МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
Юго-Западный государственный университет
(ЮЗГУ)
Кафедра вычислительной техники

УТВЕРЖДАЮ
Проректор по учебной работе
_____ О.Г.Локтионова
«21» 12 2023 г.

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Методические указания по выполнению лабораторных работ для студентов направления подготовки 09.04.01 Информатика и вычислительная техника

Курск 2023

УДК 621.3

Составитель Э.И. Ватутин

Рецензент
Кандидат технических наук, доцент Т.Н.Конаныхина

Параллельное программирование: Методические указания по выполнению лабораторных работ для студентов направления подготовки 09.04.01 Информатика и вычислительная техника / Юго-Зап. гос. ун-т; сост. Э.И. Ватутин; Курск, 2023.- 25 с.

Методические указания содержат основные теоретические положения, задания, контрольные вопросы для оценки полученных при выполнении лабораторных работ знаний, список литературы, необходимый для выполнения работ.

Методические указания соответствуют рабочей программе дисциплины «Параллельное программирование».

Методические указания предназначены для студентов направления подготовки 09.04.01 Информатика и вычислительная техника очной формы обучения.

Текст печатается в авторской редакции

Подписано в печать . Формат
Усл. печ. л. Уч. – изд.л. Тираж Заказ 353 Бесплатно.
Юго-Западный государственный университет.
305040, Курск, ул. 50 лет Октября, 94.

Содержание

Лабораторная работа №1

Введение в оптимизацию программных средств с использованием векторных расширений системы команд процессора

Цель работы

Изучение принципов организации и программирования векторных расширений системы команд современных процессоров семейства x86.

Основные теоретические сведения

Понятие и классификация векторных расширений системы команд

Векторные расширения системы команд процессоров семейства x86 позволяют получить выигрыш в скорости обработки однородных данных в соответствии с SIMD-принципом (Single Instruction Multiple Data). Расширения представляют собой группы ассемблерных команд, представленные в табл.1.

Таблица 1 - Векторные расширения системы команд

Название	Поддержка процессорами	Год выпуска	Примечание
MMX (Multi Media eXtensions)	с Intel Pentium MMX	1997 г.	–
3DNow!	с AMD K6-2	1998 г.	не поддерживается процессорами Intel
SSE (Streamed SIMD Extensions)	с Intel Pentium III	1999 г.	–
SSE2	с Intel Pentium IV, с AMD Athlon 64	2000 г.	
SSE3	с Intel Pentium IV (ядро Prescott), с AMD Athlon 64 (ядро Venice)	2004 г.	
SSSE3	с Intel Core, с AMD	2006 г.	

(Supplemental SSE3)	Bobcat		
SSE4 (4.1, 4.2, 4a)	с Intel Core (ядро Penryn)	2008 г.	частично поддерживается процессорами AMD
AVX (Advanced Vector eXtensions)	с Intel Core i7 (Sandy Bridge), с AMD Phenom (Bulldozer)	2011 г.	
AVX2	с Intel Core i7 (Haswell)	2013 г.	

В зависимости от типа расчетного кода (целочисленные вычисления, вычисления с плавающей точкой одинарной или двойной точности) для решения практических задач могут применяться команды различных векторных расширений.

Оператор Собела

Оператор Собела применяется для выделения границ (контрастных переходов) на изображениях (рис. 1).

В зависимости от типа расчетного кода (целочисленные вычисления, вычисления с плавающей точкой одинарной или двойной точности) для решения практических задач могут применяться команды различных векторных расширений.

Рисунок 1- Исходное изображение (слева) и результат применения к нему оператора Собела (справа)

Математически полутоновое изображение I представляет собой матрицу $I = \|I_{xy}\|_{n\times m}$, в которой каждый элемент I_{xy} является целым числом с

диапазоном значений $0; 255$. Для каждой точки изображения (за исключением крайних) производится операция свертки 8-связной окрестности точки $\begin{Vmatrix} I_{x-1,\,y-1} & I_{x,\,y-1} & I_{x+1,\,y-1} \\ I_{x-1,\,y} & I_{x,\,y} & I_{x+1,\,y} \\ I_{x-1,\,y+1} & I_{x,\,y+1} & I_{x+1,\,y+1} \end{Vmatrix} = \begin{Vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{Vmatrix}$ с заданной маской $M = \begin{Vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{Vmatrix}$:

$$d_{xy} = \sum_{i=1}^{3}\sum_{j=1}^{3} I_{x+i-2,\,y+j-2} m_{ij} =$$

$$= m_{11}A + m_{12}B + m_{13}C + m_{21}D + m_{22}E + m_{23}F + m_{31}G + m_{32}H + m_{33}I. \qquad (1)$$

При обработке изображения оператором Собела используются две маски:

$$M^{h} = \begin{Vmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{Vmatrix} \text{ и } M^{v} = \begin{Vmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{Vmatrix},$$

а получаемые с их применением значения сверток d_{xy}^{h} и d_{xy}^{v} объединяются по следующей формуле:

$$d_{xy} = \left\lfloor \frac{256}{1140} \sqrt{{d_{xy}^{h}}^{2} + {d_{xy}^{v}}^{2}} \right\rfloor,$$

где $\lfloor x \rfloor$ – операция округления вниз (усечения).

При реализации свертки с конкретной маской операции допускают упрощение по сравнению с (1):

$$t_1 = A - I,$$
$$t_2 = C - G,$$
$$H_h = 2(D - F) + t_1 - t_2,$$
$$H_v = 2(B - H) + t_1 + t_2,$$
$$d = \left\lfloor \frac{256}{1140} \sqrt{H_h^2 + H_v^2} \right\rfloor,$$

что уменьшает число выполняемых операций и экономит затраты вычислительного времени.

Вычисление значений t_1, t_2, H_h и H_v рациональнее производить как целочисленные операции, а вычисление значения d – как операции над

вещественными аргументами одинарной точности. Подобное разделение операций дает возможным использование следующих связок расширений в зависимости от используемого процессора:

1. MMX + FPU;
2. MMX + 3DNow;
3. MMX + SSE;
4. SSE + SSE2;
5. SSE2 + AVX;
6. AVX + AVX2.

С использованием команд первого из указанных расширений производятся целочисленные операции, второго – вещественные.

При векторной обработке выигрыш в скорости достигается за счет параллельной обработки нескольких точек изображения в зависимости от того, какое число операндов помещается в регистре (2, 4 или 8). На рис. 2 показан пример для параллельной векторной обработки 4 операндов:

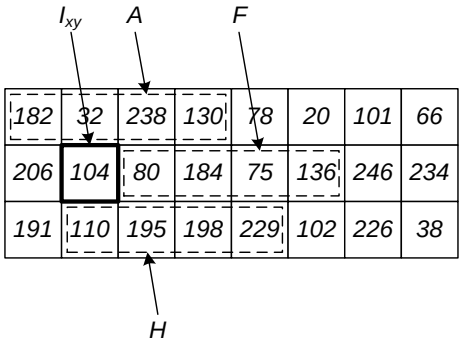

Рисунок 2 - Выборка операндов группами по 4 из памяти

Схема загрузки исходных данных из памяти представлена на рис. 3 (на примере MMX расширения).

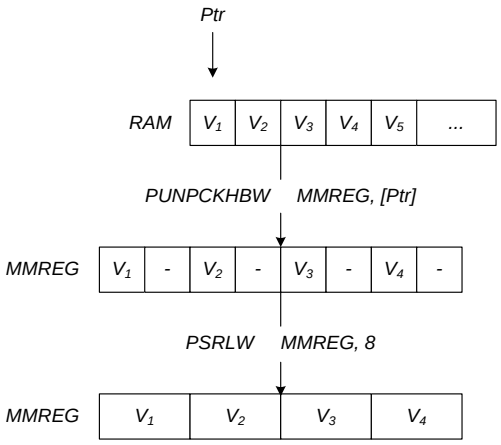

Рисунок 3 - Загрузка исходных данных из памяти

Ей соответствует следующий ассемблерный код:

```
punpckhbw mm0, [esi]
psrlw     mm0, 8
```

Схема вычисления значений сверток приведена на рис. 4 (на примере MMX расширения).

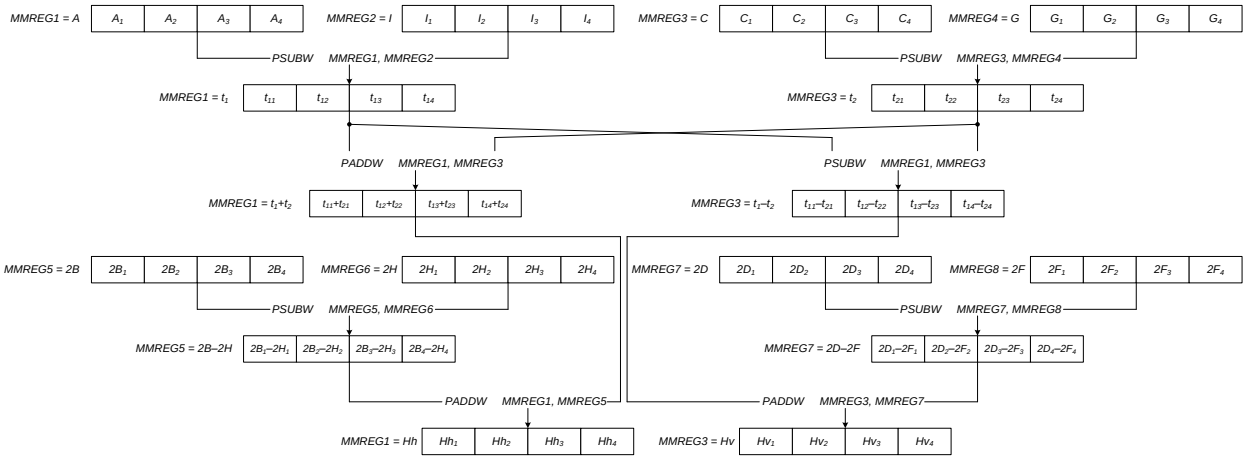

Рисунок 4 - Схема вычисления значений сверток

Схема преобразования целочисленных значений в вещественные одинарной точности приведена на рис. 5.

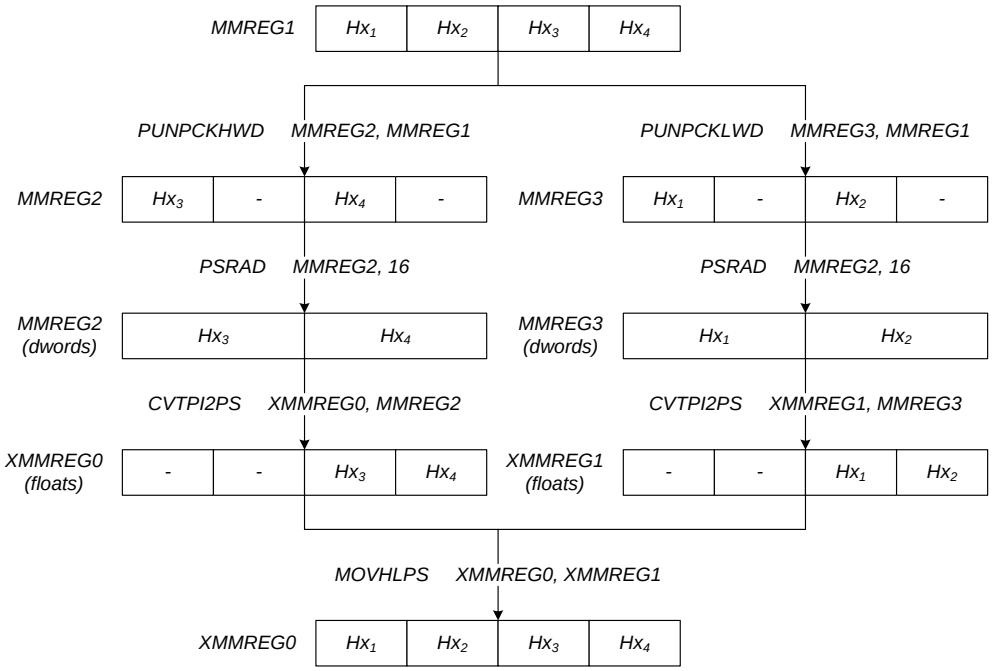

Рисунок 5 - Передача данных MMX→SSE

На рис. 6 показана схема вычисления искомого значения оператора Собела (на примере SSE расширения).

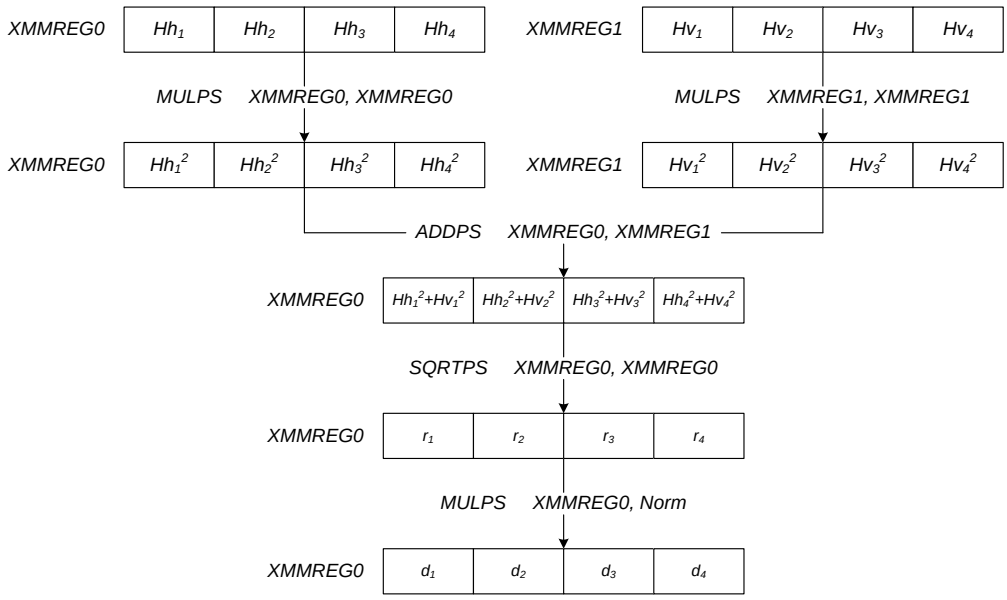

Рисунок 6 - Вычисление значения оператора Собела

Полученное значение является вещественным, перед записью в память оно должно быть преобразовано в целое. Схема преобразования приведена на рис. 7 (на примере связки SSE+MMX).

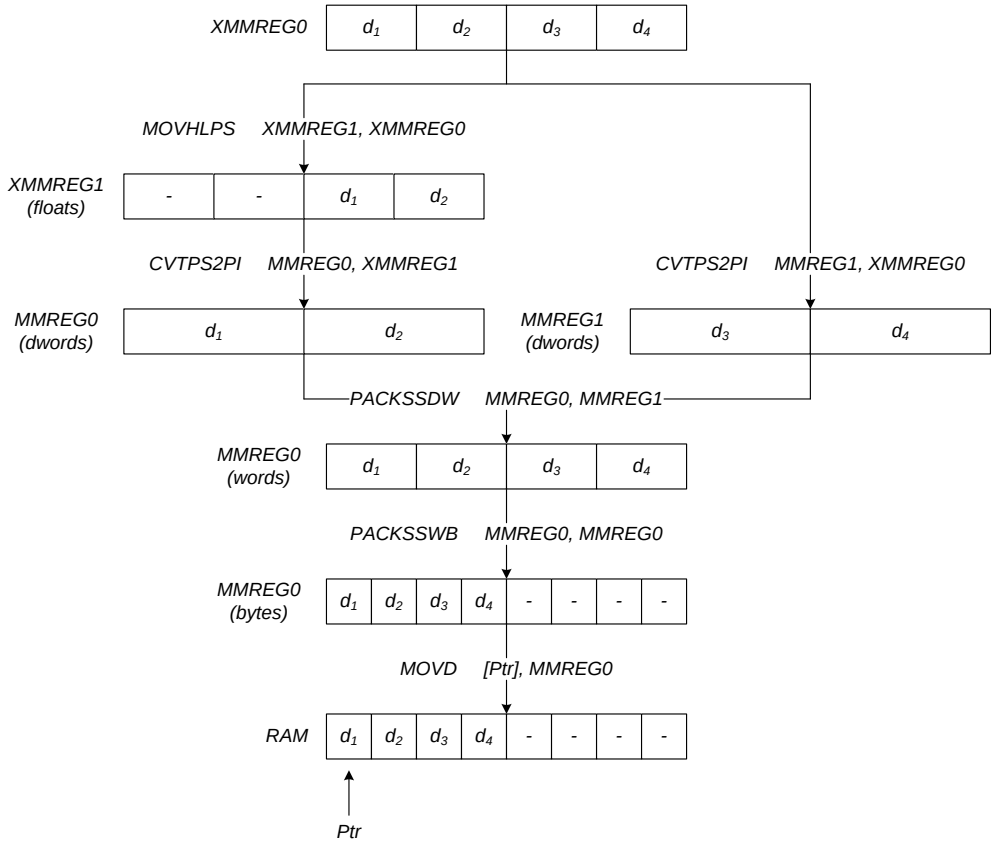

Рисунок 7 - Округление результата и запись в память

Задание

В соответствии с индивидуальным вариантом задания реализовать обработку изображения оператором Собела с использованием указанных преподавателем связок расширений. Убедиться в правильности выполняемых действий путем сопоставления результатов расчета высокоуровневой реализации с SIMD-оптимизированной версией кода. Протестировать разработанную программную реализацию на 3 различных процессорах для всех указанных связок расширений, определить необходимые затраты времени. Определить выигрыш во времени обработки SIMD-оптимизированных реализацией по сравнению с исходной высокоуровневой реализацией.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Описание условия решаемой задачи в соответствии с индивидуальным вариантом.
4. Листинг программы.
5. Результаты сопоставления оптимизированной и высокоуровневой версий кода.
6. Результаты измерения времени выполнения высокоуровневого и оптимизированного фрагмента программы, оценка достигнутого выигрыша во времени.
7. Выводы.

Контрольные вопросы

1. Что такое SIMD-принцип организации вычислительной системы?
2. Какие SIMD-расширения системы команд процессора существуют? В чем их ключевые отличия?
3. Какие регистры используются в составе SIMD-расширений системы команд процессора? Каковы их размер и логическая интерпретация?
4. За счет чего получается выигрыш во времени обработки при использовании SIMD-расширений?
5. Почему практический выигрыш во времени обработки ниже теоретического предела?

Библиографический список

1. Емельянов С.Г., Ватутин Э.И., Панищев В.С., Титов В.С. Процедурно-модульное программирование на Delphi: учебное пособие. М.: Аргамак-Медиа, 2014.- 352 с.
2. Зотов И.В., Ватутин Э.И., Борзов Д.Б. Процедурно-ориентированное программирование на С++: учебное пособие. Курск: КурскГТУ, 2008. - 211 с.

Лабораторная работа №2
Разработка программ с поддержкой технологии CUDA с использованием компилятора командной строки

Цель работы

Изучение особенностей разработки программ с использованием технологии CUDA.

Основные теоретические сведения

Для разработки программ с использованием технологии CUDA необходима установка инструментария из CUDA SDK, CUDA Driver и CUDA Toolkit, в состав которого входит компилятор командной строки `nvcc`. Инструментарий свободно доступен по адресу https://developer.nvidia.com/cuda-downloads/. Программа с поддержкой технологии CUDA разбивается на `.cu`-файлы, компилируемые с использованием `nvcc`, и обычные файлы программы (в данной работе `.cpp`), компилируемые и линкуемые обычным компилятором (в данной работе `cl.exe`, входящим в состав Microsoft Visual Studio).

В состав `.cu`-файлов входят специфичные для технологии CUDA элементы: функции CUDA-ядер (англ. kernel), выполняемые потоковыми мультипроцессорами (англ. Streaming Multiprocessor, SM) видеокарты, и функции, осуществляющие вызов CUDA-ядер. Пример CUDA-ядра, осуществляющего поэлементное умножение векторов:

```
__global__ void VecMulKernel(float *a, float *b, float *c)
{
    // Определение индекса потока
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    // Обработка соответствующей порции данных
    c[i] = a[i] * b[i];
}
```

Пример вызова CUDA-ядра:

```
VecMulKernel<<<blocks, threads>>>(a, b, c);
```

При компиляции `.cu`-файлов может потребоваться подключение ряда заголовочных файлов:

```
#include <cuda.h>
#include <cuda_runtime.h>
//...
```

располагающихся в папке

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\
```

(для Windows XP x86).

`.cpp`-файлы содержат весь остальной код (например, функцию `main()`), включающий вызовы функций из `.cu`-файлов.

Компиляция простейшего проекта с использованием компилятора `nvcc` производится из командной строки следующим образом:

```
nvcc file1.cu file2.cu … fileN.cu file1.cpp file2.cpp … fileN.cpp -o program_name.exe
```

причем для корректной работы пути к файлам `nvcc.exe` и `cl.exe` должны содержаться в переменной окружения `PATH` (при необходимости их нужно добавить в список вручную). Если компиляция завершена успешно, то в результате будет сформирован исполняемый файл с указанным именем (`program_name.exe` в данном примере).

Общая стратегия обработки информации на видеокарте сводится к трем действиям:

1. Передача исходных данных из оперативной памяти в память видеокарты.
2. Запуск ядра.
3. Передача результирующих данных из памяти видеокарты и оперативную память.

Управление динамической памятью на видеокарте происходит с использованием функций `cudaMalloc()` и `cudaFree()`, функциональность которых аналогична соответствующим функциям стандартной библиотеки для работы с динамической памятью.

Для копирования данных между оперативной памятью и памятью видеокарты применяется функция `cudaMemcpy(pDst, pSrc, SizeInBytes, FromTo)`, в параметрах которой передаются адреса областей памяти источника и приемника, размер копируемой области данных в байтах и константа, определяющая направление копирования:

```
cudaMemcpyHostToHost     // RAM (Host) := RAM (Host)

cudaMemcpyHostToDevice   // GPU (Device) := RAM (Host)

cudaMemcpyDeviceToHost   // RAM (Host) := GPU (Device)

cudaMemcpyDeviceToDevice // GPU (Device) := GPU (Device)
```

При работе с динамической памятью необходимо помнить о том, что указатели на область памяти видеокарты не действительны в основной программе, и, наоборот, указатели на область оперативной памяти не действительны в коде CUDA-ядра.

При вызове функции-ядра в параметрах `blocks` и `threads` указывается конфигурация запуска ядра (число потоков в блоке и число блоков в сетке). Например:

```
// Конфигурация запуска ядра
dim3 threads = dim3(512, 1);        // 512 потоков в блоке
dim3 blocks = dim3(n/threads.x, 1); // n/512 блоков в сетке

// Вызов ядра
MulKernel<<<blocks, threads>>>(a, b, c);
```

В функции-ядре при помощи обращения к предопределенным переменным `threadIdx` и `blockIdx` можно определить «координаты» потока в блоке и блоке в сетке.

Задание

Создать .cu-файл, поместив в него ссылки на заголовочные файлы `cuda.h` и `cuda_runtime.h`, код ядра для поэлементного умножения векторов (приведен выше) и следующий код с вызовом ядра:

```
// a, b - указатели на исходные массивы

// c - указатель на результирующий массив
// n - размер массивов (число элементов)
void vec_mul_cuda(float *a, float *b, float *c, int n)
{
    int SizeInBytes = n * sizeof(float);

    // Указатели на массивы в видеопамяти
    float *a_gpu = NULL;
    float *b_gpu = NULL;
    float *c_gpu = NULL;

    // Выделение памяти под массивы на GPU
    cudaMalloc( (void **)&a_gpu, SizeInBytes );
    cudaMalloc( (void **)&b_gpu, SizeInBytes );
    cudaMalloc( (void **)&c_gpu, SizeInBytes );

    // Копирование исходных данных из CPU на GPU
    cudaMemcpy(a_gpu, a, SizeInBytes, cudaMemcpyHostToDevice); // a_gpu = a
    cudaMemcpy(b_gpu, b, SizeInBytes, cudaMemcpyHostToDevice); // b_gpu = b

    // Задание конфигурации запуска ядра
    dim3 threads = dim3(512, 1);        // 512 потоков в блоке
    dim3 blocks = dim3(n/threads.x, 1); // n/512 блоков в сетке

    // Запуск ядра (покомпонентное умножение векторов c = a * b)
    VecMulKernel<<<blocks, threads>>>(a_gpu, b_gpu, c_gpu);

    // Копирование результата из GPU в CPU
    cudaMemcpy(c, c_gpu, SizeInBytes, cudaMemcpyDeviceToHost); // c = c_gpu

    // Освобождение памяти GPU
    cudaFree(a_gpu);
    cudaFree(b_gpu);
    cudaFree(c_gpu);
}
```

Создать `.cpp`-файл, поместив в него код для вызова разработанных подпрограмм:

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>

using namespace std;

// Описание внешней функции, располагающейся в .cu-файле
void vec_mul_cuda(float *a, float *b, float *c, int n);

// Размер вектора (должен быть кратен 512)
const int N = 1024;

// Вектора
float a[N], b[N], c[N];

void main()
{
    // Заполнение векторов исходными данными
    for (int i=0; i<N; i++)
    {
        a[i] = i;
        b[i] = i;
        c[i] = 0;
    }

    // Покомпонентное умножение векторов на GPU
    vec_mul_cuda(a, b, c, 1024);

    // Вывод первых 20 значений результата
    for (int i=0; i<20; i++)
      cout << c[i] << " ";

    getchar();
    return;
}
```

Откомпилировать проект с использованием компилятора командной строки:

```
nvcc имя_cu_файла имя_cpp_файла -o main.exe
```

Убедиться в работоспособности разработанной программы:

Модифицировать программу следующим образом:

- разработать CUDA-ядро для поэлементного сложения векторов;
- сформировать вектора из случайных исходных данных;
- найти сумму векторов с использованием разработанного CUDA-ядра и с использованием подпрограммы на CPU, сравнить полученные результаты, сделать вывод о корректности расчетов на GPU.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Задание.
4. Листинг программы.

5. Скриншот с результатами работы программы.
6. Выводы.

Контрольные вопросы

1. Для чего предназначен инструментарий CUDA?
2. Как производится программирование NVidia GPU с использованием CUDA?
3. Как производится сборка программы для ее запуска на NVidia GPU?
4. Что такое CUDA-ядро (CUDA kernel) и его конфигурация запуска?

Библиографический список

1. Емельянов С.Г., Ватутин Э.И., Панищев В.С., Титов В.С. Процедурно-модульное программирование на Delphi: учебное пособие. М.: Аргамак-Медиа, 2014.- 352 с.
2. Зотов И.В., Ватутин Э.И., Борзов Д.Б. Процедурно-ориентированное программирование на С++: учебное пособие. Курск: КурскГТУ, 2008. - 211 с.

Лабораторная работа №3
Определение параметров видеокарты с поддержкой технологии CUDA в среде Microsoft Visual Studio

Цель работы

Изучение особенностей интеграции инструментария CUDA в среде разработки Microsoft Visual Studio, определение основных параметров видеокарт, влияющих на скорость вычислений.

Основные теоретические положения

Настройка среды Microsoft Visual Studio

Инструментарий CUDA предоставляет возможность интеграции в состав популярной среды разработки Microsoft Visual Studio, что избавляет разработчика от необходимости напрямую обращаться к компилятору `nvcc` с использованием командной строки или разрабатывать `make`-файлы. Для создания проекта с поддержкой CUDA необходимо (подразумевается, что инструментарий CUDA установлен на машине):

1. Создать новый проект.
2. Добавить к нему необходимые `.cu`-файлы.
3. Для каждого из них указать инструмент, с помощью которого производится компиляция (правый клик по файлу в дереве проекта → Properties → General → Tool → в выпадающем списке выбрать «CUDA Runtime API») (рис.1).
4. Указать путь к необходимым статически подключаемым библиотекам (выбрать Главное меню → Project → Properties → Configuration properties → Linker → Input → Additional dependencies, указать значение

```
"C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v5.0\lib\Win32\cudart.lib"
```

(в кавычках, путь показан на примере Windows XP x86!)

Другой вариант – указать в коде программы ссылку на библиотеку:

```
#pragma comment(lib,"cudart.lib")
```

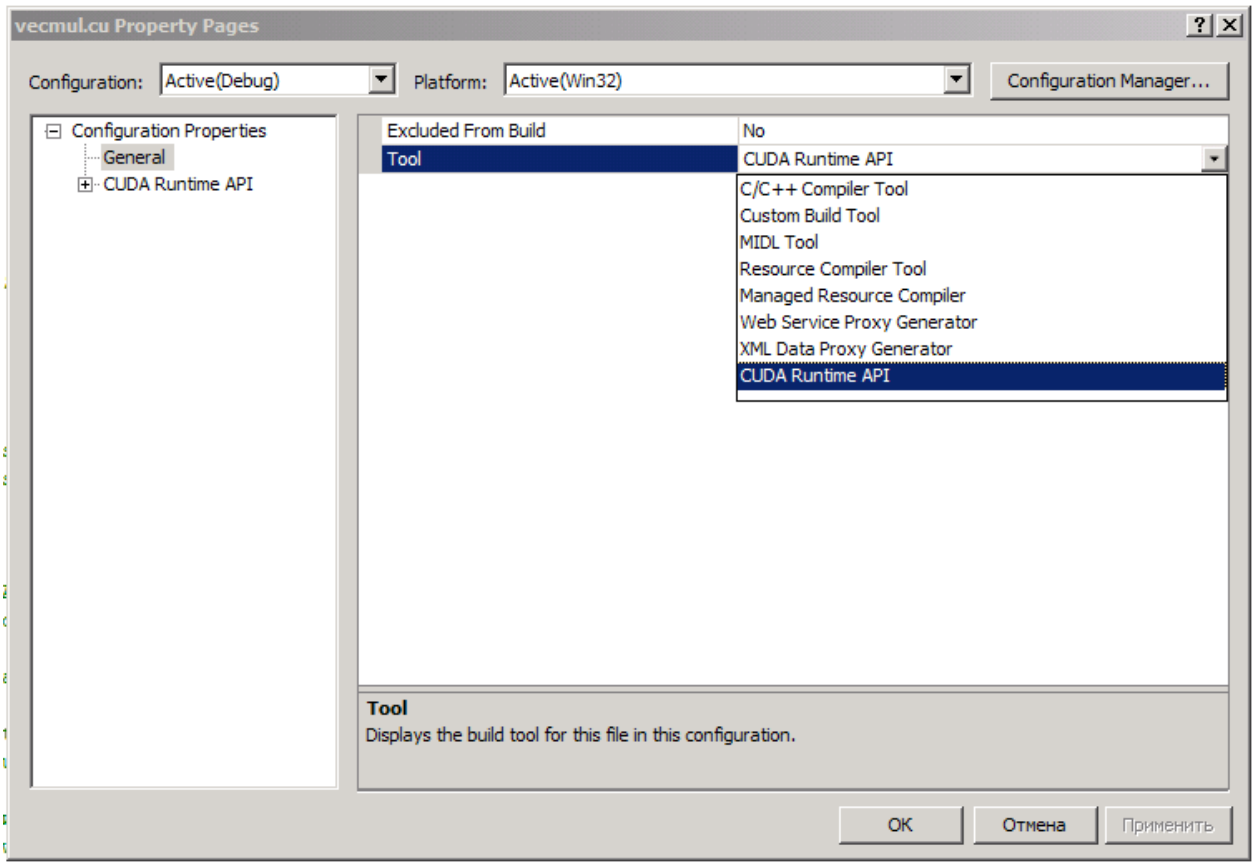

Рисунок 1 - Выбор типа компиляции для `.cu`-файлов

5. Указать путь к расположению инструментария CUDA (выбрать Главное меню → Project → Properties → Configuration properties → Linker → Additional library directories, указать значение `$(CUDA_LIB_PATH)`).
6. Указать путь к расположению заголовочных файлов CUDA (выбрать Главное меню → Project → Properties → Configuration properties → C/C++ → Additional include directories, указать значение

   `"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\"`

   (в кавычках, путь показан на примере Windows XP x86!)

После выполнения указанных действий среда разработки Microsoft Visual Studio будет самостоятельно вызывать компилятор `nvcc` для `.cu`-файлов, осуществлять компиляцию и сборку проекта.

Определение параметров видеокарты

Для определения числа установленных в системе видеокарт с поддержкой технологии CUDA используется функция `cudaGetDeviceCount()`:

```
int device_count;

cudaGetDeviceCount(&device_count);
```

Для определения свойств видеокарты используется функция `cudaGetDeviceProperties()`, принимающая в качестве параметра номер видеокарты и возвращающая в структуре `cudaDeviceProp` интересующие значения:

```
cudaDeviceProp dp;
cudaGetDeviceProperties(&dp, 0);  // Определение параметров GPU с номером 0
```

Например, для определения наименований установленных в системе видеокарт с поддержкой CUDA и их вычислительных возможностей можно использовать следующий код:

```
int device_count;
cudaDeviceProp dp;

cudaGetDeviceCount(&device_count);
cout << "CUDA device count: " << device_count << "\n";

for (int i=0; i<device_count; i++)
{
      cudaGetDeviceProperties(&dp, i);

      cout << i << ": " << dp.name << " with CUDA compute compatibility " <<
           dp.major << "." << dp.minor << "\n";
}
```

Задание

1. Создать в среде Microsoft Visual Studio проект, состоящий из пары файлов (`.cpp` и `.cu`) из предыдущей работы. Установить в среде необходимые настройки. Убедиться в том, что проект успешно собирается и запускается.
2. Определить число видеокарт с поддержкой технологии CUDA и следующие параметры видеокарты:
   - наименование;
   - общий объем графической памяти;
   - объем памяти констант;
   - объем разделяемой памяти в пределах блока;

- число регистров в пределах блока;
- размер WARP’а;
- максимально допустимое число потоков в блоке;
- версию вычислительных возможностей;
- число потоковых мультипроцессоров;
- тактовую частоту ядра;
- частоту памяти видеокарты;
- объем кэша второго уровня;
- ширину шины памяти видеокарты;
- максимальную размерность при конфигурации потоков в блоке и блоков в сетке.

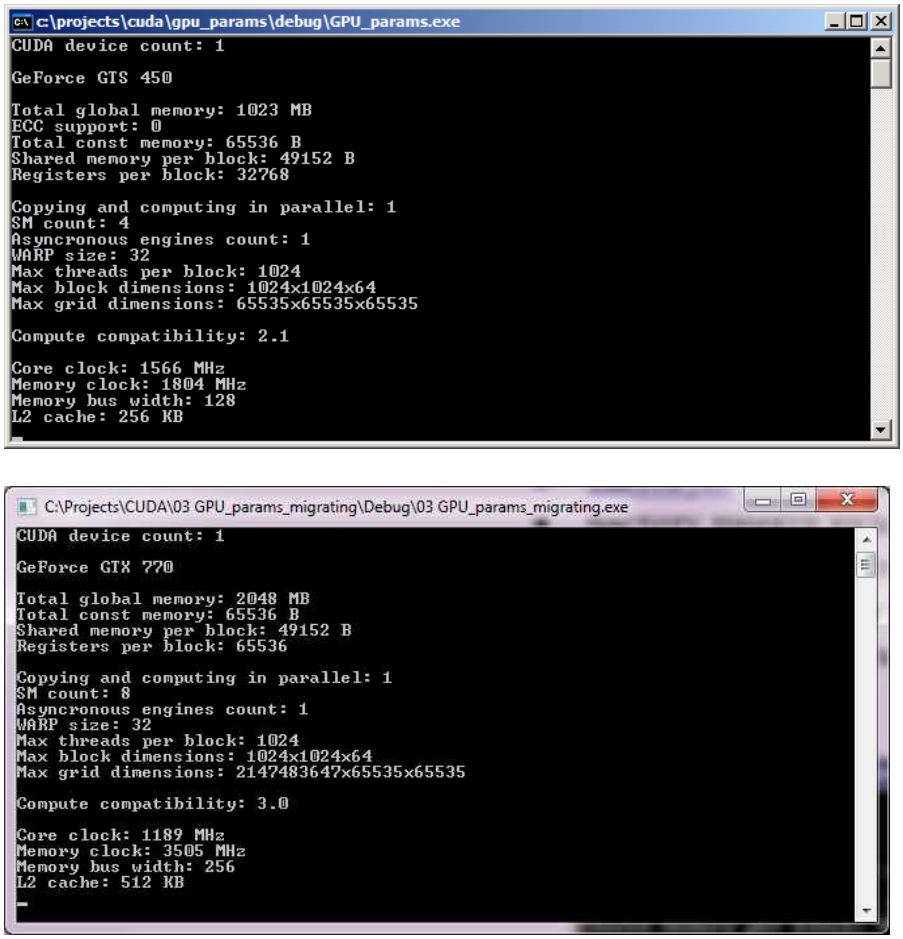

Рисунок 2 - Основные параметры видеокарты, влияющие на производительность вычислений

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Задание.

4. Листинг программы.
5. Скриншоты с результатами работы программы.
6. Выводы.

Контрольные вопросы

1. Для чего предназначен инструментарий CUDA?
2. Как производится программирование NVidia GPU с использованием CUDA?
3. Как производится сборка программы для ее запуска на NVidia GPU?
4. Как инструментарий CUDA интегрируется в состав среды Microsoft Visual Studio?

Библиографический список

1. Емельянов С.Г., Ватутин Э.И., Панищев В.С., Титов В.С. Процедурно-модульное программирование на Delphi: учебное пособие. М.: Аргамак-Медиа, 2014. - 352 с.
2. Зотов И.В., Ватутин Э.И., Борзов Д.Б. Процедурно-ориентированное программирование на C++: учебное пособие. Курск: КурскГТУ, 2008. - 211 с.

Лабораторная работа №4

Измерение пропускной способности памяти видеокарт с поддержкой технологии CUDA

Цель работы

Научиться измерять пропускную способность памяти GPU при различных типах и направлениях передачи данных.

Основные теоретические положения

Работа с памятью является одним из краеугольных камней при разработке эффективных программ с использованием технологии CUDA. Ее правильное использование зачастую позволяет увеличить скорость обработки данных до 10 и более раз.

Видеокарта не имеет доступа к оперативной памяти компьютера (host), поэтому все обрабатываемые данные должны быть загружены в динамическую память GPU (device). В данной работе используется глобальная память, для обмена данными с ней используется функция

```
cudaError_t cudaMemcpy(void *dst, void *src, size_t count,
    cudaMemcpyKind kind);
```

которая осуществляет копирование `count` байт из источника `src` в приемник `dst`. Направление копирования задается параметром `kind` и может принимать следующие значения:

```
cudaMemcpyHostToHost     /* Host   -> Host */
cudaMemcpyHostToDevice   /* Host   -> Device */
cudaMemcpyDeviceToHost   /* Device -> Host */
cudaMemcpyDeviceToDevice /* Device -> Device */
```

В первом случае копирование производится в пределах оперативной памяти (функция работает как `memcpy()`), во втором – из оперативной памяти в глобальную память GPU, в третьем – из глобальной памяти GPU в оперативную память, и в четвертом – в пределах глобальной памяти GPU.

Для выделения области памяти в глобальной памяти GPU используется функция

```
cudaError_t cudaMalloc(void **devPtr, size_t size);
```

возвращающая в первом параметре указатель на выделенную область памяти размером `size` байт. Для выделения области динамической памяти в оперативной памяти можно использовать функции

```
void * malloc(size_t size);
```

и

```
cudaError_t cudaMallocHost(void **ptr, size_t size);
```

Функция `malloc()` входит в стандартную библиотеку, она выделяет блок динамической памяти размера `size` и возвращает указатель на его начало. Функция `cudaMallocHost()` отличается от нее тем, что помечает выделяемые страницы виртуальной памяти так, что они гарантированно находятся в оперативной памяти и не могут быть вытеснены из нее в файл подкачки (т.н. page locking). Это упрощает процедуру наблюдения за данной памятью со стороны драйвера видеокарты и увеличивает скорость обмена. Однако пользоваться данной функцией необходимо с осторожностью, т.к. при выделении большого объема данных производительность системы в целом может существенно ухудшиться, могут появиться системные ошибки и т.п.

Копирование в рамках глобальной памяти GPU осуществляется асинхронно: функция `cudaMemcpy()` возвращает управление раньше, чем завершается копирование. Чтобы учесть это, можно добавить в код вызов функции

```
cudaError_t cudaThreadSynchronize();
```

которая ожидает завершения текущей асинхронной операции и лишь затем возвращает управление. Фактически, таким образом происходит синхронизация потока CPU с GPU.

Задание

1. Выделить блоки памяти одинакового размера в оперативной памяти и глобальной памяти видеокарты. Скопировать содержимое блоков между:
   - двумя буферами в оперативной памяти;
   - между оперативной памятью и глобальной памятью видеокарты в направлении CPU→GPU и GPU→CPU с использованием обычной и page-locked памяти (должно быть 4 различных варианта копирования);

- между двумя буферами в глобальной памяти видеокарты.

2.Убедиться в том, что копирование происходит корректно.

3. Измерить время копирования и, зная размер копируемого блока, определить пропускную способность (в ГБ/с) для каждого типа копирования. Сделать выводы о скорости копирования в различных режимах.
4. В отчет включить краткое описание типа и параметров видеокарты и процессора.

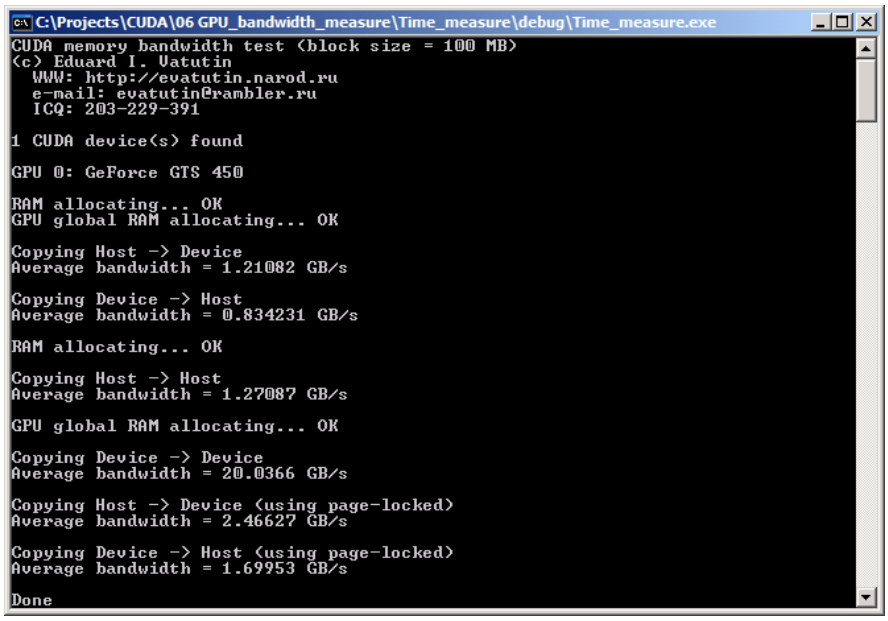

Рисунок 1- Результаты измерения пропускной способности

Содержание отчета

1. Титульный лист
2. Цель работы
3. Задание
4. Листинг программы
5. Результаты измерения пропускной способности
6. Выводы

Контрольные вопросы

1. Для чего предназначен инструментарий CUDA?
2. Как производится программирование NVidia GPU с использованием CUDA?

3. Как производится обмен данными между процессором, оперативной памятью и видеокартой?

4. Какая подсистема в составе компьютера лимитирует скорость обмена данными между оперативной памятью и глобальной памятью видеокарты?

Библиографический список

1. Емельянов С.Г., Ватутин Э.И., Панищев В.С., Титов В.С. Процедурно-модульное программирование на Delphi: учебное пособие. М.: Аргамак-Медиа, 2014. - 352 с.
2. Зотов И.В., Ватутин Э.И., Борзов Д.Б. Процедурно-ориентированное программирование на C++: учебное пособие. Курск: КурскГТУ, 2008. - 211 с.