] = rand() % 10;
        B[i][j] = rand() % 10;
        C[i][j] = 0;
      }
    double t = wtime();
    for (int k = 0; k < size; k++)
        for (int j = 0; j < size; j++){
            for(int i = 0; i < size; i++)
                C[i][j] += A[i][k] * B[k][j];
        }
    t = wtime() - t;
    printf("%lf\n", t);

    for ( int i = 0; i < size; i++ ) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A);
    free(B);
    free(C);
}

int main( int argc, char **argv ) {
    printf("|A(col) B(col)|\n");
    for (int i = 256; i <= 2048; i+=256) {
        work(i, 1);
    }
    return 0;
}
```

Умножение матриц (#pragma omp)

gemm2.c

```
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < size; i++)
        for (int k = 0; k < size; k++)
        {
            for(int j = 0; j < size; j++ )
            C[i][j] += A[i][k] * B[k][j];
        }
}
```

Умножение матриц (#pragma omp simd)

gemm3.c

```
#pragma omp parallel
{
    #pragma omp simd collapse(2)
    for (int i = 0; i < size; i++)
        for (int k = 0; k < size; k++)
        {
            for(int j = 0; j < size; j++ )
            C[i][j] += A[i][k] * B[k][j];
        }
}
```

Работа с массивом частиц (#pragma omp simd)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <sys/time.h>

#define EPS 1E-6

enum {
    n = 10000000
};

void init_particles(float *x, float *y, float *z, int n)
{
    for (int i = 0; i < n; i++) {
        x[i] = cos(i + 0.1);
        y[i] = cos(i + 0.2);
        z[i] = cos(i + 0.3);
    }
}

void distance(float *x, float *y, float *z, float *d, int n)
{
    #pragma omp for simd
    for (int i = 0; i < n; i++) {
        d[i] = sqrtf(x[i] * x[i] + y[i] * y[i] + z[i] * z[i]);
    }
}

void *xmalloc(size_t size)
{
    void *p = malloc(size);
    if (!p) {
        fprintf(stderr, "malloc failed\n");
        exit(EXIT_FAILURE);
    }
    return p;
}

double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

int main(int argc, char **argv)
{
    printf("Particles_OMP: n = %d)\n", n);

    float *d, *x, *y, *z;

    x = xmalloc(sizeof(*x) * n);
    y = xmalloc(sizeof(*y) * n);
    z = xmalloc(sizeof(*z) * n);
    d = xmalloc(sizeof(*d) * n);
```

```
    init_particles(x, y, z, n);

    double t = 0;
    for (int r = 1000; r < 10000000; r = r * 10) {
      t = wtime();
      for (int iter = 0; iter < 100; iter++) {
          distance(x, y, z, d, r);
      }
      t = wtime() - t;
      printf("%.6f sec.\n", t);
    }


    free(x);
    free(y);
    free(z);
    free(d);

    return 0;
}
```