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

КАФЕДРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

ЛАБОРАТОРНАЯ РАБОТА № 2

«ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НА АППАРАТУРНОМ УРОВНЕ»

Автор: С.Н. Мамойленко

Новосибирск - 2016

Оглавление

Цель работы 3
Теоретическое введение 3
1. Базовые сведения об архитектура электронно-вычислительных машин 3
2. Параллельное функционирование устройств ЭВМ 3
2.1 Обработка сигналов в операционных системах GNU\Linux. 4
2.2 Пример параллельной работы устройств ЭВМ. Работа с таймером 7
3. Организация оперативной памяти 9
3.1 Получение информации о процессоре и структуре кэша 9
3.2 Оптимизация доступа к оперативной памяти 10
4. Векторизация кода 11
4.1 Пример использования команд работы с векторами 11
4.2 Библиотеки функций, реализующих команды работы с векторами 12
4.3 Автоматическая оптимизация кода 13
Задание на лабораторную работу 13
Контрольные вопросы 14

Цель работы

Лабораторная работа направлена на развитие следующих общекультурных, общепрофессиональных и профессиональных компетенций:

- способностью к профессиональной эксплуатации современного оборудования и приборов (ОК-8);
- владением методами и средствами получения, хранения, переработки и трансляции информации посредством современных компьютерных технологий, в том числе в глобальных компьютерных сетях (ОПК-5);
- способностью проектировать системы с параллельной обработкой данных и высокопроизводительные системы, и их компоненты (ПК-9);
- способностью к программной реализации систем с параллельной обработкой данных и высокопроизводительных систем (ПК-14).

В результате изучения дисциплины должно быть сформировано понимание архитектуры современных электронно-вычислительных машин, асинхронного режима работы устройств, схемы работы подсистемы прерываний, системы команд процессоров семестра Intel, включая группу команд по векторизации, принципов функционирования многоуровневой памяти, а также получены базовые умения и навыки по проектированию и разработке параллельного высокопроизводительного программного обеспечения.

Теоретическое введение

1. Базовые сведения об архитектура электронно-вычислительных машин

Одним из основных исполнительных элементов распределённых вычислительных систем являются электронно-вычислительные машины.

Электронная вычислительная машина (ЭВМ) или компьютер (англ. computer – вычислитель) – это аппаратурно-программный комплекс, предназначенный для обработки информации. Под обработкой понимается: преобразование (т.е. выполнение некоторых вычислительных операций), а также ввод, вывод и хранение информации.

В архитектурном плане ЭВМ включает в себя исполнительное устройство (один или несколько центральных процессоров), устройства хранения информации (оперативная память, устройство долговременного хранения – жесткие диски, карты памяти и т.п.), внешние и дополнительные устройства. Очевидно, что все эти устройства выполняя свои функции работают параллельно, т.е. в то время как одно устройство выполняет свое действие, другие, в свою очередь, выполняют свои действия. При этом, все эти устройства имеют разную производительность и сильно влияют друг на друга, т.к. в процессе синхронной работы и взаимодействий друг с другом устройство с высокой производительностью вынуждено «подстраиваться» под устройства с низкой производительностью (чаще всего это реализуется простоем первого, пока второе выполнит необходимые от него действия). Пример архитектуры ЭВМ на базе набора микросхем Intel Z170 приведен на рисунке 1.

С точки зрения распределённой системы важным архитектурным свойством входящих в её состав ЭВМ является однородность, т.е. обеспечение возможности выполнения программного обеспечения на всех ЭВМ системы без его переделки. Такая однородность достигается путем реализации во всех ЭВМ системы команд исполнительных устройств на основе единых стандартов и использования единого интерфейса операционной системы.

2. Параллельное функционирование устройств ЭВМ

Основной проблемой, возникшей в системах, основанных на принципе общей шины, стало простаивание процессора при взаимодействии с медленно работающими устройствами. Чтобы повысить эффективность использования процессора реализовали механизм параллельного (асинхронного) функционирования устройств. В то время пока медленное устройство выполняет свои функции,

процессор переключается на выполнение другой программы, тем самым реализуя псевдопараллельный режим выполнения программ.

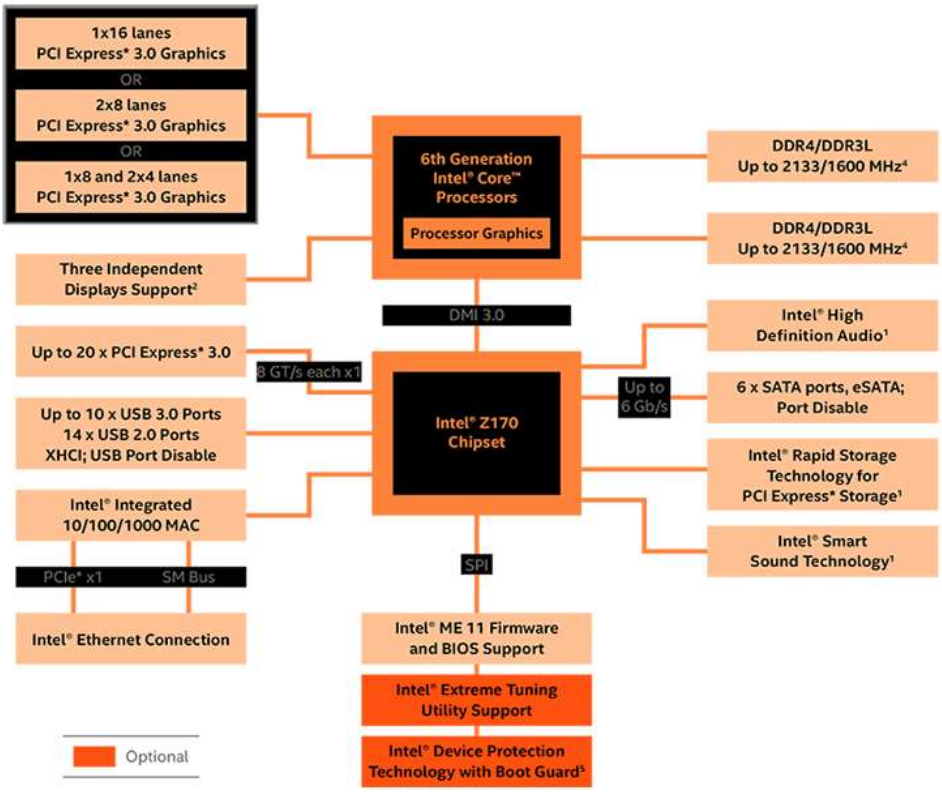

Рисунок 1 – Архитектура набора микросхем Intel Z170

Чтобы сообщить процессору, что медленное устройство снова готово взаимодействовать с ним, используется механизм прерываний. **Прерывание** – это событие, происходящие в ЭВМ, при котором процессор временно приостанавливает выполнение одной (текущей) программы и переключается на выполнение другой программы, необходимой для обработки этого события. После окончания выполнения обработчика события, вызвавшего прерывание, процессор возобновляет выполнение приостановленной программы. Такой подход позволяет устройствам, входящим в состав ЭВМ, функционировать независимо от процессора, и сообщать последнему о своей готовности взаимодействовать с ним. Кроме этого, использование прерываний позволяет реагировать на особые состояния, возникающие при работе самого процессора (т.е. при выполнении им программ).

Другим важным инструментом, позволяющим параллельное функционирование процессора и остальных устройств, является механизм прямого доступа к памяти. Благодаря этому медленное устройство, у которого процессор запросил какие-либо данные, помещается в оперативную память напрямую. Закончив передавать данные внешнее устройство генерирует прерывание и процессор уже взаимодействуя с более быстрым устройством – оперативной памятью обрабатывает необходимые данные. Подготовив данные в оперативной памяти, процессор сообщает устройству о готовности, переключается на другую программу, а устройство начинает взаимодействовать с оперативной памятью напрямую.

2.1 Обработка сигналов в операционных системах GNU\Linux.

По аналогии с аппаратурными прерываниями в каждой операционной системе предусматривается подсистема программных прерываний. В UNIX подобных операционных системах (например, GNU\Linux) таким аналогом служат сигналы. Сигнал – это способ взаимодействия программ, позволяющий сообщать о наступлении определённых событий, например, появление в очереди управляющих символов или возникновение ошибки во время работы программы (например, Segmentation Fault – выход за границы памяти).

Как и аппаратурное прерывание, сигналы описываются номерами, перечень которых содержится в заголовочном файле `signal.h`. Кроме цифрового кода, каждый сигнал имеет соответствующее символьное обозначение (макрос), например, `SIGINT`.

Большинство типов сигналов предназначены для использования ядром операционной системы, хотя есть несколько сигналов, которые посылаются от процесса к процессу. Полный список доступных сигналов приведён в электронном справочнике `man` (`man 7 signal`). Приведем некоторые из них:

- `SIGABRT` - сигнал прерывания процесса (process abort signal). Посылается процессу при вызове им функции `abort ()`. В результате сигнала `SIGABRT` произойдет аварийное завершение (abnormal termination) и запись образа памяти (core dump, иногда переводится как <дамп памяти>). Образ памяти процесса сохраняется в файле на диске для изучения с помощью отладчика;
- `SIGALRM` - сигнал таймера (alarm clock). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызовов `alarm()` или `setitimer()` (см. ниже);
- `SIGILL` - недопустимая команда процессора (illegal instruction). Посылается операционной системой, если процесс пытается выполнить недопустимую машинную команду. Иногда этот сигнал может возникнуть из-за того, что программа каким-либо образом повредила свой код. В результате сигнала SIGILL происходит аварийное завершение программы;
- `SIGINT` - сигнал прерывания программы (interrupt). Посылается ядром всем процессам, связанным с терминалом, когда пользователь нажимает клавишу прерывания (т.е., другими словами в потоке ввода появляется управляющий символ, соответствующий клавише прерывания). Примером клавиши прерывания может служить комбинация CTRL+C. Это также обычный способ остановки выполняющейся программы;
- `SIGKILL` - сигнал уничтожения процесса (kill). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса. Иногда он также посылается системой (например, при завершении работы системы). Сигнал SIGKILL - один из двух сигналов, которые не могут игнорироваться или перехватываться (то есть обрабатываться при помощи определенной пользователем процедуры);
- `SIGPROF` - сигнал профилирующего таймера (profiling time expired). Как было уже упомянуто для сигнала SIGALARM, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Сигнал SIGPROF генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством профилирования (планирования работы) программы;
- `SIGQUIT` - сигнал о выходе (quit). Очень похожий на сигнал SIGINT, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. Значение клавиши выхода по умолчанию соответствует символу ASCII F6 или Ctrl-Q. В отличие от SIGINT, этот сигнал приводит к аварийному завершению и сбросу образа памяти;
- `SIGSEGV` - обращение к некорректному адресу памяти (invalid memory reference). Сокращение `SEGV` в названии сигнала означает нарушение границ сегментов памяти (segmentation violation). Сигнал генерируется, если процесс пытается обратиться к неверному адресу памяти. Получение сигнала SIGSEGV приводит к аварийному завершению процесса;
- `SIGTERM` - программный сигнал завершения (software termination signal). Используется для завершения процесса. Программист может использовать этот сигнал для того,

чтобы дать процессу время для "наведения порядка", прежде чем посылать ему сигнал SIGKILL. Команда kill по умолчанию посылает именно этот сигнал;

- SIGWINCH – сигнал, генерируемый драйвером терминала при изменении размеров окна;
- SIGUSR1 и SIGUSR2 - пользовательские сигналы (user defined signals 1 and 2). Так же, как и сигнал SIGTERM, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя.

При получении сигнала процесс может выполнить одно из трех действий:

- выполнить действие по умолчанию. Обычно действие по умолчанию заключается в прекращении выполнения процесса. Для некоторых сигналов, например, для сигналов SIGUSR1 и SIGUSR2, действие по умолчанию заключается в игнорировании сигнала. Для других сигналов, например, для сигнала SIGSTOP, действие по умолчанию заключается в остановке процесса;
- игнорировать сигнал и продолжать выполнение. В больших программах неожиданно возникающие сигналы могут привести к проблемам. Например, нет смысла позволять программе останавливаться в результате случайного нажатия на клавишу, в то время как она производит обновление важной базы данных;
- выполнить определенное пользователем действие. Программист может задать собственный обработчик сигнала. Например, выполнить при выходе из программы операции по «наведению порядка" (такие как удаление рабочих файлов), что бы ни являлось причиной этого выхода.

Чтобы определить действие, которое необходимо выполнить при получении сигнала, используется системный вызов signal():

```
#include <signal.h>

typedef void (*sighandler_t) (int);
sighandler_t signal (int signum,  sighandler_t handler);
```

Рисунок 2 – Описание функции signal()

Вызов signal() определяет действие программы при поступлении сигнала с номером signum. Действие может быть задано как: адрес пользовательской функции (в таком случае в функцию в качестве параметра передается номер полученного сигнала) или как макросы SIG_IGN (для игнорирования сигнала) и SIG_DFL (для использования обработчика по умолчанию).

Если действие определено как пользовательская функция, то при поступлении сигнала программа будет прервана и процессор начнет выполнять указанную функцию. После её завершения выполнение программы, получившей сигнал, будет продолжено и обработчик сигнала будет установлен как SIG_DFL.

Чтобы вызвать сигнал, используется системный вызов raise():

```
#include <signal.h>
int raise (int signum);
```

Рисунок 3 – Описание функции rise()

Процессу посылается сигнал, определенный параметром signum, и в случае успеха функция raise возвращает нулевое значение.

Чтобы приостановить выполнение программы до тех пор, пока не придет хотя бы один сигнал, используется вызов pause:

```
#include <signal.h>
int pause (void);
```

Рисунок 4 – Описание функции pause()

Вызов pause приостанавливает выполнение вызывающего процесса до получения любого сигнала. Если сигнал вызывает нормальное завершение процесса или игнорируется процессом, то в результате вызова pause будет просто выполнено соответствующее действие (завершение работы или игнорирование сигнала). Если же сигнал перехватывается, то после завершения соответствующего обработчика прерывания вызов pause вернет значение -1 и поместит в переменную errno значение EINTR.

Пример программы, обрабатывающей прерывания приведён на рисунке 5.

```
1)  #include <stdio.h>
2)  #include <signal.h>
3)
4)  /* Функция-обработчик сигнала */
5)  void sighandler (int signo){
6)    printf ("Получен сигнал !!! Ура !!!\n");
7)  }
8)  /* Основная функция программы */
9)  int main (void){
10)   int x = 0;
11)
12)   /* Регистрируем обработчик сигнала */
13)   signal (SGUSR1, sighandler);
14)   do {
15)     printf ("Введите X = "); scanf ("%d", &x);
16)     if (x & 0x0A) raise (SIGUSR1);
17)   } while (x != 99);
18) }
```

Рисунок 5 – Пример программы, обрабатывающей сигналы

2.2 Пример параллельной работы устройств ЭВМ. Работа с таймером

В составе любой ЭВМ присутствует специальное устройство – таймер, отвечающий за отсчет времени. В GNU\Linux любое программное обеспечение можно использовать три таймера:

- ITIMER_REAL уменьшается постоянно и подает сигнал SIGALRM, когда значение таймера становится равным 0;
- ITIMER_VIRTUAL уменьшается только во время работы процесса и подает сигнал SIGVTALRM, когда значение таймера становится равным 0.
- ITIMER_PROF уменьшается во время работы процесса и когда система выполняет что-либо по заданию процесса. Совместно с ITIMER_VIRTUAL этот таймер обычно используется для профилирования времени работы приложения в пользовательской области и в области ядра. Когда значение таймера становится равным 0, подается сигнал SIGPROF.

Когда на одном из таймеров заканчивается время, процессу посылается сигнал и таймер обычно перезапускается.

Для установки таймеров используется функция setitimer. Величина, на которую устанавливается таймер, определяется следующими структурами: itemerval и timeval (см. рисунок 6).

Значение таймера уменьшается от величины it_value до нуля, после чего генерируется соответствующий сигнал, и значение таймера вновь устанавливается равным it_interval. Таймер, установленный на ноль (его величина it_value равна нулю или время вышло, и величина it_interval равна нулю), останавливается. Величины tv_sec и tv_usec являются основными при установке таймера.

Если устанавливаемое время срабатывания таймера измеряется в полных секундах, то для установки таймера может использоваться системный вызов alarm (см. рисунок 7).

```
struct itimerval {
   struct timeval it_interval; /* следующее значение */
```

```
   struct timeval it_value; /* текущее значение */
};

struct timeval {
   long tv_sec;  /* секунды */
   long tv_usec; /* микросекунды */
};
```

Рисунок 6 – Описание структур `itimerval` и `timeval`

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);

#include <sys/time.h>

int setitimer(int which, struct itimerval *value, struct itimerval
*ovalue);
```

Рисунок 7 – Описание функций `alarm` и `setitimer`

Функция `alarm` запускает таймер, который через `secs` секунд сгенерирует сигнал `SIGALRM`. Поэтому вызов `alarm(60)` приводит к отправке сигнала `SIGALRM` через 60 секунд. Обратите внимание, что вызов `alarm` не приостанавливает выполнение процесса, как вызов `sleep`, вместо этого сразу же происходит возврат из вызова `alarm`, и продолжается нормальное выполнение программы, по крайней мере, до тех пор, пока не будет получен сигнал `SIGALRM`. «Выключить» таймер можно при помощи вызова `alarm` с нулевым параметром: `alarm(0)`.

Вызовы `alarm` не накапливаются. Другими словами, если вызвать `alarm` дважды, то второй вызов отменит предыдущий. Но при этом возвращаемое вызовом `alarm` значение будет равно времени, оставшемуся до срабатывания предыдущего таймера.

Пример программы, настраивающей терминал и обрабатывающей сигнал от него приведен на рисунке 8.

```
1)  #include <stdio.h>
2)  #include <signal.h>
3)  #include <sys/time.h>
4)
5)  void signalhandler (int signo){ printf ("Сработал таймер\n"); }
6)
7)  int main (void) {
8)    struct itimerval nval, oval;
9)
10)   signal (SIGALRM,  signalhandler);
11)
12)   nval.it_interval.tv_sec = 3; nval.it_interval.tv_usec = 500;
13)   nval.it_value.tv_sec = 1;    nval.it_value.tv_usec = 0;
14)
15)   setitimer (ITIMER_REAL, &nval, &oval);
16)
17)   while (1){ pause(); }
18)
19)   return (0);
20) }
```

Рисунок 8 – Пример программы, использующей таймер

3. Организация оперативной памяти

Из общего описания архитектуры ЭВМ видно, что чаще всего процессор взаимодействует с одним внешним по отношению к нему устройством - оперативной памятью. Порядок действий центрального процессора электронно-вычислительной машины, в общем очень упрощённом случае, состоит из следующих этапов: выборка данных из ячейки оперативной памяти, декодирование команды, выполнение команды, запись результата в оперативную память, переход к следующей ячейке памяти. С увеличением производительности процессора возникает проблема его длительного простоя в процессе обмена данными с оперативной памятью.

Для решения этой проблемы работа процессора и оперативной памяти реализована в параллельном (асинхронном) режиме, а для ускорения доступа используется многоуровневая архитектура памяти (см. рисунок 9), в которой часть памяти, называемой кэшем, располагается между исполнительными элементами процессора и оперативной памятью и имеет более высокое быстродействие.

Рисунок 9 - Структура многоуровневой памяти

Суть работы многоуровневого доступа к памяти – ускорение получения информации за счет предварительного её чтения (блоками определённой длинны) из более медленного уровня памяти. Когда производится доступ к какой-либо ячейке оперативной памяти, то из неё считывается группа рядом расположенных ячеек (так называемая линия кэша). В результате, если следующее обращении производится к ячейке, которая попадает в область рядом стоящих ячеек, которые уже попали в более производительный уровень памяти, то доступ будет осуществлен быстрее. Кроме того, пока обрабатывается уже считанная область оперативной памяти параллельно может считываться следующая область оперативной памяти.

Формирование нескольких уровней кэша связано с параллельным функционированием устройств процессора, которые декодируют полученные из оперативной памяти данные формируя поток команд и данных (содержание кэша первого уровня) и устройств, которые непосредственно выполняют эти команды.

От размера кэша и принципов его организации и от того, каким образом организовано выполнение команд процессором зависит время простаивания процессора из-за доступа к оперативной памяти.

3.1 Получение информации о процессоре и структуре кэша

С определённого момента развития процессоров в многие системы команд стали включать специальные функции, позволяющие узнать особенности конкретного процессора. Учитывая программную совместимость ЭВМ такие функции позволяют приложениям настроить свою работу так, чтобы эффективно использовать архитектурные особенности процессоров.

В процессорах семейства Intel получить информацию о поддерживаемых функциях процессора, структуре кэша и т.п. возможно с использованием команды `CPUID`. Пример программы, использующей эту функцию, представлен на рисунке 10.

Для пользователей системы GNU\Linux разработаны различные утилиты, выводящие информацию, получаемую от команды `CPUIFO`. Например, утилита `lscpu`. Также информацию о процессоре можно получить с использованием виртуальной файловой системы `proc` (`/proc/cpuinfo`).

Для пользователей Windows также разработаны приложения, информирующие о особенностях процессора. Например, популярная утилита CPU-Z (см. рисунок 11).

```
1)  #include <stdio.h>
2)
3)  int main (void){
4)    unsigned int e[5] = {0};
5)
6)    __asm__ __volatile__ (
7)      "movl $0x02, %%eax \n\t"  "xorl %%ebx, %%ebx \n\t".
8)       "xorl %%ecx, %%ecx \n\t" "xorl %%edx, %%edx \n\t"
9)      "cpuid "
10)
11)     : "=a"(e[0]), "=b"(e[1]), "=c"(e[2]), "=d"(e[3]) );
12)
13)     if ((e[3] && 0xFF) == 0x4E) printf
14)     ("L2: 6MByte, 24-way set associative, 64 byte line size");
15)
16)   return (0);
17) }
```

Рисунок 10 – Пример программы, использующей команду CPUID

Рисунок 11 – Пример информации, выводимой приложением CPU-Z

Подробную информацию о работе с командой CPUID следует искать в техническом руководстве разработка приложений для процессоров Intel [].

3.2 Оптимизация доступа к оперативной памяти

Суть работы кэша заключается во вре́менном хранении блоков данных определённой длинны. Для организации работы кэша память условно разделяется на группы блоков соответствующей длинны (так называемых линий кэша). Запрашивая ячейку памяти, данные которой не были обнаружены ни в одном из уровней кэша, контроллер памяти считывает соответствующую ей область оперативной памяти в кэш. Учитывая, что линии кэша всегда начинаются с определённых адресов оперативной памяти (эти адреса легко вычислить, зная длину линии кэша и располагая эти линии последовательно друг за другом, начиная с самого младшего адреса памяти) информацию в оперативной памяти следует располагать так, чтобы блоки данных максимальным образом помещались в линию кэша. Кроме того, доступ к оперативной памяти также должен планироваться так, чтобы максимально локализовать работу с данными в одной линии кэша (а не «прыгать» между линиями при каждом доступе к памяти).

Пример оптимизации программного обеспечения в части повышения эффективности использования кэша приведен на рисунке 12. Суть оптимизации – изменение порядка просмотра матриц так, чтобы доступ был локализован по линиям кэша.

```
1) for (i = 0; i < ROWSA; i++) {
2)  for (j = 0; j < COLSB; j++) {
3)   for (k = 0; k < ROWSA; k ++) {
4)    C[i][j] += A[i][k] * B[k][j];
5)   }
6)  }
7) }
```

```
1) for (i = 0; i < ROWSA; i++) {
2)  for (k = 0; k < ROWSA; k ++) {
3)   for (j = 0; j < COLSB; j++) {
4)    C[i][j] += A[i][k] * B[k][j];
5)     }
6)   }
7) }
```

Рисунок 12 – Пример оптимизации работы с памятью

Эффект от оптимизации кода, представленной на рисунке 12, будет достигнут для матриц, в которых количество строк в таблице A больше, чем длинна линии кэша. Очевидно, что степень проявления этого эффекта зависит от размера матрицы (чем больше размер, тем заметнее будет эффект).

4. Векторизация кода

Наверное, самым ярким проявлением параллельности на уровне аппаратурного обеспечения является одновременное выполнение процессором одного действия над несколькими данными. Такой подход называется векторизацией кода, т.к. основная идея выполнения таких операций – работа с векторами данных.

Исторически процессоры оснащались различными группами команд, предназначенных для выполнение определённого круга операций. К таким группам относят: MMX, 3DNow, SSE, SSE2, SSE3, SSE4, SSE4.1, SSE4.2, SSE4.3 и т.п. Для выполнения этих операций процессор оснащается дополнительными регистрами, имеющими значительно большую разрядность, чем регистры общего назначения. В эти регистры загружаются группы данных (векторы) и над ними выполняется одновременно одна и та же команда.

В зависимости от операции, которую надо выполнить на данными, для их размещения в специальных регистрах используют различные форматы: упакованные, двоично-десятичные и т.п.,

В качестве примера назовем следующие группы команд:

- MMX (англ. Multimedia Extension), используется восемь 64-разрядных регистров. Команды этой группы работают с целочисленными данными (упакованными и неупакованными);
- SSE (англ. Streaming SIMD Extensions) использует шесть (в 32-х разрядном режиме) или шестнадцать (в 64-х разрядном режиме) 128-разрядных регистров. Команды этой группы работают с числами с плавающей запятой (float).

4.1 Пример использования команд работы с векторами

Для того, чтобы выполнить команду обработки вектора данных необходимо подготовить соответствующим образом один или несколько специализированных регистра, выполнить команду и записать полученные результат из специализированного регистра в оперативную память. Трудоемкость работы с дополнительными регистрами немного выше, чем трудоемкость работы с регистрами общего назначения (т.к. их разрядность выше). Однако, за счет того, что операция за один раз обрабатывает несколько данных, получается ускорение выполнения всего кода. Другими словами, использование команд по работе с векторами, приводит к сокращению общего количества выполняемых программой операций.

Пример использования команд по работе с векторами представлен на рисунке 13. В этой программе представлен цикл, занимающийся выполнением операции сложения элементов двух массивов целых чисел (массивы A и B, результат записывается в массив C, массивы содержат 1024 элементов). В программе, которая не использует команды работы с векторами происходит поочередное

сложение элементов массива следующим образом: очередной элемент массива A загружается в регистр EDX (строка 2), происходил сложение очередного элемента массива B и значения в регистре EDX, результат сложения записывается в регистр EDX (строка 3), результат сложения из регистра EDX записывается в очередной элемент массива C (строка 4). Значение счетчика (номер ячейки памяти) увеличивается на 4 (строка 5, один элемент массива, содержащий целое число, использует для своего хранения 4 байта оперативной памяти), далее происходит сравнение значение счетчика с допустимым (срока 6, 4096 = 1024 x 4) и в случае, если значение счетчика оказывается меньше, чем заданная граница, то происходит переход на начало цикла (строка 7). В программе, использующей векторные команды, производятся аналогичные действия, только за один раз обрабатывается по 4 элемента массивов A и B.

```
1)  .L3:
2)    movl  A(%rax), %edx
3)    addl  B(%rax), %edx
4)    movl  %edx, C(%rax)
5)    addq  $4, %rax
6)    cmpq  $4096, %rax
7)    jne   .L3
```

```
1)  .L3:
2)    movdqa  A(%rax), %xmm0
3)    paddd   B(%rax), %xmm0
4)    movdqa  %xmm0, C(%rax)
5)    addq    $16, %rax
6)    cmpq    $4096, %rax
7)    jne     .L3
```

Рисунок 13 – Примеры реализации программа суммирования элементов двух массивов (B и B) на ассемблере без использования потоковых команд (а) и с использованием SSE (б)

Следует отметить, что для работы с векторными функциями особую важность приобретает правильность расположения данных в памяти. Чтобы обеспечить эффективность загрузки данных из оперативной памяти в специальные регистры, данные в памяти должны быть выравнены по границе линии кэша.

Перечень векторных функций можно найти в документации по процессорам Intel (главы 9 – 12 руководства разработчиков).

4.2 Библиотеки функций, реализующих команды работы с векторами

Разработка программного обеспечения с использованием ассемблерных вставок является довольно сложным процессом. Для упрощения этого процесса для программистов, использующих языки высокого уровня разрабатываются специальные библиотеки функций (, в которых описываются новые типы данных и необходимые функции. Пример программы, написанной на языке Си и использующей функции по работе с векторами данных, представлен на рисунке 14.

```
1)  #include <xmmintrin.h>
2)  #define N = 1024
3)  void fun_sse(float *a, float *b, float *c, int n)
4)  {
5)      int i;
6)      __m128 *aa = (__m128 *)a, *bb = (__m128 *)b,
7)             *cc = (__m128 *)c;
8)      for (i = 0; i < (n / 4); i++) {
9)          *cc = _mm_add_ps(*aa, *bb); aa++; bb++; cc++;
10)     }
11) }
12)
13) int main(int argc, char **argv)
14) {
15)     int i; float *a, *b, *c;
16)     a = (float *)_mm_malloc(sizeof(float) * N, 16);
17)     b = (float *)_mm_malloc(sizeof(float) * N, 16);
18)     c = (float *)_mm_malloc(sizeof(float) * N, 16);
```

```
19)      fun_sse(a, b, c, N);
20)     _mm_free(a); _mm_free(b); _mm_free(c);
21)     return 0;
22) }
```

Рисунок 14 – Пример программы на языке Си, использующей команды обработки вектора данных

4.3 Автоматическая оптимизация кода

Еще больше упростить процесс разработки программного обеспечения призваны компиляторы или интерпретаторы языков программирования. Анализируя текст программного обеспечения с учетом возможности эффективно использовать аппаратурные особенность центральных процессоров в части использования векторных команд, компиляторы генерирую соответствующий машинный код. Например, компилятор GNU `gcc` начинает применять функции по обработке векторов данных, если в командной строке указана опция `-Ofast`.

Задание на лабораторную работу

Базовые задания.

1. Разработайте программу, реализующее псевдопараллельное выполнение двух функций: одна из которых непрерывно выводит на экран символ A, а другая непрерывно на экран выводит символ B. Переключение между выполнением функций должно осуществляться раз в три секунды по сигналу от таймера.

Основные задания.

1. Доработайте программу умножения прямоугольных матриц, разработанную в лабораторной работе 1, так, чтобы имелась возможность сгенерированную комбинацию матриц A и B записать в файл и считать из файла, чтобы провести расчет умножения матриц повторно.
2. Сгенерируйте наборы матриц A и B разных размеров (количество строк и столбов в матрицах в размерах от 16 до 4096, с шагом 16).
3. Используя ресурсы кластера Jet проведите исследования эффективности разработанной программы умножения матриц в зависимости от способа обхода матриц при расчете результата (всего 4 комбинации обхода: A по строкам и B по строкам, A по строкам и B по столбцам и т.д.). Необходимо построить для каждого из способов обхода матриц A и B графики зависимости времени умножения матриц от размера матрицы[1].

Задания повышенной сложности.

1. Разработайте программу, выполняющую над двумя векторами вещественных чисел размером 1024 элемента следующие вычисления:
$$C_i = \sqrt{A_i * B_i}$$
2. Используя компилятор `GCC` получили текст программы на ассемблере (опция `-S`) для всех уровней оптимизации кода (опции `-O0, -O1, -O2, -O3, -Ofast`).
3. Проанализируйте сколько операций выполнится для получения результата в каждом из способов оптимизации кода.

[1] Для измерения времени выполнения расчетов рекомендуется использовать счетчик таков процессора. Пример функции, возвращающий текущее значение счетчика:

```
#include <inttypes.h>
uint64_t get_time(){
  uint32_t low, high;
  __asm__ __volatile__("rdtsc\n" : "=a" (low), "=d" (high) );
    return ((uint64_t)high << 32) | low;
}
```

Контрольные вопросы

1. В каких аспектах функционирования ЭВМ проявляется параллелизм на аппаратурном уровне?
2. Что такое прерывание? Какую роль подсистема прерываний играет в реализации параллельного функционирования устройств ЭВМ?
3. Зачем используется прямой доступ к памяти? Необходим ли такой режим доступа при параллельном функционировании устройств ЭВМ?
4. Что такое «сигнал» в операционной системе GNU\Linux? Как термин «сигнал» связан с термином «прерывание»?
5. Что такое «вектор прерывания»? Зачем используется обработчик прерываения? Что является обработчиком сигнала в GNU\Linux? Какие стандартные обработчики сигнала присутствуют в Linux?
6. Каким образом происходит доступ процессора к оперативной памяти? Зачем стали использовать многоуровневый доступ к памяти?
7. Что такое «линия кэша», «промах кэша»? Как эти параметры влияют на эффективность программного обеспечения?
8. В чем выражается параллелизм при работе процессора? Какие группы векторных команд Вы знаете?
9. Зачем используются опции оптимизации в компиляторе GNU\GCC?