

Зарегистрировано № _____
« ____ » _____ 2020 г.

подпись (расшифровка подписи)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»**
(НИУ «БелГУ»)

ИНСТИТУТ ИНЖЕНЕРНЫХ И ЦИФРОВЫХ ТЕХНОЛОГИЙ

**Кафедра математического и программного обеспечения
информационных систем**

**РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНОГО АЛГОРИТМА RSA
ШИФРОВАНИЯ**

Курсовая работа
по дисциплине «Методы распараллеливания программ»
студентки очной формы обучения
направления подготовки 02.03.02 «Фундаментальная информатика и
информационные технологии».
Профиль подготовки: Супервычисления
3 курса группы 12001801
Капустина Виктора Сергеевича

Допущена к защите
« ____ » _____ 2020 г.

Подпись (расшифровка подписи)

Научный руководитель:
Доц. Петров Д.В.

Оценка
« ____ » _____ 2020 г.

Подпись (расшифровка подписи)

БЕЛГОРОД 2020

ПЛАН КУРСОВОЙ РАБОТЫ

Введение.

1. Задача реализации параллельного алгоритма RSA шифрования
 - 1.1. Последовательный алгоритм RSA шифрования
 - 1.2. Разработка параллельной версии алгоритма для систем с общей памятью
 - 1.3. Разработка параллельной версии алгоритма для систем с распределенной памятью
 - 1.4. Разработка параллельной версии алгоритма для графических ускорителей
2. Программная реализация алгоритма RSA шифрования
 - 2.1. Выбор технологии и средств реализации
 - 2.2. Реализация последовательной версии алгоритма
 - 2.3. Реализация параллельной версий алгоритма
3. Тестирование программы
 - 3.1. Проверка адекватности работы последовательной и параллельной версии алгоритма
 - 3.2. Описание набора тестовых данных и методики их генерации

Заключение

Список использованных источников

Приложение

Исполнитель _____ студент Капустин В.С.

Руководитель _____ доцент Петров Д.В

Оглавление

| | |
|--|-----------|
| Введение | 3 |
| Глава 1. Задача реализации алгоритма RSA шифрования | 5 |
| 1.1 Последовательный алгоритм RSA шифрования | 6 |
| 1.2 Разработка параллельной версии алгоритма для систем с общей памятью | 8 |
| 1.3 Разработка параллельной версии алгоритма для систем с распределенной памятью | 11 |
| 1.4 Разработка параллельной версии алгоритма для графических ускорителей | 12 |
| Глава 2. Программная реализация алгоритма RSA шифрования | 14 |
| 2.1. Выбор технологии и средств реализации | 14 |
| 2.2. Реализация последовательной версии алгоритма | 15 |
| 2.3. Реализация параллельной версий алгоритма | 19 |
| Глава 3. Тестирование программы | 21 |
| 3.1. Проверка адекватности работы последовательной и параллельной версии алгоритма | 21 |
| 3.2 Описание набора тестовых данных и методики их генерации | 24 |
| Заключение | 25 |
| Список использованных источников | 26 |
| Приложение | 27 |

Введение

На сегодняшний день информационные технологии являются практически неотъемлемой частью нашей жизни. Уже нельзя представить себе, какой будет наша жизнь без компьютеров, процессоров и остальных технологий, упрощающих нашу жизнь посреди мира с огромным количеством информации. Еще одной проблемой является не защищенность передаваемой информации, которая волновала людей еще с самых давних пор. Однако из-за развития вычислительных систем и интернета, вопрос встал под другим углом. Если раньше можно было просто ограничить доступ к сообщению, то сейчас это делать все труднее, и при передаче оно может пройти через несколько десятков узлов прежде чем попадет к своему адресату. Задача еще больше усложняется одновременными требованиями надежности, секретности, а также легкостью самого процесса.

Для решения этих проблем и существовали криптографические системы. В основном они были с симметричным шифрованием, что могло вызывать целый ряд проблем. Однако в середине 70-ых годов появляются уже асимметричные криптосистемы, что приводит к огромному прорыву. Принято считать, что отправной точкой является работа 1976 года - «Новые направления в современной криптографии» под авторством Уитфилда Диффи и Мартина Хеллмана. В ней были сформулированы идеи пересылки зашифрованной информации уже без ключей, позволяющих данное дело расшифровать. Чуть позже открывается RSA, которая стала первой полноценной криптографической системой построенная на принципе асимметричного шифрования. На сегодняшний день этот алгоритм используется во многих криптографических приложениях, таких как: PGP, S/MIME, TLS/SSL, IPSEC/IKE и т.д.

Таким образом параллельный алгоритм RSA шифрования представляют собой систему, в которой автоматически рассчитывается открытый ключ, по которому и шифруется заданное сообщение. Расшифровка полученных данных производится за счет закрытого ключа, который также рассчитывается в самом начале. В данной работе будет рассмотрен полный цикл разработки данной системы, включая её тестирование и отладку. Целью работы является получение разработанной и работоспособной программы криптографической системы шифрования RSA.

Исходя из цели выделим следующие задачи:

1. Провести изучение предметной области.
2. Разработка программы, для реализации параллельного алгоритма RSA шифрования.
3. Подведение итогов на основе анализа выполненной работы.

Глава 1. Задача реализации алгоритма RSA шифрования

Главной целью криптографических систем была защита информации при ее хранении и передаче адресатам на протяжении всей истории. До сих пор эта проблема еще актуальна и на сегодняшний день, однако же развитие вычислительных технологий и систем придало ей новое качество: вопрос стал не просто о том, чтобы можно было послать письмо адресату, не опасаясь, что оно будет прочитано третьими лицами. Сетевые компьютерные системы могут включать большое количество пользователей пользователей. в данном случае традиционная симметричная схема окажется просто неэффективной. Потому новым требованием стало обеспечение аутентификации сообщений - доказательство, что адресат получил именно то сообщение, которое ему было отправлено, и то, что оно не было скажем подделано или заменено злоумышленником в процессе передачи этого сообщения. Более того, придется мириться с тем фактом, что к зашифрованной информации могут иметь доступ потенциально большое количество людей - Так, маршрут сообщения, проходящего в Интернете, невозможно предсказать, это значит , что оно может пройти через большое количество узлов пока не попадет к своему адресату. Поэтому задача все более усложняется противоречивыми требованиями одновременно надежности и секретности передачи.

Традиционная криптографическая схема выглядит следующим образом: Несколько людей знают некий открытый ключ, с помощью которого они обмениваются зашифрованными сообщениями так, чтобы любое третье лицо не смогло ничего в ней понять, даже если ему удастся перехватить шифровку. Эта схема называется симметричной, так как и адресат, и отправитель используют один ключ для шифровки и дешифровки. Однако у

такой ситуации есть свои слабые стороны. Так, представим, что есть установка - принимающая команды по сети. Не от всех, а только от тех, кто имеет нужные права доступа. Пусть наше устройство имеет определенный ключ, по которому должны быть зашифрованы все команды, на него подаваемые. Ключ этот, должен быть у всех пользователей нашей системы, т.к. она симметрична.

Теперь предположим, что другой пользователь перехватил это сообщение и решил получить доступ к сообщению, а зная ключ, с легкостью расшифровал его. Проблема может легко решиться введением разных паролей для разных пользователей. Ну а если доступ должен быть более или менее свободным? Более того это поставит новую проблему - как отличить одного пользователя от другого? Вариант решения всего один - перед началом запрашивать у пользователя его пароль и проводить сверку с хранящимся на сервере. Но тогда он с легкостью может быть перехвачен в том же самом канале связи. Данная проблема носит название проблемы распределения ключей и её существование делает симметричную схему весьма не приспособленной для обмена данными с огромным количеством пользователей, так как для шифровки и дешифровки используется один и тот же ключ.

1.1 Последовательный алгоритм RSA шифрования

Все эти проблемы была призвана решить схема асимметричного шифрования. Самой популярной из них на сегодняшний день считается RSA(аббревиатура от фамилий Rivest, Shamir и Adleman) шифрование. Она стала первой полноценной системой асимметричного шифрования и более того, используется в цифровой подписи. Основывается эта криптографическая схема на вычислительной сложности задачи факторизации больших целых чисел. Т.е. на различии в том, насколько легко

можно найти большие простые числа и насколько сложно будет их раскладывать на множители произведение двух больших простых чисел.

Перед шифрованием необходимо произвести дополнительные расчеты, такие как:

- Выбор двух простых чисел (p, q)
- Нахождение модуля их произведения (n)
- Вычисление функции Эйлера (φ)
- Выбор числа e , соответствующего трем критериям:
 - (I) оно должно быть простое
 - (II) оно должно быть меньше φ
 - (III) оно должно быть взаимно простое с φ .
- Вычисление числа d , обратное e по модулю φ

Теперь считается, что $\{e, n\}$ - открытый ключ, по которому шифруется сообщение. Замечательно, что по нему практически нереально расшифровать полученное сообщение. Для этого будет использоваться полученный закрытый ключ $\{d, n\}$, который рекомендуется никому не сообщать.

Заданное сообщение разбивается посимвольно и каждый из них кодируется по открытому ключу. Мы получаем зашифрованную последовательность, что и высылается адресату. Уже на приеме идет дешифровка по закрытому ключу каждого элемента сообщения, где мы и получаем наши символы обратно.

Также для дополнительной защиты шифрования, после кодирования символа в число, к его значению прибавляется дополнительная значимость равная $x + (14 * n)$ где n -номер итерации. Для первого символа n будет равно 0, т.е. его закодированному значению просто добавится значение x . Во время дешифровки наоборот - вначале будет отниматься $x + (14 * n)$ по такому же принципу и только потом расшифровываться по закрытому ключу.

В данной работе будет производиться генерация текста, шифрование этого текста, вывод получившихся данных, их расшифровывание и вывод расшифрованного сообщения.

1.2 Разработка параллельной версии алгоритма для систем с общей памятью

Криптографические схемы, а также в частности и алгоритм RSA работают с очень крупными числами, размеры которых могут быть сотни и тысячи бит, а то и больше. Поэтому операция модульного возведения в степень над такими числами связана с существенными машинными затратами, что может весьма ограничить работу алгоритма на аппаратном уровне, т.е. серьезно замедлить его работу. Более того, обычно на практике требуется, чтобы процесс шифрования происходил в режиме реального времени, а это уже может быть критично. Поэтому последовательная реализация алгоритма скорее всего окажется несостоятельной на огромных значениях. Поэтому становится очевидным необходимость распараллеливания алгоритма. Самым эффективным будет внедрять параллелизм в месте самого шифрования/дешифрования где действие над каждым элементом будет происходить одновременно по отдельности, а после весь результат будет передан обратно в главный поток.

Самым простым является разработка системы с общей памятью для чего подойдет использование OpenMP(Open Multi-Processing) - Открытого стандарта распараллеливания программ. В нем параллелизм реализуется путем многопоточности, где существует главный поток, который может создавать подчиненные ему потоки и распределять задачи между ними. В общем плане алгоритм остается таким же, но на месте функции шифрования/дешифрования как раз-таки и создаются подчиненные потоки, среди которых будут распределяться элементы передаваемого сообщения и на

каждом потоке он будет шифроваться/дешифроваться, а после записываться по очередным номерам элемента массива и передано в главный поток. Общий план распараллеливания представлен на Рисунке 1.

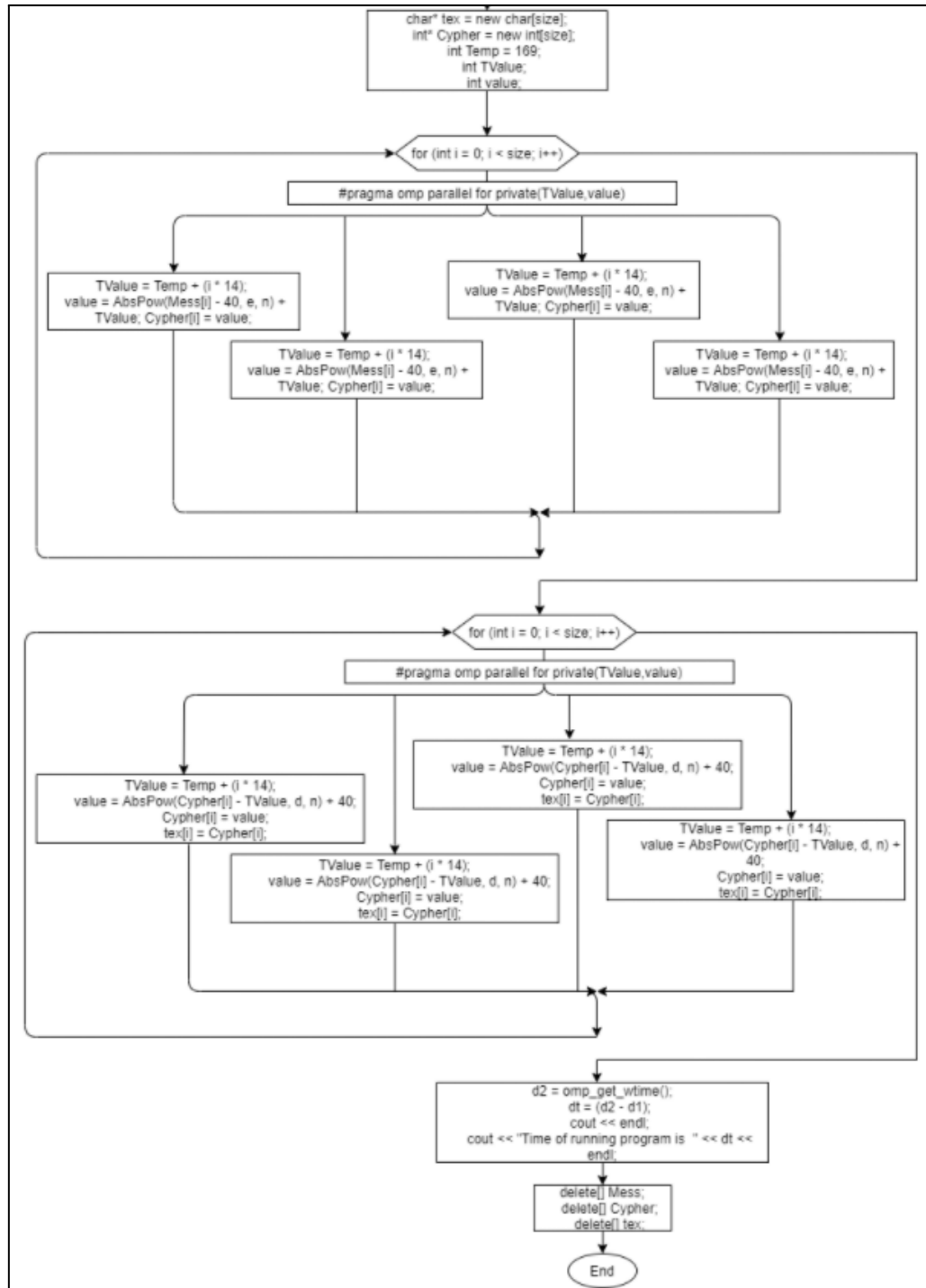


Рисунок 1. Блок-схема алгоритма для систем с общей памятью

1.3 Разработка параллельной версии алгоритма для систем с распределенной памятью

Более продвинутым уровнем распараллеливания будет являться реализация алгоритма для систем уже с распределенной памятью. Основным стандартом является - MPI(Message Passing Interface), программный интерфейс, позволяющий обмениваться данными между процессами, выполняющими одну задачу. Здесь уже данные отдаются не от главного потока подчиненным ему, а от одного процесса другим, где каждый из них может быть главным и обычно каждый из них однопоточный. Более того это может быть отличной связкой установить распараллеливание через MPI, где каждый процессор распределит данные между потоками через OpenMP. Общий план распараллеливания на системе с распределенной памятью представлен на Рисунке 2.

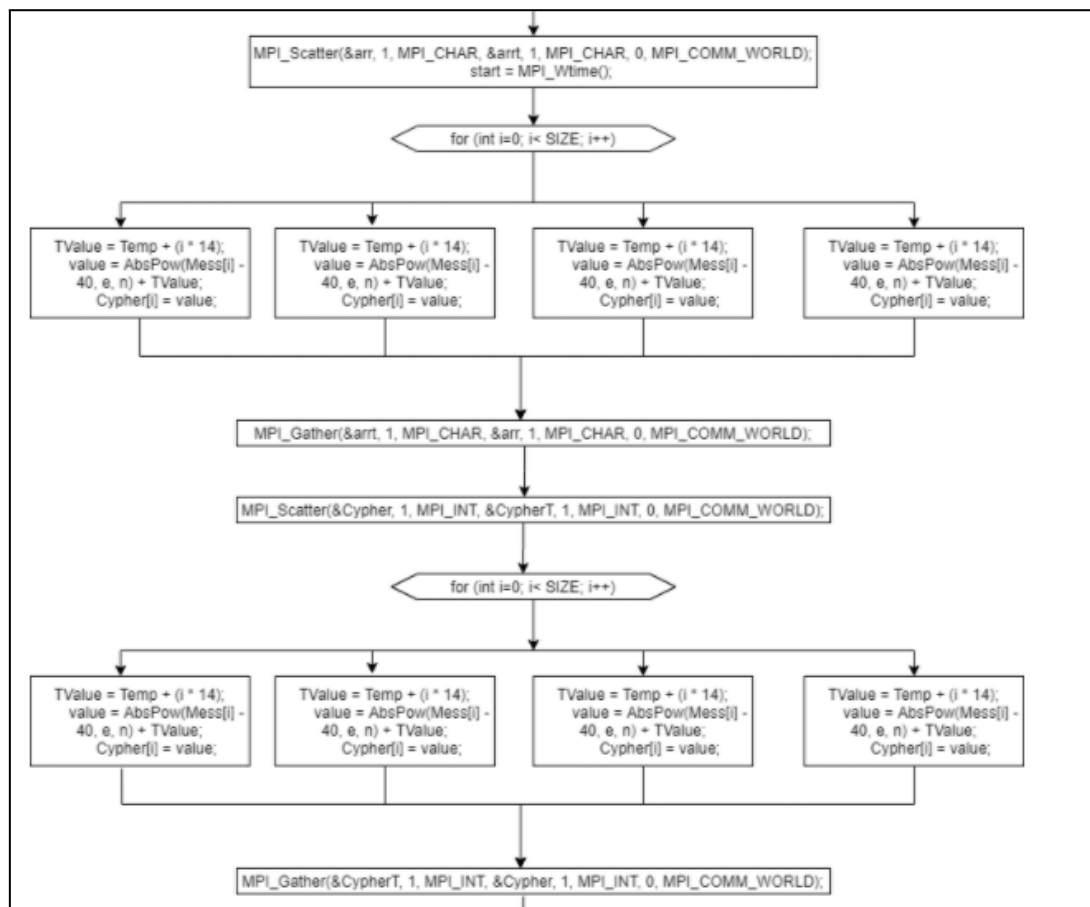


Рисунок 2. Блок-схема алгоритма для систем с распределенной памятью

В рамках данной работы использование алгоритма для систем с распределенной памятью очень схоже с использованием для систем с общей памятью. Распараллеливание происходит на этапе шифровки/дешифровки и распределяет все элементы сообщения между процессами где и происходит основные вычислительные действия. После этого с них собираются результаты на один процесс и выписываются по очередным номерам.

1.4 Разработка параллельной версии алгоритма для графических ускорителей

Но самым эффективным и востребованным распараллеливанием на сегодняшний день является использование графических процессоров и CUDA(Compute Unified Device Architecture). Эта программно-аппаратная архитектура для параллельных вычислений позволяет многократно увеличить производительность за счет передачи вычислительных задач с центрального процессора на графический, на котором благодаря кластерному моделированию потоков и SIMD-инструкциями достигается существенное увеличение вычислительных мощностей. Основным отличительным и преимущественным отличием в данном случае будет то, что GPU проектировалось для быстрого выполнения огромного количества параллельно выполняемых потоков заданных инструкций. Это помогает достичь высокой производительности, используя единственный поток команд, обрабатывающего и целые числа и числа с плавающей точкой. Правда при этом доступ к памяти оказывается случайным, но при определенных хитростях и это не будет проблемой. Но сложность заключается в трудоемкости данного программирования, и требует требует очень хорошего понимания организации памяти от программиста. Высокая производительность будет заметна в основном при большом количестве данных, иначе разбиение между процессорами будет более затратно, нежели

полученная от этого полезность. Общий план распараллеливания представлен на Рисунке 3.

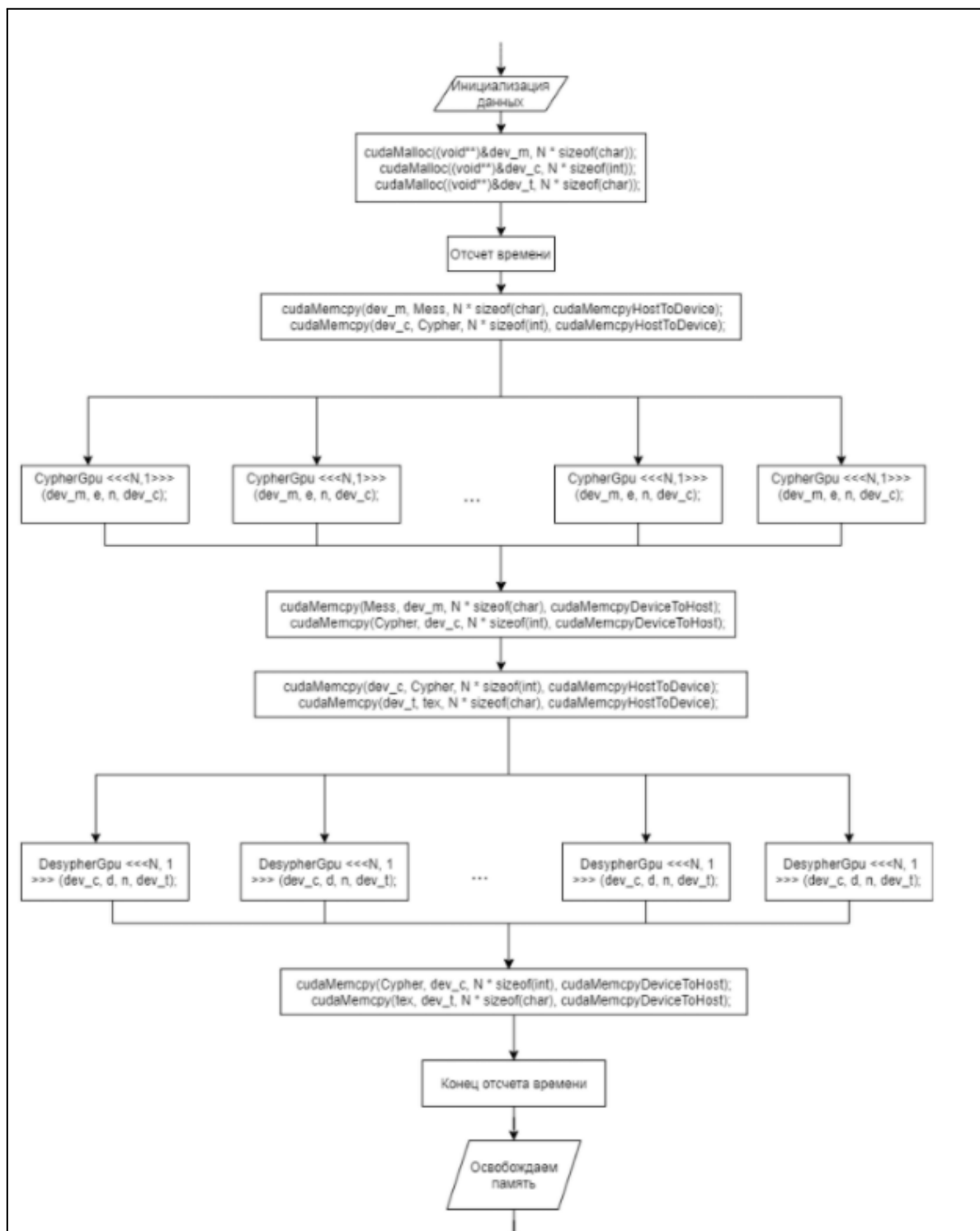


Рисунок 3. Блок-схема алгоритма для графических ускорителей

Глава 2. Программная реализация алгоритма RSA шифрования

2.1. Выбор технологии и средств реализации

Разработка будет происходить в среде программирования - Visual Studio 2019, полнофункциональной интегрированная среда разработки (IDE), поддерживающая многие популярные языки программирования: C#, C++, VB.NET, JavaScript, Python и т.д.

Функциональность Visual Studio охватывает абсолютно все этапы разработки программного обеспечения, умело предоставляя многие современные инструменты написания кода, сборки, отладки и тестирования приложений, а также проектирования графических интерфейсов. Также возможности данной среды вполне могут быть дополнены подключением необходимых расширений и библиотек.

Редактор кода Visual Studio очень дружелюбен к пользователю, путем поддержания подсветки синтаксиса, вставки целых фрагментов кода, отображение всей структуры и связанных с ней функций. Более того ускорить работу помогает технология IntelliSense - умного заполнения, тем самым автозаполнения кода по мере его ввода. А встроенный отладчик Visual Studio используется для поиска и исправления ошибок уже во время написания самого кода, в том числе на низком аппаратном уровне.

Для увеличения эффективности и производственных мощностей была выбрана версия алгоритма распараллеливания для графических ускорителей. Для этого будет использована программно-аппаратная архитектура параллельных вычислений - CUDA. Для данной работы лучше всего подойдут вычислительные мощности именно графического процессора, т.к.

они проектировались под данный типа работы, когда необходимо выполнение одного потока команд очень много раз. Кластерное моделирование потоков обеспечит наилучший результат при шифровании/дешифровании отдельных элементов сообщения. Таким образом, передаваемые данные разобьются по потоком и с помощью SIMD-инструкций запросто произведут большое количество вычислительных процессов.

2.2. Реализация последовательной версии алгоритма

Как говорилось ранее, прежде чем приступить к самому процессу шифровки/дешифровки. необходимо произвести дополнительные вычисления.

Необходимые расчеты:

Листинг 1. Нахождение P и Q

```
switch (rnd)
{
    case 0: p = 7; break;
    case 1: p = 13; break;
}

q = 17;
```

Конец листинга 1

-Нахождение модуля произведения P и Q (n): $n = |P * Q|$

Листинг 2. Нахождение n

```
n = abs(p * q);
```

Конец листинга 2

-Вычисление функции Эйлера (ϕ) : $\phi = (P-1)*(Q-1)$

Листинг 3. Вычисление ϕ

```
fi = (p - 1) * (q - 1);
```

Конец листинга 3

-Проверка взаимной простоты ϕ и e - Алгоритм Евклида:

1. Большее число делим на меньшее.
2. Если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла).
3. Если есть остаток, то большее число заменяем на остаток от деления.
4. Переходим к пункту 1.

Листинг 4. Проверка взаимной простоты

```
int i = 2;
while (e < 1) {
    if ((gcd(i, fi) == 1) && (i != p) && (i != q))
    {
        e = i;
    }
    i++;
}
```

Конец листинга 4

Листинг 4.1 Функция *gcd*

```
int gcd(int x, int y)
{
    return y ? gcd(y, x % y) : x;
}
```

Конец листинга 4.1

-Вычисление d : $(d \times e) \% \phi = 1$, где e и ϕ были получены в предыдущих шагах.

Листинг 5 Вычисление d

```
while (flag != true)
{
    if (((d * e) % fi) == 1)
    {
        if (ff == 1)
        {
            flag = true;
        }
        ff = 1;
    }
    else d++;
}
```

Конец листинга 5

Сам процесс шифрование происходит путем возведения каждого элемента в степень по модулю. Так, чтобы возвести $a^x \bmod p$ можно воспользоваться алгоритмом быстрого возведения в степень по модулю, где a соответствует элементу массива arr , x соответствует e , и p соответствует n :

1. Перевод степень числа x в двоичную систему (Doub)
2. Определить *quant*, которое равно количеству всех цифр степени в двоичной системе.
3. Составить массив данных, где первый элемент равен исходному числу a , а каждый последующий вычисляется так: Возводим в квадрат текущее значение и находим остаток от деления на p . По итогу получим массив значений arr .
4. Вычислить произведение элементов массива arr , которые возведены в соответствующие степени из двоичной записи числа на 1 шаге.
Требуется каждый элемент массива возвести в степень (либо в первую,

либо в нулевую), затем перемножить то, что получилось. *Степени соответствуют обратному порядку двоичной системы из 1 шага.*

5. Вычислить остаток от деления полученного произведения.

В данной работе алгоритм быстрого возведения в степень по модулю реализован по схеме «справа налево», основная идея, которой заключается в циклическом частичном умножении с уменьшением длины промежуточных результатов. На первом этапе показатель преобразуется в двоичную форму. После этого полученное значение побитово считывается справа налево, т.е. начиная с младших бит. Количество итераций равно количеству бит в двоичном представлении показателя, однако в результат включаются только те, в которых соответствующий бит равен 1.

Листинг 6. Алгоритм быстрого возведения в степень по модулю

```
int tid = 0;
while (tid < size)
{
    int TValue = 169 + (tid * 14);
    int g = Mess[tid] - 40;
    int result = 1;
    int exponent = e;
    while (exponent > 0)
    {
        if (exponent % 2 == 1)
        {
            result = (result * g) % n;
        }
        exponent = exponent >> 1;
        g = (g * g) % n;
    }
    Cypher[tid] = result + TValue;
```

```
tid += 1;  
}
```

Конец листинга 6

Для дополнительной защиты шифрования, после кодирования символа в число, к его значению прибавляется $x + (14 * n)$ где n -номер итерации. Для первого символа n будет равно 0, т.е. его закодированному значению добавится x . Во время дешифровки вначале отнимается $x + (14 * n)$ по такому же принципу и только потом декодируется в символ. В данной работе за x было выбрано значение 169.

2.3. Реализация параллельной версий алгоритма

Для реализации распараллеливания средствами CUDA используем следующий алгоритм:

1. Инициализация копий переменных хоста и устройства: `*dev_m`, `*dev_t` и `*dev_c`.

Листинг 7. Инициализация копий переменных

```
char *dev_m = 0;  
char *dev_t = 0;  
int *dev_c = 0;
```

Конец листинга 7

2. Выделение пространства на GPU для данных переменных

Листинг 8. Выделение пространства на GPU

```
cudaMalloc((void**)&dev_m, size * sizeof(char));  
cudaMalloc((void**)&dev_c, size * sizeof(int));  
cudaMalloc((void**)&dev_t, size * sizeof(char));
```

Конец листинга 8

3. Передача входных данных в GPU

Листинг 9. Передача входных данных

```
cudaMemcpy(dev_m, Mess, size * sizeof(char), cudaMemcpyHostToDevice);
```

```
cudaMemcpy(dev_c, Cypher, size * sizeof(int), cudaMemcpyHostToDevice);
```

Конец листинга 9

4. Вызов ядра CUDA на GPU

Листинг 10. Вызов ядра CUDA

```
CypherGpu <<<size,1>>> (dev_m, e, n, dev_c);
```

Конец листинга 10

5. Копирование результатов обратно на хост

Листинг 11. Копирование результатов

```
cudaMemcpy(Mess, dev_m, size * sizeof(char), cudaMemcpyDeviceToHost);
```

```
cudaMemcpy(Cypher, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Конец листинга 11

6. Очистка памяти.

Листинг 12. Очистка памяти.

```
cudaFree(dev_m);
```

```
    cudaFree(dev_t);
```

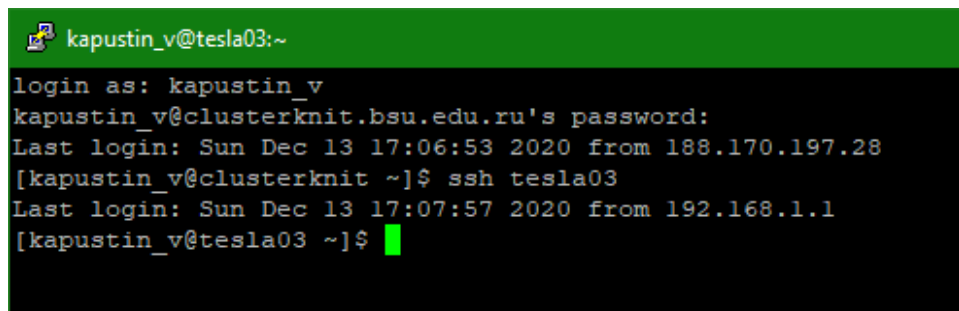
```
    cudaFree(dev_c);
```

Конец листинга 12

Глава 3. Тестирование программы

3.1. Проверка адекватности работы последовательной и параллельной версии алгоритма

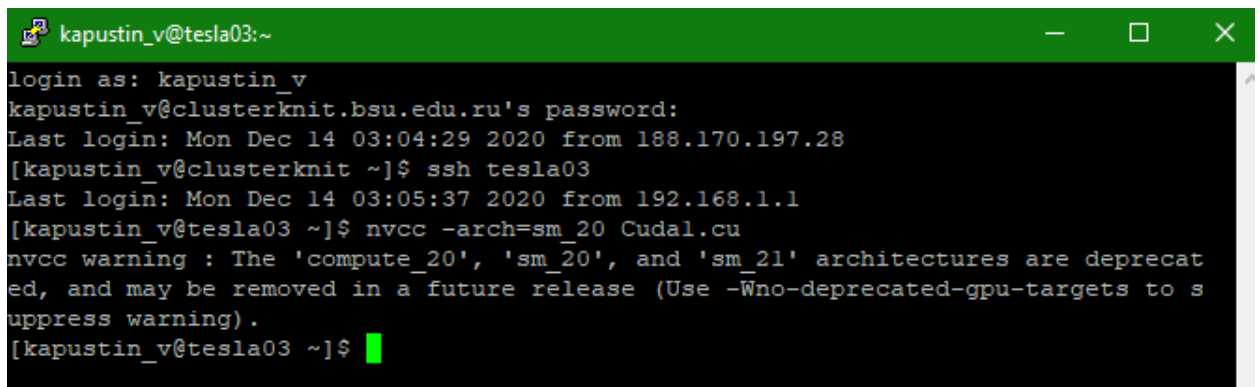
Для начала начнем с пробных запусков и просмотром работоспособности программы. Запуск CUDA производится через прокладывание туннеля на удаленном кластере(Рис. 5)



```
kapustin_v@tesla03:~  
login as: kapustin_v  
kapustin_v@clusterknit.bsu.edu.ru's password:  
Last login: Sun Dec 13 17:06:53 2020 from 188.170.197.28  
[kapustin_v@clusterknit ~]$ ssh tesla03  
Last login: Sun Dec 13 17:07:57 2020 from 192.168.1.1  
[kapustin_v@tesla03 ~]$
```

Рисунок 5. Прокладывание туннеля.

После этого мы компилируем нашу программу (Cuda1.cu) и видим, что все прошло успешно и без ошибок (Рис. 6)



```
kapustin_v@tesla03:~  
login as: kapustin_v  
kapustin_v@clusterknit.bsu.edu.ru's password:  
Last login: Mon Dec 14 03:04:29 2020 from 188.170.197.28  
[kapustin_v@clusterknit ~]$ ssh tesla03  
Last login: Mon Dec 14 03:05:37 2020 from 192.168.1.1  
[kapustin_v@tesla03 ~]$ nvcc -arch=sm_20 Cuda1.cu  
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecated, and may be removed in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).  
[kapustin_v@tesla03 ~]$
```

Рисунок 6. Успешная компиляция программы.

После успешной компиляции мы получили файл a.out, который и является нашей программой. Запустим его с выводом тестовых значений(Рис. 7). На нем мы наглядно видим, что программа все вывела исправно. Ошибок не выявлено. По полученным данным видно, что алгоритм работает. Нам выдается изначально заданное сообщение (Надпись - Original text), после чего

выводится зашифрованные данные (Надпись - Encrypted) и уже расшифрованное с этих данных сообщение (Надпись - Conclusion). На основе того, что изначальный текст и расшифрованный совпадают, очевидно можно сделать вывод, что программа работает корректно. Также еще одним критерием корректности программы является увеличивающиеся значения зашифрованных данных, что означает, что дополнительные меры по защите зашифрованного текста так же работают исправно.

```
[kapustin_v@tesla03 ~]$ ./a.out
Original text
T T U _ F R R H g J g U K S v q
Encrypted
226 240 394 402 391 248 262 269 301 329 329 520 457 478 456 478
Conclusion
T T U _ F R R H g J g U K S v q
0.233056 time
[kapustin_v@tesla03 ~]$
```

Рис. 7 Вывод программы с тестовыми значениями.

Теперь можно приступить к оценке адекватности получаемых данных от работы последовательной и параллельных версий алгоритма. Для этого измерим время работы на одних и тех же объемах задачи последовательный и параллельные версии. Все данные представлены в Таблице 1.

Таблица 1. Зависимость времени вычислений от объема исходных данных и количества задействованных узлов

| Объем задачи | Время расчета на CPU | Время расчета на GPU |
|--------------|----------------------|----------------------|
| 16 | 0.000001 | 0.00137760 |
| 1600 | 0.000001 | 0.00245952 |
| 16000 | 0.000001 | 0.01189152 |

| | | |
|-------------------|------------|-------------|
| 160000 | 0.000001 | 0.01883200 |
| 1600000 | 0.240000 | 0.20255936 |
| 16000000 | 2.200000 | 1.40146240 |
| 160000000 | 20.040000 | 13.65425842 |
| 1600000000 | 200.380000 | 23.27707642 |

На основе полученных данных, видно, что при сильно возрастающих объемах задачи, параллельная версия алгоритма показывает более эффективный результат нежели последовательная. Данный результат можно смело назвать адекватным и соответствующим заявленным ожиданиям.

3.2 Описание набора тестовых данных и методики их генерации

В качестве вводимых данных подразумевается ввод букв на латинице, а также некоторых близких по ASCII таблице символов. Генерирование сообщения происходит путем рандомизации каждого символа на интервале от 65 по 122 символ таблицы ASCII (От А до z) чем и заполняет массив изначального сообщения.

Листинг 13.Генерация тестовых данных.

```
char* Mess = new char[size];
for (int i = 0; i < size; i++)
{
    Mess[i] = 65 + rand() % 57;
    printf(" %c ", Mess[i]); }

```

Конец листинга 13

Заключение

В процессе выполнения курсовой работы был рассмотрен алгоритм шифрования RSA. Были описаны методы распараллеливания с общей, с распределенной памятью и версией алгоритма для графических ускорителей. Поставленная цель была выполнена.

Для ее достижения все следующие задачи были решены:

1. Изучена предметная область
2. Реализован алгоритм RSA шифрования, и разработана параллельная версия алгоритма для графических ускорителей
3. Проведены тесты и доказана адекватность расчетов

Таким образом был реализован алгоритм асимметричного шифрования. Была изучена тема и получен опыт на практике шифрования сообщения и криптографии. Программа была протестирована на работоспособность и проведены расчеты работы параллельного и процессорной работы.

Список использованных источников

1. <https://www.ixbt.com/video3/cuda-1.shtml>
2. <http://www.michurin.net/computer-science/rsa.html>
3. <https://www.e-nigma.ru/stat/rsa/>
4. <https://neerc.ifmo.ru/wiki/index.php?title=RSA>
5. https://www.nvidia.ru/content/EMEA/CUDA/lecture_documents/Lecture_3_3.pdf
6. <http://steps3d.narod.ru/tutorials/cuda-dynamic-parallelism.html>
7. https://habr.com/ru/company/epam_systems/blog/245503/
8. http://www.thg.ru/graphic/nvidia_cuda_test/index.html
9. <https://ru.bmstu.wiki/CUDA>
10. <https://www.overclockers.ua/software/nvidia-cuda/>

Приложение

Листинг программы

```
#include <iostream>
#include <cstdlib>
#include <cctype>
#include <string>
#include <bitset>
#include <cmath>
#include <time.h>
#include <stdio.h>
#include <ctime>
#include <iomanip>
#include <locale.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cuda.h>

#define size 16          //Размер сообщения

int gcd(int x, int y)
{
    return y ? gcd(y, x % y) : x;
}

/*
void CPUC(char *Mess, int e, int n, int *Cypher)
{
    int tid = 0;
    while (tid < size)
    {
        int TValue = 169 + (tid * 14);
```

```

int g = Mess[tid] - 40;
int result = 1;
int exponent = e;
while (exponent > 0)
{
    if (exponent % 2 == 1)
    {
        result = (result * g) % n;
    }
    exponent = exponent >> 1;
    g = (g * g) % n;
}

Cypher[tid] = result + TValue;
tid += 1;
}
}
void CPUD(int* Cypher, int d, int n, char* tex)
{
    int tid = 0;
    while (tid < size)
    {

        int TValue = 169 + (tid * 14);
        int g = Cypher[tid] - TValue;
        int result = 1;
        int exponent = d;
        while (exponent > 0)
        {
            if (exponent % 2 == 1)

```

```

    {
        result = (result * g) % n;
    }
    exponent = exponent >> 1;
    g = (g * g) % n;

}
Cypher[tid] = result + 40;
tex[tid] = Cypher[tid];
tid += 1;
}
}
*/

```

```

__global__ void DesypherGpu(int *Cypher, int d, int n, char *tex)
{
    int TValue = 169 + (blockIdx.x * 14);
    int g = Cypher[blockIdx.x] - TValue;
    int result = 1;
    int exponent = d;
    while (exponent > 0)
    {
        if (exponent % 2 == 1)
        {
            result = (result * g) % n;
        }
        exponent = exponent >> 1;
        g = (g * g) % n;
    }
}

```

```

    Cypher[blockIdx.x] = result + 40;
    tex[blockIdx.x] = Cypher[blockIdx.x];

}

__global__ void CypherGpu(char *Mess, int e, int n, int *Cypher)
{
    int TValue = 169 + (blockIdx.x * 14);
    int g = Mess[blockIdx.x] - 40;
    int result = 1;
    int exponent = e;
    while (exponent > 0)
    {
        if (exponent % 2 == 1)
        {
            result = (result * g) % n;
        }
        exponent = exponent >> 1;
        g = (g * g) % n;
    }

    Cypher[blockIdx.x] = result + TValue;

}

int main()
{
    srand(time(0));

```

```

int p;
int q;
int n;
int fi;
int e = 0;
int rnd = rand() % 2;

printf(" Original text \n ");
char* Mess = new char[size];
for (int i = 0; i < size; i++)
{
    Mess[i] = 65 + rand() % 57;
    printf(" %c ", Mess[i]);
}
printf("\n ");
switch (rnd)
{
case 0: p = 7; break;
case 1: p = 13; break;
}

q = 17;
n = abs(p * q);
fi = (p - 1) * (q - 1);

int i = 2;
while (e < 1) {

    if ((gcd(i, fi) == 1) && (i != p) && (i != q))
    {
        e = i;
    }
}

```

```

    }
    i++;
}
bool flag = false;
int ff = 0;
int d = 1;
while (flag != true)
{
    if (((d * e) % fi) == 1)
    {
        if (ff == 1)
        {
            flag = true;
        }
        ff = 1;
    }
    else d++;

}

char* tex = new char[size];
int* Cypher = new int[size];

/*
double startTime = clock();
srand(time(NULL));
CPUC(Mess, e, n, Cypher);    /
CPUD(Cypher, d, n, tex);    //
uble endTime = clock();
double cpuTime = (endTime - startTime) / CLOCKS_PER_SEC * 1000;

```

```

printf("\n %f time \n", cpuTime);
*/

char *dev_m = 0;
char *dev_t = 0;
int *dev_c = 0;

cudaEvent_t start, stop;
float gpuTime = 0.0f;

cudaMalloc((void**)&dev_m, size * sizeof(char));
cudaMalloc((void**)&dev_c, size * sizeof(int));
cudaMalloc((void**)&dev_t, size * sizeof(char));

cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

//Шифровка

cudaMemcpy(dev_m, Mess, size * sizeof(char), cudaMemcpyHostToDevice);
cudaMemcpy(dev_c, Cypher, size * sizeof(int), cudaMemcpyHostToDevice);

CypherGpu <<<size,1>>> (dev_m, e, n, dev_c);

cudaMemcpy(Mess, dev_m, size * sizeof(char), cudaMemcpyDeviceToHost);
cudaMemcpy(Cypher, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

```



```

printf(" Encrypted \n");
for (int i = 0; i < size; i++)
{
    printf(" %d ", Cypher[i]);
}
printf(" \n");

//Расшифровка
cudaMemcpy(dev_c, Cypher, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_t, tex, size * sizeof(char), cudaMemcpyHostToDevice);

DesypherGpu <<<size, 1 >>> (dev_c, d, n, dev_t);

cudaMemcpy(Cypher, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tex, dev_t, size * sizeof(char), cudaMemcpyDeviceToHost);

printf(" Conclusion \n ");
for (int i = 0; i < size; i++)
{
    printf(" %c ", tex[i]);
}
cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("\n %f time \n", gpuTime);

```

```
    cudaFree(dev_m);  
    cudaFree(dev_t);  
    cudaFree(dev_c);  
  
    return 1;  
  
}
```