Государственное образовательное учреждение
"Приднестровский государственный университет им. Т.Г. Шевченко"

Инженерно-технический институт

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

УТВЕРЖДАЮ

Заведующий кафедрой ПОВТ и АС


С.Г. Федорченко

«28» августа 2020 г.

ФОНД ОЦЕНОЧНЫХ СРЕДСТВ

по дисциплине

ПРОГРАММИРОВАНИЕ СПЕЦИАЛИЗИРОВАННЫХ ВЫЧИСЛИТЕЛЬНЫХ УСТРОЙСТВ

Направление подготовки
2.09.04.04 Программная инженерия

Профиль подготовки
Разработка программно-информационных систем

Квалификация (степень) выпускника: **магистр**

Форма обучения: **очная, заочная**

Год набора: **2019 г.**

Разработал:
к.т.н., доцент кафедры ПОВТ и АС,


С.Г. Федорченко

«28» августа 2020 г.

Тирасполь, 2020

Паспорт фонда оценочных средств по учебной дисциплине

1. В результате изучения дисциплины «Программирование специализированных вычислительных устройств» у обучающегося должны быть сформированы следующие компетенции:

Категория общепрофессиональных компетенций	Код и наименование общепрофессиональной компетенции	Код и наименование индикатора достижения общепрофессиональной компетенции
-	ОПК-5. Способен разрабатывать и модернизировать программное и аппаратное обеспечение информационных и автоматизированных систем	ИД-1$_{ОПК-5}$ Знать современное программное и аппаратное обеспечение информационных и автоматизированных систем
		ИД-2$_{ОПК-5}$ Уметь модернизировать программное и аппаратное обеспечение информационных и автоматизированных систем для решения профессиональных задач;
		ИД-3$_{ОПК-5}$ Иметь навыки разработки программного и аппаратного обеспечения информационных и автоматизированных систем для решения профессиональных задач

2. Программа оценивания контролируемой компетенции:

Текущая аттестация	Контролируемые модули, разделы (темы) дисциплины их название	Код контролируемой компетенции (или ее части)	Наименование оценочного средства
РУБЕЖНЫЙ КОНТРОЛЬ	Раздел 1. Графические ускорители на основе технологии CUDA.	ОПК-5	Лабораторная работа №1 Лабораторные работы №2 Контрольная работа №1
РУБЕЖНАЯ АТТЕСТАЦИЯ	Раздел 2. Архитектура микропроцессора Cell BE IBM.		Лабораторная работа №3 Лабораторная работа №4 Лабораторная работа №5 Контрольная работа №2
Промежуточная аттестация		Код контролируемой компетенции (или ее части)	Наименование оценочного средства
№1		ОПК-5	Экзамен

3. Показатели и критерии оценивания компетенции по этапам формирования, описание шкал оценивания

Этапы оценивания компетенции	Показатели достижения заданного уровня освоения компетенции	Критерии оценивания результатов обучения			
		2	**3**	**4**	**5**
Первый этап	ИД-1$_{ОПК-5}$ **Знать** современное программное и аппаратное обеспечение информационных и автоматизированных систем;	Не знает	Знает современное программное и аппаратное обеспечение информационных и автоматизированных систем но не знает способы применения их	Знает современное программное и аппаратное обеспечение информационных и автоматизированных систем, но делает ошибки, не влияющие на результаты	Знает современное программное и аппаратное обеспечение информационных и автоматизированных систем. Умеет адекватно его применять
Второй этап	ИД-2$_{ОПК-5}$ **Уметь** модернизировать программное и аппаратное обеспечение информационных и автоматизированных систем для решения профессиональных задач;	Не умеет	Правильно определяет необходимость модернизировать программное и аппаратное обеспечение информационных и автоматизированных систем, но не умеет применять методы модернизации для решения профессиональных задач	Умеет модернизировать программное и аппаратное обеспечение информационных и автоматизированных систем, но не умеет обрабатывать результаты	Умеет модернизировать программное и аппаратное обеспечение информационных и автоматизированных систем для решения профессиональных задач
Третий этап	ИД-3$_{ОПК-5}$ **Иметь** навыки разработки программного и аппаратного обеспечения информационных и автоматизированных систем для решения профессиональных задач	Не владеет	Владеет навыками разработки программного и аппаратного обеспечения информационных и автоматизированных систем, но не владеет навыками оценки качества	Владеет навыками разработки программного и аппаратного обеспечения информационных и автоматизированных систем, но ошибается в обработке их результатов	Владеет навыками разработки программного и аппаратного обеспечения информационных и автоматизированных систем для решения профессиональных задач

4. Шкала оценивания

Согласно Положению «О порядке организации аттестации в ИТИ ПГУ им. Т.Г. Шевченко, итоговая оценка представляет собой сумму баллов, полученных студентом по итогу освоения дисциплины (модуля):

Оценка в традиционной шкале	Оценка в 100-балльной шкале	Буквенные эквиваленты оценок в шкале ЗЕ (% успешно аттестованных)
5 (отлично)	88–100	A (отлично) – 88-100 баллов
4 (хорошо)	70–87	B (очень хорошо) – 80-87баллов
		C (хорошо) – 70-79 баллов
3 (удовлетворительно)	50–69	D (удовлетворительно) – 60-69 баллов
		E (посредственно) – 50-59 баллов
2 (неудовлетворительно)	0–49	Fx – неудовлетворительно, с возможной пересдачей – 21-49 баллов
		F – неудовлетворительно, с повторным изучением дисциплины – 0-20 баллов

Расшифровка уровня знаний, соответствующего полученным баллам, дается в таблице, указанной ниже

A	“Отлично” - теоретическое содержание курса освоено полностью, без пробелов, необходимые практические навыки работы с освоенным материалом сформированы, все предусмотренные программой обучения учебные задания выполнены, качество их выполнения оценено числом баллов, близким к максимальному.
B	“Очень хорошо” - теоретическое содержание курса освоено полностью, без пробелов, необходимые практические навыки работы с освоенным материалом в основном сформированы, все предусмотренные программой обучения учебные задания выполнены, качество выполнения большинства из них оценено числом баллов, близким к максимальному.
C	“Хорошо” - теоретическое содержание курса освоено полностью, без пробелов, некоторые практические навыки работы с освоенным материалом сформированы недостаточно, все предусмотренные программой обучения учебные задания выполнены, качество выполнения ни одного из них не оценено минимальным числом баллов, некоторые виды заданий выполнены с ошибками.
D	“Удовлетворительно” - теоретическое содержание курса освоено частично, но пробелы не носят существенного характера, необходимые практические навыки работы с освоенным материалом в основном сформированы, большинство предусмотренных программой обучения учебных заданий выполнено, некоторые из выполненных заданий, возможно, содержат ошибки.
E	“Посредственно” - теоретическое содержание курса освоено частично, некоторые практические навыки работы не сформированы, многие предусмотренные программой обучения учебные задания не выполнены, либо качество выполнения некоторых из них оценено числом баллов, близким к минимальному.
FX	“Условно неудовлетворительно” - теоретическое содержание курса освоено частично, необходимые практические навыки работы не сформированы, большинство предусмотренных программой обучения учебных заданий не выполнено, либо качество их выполнения оценено числом баллов, близким к минимальному; при дополнительной самостоятельной работе над материалом курса возможно повышение качества выполнения учебных заданий.
F	“Безусловно неудовлетворительно” - теоретическое содержание курса не освоено, необходимые практические навыки работы не сформированы, все выполненные учебные задания содержат грубые ошибки, дополнительная самостоятельная работа над материалом курса не приведет к какому-либо значимому повышению качества выполнения учебных заданий.

5. Типовые контрольные задания или иные материалы, необходимые для оценки знаний, умений, навыков и (или) опыта деятельности, характеризующих этапы формирования компетенций при изучении учебной дисциплины в процессе освоения образовательной программы

5.1 Типовой вариант лабораторной работы № 1.

Тема: Программная модель CUDA

Приступая к работе, нужно убедиться, что на вашем рабочем месте, будь то компьютер или ноутбук, присутствует дискретная видеокарта NVIDIA с чипом восьмого поколения G80 (NVIDIA GeForce 8).

Чтобы убедиться, что все условия выполнены, нужно подробно разобрать вопрос с видеокартами восьмого поколения, так сказать, какие модели нам подходят. Сейчас вряд ли у кого-то на компьютере можно обнаружить видеокарту GeForce 8800 GTX или старее, так как данную модель выпустили аж в 2006 году, но тем не менее не будет лишним узнать о том, какие GPU поддерживают CUDA. Полный перечень всех видеокарт можно увидеть на официальной странице NVIDIA:

https://developer.nvidia.com/cuda-gpus

Из программного обеспечения потребуется среда разработки CUDA Toolkit, которую можно скачать с официального сайта компании NVIDIA (https://developer.nvidia.com/cuda-toolkit).

CUDA Toolkit можно установить на Windows, Linux и Mac OSX.

Ссылки на установки по каждой из операционных систем приведены ниже:

Установка для Windows — https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows

Установка для Mac OSX — https://docs.nvidia.com/cuda/cuda-installation-guide-mac-os-x

Установка для Linux — https://docs.nvidia.com/cuda/cuda-installation-guide-linux

Также потребуется стандартный компилятор языка C. Лучше всего использовать Visual Studio 2008 и последующих версий. В связке с Visual Studio при установке Toolkit автоматически прописываются пути в переменной PATH. Выглядит это примерно следующим образом (если вдруг у кого-то автоматически не настроилось):

(Название переменной = Путь)

CUDA_BIN_PATH = C:\Programs\CUDA toolkit 2.0\bin

CUDA_INC_PATH = C:\Programs\CUDA toolkit 2.0\include

CUDA_LIB_PATH = C:\Programs\CUDA toolkit 2.0\lib

NVSDKCUDA_ROOT = C:\Programs\CUDA SDK 2.02.0811.0240

Чтобы при создании нового проекта на Visual Studio появился раздел CUDA нужно выполнить следующие действия.

Зайти в **Tools > Options > Projects and Solutions > VC++ Directories** вкладка **Executable files** добавить новый параметр и ввести:

D:\Programs\CUDA SDK 2.02.0811.0240\bin,

либо **$(CUDA_BIN_PATH)**;

вкладка **Include files** добавить новый параметр и ввести:

D:\Programs\CUDA toolkit 2.0\include, либо **$(CUDA_INC_PATH)**

добавить новый параметр и ввести:

D:\Programs\CUDA SDK 2.02.0811.0240\common\inc,

либо **$(NVSDKCUDA_ROOT)\common\inc**

вкладка **Library files** добавить новый параметр и ввести:

D:\Programs\CUDA toolkit 2.0\lib, либо **$(CUDA_LIB_PATH)** добавить новый параметр и ввести:

D:\Programs\CUDA SDK 2.02.0811.0240\common\lib,

либо **$(NVSDKCUDA_ROOT)\common\lib**

В конечном итоге при создании проекта появится вкладка **NVIDIA,** в которой будут подразделы с актуальными версиями CUDA.

Помимо работы на локальном компьютере существует возможность работать и вне среды Visual Studio или ей подобной, то есть через удаленное подключение с использованием SSH-клиента. Выбор SSH-клиента уже остается за вами. В данном учебном пособии будут приведены скриншоты из PuTTY 0.70 версии и из WinSCP версии 5.13.

Чтобы проверить, что на вашем компьютере все настроено правильно, нужно через терминал запустить команду:

nvcc -V

Данная команда должна отобразить всю информацию по CUDA на вашем устройстве. Должно быть примерно так:

Copyright (c) 2005-2016 NVIDIA Corporation

Built on Tue_Jan_10_13:28:28_CST_2017

Cuda compilation tools, release 8.0, V8.0.61

nvcc:NVIDIA (R) Cuda compiler driver означает, что компилятор CUDA *nvcc* присутствует в вашей системе.

Cuda compilation tools, release 8.0, V8.0.61 указывает релиз и версию CUDA Toolkit.

5.2 Типовой вариант лабораторной работы № 2.

Тема: Арифметические операции

Познакомимся с основами практики программирования на CUDA C – будет подробно описано написание кода, его суть и исполнение. В качестве примеров будут разобраны самые популярные и полезные задачи, среди которых можно выделить:

- арифметические операции (сложение, умножение) с числами и матрицами,
- добавление массива,
- генерация псевдослучайных чисел,
- увеличение значений в матрице,
- алгоритмы сортировки и т.д.

В примерах будут разобраны такие моменты, как комбинация тредов и блоков, применение shared-памяти в решении задачи, использование SAXPY («Single-Precision A · X Plus Y»).

Сложение двух чисел

```
#include <stdio.h>
 __global__ void add( int *a, int *b, int *c ) {
*c = *a + *b;
}
 int main( void ) {
```

```
int a, b, c; // host копии a, b, c
int *dev_a, *dev_b, *dev_c; // device копии of a, b, c
int size = sizeof( int );
 //выделяем память для device копий для a, b, c
 cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
 a = 5;
b = 10;
// копируем ввод на device
 cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );
 // запускаем add() kernel на GPU, передавая параметры
 add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
 // copy device result back to host copy of c
 cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
 return 0;
 }
```

Прописываем спецификатор __global__, внутри которого 3 аргумента — 2 числа и результат их сложения. В теле этой функции прописываем саму арифметическую операцию

*c = *a + *b.

В главной функции переменные разделяются на две части: на хосте и на девайсе (названия переменных помечены соответственно), кроме этого указывается переменная, отвечающая за размер данных — int size. Далее выделяем память на GPU с помощью команды cudaMalloc, в аргументах прописаны указатели памяти на указатели типа void по-адресно (dev_a, dev_b, dev_c) и выделяемый размер памяти для каждой переменной. Прописываем значения переменных *a* и *b* (например, 5 и 10). Копируем значения на GPU и запускаем ядро GPU с помощью команды add<<< 1, 1 >>>(dev_a, dev_b, dev_c);. Копируем результат переменной *c* обратно на CPU и освобождаем память.

Перейдем к примерам посложнее, когда нужно менять значение блоков, тредов поочередно и одновременно. Подробное описание в них будет опущено, за исключением комментариев в самом коде.

Сложение двух чисел с использованием N-блоков по одному треду

```
__global__ void add( int *a, int *b, int *c ) {
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
 #define N 512
 int main( void ) {
int *a, *b, *c; // host копии a, b, c
int *dev_a, *dev_b, *dev_c; // device копии a, b, c
int size = N * sizeof( int );
 //выделяем память для device копий a, b, c
 cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
 a = (int*)malloc( size );
b = (int*)malloc( size );
```

```
c = (int*)malloc( size );
 random_ints( a, N );
random_ints( b, N );
 // копируем ввод на device
 cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );
 // launch add() kernel with N parallel blocks
 add<<< N, 1 >>>( dev_a, dev_b, dev_c );
 // копируем результат работы device обратно на host - копию c
 cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );
free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
 return 0;
 }
```

Сложение двух чисел с одним блоком и N-тредов

```
__global__ void add( int *a, int *b, int *c ) {
c[ threadIdx.x ] = a[ threadIdx.x ] + b[ threadIdx.x ];
// blockIdx.x blockIdx.x blockIdx.x
 }
 #define N 512
 int main( void ) {
int *a, *b, *c; //host копии a, b, c
int *dev_a, *dev_b, *dev_c; //device копии of a, b, c
int size = N * sizeof( int );
 //выделяем память для копий a, b, c
 cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
 a = (int*)malloc( size );
b = (int*)malloc( size );
c = (int*)malloc( size );
 random_ints( a, N );
random_ints( b, N );
 // копируем ввод device
 cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );
 // запускаем на выполнение add() kernel с N тредами в блоке
 add<<< 1, N >>>( dev_a, dev_b, dev_c );
 // N, 1
 // копируем результат работы device обратно на host (копия c)
 cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );
free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
 return 0;
 }
```

Сложение двух чисел с использованием N-блоков и N-тредов

```
__global__ void add( int *a, int *b, int *c ) {
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

```
c[index] = a[index] + b[index];
 }
 #define N (2048*2048)
 #define THREADS_PER_BLOCK 512
 int main( void ) {
int *a, *b, *c; // host копии a, b, c
int *dev_a, *dev_b, *dev_c; // device копии of a, b, c
int size = N * sizeof( int );
 //выделяем память на device для of a, b, c
 cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
 a = (int*)malloc( size );
b = (int*)malloc( size );
c = (int*)malloc( size );

random_ints( a, N );
random_ints( b, N );
 //копируем ввод на device
 cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );
 //запускаем на выполнение add() kernel с блоками и тредами
 add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b,
dev_c );
 // копируем результат работы device на host ( копия c )
 cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );
free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
 return 0;

}
```

ЗАДАНИЕ

1. Реализовать программы, тексты которых приведены в лабораторной работе.
2. Выполнить индивидуальное задание преподавателя

ЛИТЕРАТУРА

https://cc.dvfu.ru/ru/lesson-5/

Введение в программирование на CUD

5.3 Типовой вариант контрольной работы № 1.

1. Закон Амдала:
   1) показывает зависимость ускорения выполнения программы от количества параллельно работающих ядер;
   2) показывает зависимость ускорения выполнения программы от частоты системной шины;
   3) показывает зависимость ускорения выполнения программы от частоты процессора;
   4) показывает зависимость ускорения выполнения программы от применяемых алгоритмов.

2. SIMD-процессор:

1) одна и та же операция применяется одновременно ко множеству зависимых данных;
2) одна и та же операция применяется одновременно ко множеству независимых данных;
3) одна и та же операция применяется ко множеству независимых данных;
4) одна и та же операция применяется ко множеству зависимых данных.

3. Архитектура GPU:
   1) содержит один арифметико-логический блок;
   2) работает на высокой тактовой частоте;
   3) работает на низкой тактовой частоте;
   4) весь состоит из арифметико-логических блоков;
   5) похожа на структуру центрального процессора.

4. Программная модель CUDA:
   1) система компиляции и исполнения программ, работающих на CPU;
   2) система компиляции и исполнения программ, части которых работают на CPU и GPU;
   3) система компиляции и исполнения программ, работающих на GPU;
   4) система компиляции и исполнения программ, части которых работают на разных операционных системах.

5. Общий прием программирования для CUDA:
   1) группировка множества нитей в блоки;
   2) выделение главной нити;
   3) формирование блоков, реализующих ядро алгоритма вычислений;
   4) нити в рамках одного блока взаимодействуют.

6. Оптимизация работы с глобальной памятью CUDA:
   1) адреса должны быть кратны 64;
   2) выполняется выравнивание адресов (кратные 256);
   3) запросы всех нитей варпа (полуварпа) выполняются независимо друг о друга;
   4) объединяются запросы всех нитей варпа (полуварпа).

7. Асинхронные функции CUDA:
   1) хранение информации;
   2) запуск ядра;
   3) дублирование информации;
   4) функции, имена которых заканчиваются на Async.

8. Атомарные операции CUDA:
   1) AtomaticAdd;
   2) AtomaticSub;
   3) AtomaticIn;
   4) AtomaticDe.

9. Разделяемая память CUDA:
   1) выделяется на уровне нитей;
   2) выделяется на уровне блоков;
   3) выделяется для каждого процесса;
   4) глобальная память.

10. Константная и текстурная память CUDA:
   1) используется для хранения кода программы;

2) расположена в DRAM;
3) обладает независимым кэшем;
4) не обладает независимым кэшем.

11.Максимальная размерность блока памяти CUDA:
1) <64K;
2) 64K;
3) 512K;
4) <1024K.

12.Дивергентное ветвление, это:
1) нити в составе варпа выполняют параллельно одну и ту же команду;
2) если нити одного варпа должны идти по разным веткам кода, то выполнятся все проходимые ветки кода;
3) пронумерованная группа нитей называется варпом;
4) варп содержит 32 нити.

5.4 Типовой вариант лабораторной работы № 3.

Тема: Обработчик событий

Увеличение значение у элементов в матрице на единицу

Следующая программа будет увеличивать значение элемента в матрице на единицу. Ниже приведен листинг всего кода с комментариями.

```
#include <stdio.h>
 __global__ void incKernel ( float * data )
{
int idx = blockIdx.x * blockDim.x + threadIdx.x;
data [idx] = data [idx] + 1.0f;
}
 int main ( int argc, char * argv [] )
{
int n = 16 * 1024 * 1024;
int numBytes = n * sizeof ( float );
 // выделение памяти на хосте
float * a = new float [n];
for ( int i = 0; i < n; i++ )
a [i] = 0.0f;

// выделение памяти на девайсе
float * dev = NULL;
cudaMalloc ( (void**)&dev, numBytes );
 // Установка конфигурации запуска ядра
 dim3 threads = dim3(512, 1);
dim3 blocks = dim3(n / threads.x, 1);
 // создание обработчиков событий cuda
 cudaEvent_t start, stop;
float gpuTime = 0.0f;
 cudaEventCreate ( &start );
cudaEventCreate ( &stop );
 // асинхронно выдаем работу на GPU (все в поток 0)
 cudaEventRecord ( start, 0 );
```

```
cudaMemcpy ( dev, a, numBytes, cudaMemcpyHostToDevice );
 incKernel<<<blocks, threads>>>(dev);
 cudaMemcpy ( a, dev, numBytes, cudaMemcpyDeviceToHost );
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &gpuTime, start, stop );
 // Печатаем время работы на CPU и GPU
 printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );
 // проверка аутпута на корректность
 printf("------------------------------------------------------------\n");
 for ( int i = 0; i < n; i++ )
if ( a [i] != 1.0f )
 {
 printf ( "Error in pos %d, %f\n", i, a [i] );
 break;
 }
 // освобождение ресурсов
 cudaEventDestroy ( start );
cudaEventDestroy ( stop );
cudaFree( dev );
 delete a;
 return 0;
 }
```

В примере продемонстрирован обработчик событий, который не был ранее описан. Итак, обработчик событий CUDA cudaEvent_t start**,** stop задает начало и конец события. С помощью переменной float gpuTime = 0.0f будет производиться расчет времени выполнения программы в миллисекундах. Запись события производится с помощью специального объявления cudaEventRecord **(**start/stop**,** 0**).** Вычисление времени производится с помощью cudaEventElapsedTime. Это время, которое потребовалось для вычисления только на стороне GPU. В последующих примерах работа с записью времени подробно описываться не будет, за исключением того, что не было описано здесь.

В данном примере ядро устроено следующим образом: каждому треду соответствует один тред, блоки и grid одномерны. Ядро (она же функция incKernel) на вход получает только указатель на массив с данными в глобальной памяти. Задача ядра — по threadIdx и blockIdx определить какой именно элемент соответствует данному треду и увеличить именно его.

Так как и блоки и grid одномерны, то номер треда будет определятся как произведение номера блока на количество тредов в блоке, плюс номер треда внутри блока, т.е. blockIdx.x * blockDim.x + threadIdx.x.

Функция main несколько сложнее — она должна подготовить массив с данными в памяти CPU, после чего при помощи cudaMalloc выделяется память под копию массива с данными в глобальной памяти (DRAM GPU). Далее данные копируются функцией cudaMemcpy из памяти CPU к глобальную память GPU.

По завершению копирования данных в глобальную память запускается ядро для обработки данных и после его вызова копируются результаты вычислений обратно из глобальной памяти GPU в память CPU.

Результатом выполнения будет замер затраченного на копирование и вычисления времени программы, а также проверка на корректность получаемого результата:

Перемножение двух матриц

Следующий пример будет чуть сложнее, но не менее интереснее — перемножение двух квадратных матриц размерности N*N.

Допустим у нас есть две квадратные матрицы A и B размера N*N (будем считать, что N кратно 16). Простейший вариант использует по одному треду на каждый элемент получающейся матрицы C, при этом тред извлекает все необходимые элементы из глобальной памяти и производит требуемые вычисления.

Элемент $c_{i,j}$ произведения двух матриц A и B вычисляется следующим фрагментом псевдокода:

c [i][j] считать равным нулю;
Для каждого k-го элемента от 0 до N-1 считать:
*c [i][j] += a [i][k] * b [k][j]*

Таким образом для вычисления одного элемента произведения матриц нужно выполнить 2*N арифметических операций и 2*N чтений из глобальной памяти. Очевидно, что в основным лимитирующим фактором является скорость доступа к глобальной памяти, которая весьма низка

```
#include <stdio.h>
#define BLOCK_SIZE 16 // submatrix size
#define N 1024 // matrix size is N*N
 __global__ void matMult ( float * a, float * b, int n, float * c )
 {
int bx = blockIdx.x; // block index
int by = blockIdx.y;
int tx = threadIdx.x; // thread index
int ty = threadIdx.y;
 float sum = 0.0f; // computed subelement
int ia = n * BLOCK_SIZE * by + n * ty; // a [i][0]
int ib = BLOCK_SIZE * bx + tx;
 // Multiply the two matrices together;
 for ( int k = 0; k < n; k++ )
sum += a [ia + k] * b [ib + k*n];
 // Write the block sub-matrix to global memory;
 // each thread writes one element
 int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
c [ic + n * ty + tx] = sum;
}
 int main ( int argc, char * argv [] )
{
int numBytes = N * N * sizeof ( float );
 // выделение памяти на хосте
 float * a = new float [N*N];
float * b = new float [N*N];
float * c = new float [N*N];
 for ( int i = 0; i < N; i++ )
for ( int j = 0; j < N; j++ )
{
int k = N*i + j;
a [k] = 0.0f;
b [k] = 1.0f;
}
```

```
// выделение памяти на девайсе
float * adev = NULL;
float * bdev = NULL;
float * cdev = NULL;
cudaMalloc ( (void**)&adev, numBytes );
cudaMalloc ( (void**)&bdev, numBytes );
cudaMalloc ( (void**)&cdev, numBytes );
// Установка конфигурации запуска ядра
dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
dim3 blocks ( N / threads.x, N / threads.y);
// Создание обработчика событий CUDA
cudaEvent_t start, stop;
float gpuTime = 0.0f;
cudaEventCreate ( &start );
cudaEventCreate ( &stop );
// асинхронно выдаваем работу на GPU (все в поток 0)
cudaEventRecord ( start, 0 );
cudaMemcpy ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy ( bdev, b, numBytes, cudaMemcpyHostToDevice );
matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );
cudaMemcpy ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &gpuTime, start, stop );
// Печатаем время работы на GPU и CPU
printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );
// Освобождение ресурсов
cudaEventDestroy ( start );
cudaEventDestroy ( stop );
cudaFree ( adev );
cudaFree ( bdev );
cudaFree ( cdev );
delete a;
delete b;
delete c;
return 0;
}
```

Саму матрицу можно задать произвольно, в данном примере была указана лишь размерность без каких-либо значений элементов. Данный пример демонстрирует производительность перемножения двух матриц размерностей N*N:

Перемножение двух матриц с использованием shared-памяти

Разберем быстродействие программы за счет использования shared-памяти. Для этого нужно разбить результирующую матрицы на подматрицы 16*16, вычислением каждой такой подматрицы будет заниматься один блок. Обратите внимание, что для вычисления такой подматрицы нужны только небольшие «бэнды» матриц A и B .

Нет возможности копировать «бэнды» в shared-память из-за довольно небольшого объема shared-памяти. Поэтому можно поступить другим образом — разбить бэнды на матрицы 16*16 и вычисление подматрицы произведения матриц будем проводить в N/16 шагов.

Поскольку используется разбиение бэндов на квадратные подматрицы, то псевдокод расчета элемент $c_{i,j}$ изменится:

c [i][j] равен нулю;
Для каждого шага от 0 до N/16 считать:
По каждому k-му элементу от 0..16 считать:
*c [i][j] += a [i][k+step*16] * b [k+step*16][j]*

Следует обратить внимание, что для каждого значения step значения из матриц A и B берутся из двух подматриц размером 16*16.

На каждом шаге будем загружать в shared-память по одной 16*16 подматрице A и одной 16*16 подматрице B. Далее будем вычислять соответствующую им сумму для элементов произведения, потом загружаем следующие 16*16-подматрицы и т.д.

При этом на каждом шаге один тред загружает ровно по одному элементу из каждой из матриц A и B и вычисляет соответствующую им сумму членов. По окончании всех вычислений производится запись элемента в итоговую матрицу.

Следует обратить внимание, что после загрузки элементов из A и B нужно выполнить синхронизацию тредов при помощи вызова __synchronize для того, чтобы к моменту начала расчетов все нужные элементы (загружаемые остальными тредами блока) были бы уже загружены. Точно также по окончании обработки загруженных подматриц и перед загрузкой следующих также нужна синхронизация.

Ниже приводится соответствующий исходный код:

```
#include <stdio.h>
 #define BLOCK_SIZE 16 // submatrix size
#define N 1024 // matrix size is N*N
 __global__ void matMult ( float * a, float * b, int n, float * c )
{
int bx = blockIdx.x; // block index
int by = blockIdx.y;
int tx = threadIdx.x; // thread index
int ty = threadIdx.y;
 // Index of the first sub-matrix of A processed by the block
int aBegin = n * BLOCK_SIZE * by;
int aEnd = aBegin + n - 1;
 // Step size used to iterate through the sub-matrices of A
int aStep = BLOCK_SIZE;
 // Index of the first sub-matrix of B processed by the block
int bBegin = BLOCK_SIZE * bx;

// Step size used to iterate through the sub-matrices of B
int bStep = BLOCK_SIZE * n;
float sum = 0.0f; // computed subelement
for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )
{
 // Shared memory for the sub-matrix of A
__shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
 // Shared memory for the sub-matrix of B
__shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
 // Load the matrices from global memory to shared memory;
as [ty][tx] = a [ia + n * ty + tx];
bs [ty][tx] = b [ib + n * ty + tx];
```

```
 __syncthreads(); // Synchronize to make sure the matrices are loaded

// Multiply the two matrices together;
for ( int k = 0; k < BLOCK_SIZE; k++ )
sum += as [ty][k] * bs [k][tx];
 // Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
 __syncthreads();
}
 // Write the block sub-matrix to global memory;
// each thread writes one element
 int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
c [ic + n * ty + tx] = sum;
 }
int main ( int argc, char * argv [] )
{
int numBytes = N * N * sizeof ( float );
 // allocate host memory
 float * a = new float [N*N];
float * b = new float [N*N];
float * c = new float [N*N];
 for ( int i = 0; i < N; i++ )
for ( int j = 0; j < N; j++ )
{
a [i] = 0.0f;
b [i] = 1.0f;
}
 // allocate device memory
 float * adev = NULL;
float * bdev = NULL;
float * cdev = NULL;
 cudaMalloc ( (void**)&adev, numBytes );
cudaMalloc ( (void**)&bdev, numBytes );
cudaMalloc ( (void**)&cdev, numBytes );
 // set kernel launch configuration
 dim3 threads ( BLOCK_SIZE, BLOCK_SIZE );
dim3 blocks ( N / threads.x, N / threads.y);
 // create cuda event handles
 cudaEvent_t start, stop;
float gpuTime = 0.0f;
cudaEventCreate ( &start );
cudaEventCreate ( &stop );
 // asynchronously issue work to the GPU (all to stream 0)
 cudaEventRecord ( start, 0 );
cudaMemcpy ( adev, a, numBytes, cudaMemcpyHostToDevice );
cudaMemcpy ( bdev, b, numBytes, cudaMemcpyHostToDevice );
 matMult<<<blocks, threads>>> ( adev, bdev, N, cdev );
 cudaMemcpy ( c, cdev, numBytes, cudaMemcpyDeviceToHost );
cudaEventRecord ( stop, 0 );
cudaEventSynchronize ( stop );
cudaEventElapsedTime ( &gpuTime, start, stop );
 // print the cpu and gpu times
```

```
printf("time spent executing by the GPU: %.2f millseconds\n", gpuTime );
// release resources
cudaEventDestroy ( start );
cudaEventDestroy ( stop );
cudaFree ( adev );
cudaFree ( bdev );
cudaFree ( cdev );
delete a;
delete b;
delete c;
return 0;
}
```

Теперь для вычисления одного элемента произведения матриц нам нужно всего 2*N/16 чтений из глобальной памяти. И по результатам сразу видно за счет использования shared-памяти удалось поднять быстродействие более чем на порядок – 3.66 милисекунд.

ЗАДАНИЕ

1. Реализовать программы, тексты которых приведены в лабораторной работе.
2. Выполнить индивидуальное задание преподавателя

ЛИТЕРАТУРА

1. https://cc.dvfu.ru/ru/lesson-6/
2. Введение в программирование на CUD

5.5 Типовой вариант лабораторной работы № 4.

Тема: Битонная сортировка

Битонная сортировка – это параллельный алгоритм сортировки, основанный на битонной последовательности. Такой последовательностью называют последовательность, которая монотонно не убывает, а затем монотонно не возрастает. Пример такой последовательности – {8, 10, 16, 12, 4, -2, -8}.

Ее можно эффективно отсортировать следующим образом: разобьем массив на две части, создадим два массива, в первый добавим все элементы, равные минимуму из соответственных элементов каждой из двух частей, а во второй – равные максимуму. Утверждается, что получатся две битонные последовательности, каждую из которых можно рекурсивно отсортировать тем же образом, после чего можно склеить два массива (так как любой элемент первого меньше или равен любого элемента второго). Для того, чтобы преобразовать исходный массив в битонную последовательность, сделаем следующее: если массив состоит из двух элементов, можно просто завершиться, иначе разделим массив пополам, рекурсивно вызовем от половинок алгоритм, после чего отсортируем первую часть по порядку, вторую в обратном порядке и склеим.

С помощью CUDA будет реализован данный алгоритм.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
/* Каждый поток получает ровно одно значение в несортированном массиве */
#define THREADS 512 // 2^9
#define BLOCKS 32768 // 2^15
```

```
#define NUM_VALS THREADS*BLOCKS
 /* Функция печати результата выполнения программы*/
 void print_elapsed(clock_t start, clock_t stop) {
 double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC;
printf("Elapsed time: %.3fs\n", elapsed);
}
 /* Генерация случайных чисел массива */
 float random_float() {
 return (float)rand()/(float)RAND_MAX;
}
 /* Печать массива */
 void array_print(float *arr [], int length) {
 int i;
for (i = 0; i < length; ++i) {
printf("%1.3f ", arr[i]);
}
 printf("\n");
}
 /* Заполнение массива */
 void array_fill(float *arr, int length) {
 srand(time(NULL));
int i;
for (i = 0; i < length; ++i) {
arr[i] = random_float();
}
}
 __global__ void bitonic_sort_step(float *dev_values, int j, int k) {
 unsigned int i, ixj; /* Сортировка i и ixj */
i = threadIdx.x + blockDim.x * blockIdx.x;
ixj = i^j;
 /* Нити с наименьшими идентификаторами сортируют массив. */
 if ((ixj)>i) {
if ((i&k)==0) {
 /* Сортировка по возрастанию */
 if (dev_values[i]>dev_values[ixj]) {
 /* обмен(i,ixj) */
 float temp = dev_values[i];
 dev_values[i] = dev_values[ixj];
dev_values[ixj] = temp;
}
 }
 if ((i&k)!=0) {
 /* Сортировка по убыванию */
 if (dev_values[i]<dev_values[ixj]) {
 /* обмен(i,ixj); */
 float temp = dev_values[i];
 dev_values[i] = dev_values[ixj];
dev_values[ixj] = temp;
 }
 }
}
}
 /*
```

```
* Битонная сортировка на CUDA.
*/
 void bitonic_sort(float *values) {
 float *dev_values;
 size_t size = NUM_VALS * sizeof(float);
 cudaMalloc((void**) &dev_values, size);
 cudaMemcpy(dev_values, values, size, cudaMemcpyHostToDevice);
 dim3 blocks(BLOCKS,1); /* Количество блоков */
 dim3 threads(THREADS,1); /* Количество тредов */
 int j, k;
 /* Основной шаг выполнения сортировки */
 for (k = 2; k <= NUM_VALS; k <<= 1) {
 /* Второстепенный шаг выполнения сортировки */
 for (j=k>>1; j>0; j=j>>1) {
 bitonic_sort_step<<<blocks, threads>>>(dev_values, j, k);
 }
}
 cudaMemcpy(values, dev_values, size, cudaMemcpyDeviceToHost);
 cudaFree(dev_values);
 }
 int main(void)
 {
 clock_t start, stop;
 float *values = (float*) malloc( NUM_VALS * sizeof(float));
 array_fill(values, NUM_VALS);
 start = clock();
 bitonic_sort(values);
 stop = clock();
 print_elapsed(start, stop);
 }
```

Результатом выполнения программы будет затраченное время на сортировку случайных значений несортированного массива по 512 тредам и 32768 блокам. Количество значений будет равно их произведению – NUM_VALS THREADS*BLOCKS.

ЗАДАНИЕ

1. Реализовать программы, тексты которых приведены в лабораторной работе.
2. Выполнить индивидуальное задание преподавателя

ЛИТЕРАТУРА

1. https://cc.dvfu.ru/ru/lesson-10/
2. Введение в программирование на CUD

5.6 Типовой вариант лабораторной работы № 5.

Тема: Программирование процессора SPU на C/C++

Для того чтобы вы могли использовать SPU-расширения языка C/C++, в начало вашего кода должен быть включен заголовочный файл spu_intrinsics.h.

Основы векторов SPU

Главное отличие векторных процессоров от невекторных заключается в том, что векторные процессоры имеют большие регистры, которые позволяют им хранить несколько значений (называющихся *элементами*) одного и того же типа и обрабатывать их за раз с помощью одной операции. В векторных процессорах регистр рассматривается и как простое устройство, и как составное. Для представления этой концепции в языке C/C++ добавлено ключевое слово vector , которое берет простой тип данных и использует его в пределах всего регистра. Например, инструкция vector unsigned int myvec; создает вектор из четырех целочисленных элементов, которые загружаются, обрабатываются и сохраняются в совокупности, а переменная myvec относится ко всем четырем элементам одновременно. Ключевое слово signed/unsigned необходимо использовать для объявлений типов данных, отличных от floating point. Векторные константы создаются путем помещения типа вектора в круглые скобки, за которыми следует содержимое вектора в фигурных скобках. Например, вы можете присвоить значения элементам вектора с именем myvec следующим образом:

```
vector unsigned int myvec = (vector unsigned int){1, 2, 3, 4};
```

Помимо непосредственного присвоения существуют четыре основных примитива, которые используются для выполнения преобразований между скалярными и векторными данными: spu_insert, spu_extract, spu_promote и spu_splats. spu_insert используется для помещения скалярного значения в определенный элемент вектора spu_insert(5, myvec, 0) возвращает копию вектора myvec, в котором первый элемент (элемент 0) равен 5. spu_extract извлекает определенный элемент из вектора и возвращает его в качестве скалярной величины. spu_extract(myvec, 0) возвращает в качестве скалярной величины первый элемент вектора myvec. spu_promote преобразовывает значение в вектор, но определяет только один элемент. Тип вектора зависит от типа преобразовываемого значения. spu_promote((unsigned int)5, 1) создает вектор типа unsigned int со значением 5 во втором элементе (элемент 1), а остальные элементы остаются не определены. spu_splats работает подобно spu_promote за исключением того, что значение копируется во *все* элементы вектора. spu_splats((unsigned int)5) создает вектор типа unsigned int со значением 5 в каждом элементе.

Заманчиво было бы представлять себе векторы, как небольшие массивы, но фактически они ведут себя иначе в некоторых отношениях. Векторы, по существу, рассматриваются в качестве скалярных значений, тогда как массивы обрабатываются как ссылки. Например, spu_insert *не изменяет содержимое вектора*. Вместо этого возвращается совершенно новая копия вектора со вставленным элементом. Это выражение, результатом которого является значение, а не изменение самого значения. Например, также как myvar + 1 возвращает новое значение вместо изменения вектора myvar, команда spu_insert(1, myvec, 0) не изменяет вектор myvec, а вместо этого возвращает новое значение вектора, которое эквивалентно вектору myvec за исключением того, что первый элемент содержит значение 1.

Ниже приведена небольшая программа, использующая эти концепции:

Листинг 1. Программа, которая знакомит вас с SPU-расширениями языка C/C++

```
#include <spu_intrinsics.h>
 void print_vector(char *var, vector unsigned int val) {
   printf("Vector %s is: {%d, %d, %d, %d}\n", var, spu_extract(val, 0),
    spu_extract(val, 1), spu_extract(val, 2), spu_extract(val, 3));
 }
 int main() {
   /* Создание четырех векторов */
   vector unsigned int a = (vector unsigned int){1, 2, 3, 4};
   vector unsigned int b;
   vector unsigned int c;
   vector unsigned int d;
    /* вектор b идентичен вектору a, но последний элемент изменен на 9 */
   b = spu_insert(9, a, 3);
    /* вектор c содержит все четыре значения, установленные в 20 */
   c = spu_splats((unsigned int) 20);
    /* вектор d содержит второе значение,
   установленное в 5, остальные – мусор */
/* (в данном случае они все будут установлены в 5, но на это не стоит полагаться) */
    d = spu_promote((unsigned int)5, 1);
    /* Вывод результатов */
   print_vector("a", a);
   print_vector("b", b);
   print_vector("c", c);
   print_vector("d", d);
    return 0;
 }
```

Для компиляции и сборки программы в среде elfspe просто выполните следующие команды:

```
spu-gcc   vec_test.c   -o
vec_test
./vec_test
```

Встроенные функции для работы с векторами

Расширения языка C/C++ включают в себя типы данных и встроенные функции, которые обеспечивают программисту почти полный доступ к инструкциям языка ассемблера SPU. Однако, существует множество встроенных функций, которые весьма упрощают язык ассемблера SPU путем объединения многих похожих инструкций в одну функцию. Инструкции, которые различаются лишь типом операнда (такие как a, ai, ah, ahi, fa и, в дополнение, dfa) представлены одной функцией C/C++, которая выбирает подходящую инструкцию на основании типа операнда. Вдобавок к этому, когда в качестве параметров указаны два параметра типа vector unsigned int, инструкция spu_add генерирует инструкцию a (32-bit add). Тем не менее, если в качестве параметров указаны два параметра типа vector float, будет сгенерирована инструкция fa (float add). Обратите внимание на то, что встроенные функции в основном имеют те же ограничения, что и соответствующие им инструкции языка ассемблера. Однако в случаях, когда непосредственное значение оказывается слишком большим для соответствующей инструкции непосредственного (immediate-mode) режима, компилятор переведет непосредственное значение в вектор и выполнит соответствующую операцию вектор/вектор. Например, spu_add(myvec, 2) гене-

рирует инструкцию ai (add immediate), тогда как spu_add(myvec, 2000) сначала загружает значение 2000 в собственный вектор, используя инструкцию il, а затем выполняет инструкцию a (add).

Порядок операндов во встроенных функциях, по существу, тот же самый, что и в инструкции ассемблера за исключением того, что первый операнд (тот, который в ассемблере содержит регистр назначения) в языке C/C++ не указан, а вместо этого он используется в качестве возвращаемого значения функции. Компилятор подставляет соответствующий операнд в код, генерируемый на языке ассемблера.

Ниже перечислены некоторые наиболее распространенные встроенные функции SPU (их типы не указаны, поскольку большинство из них являются полиморфными).

- spu_add(val1, val2)
- Добавляет каждый элемент val1 к соответствующему элементу val2. Если val2 не является векторным значением, оно добавляется к каждому элементу val1.
- spu_sub(val1, val2)
- Вычитает каждый элемент val2 из соответствующего элемента val1. Если val1 не является векторным значением, оно дублируется во всех элементах вектора, после чего из него вычитается val2.
- spu_mul(val1, val2)
- Поскольку инструкции умножения действуют по-другому, встроенные функции SPU не объединяют их так, как это происходит с другими операциями. spu_mul выполняет умножение типа floating point (одинарной и двойной точности). Результатом является вектор, в котором каждый элемент является результатом умножения соответствующих элементов val1 и val2.
- spu_and(val1, val2), spu_or(val1, val2), spu_not(val), spu_xor(val1, val2), spu_nor(val1, val2), spu_nand(val1, val2), spu_eqv(val1, val2)
- Логические операции выполняются поразрядно, поэтому типы получаемых ими операндов имеют значение лишь при вычислении типа возвращаемого ими значения. spu_eqv является поразрядной, а не поэлементной операцией эквивалентности.
- spu_rl(val, count), spu_sl(val, count)
- spu_rl выполняет циклический сдвиг каждого элемента val влево на число битов, указанное в соответствующем элементе count. Биты, вытесненные за пределы, переносятся вправо. Если count является скалярным значением, оно используется в качестве счетчика для всех элементов val. spu_sl действует точно также, но вместо циклического сдвига выполняется последовательный сдвиг.
- spu_rlmask(val, count), spu_rlmaska, spu_rlmaskqw(val, count), spu_rlmaskqwbyte(val, count)
- Названия этих операций очень запутаны. Операции называются "циклический сдвиг влево и маскирование" (rotate left and mask), но на самом деле они выполняют *сдвиг вправо* (эти операции *реализованы* путем комбинации сдвигов влево и маскирования, однако интерфейс программирования предназначен для сдвигов вправо). spu_rlmask и spu_rlmaska сдвигают каждый элемент val вправо на число битов, указанное в соответствующем элементе count (или на значение count, если count является скаляром). spu_rlmaska дублирует знаковый разряд по мере сдвига битов. spu_rlmaskqw оперирует целым четверным словом за раз, но только до 7 битов (вычисляется модуль count для помещения его в соответствующий диапазон).

spu_rlmaskqwbyte работает точно также за исключением того, что count является числом байтов, а не битов, а также является модулем 16, а не 8.

- spu_cmpgt(val1, val2), spu_cmpeq(val1, val2)
- Эти инструкции выполняют поэлементное сравнение двух операндов. Результаты сохраняются в качестве всех единиц (в случае совпадения) и всех нулей (в случае несовпадения) в результирующем векторе в соответствующем элементе. spu_cmpgt выполняет сравнение "больше чем", а spu_cmpeq – сравнение эквивалентности.
- spu_sel(val1, val2, conditional)
- Эта инструкция соответствует инструкции selb языка ассемблера. Сама инструкция является битовой, поэтому все типы используют одну и ту же базовую инструкцию. Тем не менее, вспомогательная функция возвращает значение того же типа, что и операнды. Так же, как и в языке ассемблера, spu_sel проверяет каждый бит, содержащийся в операнде conditional. Если этот бит нулевой, результирующий соответствующий бит выбирается из соответствующего бита val1; в противном случае он выбирается из соответствующего бита val2.
- spu_shuffle(val1, val2, pattern)
- Эта интересная инструкция позволяет вам переставлять байты в val1 и val2 в соответствии с шаблоном, указанном в pattern. Инструкция проверяет каждый байт в pattern, и если байт начинается с битов 0b10, соответствующий результирующий байт устанавливается равным 0x00; если байт начинается с битов 0b110, соответствующий результирующий байт устанавливается равным 0xff; если байт начинается с битов 0b111, соответствующий результирующий байт устанавливается равным 0x80; наконец (и это важнее всего), если не выполняется ни одно из предыдущих условий, последние пять битов байта шаблона используются для выбора того, какой байт из val1 или val2 должен использоваться в качестве значения для текущего байта. Два значения объединены, и пятибитовое значение используется в качестве индекса байта объединенного значения. Это используется для вставки элементов в векторы, а также для выполнения быстрого табличного поиска.

Все инструкции, начинающиеся с префикса spu_, будут пытаться подобрать наилучшее совпадение, основываясь на типах операндов. Тем не менее, не все векторные типы поддерживаются всеми инструкциями – данная функциональность основана на возможностях обработки этих типов инструкциями языка ассемблера. В дополнение к этому, если вы хотите использовать инструкцию, отличную от той, что выбирает компилятор, вы можете выполнять почти все инструкции, не связанные с ветвлениями, с помощью *специальных встроенных функций*. Все специальные встроенные функции имеют форму si_assemblyinstructionname, где assemblyinstructionname – это имя инструкции языка ассемблера, определенное в спецификации SPU Assembly Language Specification. Таким образом, команда si_a(a, b) дополнительно приводит к выполнению инструкции a. Все операнды специальных встроенных функций приводятся к специальному типу, который называется qword и является, по существу, непрозрачным типом регистрового значения. Возвращаемые специальными встроенными функциями значения также имеют тип qword, который впоследствии можно привести к любому выбранному вами векторному типу.

Использование встроенных функций

Давайте теперь посмотрим, как написать функцию преобразования в верхний регистр на языке C/C++, а не на ассемблере. Ниже перечислены основные шаги для преобразования одного вектора:

1. Преобразовать все значения с использованием преобразования в верхний регистр.
2. Выполнить сравнения вектора со всеми байтами и выяснить, попадают ли они в диапазон между 'a' и 'z'.
3. Путем сравнения выбрать между преобразованными и не преобразованными значениями, используя инструкцию выбора.

Кроме того, чтобы помочь лучше запланировать выполнение инструкций, некоторые из этих преобразований будут параллельно выполняться на языке ассемблера. В языке C/C++ вы можете вызывать встраиваемую функцию несколько раз и позволить компилятору соответствующим образом запланировать ее выполнение. Это не означает, что ваши знания о планировании выполнения инструкций бесполезны, ведь поскольку вы знаете, как работает планирование, вы сможете лучше предоставить компилятору исходный материал для работы. Если бы вы не знали, что планирование выполнения инструкций позволяет улучшить код, и что оно может быть дополнено раскрутками циклов, то вы не смогли бы помочь компилятору оптимизировать ваш код.

Итак, ниже приведен код функции convert_buffer_to_upper на языке C/C++ (введите как convert_buffer_c.c, сохранив в той же папке, в которой находятся файлы из предыдущих статей – эти файлы понадобятся вам для компиляции полного приложения).

Листинг 2. Преобразование в верхний регистр на языке C/C++

```
#include <spu_intrinsics.h>

unsigned char conversion_value = 'a' - 'A';

inline vec_uchar16 convert_vec_to_upper(vec_uchar16 values) {
   /* Обработка всех символов */
   vec_uchar16 processed_values = spu_absd(values, spu_splats(conversion_value));
   /* Выясним, какие из символов нуждаются в обработке
   (те, что находятся между 'a' and 'z') */
   vec_uchar16 should_be_processed = spu_xor(spu_cmpgt(values, 'a'-1),
   spu_cmpgt(values, 'z'));
   /* Использование should_be_processed для выбора
   между исходными и обработанными значениями */
   return spu_sel(values, processed_values, should_be_processed);
}

void convert_buffer_to_upper(vec_uchar16 *buffer, int buffer_size) {
   /* Найдем конец буфера (сначала необходимо привести к
   определенному виду, поскольку размер – это байты) */
   vec_uchar16 *buffer_end = (vec_uchar16 *)((char *)buffer + buffer_size);

   while(__builtin_expect(buffer < buffer_end, 1)) {
      *buffer = convert_vec_to_upper(*buffer);
      buffer++;
      *buffer = convert_vec_to_upper(*buffer);
      buffer++;
      *buffer = convert_vec_to_upper(*buffer);
```

```
            buffer++;
            *buffer = convert_vec_to_upper(*buffer);
            buffer++;
        }
    }
```

Чтобы выполнить компиляцию и сборку программы, просто выполните следующие команды:

```
spu-gcc convert_buffer_c.c convert_driver.s dma_utils.s -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

Как вы, вероятно, заметили, эта программа использует слегка отличающуюся форму записи для имен векторных типов, нежели раньше. В документации по встроенным функциям SPU (обратитесь к разделу Ресурсы) определены упрощенные имена векторных типов, начинающиеся с префикса vec_. Для целочисленных типов следующим символом является u (для подписанных) или s (для неподписанных). Далее следует имя используемого базового типа (char, int, float и так далее). Наконец, в конце указывается количество элементов этого типа, содержащихся в векторе. Например, vec_uchar16 является вектором из 16 элементов типа unsigned char, а vec_float4 – вектором из 4 элементов типа float. Эта форма записи существенно упрощает объем вводимой информации.

При вычислении значения buffer_end программа выполняет некоторые преобразующие действия. Поскольку переменная size измерялась в байтах, мне пришлось привести указатель к типу char * таким образом, чтобы, когда я изменяю размер, он бы изменялся в байтах, а не в четверных словах. Поскольку длина значения, на которое указывают указатели вектора, составляет 16 байтов, их инкремент составляет 16 байтов, тогда как инкремент указателей типа char составляет один байт. Вот почему работает инструкция buffer++ – она увеличивается на длину одного вектора, равную 16 байтам.

Другой интересной вещью в коде на C/C++ является конструкция __builtin_expect, которая помогает компилятору выполнять прогнозирование ветвлений. В C/C++ вы не можете напрямую выполнять прогнозирование ветвлений, потому что вы не имеете ни адреса ветвления, ни конечного адреса. Поэтому вы оставляете данную задачу компилятору, который затем может сгенерировать соответствующее прогнозирование ветвлений. __builtin_expect(buffer < buffer_end, 1) генерирует код ветвления на основе первого аргумента, buffer < buffer_end, но выполняет прогнозирование ветвления на основе второго аргумента, 1. Это указывает компилятору сгенерировать прогнозирования, которые ожидают, что значение buffer < buffer_end будет равно 1.

На этом этапе для программирования SPU доступны два компилятора, и, как можно предположить, каждый из них имеет преимущества в определенной области. GCC, например, выполняет фантастическую работу по чередованию инструкций между вызовами convert_vec_to_upper, таким образом, минимизируя задержки. Тем не менее, в этой программе конструкция __builtin_expect оказывается для нас практически бесполезной. Компилятор IBM XLC, с другой стороны, действует наоборот. Он вообще не организует чередование инструкций между вызовами convert_vec_to_upper, но структурирует цикл таким образом, что мы получаем максимальный эффект от прогнозирования ветвлений. Неудивительно, что ни один из этих компиляторов не обеспечивает такой же быстрой работы, как код из предыдущей статьи, вручную созданный на языке ассемблера, однако для этой

программы компилятор XLC работает лучше, чем GCC. Компилирование кода без каких-либо флагов оптимизации приводит к тому, что приложение работает примерно *в пять раз медленнее*, поэтому не забывайте всегда использовать при компиляции опцию -O2 или -O3.

Составные встроенные функции и программирование контроллера MFC

Составные встроенные функции – это те функции, которые компилируются в несколько инструкций. Составные встроенные функции инкапсулируют широко используемые шаблоны элементов SPE для упрощения их программирования. Две самые важные составные встроенные функции – это spu_mfcdma64 и spu_mfcstat. spu_mfcdma64 почти в точности повторяет функцию dma_transfer, которую я написал и использовал в предыдущей статье, за исключением того, что верхние и нижние части адреса основной памяти разделены на два 32-разрядных параметра (dma_transfer использует один 64-разрядный параметр для адреса основной памяти).

spu_mfcdma64 принимает шесть параметров:

1) адрес основной памяти для передачи
2) старшие 32 бита адреса основной памяти
3) младшие 32 бита адреса основной памяти
4) размер передачи
5) "тэг", назначаемый передаче
6) передаваемая команда DMA

Часто адрес основной памяти будет представлен в виде 64-разрядного значения. Для того чтобы разделить его на части, используйте инструкцию mfc_ea2h для получения старших битов и инструкцию mfc_ea2l – для получения младших битов. Тэг – это число от 0 до 31, назначенное программистом для идентификации передачи или группы передач в запросах статуса и операциях упорядочивания. Команда DMA может принимать ряд значений (обратитесь к разделу Ресурсы для получения информации о том, где искать не перечисленные здесь значения). Передача DMA называется PUT, если данные передаются из локальной памяти SPU в основную память, и GET – если передача происходит в обратном направлении. Имена этих команд DMA начинаются либо с префикса MFC_PUT, либо с префикса MFC_GET, соответственно. Далее, команды MFC работают либо по отдельности, либо в списке. Если команда DMA работает в списке, к ее имени добавляется символ L (обратитесь к разделу Ресурсы для получения дополнительной информации о командах DMA, работающих в списках). Команды DMA также могут иметь определенные уровни синхронизации, применяемые к ним. Для барьерной синхронизации добавьте символ B, для fence-синхронизации добавьте символ F, а в случае отсутствия синхронизации вам не нужно добавлять ничего. Наконец, все имена команд DMA имеют суффикс _CMD. Таким образом, команда для выполнения одной передачи из локальной памяти в основную с использованием fence-синхронизации будет называться MFC_PUTF_CMD.

По умолчанию команды DMA на контроллере MFC совершенно неупорядочены – контроллер MFC может обрабатывать их в любой последовательности, на свое усмотрение. Тем не менее, для наложения упорядочивающих ограничений на передачи MFC DMA можно использовать тэги, ограничители и барьеры. *Ограничитель* (fence) устанавливает ограничение, приводящее к тому, что передача DMA выполняется только *после* завершения выполнения всех предыдущих команд с тем же самым тэгом. *Барьер* (barrier) устанав-

ливает ограничение, приводящее к тому, что указанная передача DMA выполняется только *после* завершения выполнения всех предыдущих команд с тем же тэгом (как и в случае с ограничителем), но *до* начала выполнения всех последующих команд с тем же тэгом.

Ниже приведены несколько примеров:

Листинг 3. Использование spu_mfcdma64

```
        typedef unsigned long long uint64;
        typedef unsigned long uint32;
        uint64 ea1, ea2, ea3, ea4, ea5; /* предполагаем, что
        все переменные содержат разумные значения */
        void *ls1, *ls2, *ls3, *ls4; /* предполагаем, что все
         переменные содержат разумные значения */
        uint32 sz1, sz2, sz3, sz4; /* предполагаем, что все
        переменные содержат разумные значения */
        int tag = 3; /* Случайное значение, но оно должно быть
         одинаковым для всех синхронизированных передач */

/* Передача 1: Системная память -> Локальная память, без упорядочивания */
        spu_mfcdma64(ls1, mfc_ea2h(ea1), mfc_ea2l(ea1), sz1, tag, MFC_GET_CMD);

/* Передача 2: Локальная память -> Системная память, должна быть
        выполнена после предыдущих передач */
        spu_mfcdma64(ls2, mfc_ea2h(ea2), mfc_ea2l(ea2), sz2, tag, MFC_PUTF_CMD);

/* Передача 3: Локальная память -> Системная память, без упорядочивания */
        spu_mfcdma64(ls3, mfc_ea2h(ea3), mfc_ea2l(ea3), sz3, tag, MFC_PUT_CMD);

/* Передача 4: Локальная память -> Системная память, должна быть синхронизована */
        spu_mfcdma64(ls4, mfc_ea2h(ea4), mfc_ea2l(ea4), sz4, tag, MFC_PUTB_CMD);

/* Передача 5: Системная память -> Локальная память, без упорядочивания */
        spu_mfcdma64(ls4, mfc_ea2h(ea5), mfc_ea2l(ea5), sz4, tag, MFC_GET_CMD);
```

В вышеприведенном примере могли использоваться несколько возможных вариантов упорядочивания - они перечислены ниже:

- 1, 2, 3, 4, 5
- 3, 1, 2, 4, 5
- 1, 3, 2, 4, 5

Поскольку передача 2 использует только ограничитель, а в передаче 3 порядок выполнения вообще не указан, то передача 3 может выполняться в любом месте до барьера (передача 4). Единственным требованием для первых трех передач является то, что передача 2 должна быть выполнена после передачи 1. Передача 4, тем не менее, требует полной синхронизации передач, расположенных до и после нее.

Присмотритесь внимательнее к передачам 4 и 5. Это полезный прием, который можно взять на заметку – сохранение и перезагрузка. Если вы по частям передаете данные из системной памяти в локальную память и сохраняете их назад в системную память, вы можете запланировать одновременно сохранение и загрузку, используя ограничитель или барьер для их упорядочивания. Этот прием помещает всю логическую часть передачи в контроллер MFC, разгружая вашу программу и позволяя ей выполнять другие вычисления, пока буфер ожидает новые данные. Мы воспользуемся этим в следующей статье, когда будем говорить о двойной буферизации.

spu_mfcdma64 является достаточно удобным инструментом, но немного утомительным, особенно когда вам приходится продолжать использовать команды mfc_ea2h и mfc_ea2l для преобразования адресов. Поэтому спецификацией дополнительно предусмотрен ряд сервисных утилит, сокращающих необходимый объем ручного набора кода. Функции класса mfc_ принимают все те же параметры, что и функция spu_mfcdma64 за исключением того, что адрес основной памяти является простым 64-разрядным параметром, а команда DMA зашифрована в имени функции. Кроме того, эти функции принимают два дополнительных параметра: *идентификатор класса передачи* и *идентификатор класса замены*. Оба из них могут быть безо всяких последствий приравнены к нулю в приложениях, не работающих в режиме реального времени (обратитесь к разделу Ресурсы для получения ссылок на дополнительную информацию по этим двум параметрам). Таким образом, передачу 2 из вышеприведенного листинга можно переписать следующим образом:

```
mfc_putf(ls2, ea2, sz2, tag, 0, 0);
```

Тэги оказываются полезными не только для синхронизации передач данных, но также и для проверки их статуса. В элементе SPE существуют следующие каналы: канал маски тэгов, указывающий, какой тэг используется в настоящий момент для проверок статуса; канал для формирования запросов статуса и еще один канал для обратного считывания статуса канала. Хотя это достаточно простые операции, спецификация также содержит специальные методы для их выполнения. mfc_write_tag_mask принимает 32-разрядное целое число и использует его в качестве маски канала для последующих обновлений статуса. Установите в этой маске разряд каждого тэга, статус которого вы хотите проверить, равным 1. Так, для проверки статуса каналов 2 и 4 следует использовать инструкцию mfc_write_tag_mask(20), а для лучшего восприятия, вы можете использовать команду mfc_write_tag_mask(1<<2 | 1<<4);. Для фактического обновления статуса вам необходимо выбрать команду для обработки статуса и послать ее с помощью инструкции spu_mfcstat(unsigned int command). Команды для обработки статуса приведены в следующем списке:

- MFC_TAG_UPDATE_IMMEDIATE

Эта команда заставляет элемент SPE немедленно возвратить статус каналов DMA. Каждый канал, указанный в маске каналов, будет установлен в 1, если в очереди не осталось команд с этим тэгом (другими словами, если все операции, которые могли быть активны ранее, завершены), и установлен в 0, если в очереди присутствуют команды.

- MFC_TAG_UPDATE_ANY

Эта команда заставляет элемент SPE ожидать, пока не завершатся все команды хотя бы с одним из указанных в маске тэгом, а затем возвращает статус каналов DMA, указанных в маске тэгов.

- MFC_TAG_UPDATE_ALL

Эта команда заставляет элемент SPE ожидать, пока не завершатся все команды со всеми тэгами, указанными в маске тэгов, и только затем возвращает статус. Возвращаемым значением будет являться 0.

Для использования этих констант вам необходимо включить в код заголовочный файл spu_mfcio.h.

Использование spu_mfcstat позволяет вам выполнять проверку статуса и ожидание запросов DMA. Использование MFC_TAG_UPDATE_ANY позволяет вам выполнять не-

сколько запросов DMA, предоставляет контроллеру MFC возможность обрабатывать их в наилучшем с его точки зрения порядке, на основании которого происходит соответствующее взаимодействие с вашим кодом.

Пример программы для контроллера MFC

Теперь я применю знания об этих составных встроенных функциях MFC в программе преобразования в верхний регистр. Ранее в этой статье я переписал на языке C главную функцию преобразования, а теперь я перепишу на C главный цикл. Ниже приведен новый код (введите как convert_driver_c.c):

Листинг 4. Код передачи MFC для преобразования в верхний регистр

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
typedef unsigned long long uint64;
 #define CONVERSION_BUFFER_SIZE 16384
#define DMA_TAG 0
 void convert_buffer_to_upper(char *conversion_buffer, int current_transfer_size);
 char conversion_buffer[CONVERSION_BUFFER_SIZE];
 typedef struct {
   int length __attribute__((aligned(16)));
   uint64 data __attribute__((aligned(16)));
} conversion_structure;
 int main(uint64 spe_id, uint64 conversion_info_ea) {
   conversion_structure conversion_info; /*
   Information about the data from the PPE */
    /* В этой программе мы используем только один тэг */
   mfc_write_tag_mask(1<<DMA_TAG);
    /* Соберем информацию о преобразовании */
   mfc_get(&conversion_info, conversion_info_ea,
   sizeof(conversion_info), DMA_TAG, 0, 0);
   spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
    /* Получим реальные данные */
   mfc_get(conversion_buffer, conversion_info.data, conversion_info.length,
   DMA_TAG, 0, 0);
   spu_mfcstat(MFC_TAG_UPDATE_ALL);
    /* Выполним преобразование */
   convert_buffer_to_upper(conversion_buffer, conversion_info.length);
    /* Поместим данные обратно в системную память */
   mfc_put(conversion_buffer, conversion_info.data, conversion_info.length,
   DMA_TAG, 0, 0);
   spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
}
```

Для компиляции и запуска программы, просто выполните следующие команды:

```
spu-gcc convert_buffer_c.c convert_driver_c.c -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

Эта реализация на языке C следует той же основной структуре, что и исходный код, за исключением того, что новый код более удобен для восприятия. Это упрощает процесс его проверки и доработки. Например, одна из проблем исходного кода заключалась в том, что он был ограничен размером передачи DMA. Если бы вы захотели избавиться от этого ограничения, вы могли бы просто организовать цикл и обрабатывать данные по частям до

тех пор, пока не была бы обработана вся строка целиком. Ниже приведен переработанный код, позволяющий сделать это.

Листинг 5. Цикл в коде передачи MFC

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h> /* объявление константы для MFC */
typedef unsigned long long uint64;
typedef unsigned int uint32;
 /* CONVERSION_BUFFER_SIZE переименована в MAX_TRANSFER_SIZE, поскольку теперь она
главным образом используется для ограничения размера передач DMA */
#define MAX_TRANSFER_SIZE 16384
 void convert_buffer_to_upper(char *conversion_buffer, int current_transfer_size);
 char conversion_buffer[MAX_TRANSFER_SIZE];
 typedef struct {
   uint32 length __attribute__((aligned(16)));
   uint64 data __attribute__((aligned(16)));
} conversion_structure;
 int main(uint64 spe_id, uint64 conversion_info_ea) {
   conversion_structure conversion_info; /* Информация о данных из PPE */
   /* Новые переменные для отслеживания нашего местоположения в данных */
   uint32 remaining_data; /* Сколько данных осталось в целой строке */
   uint64 current_ea_pointer; /* Наше местоположение в системной памяти */
   uint32 current_transfer_size; /* Насколько велик размер текущей передачи
    (может быть
   меньше, чем MAX_TRANSFER_SIZE) */
    /* В этой программе мы используем только один тэг */
   mfc_write_tag_mask(1<<0);
    /* Соберем информацию о преобразовании */
   mfc_get(&conversion_info, conversion_info_ea, sizeof(conversion_info), 0, 0, 0);
   spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */

   /* Настройка цикла */
   remaining_data = conversion_info.length;
   current_ea_pointer = conversion_info.data;
   while(remaining_data > 0) {
      /* Определим, сколько данных осталось передать */
      if(remaining_data < MAX_TRANSFER_SIZE)
         current_transfer_size = remaining_data;
      else
         current_transfer_size = MAX_TRANSFER_SIZE;
      /* Получим реальные данные */
      mfc_getb(conversion_buffer, current_ea_pointer,
       current_transfer_size, 0, 0, 0);
      spu_mfcstat(MFC_TAG_UPDATE_ALL);
      /* Выполним преобразование */
      convert_buffer_to_upper(conversion_buffer,
      current_transfer_size);
      /* Поместим данные обратно в системную память */
      mfc_putb(conversion_buffer, current_ea_pointer,
      current_transfer_size, 0, 0, 0);
      /* Перейдем к следующей части данных */
      remaining_data -= current_transfer_size;
      current_ea_pointer += current_transfer_size;
   }
   spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
}
```

Для компиляции и запуска выполните те же команды, что и в предыдущем примере:

```
spu-gcc convert_buffer_c.c convert_driver_c.c -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

Итак, вы только что увеличили размер данных, которые вы можете обрабатывать, до 4 гигабайтов, хотя вы с легкостью могли бы еще более увеличить его, используя для работы с данными 64-разрядные переменные вместо 32-разрядных. Заметьте, что ваш код не содержит явных указаний контроллеру MFC ожидать завершения операций PUT прежде, чем выполняется повторный запуск операций GET. Это возможно благодаря тому, что при работе с передачами вы используете барьеры и одинаковый тэг DMA. В результате передачи переводятся в последовательный режим самим контроллером MFC, и, таким образом, он всегда ожидает, пока для текущего преобразования завершится операция PUT и данные будут помещены в системную память, прежде чем в буфер будет загружена (GET) следующая часть данных. Помните только о том, что нужно дождаться завершения последней операции (обратите внимание на команду spu_mfcstat за пределами цикла), иначе передача последнего бита ваших данных может не завершиться прежде, чем он будет использован в программе!

Еще одна вещь, с которой следует обращаться аккуратно при программировании на языке C, заключается в том, что всегда нужно следить за тем, что вы указываете прототипы функций (имя функции и список ее формальных параметров с указанием их типов). На самом деле, очень легко случайно перепутать 32- и 64-разрядные значения. В случае с PPE ничего плохого не произойдет, поскольку значения просто усекаются или расширяются, однако если прототипы окажутся неверными при работе с элементом SPE, привилегированный слот для 32- и 64-разрядных значений окажется смещенным так, что их необходимо будет преобразовывать явным образом.

Полезные советы по программированию SPE на языке C

Воспользуйтесь следующими советами при разработке приложений SPE на языке C:

- Векторы *могут* быть преобразованы в векторы других типов. Также можно выполнять прямые и обратные преобразования между типами векторов и специальным типом quad. Однако *никакое из этих преобразований не приводит к какому-либо преобразованию данных*. Если вам необходимо выполнить преобразование типов, используйте соответствующие встроенные функции SPU.
- Векторные и не векторные указатели *могут* быть преобразованы из одного типа в другой, но при преобразовании скалярного указателя в векторный *программист должен самостоятельно убедиться в том, что указатель выровнен по четверному слову*.
- При выделении памяти объявленные векторы всегда выровнены по четверному слову.
- Помните, что передачи DMA размером в 16 или более байтов *должны быть кратными 16 байтам и выровненными в пределах 16-байтовых границ* как для SPE, так и для PPE. Передачи меньшего размера должны являться степенями двойки и

быть естественно выровненными. Оптимальными передачами являются кратные 128 байтам передачи, которые выровнены в пределах 128-байтовый границ.

- Если вы не уверены насчет выравнивания данных в PPE, используйте инструкцию memalign или posix_memalign для выделения памяти выровненному указателю из кучи, и используйте инструкцию memcpy или подобную ей для перемещения данных в выровненную область.
- Всегда компилируйте программы с ключом -Wall и *в особенности уделяйте внимание сообщениям о пропущенных прототипах*. Неправильно определенные прототипы (в особенности это касается 32- и 64-разрядных типов) могут привести к непредсказуемым сбойным ситуациям.
- Всегда сохраняйте адреса основной памяти в виде unsigned long long как для PPE, так и для SPE. При таком подходе адреса могут интерпретироваться в качестве унифицированной формы для PPE и SPE независимо от того, был ли код PPE скомпилирован для выполнения в 32- или 64-разрядной среде.
- Избегайте операций умножения целочисленных типов (особенно 32-разрядных операций умножения). Для выполнения таких операций требуется пять инструкций. Если вам необходимо выполнить умножение, предварительно преобразуйте данные к типу unsigned short.
- Объявление скалярных значений в качестве векторов и векторных указателей в скалярном коде для SPE (даже если вы не используете их в качестве векторов) может ускорить выполнение кода, поскольку в этом случае не требуется выполнять невыровненные загрузки и сохранения.
- При работе с SPE остерегайтесь следующего: типы данных float и double реализованы по-разному и округляются также по-разному. В частности, значения типа float отклоняются от стандарта C99. В следующей статье мы рассмотрим это более подробно.

Заключение

Встроенные функции языка C позволяют программистам наилучшим образом использовать знания языков C и ассемблера. Встроенные функции SPU позволяют приложениям свободно переключаться между высокоуровневым и низкоуровневым кодом, написанным в рамках семантической структуры языка C.

Литература

1.https://www.ibm.com/developerworks/ru/library/pa-linuxps3-5/index.html?S_TACT=105AGX99&S_CMP=GR01
2. https://www.ibm.com/developerworks/ru/library/pa-linuxps3-5/pa-linuxps3-5-pdf.pdf

ЗАДАНИЕ

1. Реализовать программы, тексты которых приведены в лабораторной работе.
2. Выполнить индивидуальное задание преподавателя

5.7 Типовой вариант контрольной работы № 2.

1. Для оптимизации CUDA – программ используют:
   1) оптимизацию доступа к памяти;
   2) таймеры компьютеров;
   3) вмешательство операторов;

4) параллелизм задачи.

2. Максимальная производительность константной памяти, достигается при:
1) обращении всеми потоками варпа по двум адресам;
2) обращении всеми потоками варпа по одному адресу;
3) увеличении числа потоков варпа;
4) уменьшении числа потоков варпа.

3. Библиотека CUSPARSE позволяет:
1) учитывать физическое время;
2) реализует основные операции линейной алгебры для разреженных векторов и матриц;
3) реализовать преобразование Фурье;
4) реализует основные операции линейной алгебры для векторов и матриц;
5) поддерживает вещественные и комплексные типы данных.

4. Библиотека CUFFT реализует:
1) преобразование Фурье;
2) операции над матрицами;
3) решение систем линейных уравнений;
4) решение систем дифференциальных уравнений.

5. Библиотека CURAND содержит:
1) функции для вычисления стандартных математических функций;
2) набор констант;
3) генераторы случайных чисел;
4) генерацию стандартных сообщений.

6. Библиотека Thrust поддерживает:
1) нечеткую логику;
2) обработку сообщений без временных меток;
3) параллельные алгоритмы обработки данных;
4) алгоритмы редукции;
5) оптимизацию потока сообщений.

7. При использовании нескольких GPU:
1) используется уровень пользовательского интерфейса;
2) нет механизма управления распределенным кластером;
3) создаются контексты устройств;
4) работают на уровне протоколов.

8. Микропроцессор Cell:
1) содержит 2 блок PPE и 16 блоков SPE;
2) содержит 1 блок PPE и 8 блоков SPE;
3) содержит 4 блок PPE и 32 блока SPE;
4) содержит 16 блоков SPE;

9. Микропроцессор Cell:
1) одно ядро имеет доступ к общей памяти;
2) все ядра имеют доступ к общей памяти;
3) каждое ядро SPE имеет свою локальную память;
4) каждое ядро SPE работает только с общей памятью.

10. Блок SPE:
    1) содержит синергетическое процессорное устройство- SPU;
    2) не содержит синергетическое процессорное устройство- SPU;
    3) содержит несколько SPU;
    4) представляет собой RISC-процессор.

11. Программирование Cell:
    1) выполнено расширение языка Java;
    2) выполнено расширение языка C++;
    3) разработан собственный язык программирования;
    4) использует удаленный вызов процедур (RPC).

12. Модель программирования Cell:
    1) синхронизированная модель;
    2) сетевая модель;
    3) однопоточная модель;
    4) многопоточная модель.

13. Программирование SPE:
    1) обычная программа на языке C++;
    2) обычная программа на языке Java;
    3) программа на языке C++ в терминах векторов значений;
    4) программа на языке Java в терминах векторов значений.

5.8 Типовой тест для промежуточной аттестации

1. Закон Амдала:
    1) показывает зависимость ускорения выполнения программы от количества параллельно работающих ядер;
    2) показывает зависимость ускорения выполнения программы от частоты системной шины;
    3) показывает зависимость ускорения выполнения программы от частоты процессора;
    4) показывает зависимость ускорения выполнения программы от применяемых алгоритмов.

2. SIMD-процессор:
    1) одна и та же операция применяется одновременно ко множеству зависимых данных;
    2) одна и та же операция применяется одновременно ко множеству независимых данных;
    3) одна и та же операция применяется ко множеству независимых данных;
    4) одна и та же операция применяется ко множеству зависимых данных.

3. Архитектура GPU:
    1) содержит один арифметико-логический блок;
    2) работает на высокой тактовой частоте;
    3) работает на низкой тактовой частоте;
    4) весь состоит из арифметико-логических блоков;
    5) похожа на структуру центрального процессора.

4. Программная модель CUDA:

1) система компиляции и исполнения программ, работающих на CPU;
2) система компиляции и исполнения программ, части которых работают на CPU и GPU;
3) система компиляции и исполнения программ, работающих на GPU;
4) система компиляции и исполнения программ, части которых работают на разных операционных системах.

5. Общий прием программирования для CUDA:

1) группировка множества нитей в блоки;
2) выделение главной нити;
3) формирование блоков, реализующих ядро алгоритма вычислений;
4) нити в рамках одного блока взаимодействуют.

6. Оптимизация работы с глобальной памятью CUDA:

5) адреса должны быть кратны 64;
6) выполняется выравнивание адресов (кратные 256);
7) запросы всех нитей варпа (полуварпа) выполняются независимо друг о друга;
8) объединяются запросы всех нитей варпа (полуварпа).

7. Асинхронные функции CUDA:

1) хранение информации;
2) запуск ядра;
3) дублирование информации;
4) функции, имена которых заканчиваются на Async.

8. Атомарные операции CUDA:

1) AtomaticAdd;
2) AtomaticSub;
3) AtomaticIn;
4) AtomaticDe.

9. Разделяемая память CUDA:

1) выделяется на уровне нитей;
2) выделяется на уровне блоков;
3) выделяется для каждого процесса;
4) глобальная память.

10. Константная и текстурная память CUDA:

1) используется для хранения кода программы;
2) расположена в DRAM;
3) обладает независимым кэшем;
4) не обладает независимым кэшем.

11. Максимальная размерность блока памяти CUDA:

1) <64K;
2) 64K;
3) 512K;
4) <1024K.

12. Дивергентное ветвление, это:

1) нити в составе варпа выполняют параллельно одну и ту же команду;
2) если нити одного варпа должны идти по разным веткам кода, то выполнятся все проходимые ветки кода;
3) пронумерованная группа нитей называется варпом;
4) варп содержит 32 нити.

13. Для оптимизации CUDA – программ используют:
1) оптимизацию доступа к памяти;
2) таймеры компьютеров;
3) вмешательство операторов;
4) параллелизм задачи.

14. Максимальная производительность константной памяти, достигается при:
1) обращении всеми потоками варпа по двум адресам;
2) обращении всеми потоками варпа по одному адресу;
3) увеличении числа потоков варпа;
4) уменьшении числа потоков варпа.

15. Библиотека CUSPARSE позволяет:
1) учитывать физическое время;
2) реализует основные операции линейной алгебры для разреженных векторов и матриц;
3) реализовать преобразование Фурье;
4) реализует основные операции линейной алгебры для векторов и матриц;
5) поддерживает вещественные и комплексные типы данных.

16. Библиотека CUFFT реализует:
1) преобразование Фурье;
2) операции над матрицами;
3) решение систем линейных уравнений;
4) решение систем дифференциальных уравнений.

17. Библиотека CURAND содержит:
1) функции для вычисления стандартных математических функций;
2) набор констант;
3) генераторы случайных чисел;
4) генерацию стандартных сообщений.

18. Библиотека Thrust поддерживает:
1) нечеткую логику;
2) обработку сообщений без временных меток;
3) параллельные алгоритмы обработки данных;
4) алгоритмы редукции;
5) оптимизацию потока сообщений.

19. При использовании нескольких GPU:
1) используется уровень пользовательского интерфейса;
2) нет механизма управления распределенным кластером;
3) создаются контексты устройств;
4) работают на уровне протоколов.

20. Микропроцессор Cell:

1) содержит 2 блок PPE и 16 блоков SPE;
2) содержит 1 блок PPE и 8 блоков SPE;
3) содержит 4 блок PPE и 32 блока SPE;
4) содержит 16 блоков SPE;

21. Микропроцессор Cell:
1) одно ядро имеет доступ к общей памяти;
2) все ядра имеют доступ к общей памяти;
3) каждое ядро SPE имеет свою локальную память;
4) каждое ядро SPE работает только с общей памятью.

22. Блок SPE:
1) содержит синергетическое процессорное устройство- SPU;
2) не содержит синергетическое процессорное устройство- SPU;
3) содержит несколько SPU;
4) представляет собой RISC-процессор.

23. Программирование Cell:
1) выполнено расширение языка Java;
2) выполнено расширение языка C++;
3) разработан собственный язык программирования;
4) использует удаленный вызов процедур (RPC).

24. Модель программирования Cell:
1) синхронизированная модель;
2) сетевая модель;
3) однопоточная модель;
4) многопоточная модель.

25. Программирование SPE:
1) обычная программа на языке C++;
2) обычная программа на языке Java;
3) программа на языке C++ в терминах векторов значений;
4) программа на языке Java в терминах векторов значений.

Ответы на тест по предмету «Программирование специализированных вычислительных устройств»:

1. – 1,4	2. – 2	3. – 3,4.	4. – 2	5. – 1,4
6. – 2,4	7. – 2,4	8. – 1,2	9. – 2	10. – 2,3.
11. – 2.	12. – 2.	13. – 1,4.	14. – 2	15. – 2,5
16. – 1	17. – 3.	18. – 3,4	19. – 2,3	20. – 2
21. – 2	22. - 1,4.	23. - 2.	24. – 3,4.	25. – 3.

5.9 Список вопросов к экзамену по дисциплине «**Программирование специализированных вычислительных устройств**»

1 CUDA-программная модель
2 Нити и блоки
3 Расширения языка для CUDA
4 Встроенные типы

5 Встроенные переменные
6 Встроенные функции
7 Асинхронное исполнение
8 Обработка ошибок
9 Доступ к свойствам установленных GPU
10 Атомарные операции
11 Иерархия памяти
12 Константная память
13 Глобальная память
14 Оптимизация работы с глобальной памятью
15 Текстурная память
16 Общее виртуальное адресное пространство
17 Обмен данными напрямую между GPU
18 Разделяемая память
19 Общее виртуальное пространство
20 Взаимодействие CUDA и языка высокого уровня
21 Некоторые алгоритмы обработки массивов
22 Архитектура GPU
23 Общие методы оптимизации CUDA- программ
24 Библиотека Thrust
25 Трансформация общего вида
26 Взаимодействие Thrust и CUDA C
27 Переключение целевой платформы Thrust
28 Библиотека PyCUDA
29 Прикладные библиотеки
30 Технологии для разработки на основе CUDA
31 Анализ работы приложений на GPU
32 Использование нескольких GPU
33 Примеры программирования на основе CUDA
34 Микропроцессоры Cell
35 Программирование МП CELL