Московский государственный университет

приборостроения и информатики

Кафедра ИТ-6 «Управление и моделирование систем»

Специальность 230105

«Программное обеспечение вычислительной техники и автоматизированных систем»

Дисциплина «Операционные системы»

Отчет по курсовой работе

Подготовил: студент 3 курса

Кулыгин А.А.

Проверил: Никифоров А.А.

Москва 2011

Оглавление

[Умножение матриц 3](#_Toc310083121)

[Обычное умножение VS параллельное умножение 4](#_Toc310083122)

[Оборудование 7](#_Toc310083123)

[OpenMP 10](#_Toc310083124)

[CUDA 12](#_Toc310083125)

[OpenCL 16](#_Toc310083126)

[Исходные коды программ 18](#_Toc310083127)

[CPU-версия 18](#_Toc310083128)

[OpenMP-версия 19](#_Toc310083129)

[Cuda-версия 20](#_Toc310083130)

[OpenCL-версия 22](#_Toc310083131)

[Device code. 22](#_Toc310083132)

[Host code. 23](#_Toc310083133)

[Export C interface 31](#_Toc310083134)

[Size setup 31](#_Toc310083135)

[Информационные источники 32](#_Toc310083136)

# Умножение матриц

**Умноже́ние ма́триц** — одна из основных операций над матрицами. Матрица, получаемая в результате операции умножения, называется **произведе́нием ма́триц**.

Пусть даны две прямоугольные матрицы *A* и *B* размерности mxn и nxqсоответственно:


A = 
  \begin{bmatrix} 
    a_{11} & a_{12} & \cdots & a_{1n} \\
    a_{21} & a_{22} & \cdots & a_{2n} \\ 
    \vdots & \vdots & \ddots & \vdots \\ 
    a_{m1} & a_{m2} & \cdots & a_{mn}
  \end{bmatrix},\;\;\;
B =   
  \begin{bmatrix} 
    b_{11} & b_{12} & \cdots & b_{1q} \\
    b_{21} & b_{22} & \cdots & b_{2q} \\ 
    \vdots & \vdots & \ddots & \vdots \\ 
    b_{n1} & b_{n2} & \cdots & b_{nq}
  \end{bmatrix}.


Тогда матрица *C* размерностью mxq называется их *произведением*:

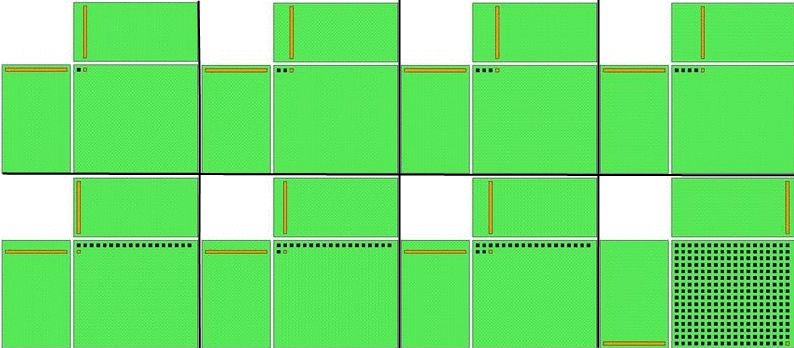

C = 
  \begin{bmatrix} 
    c_{11} & c_{12} & \cdots & c_{1q} \\
    c_{21} & c_{22} & \cdots & c_{2q} \\ 
    \vdots & \vdots & \ddots & \vdots \\ 
    c_{m1} & c_{m2} & \cdots & c_{mq}
  \end{bmatrix},


где:

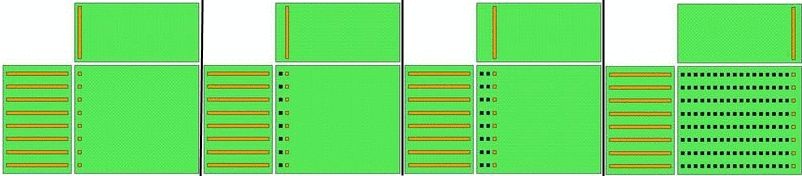
 c_{i,j} = \sum_{r=1}^n a_{i,r}b_{r,j} \;\;\; \left(i=1, 2, \ldots m;\;j=1, 2, \ldots q \right).

# Обычное умножение VS параллельное умножение

Обычное



Параллельное



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Размерность** | **Время перемножения в миллисекундах** | | | | | | |
|  | **CPU** | | **OpenMP** | | **OpenCL** | **CUDA** | |
|  | *Нетбук\** | *ПК\** | *Нетбук* | *ПК* | *Нетбук* | *Нетбук* | *ПК* |
| **800x800** | 20327 | 2137 | 10161,2 | 569,13 | 296,85 | 526 | 47 |
| **1000x1000** | 37768 | 5974 | 14141,5 | 1664,98 | 590,28 | 979 | 86 |
| **2000x2000** | 559607 | 60232 | 386756 | 16150,3 | 4557,24 | FAIL | 640 |
| **4000x4000** | -- | 626625 | -- | -- | FAIL | FAIL | 4710 |

\*характеристики пк и нетбука после схем

*Время (миллисекунды) перемножения матриц 800х800*

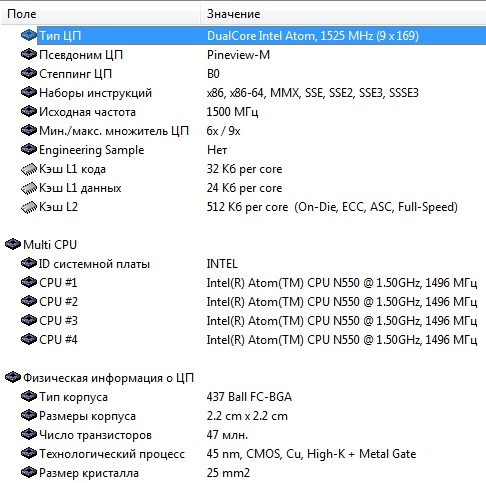
*Время (миллисекунды) перемножения матриц 1000х1000*

*Время (миллисекунды) перемножения матриц 2000х2000*

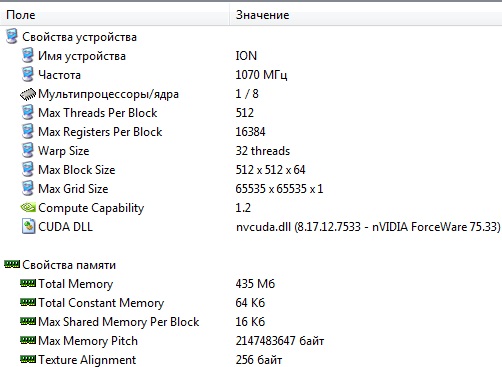
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Размерность** | **Выигрыш по сравнению с CPU** | | | | | |
|  | **OpenMP** | | | **OpenCL** | **CUDA** | |
|  | *Нетбук* | | *ПК* | *Нетбук* | *Нетбук* | *ПК* |
| **800\*800** | 2,000453 | 3,754854 | | 68,47566 | 38,64449 | 45,46809 |
| **1000\*1000** | 2,670721 | 3,588031 | | 63,98319 | 38,57814 | 69,46512 |
| **2000\*2000** | 1,446925 | 3,729466 | | 122,7952 |  | 94,1125 |
| **4000\*4000** |  |  | |  |  | 133,0414 |

# Оборудование

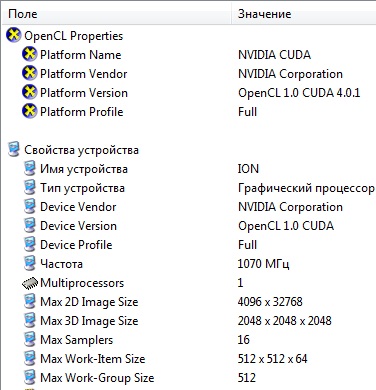
Нетбук CPU



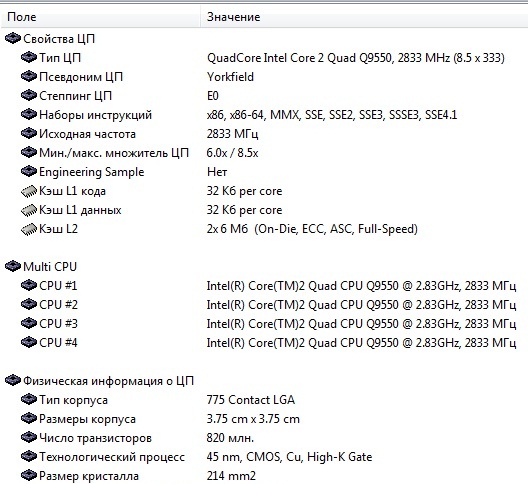
Нетбук CUDA



Нетбук OpenCL



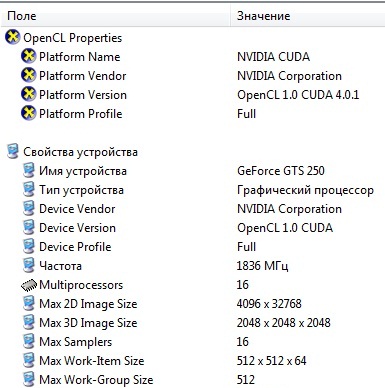
ПК CPU



ПК CUDA



ПК OpenCL



# OpenMP

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых [SMP-системах](http://parallel.ru/parallel/computers/classes.html#smp) (SSMP,ccNUMA, etc.) в модели общей памяти (shared memory model). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды.

Разработкой стандарта занимается организация OpenMP ARB (ARchitecture Board), в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения. Спецификации для языков Fortran и C/C++ появились соответственно в октябре 1997 года и октябре 1998 года. Открыт список рассылки для публичного обсуждения OpenMP ([omp@openmp.org](mailto:omp@openmp.org)).

До появления OpenMP не было подходящего стандарта для эффективного программирования на SMP-системах.

Наиболее гибким, переносимым и общепринятым интерфейсом параллельного программирования является [MPI](http://parallel.ru/tech/tech_dev/mpi.html) (интерфейс передачи сообщений). Однако модель передачи сообщений 1) недостаточно эффективна на SMP-системах; 2) относительно сложна в освоении, так как требует мышления в "невычислительных" терминах.

Проект стандарта **X3H5** провалился, так как был предложен во время всеобщего интереса к MPP-системам, а также из-за того, что в нем поддерживается только параллелизм на уровне циклов. OpenMP развивает многие идеи X3H5.

POSIX-интерфейс для организации нитей (**Pthreads**) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического параллельного программирования:

1. нет поддержки Fortran-а,
2. слишком низкий уровень,
3. нет поддержки параллелизма по данным,
4. механизм нитей изначально разрабатывался не для целей организации параллелизма.

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей).

Многие поставщики SMP-архитектур (Sun,HP,SGI) в своих компиляторах поддерживают **спецдирективы для распараллеливания** циклов. Однако эти наборы директив, как правило, 1) весьма ограничены; 2) несовместимы между собой; в результате чего разработчикам приходится распараллеливать приложение отдельно для каждой платформы. OpenMP является во многом обобщением и расширением упомянутых наборов директив.

**Преимущества OpenMP**

1. За счет идеи **"инкрементального распараллеливания"** OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы.

2. При этом, OpenMP - достаточно **гибкий механизм**, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения.

3. Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована **в качестве последовательной** программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки (stubs), текст которых приведен в спецификациях.

4. Одним из достоинств OpenMP его разработчики считают поддержку так называемых **"orphan" (оторванных) директив**, то есть директивы синхронизации и распределения работы могут не входить непосредственно в лексический контекст параллельной области.

**Ключевыми элементами** OpenMP являются

* конструкции для создания потоков (директива parallel),
* конструкции распределения работы между потоками (директивы DO/for и section),
* конструкции для управления работой с данными (выражения shared и private),
* конструкции для синхронизации потоков (директивы critical, atomic и barrier),
* процедуры библиотеки поддержки времени выполнения (например, omp\_get\_thread\_num),
* переменные окружения (например, OMP\_NUM\_THREADS).

# CUDA

**CUDA** (англ. *Compute Unified Device Architecture*) — программно-аппаратная архитектура, позволяющая производить вычисления с использованием графических процессоров NVIDIA, поддерживающих технологию GPGPU (произвольных вычислений на видеокартах)

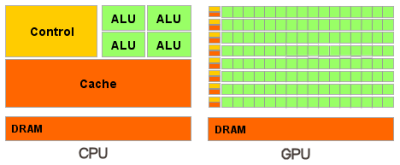
Преимущества CUDA перед традиционным подходом к GPGPU вычислениям:

* интерфейс программирования приложений CUDA основан на стандартном языке программирования Си с расширениями, что упрощает процесс изучения и внедрения архитектуры CUDA;
* CUDA обеспечивает доступ к разделяемой между потоками памяти размером в 16 Кб на мультипроцессор, которая может быть использована для организации кэша с широкой полосой пропускания, по сравнению с текстурными выборками;
* более эффективная передача данных между системной и видеопамятью
* отсутствие необходимости в графических API с избыточностью и накладными расходами;
* линейная адресация памяти, и gather и scatter, возможность записи по произвольным адресам;
* аппаратная поддержка целочисленных и битовых операций.

Основные ограничения CUDA:

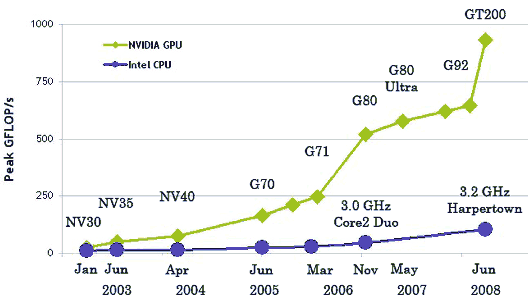
* отсутствие поддержки рекурсии для выполняемых функций;
* минимальная ширина блока в 32 потока;
* закрытая архитектура CUDA, принадлежащая NVIDIA.

Вкратце можно сказать, что в отличие от современных универсальных CPU, видеочипы предназначены для параллельных вычислений с большим количеством арифметических операций. И значительно большее число транзисторов GPU работает по прямому назначению — обработке массивов данных, а не управляет исполнением (flow control) немногочисленных последовательных вычислительных потоков. Это схема того, сколько места в CPU и GPU занимает разнообразная логика:

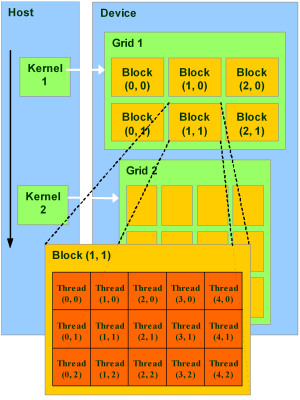
* 

В итоге, основой для эффективного использования мощи GPU в научных и иных неграфических расчётах является распараллеливание алгоритмов на сотни исполнительных блоков, имеющихся в видеочипах

В результате всех описанных выше отличий, теоретическая производительность видеочипов значительно превосходит производительность CPU. Компания NVIDIA приводит такой график роста производительности CPU и GPU за последние несколько лет:



Модель программирования в CUDA предполагает группирование потоков. Потоки объединяются в блоки потоков (thread block) — одномерные или двумерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (ядро, kernel) исполняется над сеткой (grid) блоков потоков (thread blocks), см. рисунок ниже. Одновременно исполняется одна сетка. Каждый блок может быть одно-, двух- или трехмерным по форме, и может состоять из 512 потоков на текущем аппаратном обеспечении.

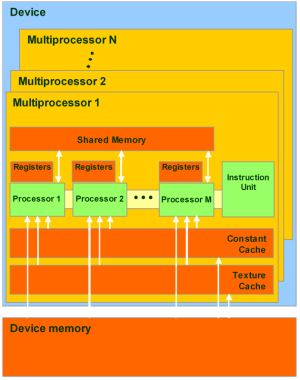


Блоки потоков выполняются в виде небольших групп, называемых варп (warp), размер которых — 32 потока. Это минимальный объём данных, которые могут обрабатываться в мультипроцессорах. И так как это не всегда удобно, CUDA позволяет работать и с блоками, содержащими от 64 до 512 потоков.

Группировка блоков в сетки позволяет уйти от ограничений и применить ядро к большему числу потоков за один вызов. Это помогает и при масштабировании. Если у GPU недостаточно ресурсов, он будет выполнять блоки последовательно. В обратном случае, блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеочипах разного уровня, начиная от мобильных и интегрированных.

Модель памяти в CUDA отличается возможностью побайтной адресации, поддержкой как gather, так и scatter. Доступно довольно большое количество регистров на каждый потоковый процессор, до 1024 штук. Доступ к ним очень быстрый, хранить в них можно 32-битные целые или числа с плавающей точкой.

Каждый поток имеет доступ к следующим типам памяти:



**Глобальная память** — самый большой объём памяти, доступный для всех мультипроцессоров на видеочипе, размер составляет от 256 мегабайт до 1.5 гигабайт на текущих решениях (и до 4 Гбайт на Tesla). Обладает высокой пропускной способностью, более 100 гигабайт/с для топовых решений NVIDIA, но очень большими задержками в несколько сот тактов. Не кэшируется, поддерживает обобщённые инструкции load и store, и обычные указатели на память.

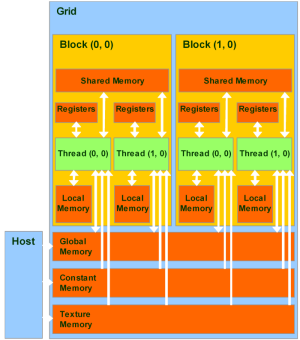
**Локальная память** — это небольшой объём памяти, к которому имеет доступ только один потоковый процессор. Она относительно медленная — такая же, как и глобальная.

**Разделяемая память** — это 16-килобайтный (в видеочипах нынешней архитектуры) блок памяти с общим доступом для всех потоковых процессоров в мультипроцессоре. Эта память весьма быстрая, такая же, как регистры. Она обеспечивает взаимодействие потоков, управляется разработчиком напрямую и имеет низкие задержки. Преимущества разделяемой памяти: использование в виде управляемого программистом кэша первого уровня, снижение задержек при доступе исполнительных блоков (ALU) к данным, сокращение количества обращений к глобальной памяти.

**Память констант** — область памяти объемом 64 килобайта (то же — для нынешних GPU), доступная только для чтения всеми мультипроцессорами. Она кэшируется по 8 килобайт на каждый мультипроцессор. Довольно медленная — задержка в несколько сот тактов при отсутствии нужных данных в кэше.

**Текстурная память** — блок памяти, доступный для чтения всеми мультипроцессорами. Выборка данных осуществляется при помощи текстурных блоков видеочипа, поэтому предоставляются возможности линейной интерполяции данных без дополнительных затрат. Кэшируется по 8 килобайт на каждый мультипроцессор. Медленная, как глобальная — сотни тактов задержки при отсутствии данных в кэше.

Естественно, что глобальная, локальная, текстурная и память констант — это физически одна и та же память, известная как локальная видеопамять видеокарты. Их отличия в различных алгоритмах кэширования и моделях доступа. Центральный процессор может обновлять и запрашивать только внешнюю память: глобальную, константную и текстурную.



Из написанного выше понятно, что CUDA предполагает специальный подход к разработке, не совсем такой, как принят в программах для CPU. Нужно помнить о разных типах памяти, о том, что локальная и глобальная память не кэшируется и задержки при доступе к ней гораздо выше, чем у регистровой памяти, так как она физически находится в отдельных микросхемах.

# OpenCL

Архитектура современных процессоров для увеличения производительности все чаще использует принципы параллелизма. Сталкиваясь с техническими проблемами высоких тактовых частот при фиксированном диапазоне электрического напряжения, производительность центральных процессоров (CPU, Central Processing Units) сейчас улучшается добавлением процессорных ядер. Графические процессоры (GPU, Graphics Processing Units) так же развились из устройств фиксированной функциональности построения графических сцен в программируемые параллельные процессоры. Ввиду того, что современные компьютеры часто включают высоко параллельные CPU, GPU и другие типы процессоров, важно дать возможность разработчикам программного обеспечения (ПО) использовать эти гетерогенные процессорные системы в полной мере.

Создание приложений для гетерогенных параллельных вычислительных платформ является сложным процессом, поскольку классические подходы к программированию для многоядерных CPU и GPU значительно различаются. Модели программирования CPU хотя в большинстве случаев и основаны на стандартах, но обычно они предполагают наличие общего адресного пространства и не учитывают возможность векторных операций.  
   
Модели программирования GPU общего назначения, включая сложные иерархии памяти и векторные операции, традиционно остаются платформо-зависимыми. Эти ограничения затрудняют доступ разработчиков к обширной базе исходных кодов для CPU, GPU и других типов процессоров. Как никогда, сейчас требуется предоставить возможность разработчикам программного обеспечения эффективно использовать все преимущества гетерогенных вычислительных платформ - от высокопроизводительных вычислительных серверов, через персональные компьютеры к мобильным устройствам, которые включают разнообразные параллельные CPU, GPU и другие процессоры, такие как сигнальные (DSP) и ускорители Cell/B.E.    
   
**OpenCL** (Open Computing Language) - открытый, не требующий лицензионных отчислений стандарт для универсального параллельного программирования различных типов процессоров. Стандарт предоставляет программистам переносимый и эффективный доступ ко всей мощи гетерогенных вычислительных платформ. OpenCL поддерживает широкий круг ПО: от встроенных и клиентских приложений до высокопроизводительных решений через достаточно низкоуровневневый, пригодный для использования в высокопроизводительных решениях, переносимый программный интерфейс.   
   
В результате создания эффективного, учитывающего аппаратные особенности платформы программного интерфейса, OpenCL способен сформировать базовый уровень параллельного вычислительного кода независимых от аппаратной платформы программных инструментов, промежуточного ПО и других видов приложений.

OpenCL также хорошо подходит на важную роль в интенсивно развивающихся интерактивных графических приложений, которые включают как универсальные алгоритмы параллельных вычислений, так и алгоритмы построения сложных графических сцен.   
Стандарт OpenCL описывает API для управления процессом параллельных вычислений среди гетерогенных процессоров; так же кросс-платформенный язык программирования с детально описанным вычислительным окружением.  
   
Стандарт OpenCL:

* поддерживает различные модели параллелизма;
* использует подмножество стандарта С99 с расширениями поддержки параллелизма;
* поддерживает стандарт арифметики чисел с плавающей точкой IEEE 754;
* определяет профили конфигураций для переносных и встраиваемых устройств.

Ключевыми отличиями используемого языка от C99 являются:

* Отсутствие поддержки указателей на функции, рекурсии, битовых полей, массивов переменной длины (VLA), стандартных заголовочных файлов
* Расширения языка для параллелизма: векторные типы, синхронизация, функции для Work-items/Work-Groups
* Квалификаторы типов памяти: \_\_global, \_\_local, \_\_constant, \_\_private
* Иной набор встроенных функций

# Исходные коды программ

## CPU-версия

#include "stdafx.h"

#include <conio.h>

#include <intrin.h>

#include <stdlib.h>

#include <math.h>

#include <iostream>

#include <windows.h>

using namespace std;

void MatrixInit(double \*M,int dim)

{

for (int i=0;i<dim;i++)

{

for (int j=0;j<dim;j++)

{ M[i\*dim+j]=rand();}

}

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

int n;

double sum;

int i, j, k;

DWORD timer1,timer2;

cout<<"Enter size n: ";

cin>>n;

double \*MatrixA=new double [n\*n];

double \*MatrixB=new double [n\*n];

double \*MatrixC=new double [n\*n];

MatrixInit(MatrixA,n);

MatrixInit(MatrixB,n);

timer1 = GetTickCount();

for(i=0;i<n;i++)

{

for(k=0;k<n;k++)

{

sum=0;

for(j=0;j<n;j++)

{

sum+=MatrixA[i\*n+j]\*MatrixB[j\*n+k];

}

MatrixC[i\*n+k]=sum;

}

}

timer2 = GetTickCount();

cout<<"time="<<(timer2-timer1)<<" milisec\n";

delete[] MatrixA;

delete[] MatrixB;

delete[] MatrixC;

system("Pause");

}

## OpenMP-версия

#include "stdafx.h" #include "stdafx.h" #include <omp.h> #include <conio.h>

#include <intrin.h> #include <stdlib.h> #include <math.h> #include <iostream>

#include <windows.h>

using namespace std;

void MatrixInit(double \*M,int dim)

{

for (int i=0;i<dim;i++)

{

for (int j=0;j<dim;j++)

{

M[i\*dim+j]=rand();

// cout<<M[i\*dim+j]<<"\n";

}

}

//cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

int N; char otv; double sum; int i, j, k; int tid,chunk,nthreads; double timer1,timer2;

cout<<"Enter size n: "; cin>>N;

double \*MatrixA=new double [N\*N]; double \*MatrixB=new double [N\*N];

double \*MatrixC=new double [N\*N]; MatrixInit(MatrixA,N); MatrixInit(MatrixB,N);

chunk = N/4;

timer1 =omp\_get\_wtime();

#pragma omp parallel shared(MatrixA,MatrixB,MatrixC,chunk,nthreads) private(i,j,k,tid,sum)

{

tid = omp\_get\_thread\_num();

if (tid == 0)

{

nthreads = omp\_get\_num\_threads();

printf("Starting matrix multiple with %d threads\n",nthreads);

}

#pragma omp for schedule(static,chunk)

for(i=0;i<N;i++)

{

//printf("Thread=%d did row=%d\n",tid,i);

for(k=0;k<N;k++)

{

sum=0;

for(j=0;j<N;j++)

{

sum+=MatrixA[i\*N+j]\*MatrixB[j\*N+k];

}

MatrixC[i\*N+k]=sum;

// printf("%10.0f\n", MatrixC[i\*N+k]);

}

}

}

timer2 =omp\_get\_wtime();

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n time="<<(timer2-timer1)<<" sec\n";

delete[] MatrixA;

delete[] MatrixB;

delete[] MatrixC;

system("Pause");

}

## Cuda-версия

#include <cuda.h>

#include <stdio.h>

#include <windows.h>

#define BLOCK\_SIZE 16

#define N 2000

\_\_global\_\_ void matMult ( double \* a, double \* b, int n, double \* c )

{

int bx = blockIdx.x;

int by = blockIdx.y;

int tx = threadIdx.x;

int ty = threadIdx.y;

int aBegin = n \* BLOCK\_SIZE \* by;

int aEnd = aBegin + n - 1;

int aStep = BLOCK\_SIZE;

int bBegin = BLOCK\_SIZE \* bx;

int bStep = BLOCK\_SIZE \* n;

double sum = 0;

for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep )

{

\_\_shared\_\_ float as [BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float bs [BLOCK\_SIZE][BLOCK\_SIZE];

as [ty][tx] = a [ia + n \* ty + tx];

bs [ty][tx] = b [ib + n \* ty + tx];

\_\_syncthreads();

for ( int k = 0; k < BLOCK\_SIZE; k++ ) sum += as [ty][k] \* bs [k][tx];

\_\_syncthreads();

}

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 ( double );

double \* a = new double [N\*N];

double \* b = new double [N\*N];

double \* c = new double [N\*N];

for ( int i = 0; i < N; i++ )

for ( int j = 0; j < N; j++ )

{

a [i] = 5\*i+j\*7+13;

b [i] = 5\*i+j\*7+13;

}

double \* adev = NULL;

double \* bdev = NULL;

double \* cdev = NULL;

cudaMalloc ( (void\*\*)&adev, numBytes );

cudaMalloc ( (void\*\*)&bdev, numBytes );

cudaMalloc ( (void\*\*)&cdev, numBytes );

dim3 threads ( BLOCK\_SIZE, BLOCK\_SIZE );

dim3 blocks ( N / threads.x, N / threads.y);

cudaEvent\_t start, stop;

float gpuTime = 0.0f;

cudaEventCreate ( &start );

cudaEventCreate ( &stop );

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 );

printf("time spent executing by the GPU: %0.f millseconds\n", gpuTime );

cudaEventDestroy ( start );

cudaEventDestroy ( stop );

cudaFree ( adev );

cudaFree ( bdev );

cudaFree ( cdev );

delete a;

delete b;

delete c;

system("Pause");

return 0;

}

## OpenCL-версия

Device code.

#define AS(i, j) As[j + i \* BLOCK\_SIZE]

#define BS(i, j) Bs[j + i \* BLOCK\_SIZE]

///////////////////////////////////////////////////////////////////////////////

//! Matrix multiplication on the device: C = A \* B

//! uiWA is A's width and uiWB is B's width

////////////////////////////////////////////////////////////////////////////////

\_\_kernel void

matrixMul( \_\_global float\* C, \_\_global float\* A, \_\_global float\* B,

\_\_local float\* As, \_\_local float\* Bs, int uiWA, int uiWB)

{

// Block index

int bx = get\_group\_id(0);

int by = get\_group\_id(1);

// Thread index

int tx = get\_local\_id(0);

int ty = get\_local\_id(1);

// Index of the first sub-matrix of A processed by the block

int aBegin = uiWA \* BLOCK\_SIZE \* by;

// Index of the last sub-matrix of A processed by the block

int aEnd = aBegin + uiWA - 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 \* uiWB;

// Csub is used to store the element of the block sub-matrix

// that is computed by the thread

float Csub = 0.0f;

// Loop over all the sub-matrices of A and B

// required to compute the block sub-matrix

for (int a = aBegin, b = bBegin;

a <= aEnd;

a += aStep, b += bStep) {

// Load the matrices from device memory

// to shared memory; each thread loads

// one element of each matrix

AS(ty, tx) = A[a + uiWA \* ty + tx];

BS(ty, tx) = B[b + uiWB \* ty + tx];

// Synchronize to make sure the matrices are loaded

barrier(CLK\_LOCAL\_MEM\_FENCE);

// Multiply the two matrices together;

// each thread computes one element

// of the block sub-matrix

#pragma unroll

for (int k = 0; k < BLOCK\_SIZE; ++k)

Csub += 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

barrier(CLK\_LOCAL\_MEM\_FENCE);

}

// Write the block sub-matrix to device memory;

// each thread writes one element

C[get\_global\_id(1) \* get\_global\_size(0) + get\_global\_id(0)] = Csub;

}

Host code.

// standard utilities and system includes

#include <oclUtils.h>

#include <shrQATest.h>

// project include

#include "matrixMul.h"

// max GPU's to manage for multi-GPU parallel compute

const unsigned int MAX\_GPU\_COUNT = 8;

// Globals for size of matrices

unsigned int uiWA, uiHA, uiWB, uiHB, uiWC, uiHC;

int iSizeMultiple = 1;

// global variables

cl\_context cxGPUContext;

cl\_kernel multiplicationKernel[MAX\_GPU\_COUNT];

cl\_command\_queue commandQueue[MAX\_GPU\_COUNT];

////////////////////////////////////////////////////////////////////////////////

// declaration, forward

int runTest(int argc, const char\*\* argv);

void printDiff(float\*, float\*, int, int, int, float);

void matrixMulGPU(cl\_uint ciDeviceCount, cl\_mem h\_A, float\* h\_B\_data, unsigned int mem\_size\_B, float\* h\_C );

extern "C"

void computeGold(float\*, const float\*, const float\*, unsigned int, unsigned int, unsigned int);

////////////////////////////////////////////////////////////////////////////////

// Helper functions

////////////////////////////////////////////////////////////////////////////////

double executionTime(cl\_event &event)

{

cl\_ulong start, end;

clGetEventProfilingInfo(event, CL\_PROFILING\_COMMAND\_END, sizeof(cl\_ulong), &end, NULL);

clGetEventProfilingInfo(event, CL\_PROFILING\_COMMAND\_START, sizeof(cl\_ulong), &start, NULL);

return (double)1.0e-9 \* (end - start); // convert nanoseconds to seconds on return

}

////////////////////////////////////////////////////////////////////////////////

// Program main

////////////////////////////////////////////////////////////////////////////////

int main(int argc, char\*\* argv)

{

shrQAStart(argc, argv);

// start the logs

shrSetLogFileName ("oclMatrixMul.txt");

shrLog("%s Starting...\n\n", argv[0]);

// run the code

bool bOK = (runTest(argc, (const char \*\*)argv) == CL\_SUCCESS);

shrLog("%s\n\n", (bOK ? "PASSED" : "FAILED"));

// finish

shrQAFinishExit(argc, (const char \*\*)argv, (bOK ? QA\_PASSED : QA\_FAILED));

}

void matrixMulGPU(cl\_uint ciDeviceCount, cl\_mem h\_A, float\* h\_B\_data, unsigned int mem\_size\_B, float\* h\_C )

{

cl\_mem d\_A[MAX\_GPU\_COUNT];

cl\_mem d\_C[MAX\_GPU\_COUNT];

cl\_mem d\_B[MAX\_GPU\_COUNT];

cl\_event GPUDone[MAX\_GPU\_COUNT];

cl\_event GPUExecution[MAX\_GPU\_COUNT];

// Start the computation on each available GPU

// Create buffers for each GPU

// Each GPU will compute sizePerGPU rows of the result

int sizePerGPU = uiHA / ciDeviceCount;

int workOffset[MAX\_GPU\_COUNT];

int workSize[MAX\_GPU\_COUNT];

workOffset[0] = 0;

for(unsigned int i=0; i < ciDeviceCount; ++i)

{

// Input buffer

workSize[i] = (i != (ciDeviceCount - 1)) ? sizePerGPU : (uiHA - workOffset[i]);

d\_A[i] = clCreateBuffer(cxGPUContext, CL\_MEM\_READ\_ONLY, workSize[i] \* sizeof(float) \* uiWA, NULL,NULL);

// Copy only assigned rows from host to device

clEnqueueCopyBuffer(commandQueue[i], h\_A, d\_A[i], workOffset[i] \* sizeof(float) \* uiWA,

0, workSize[i] \* sizeof(float) \* uiWA, 0, NULL, NULL);

// create OpenCL buffer on device that will be initiatlize from the host memory on first use

// on device

d\_B[i] = clCreateBuffer(cxGPUContext, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR,

mem\_size\_B, h\_B\_data, NULL);

// Output buffer

d\_C[i] = clCreateBuffer(cxGPUContext, CL\_MEM\_WRITE\_ONLY, workSize[i] \* uiWC \* sizeof(float), NULL,NULL);

// set the args values

clSetKernelArg(multiplicationKernel[i], 0, sizeof(cl\_mem), (void \*) &d\_C[i]);

clSetKernelArg(multiplicationKernel[i], 1, sizeof(cl\_mem), (void \*) &d\_A[i]);

clSetKernelArg(multiplicationKernel[i], 2, sizeof(cl\_mem), (void \*) &d\_B[i]);

clSetKernelArg(multiplicationKernel[i], 3, sizeof(float) \* BLOCK\_SIZE \*BLOCK\_SIZE, 0 );

clSetKernelArg(multiplicationKernel[i], 4, sizeof(float) \* BLOCK\_SIZE \*BLOCK\_SIZE, 0 );

clSetKernelArg(multiplicationKernel[i], 5, sizeof(cl\_int), (void \*) &uiWA);

clSetKernelArg(multiplicationKernel[i], 6, sizeof(cl\_int), (void \*) &uiWB);

if(i+1 < ciDeviceCount)

workOffset[i + 1] = workOffset[i] + workSize[i];

}

// Execute Multiplication on all GPUs in parallel

size\_t localWorkSize[] = {BLOCK\_SIZE, BLOCK\_SIZE};

size\_t globalWorkSize[] = {shrRoundUp(BLOCK\_SIZE, uiWC), shrRoundUp(BLOCK\_SIZE, workSize[0])};

// Launch kernels on devices

#ifdef GPU\_PROFILING

int nIter = 30;

for (int j = -1; j < nIter; j++)

{

// Sync all queues to host and start timer first time through loop

if(j == 0){

for(unsigned int i = 0; i < ciDeviceCount; i++)

{

clFinish(commandQueue[i]);

}

shrDeltaT(0);

}

#endif

for(unsigned int i = 0; i < ciDeviceCount; i++)

{

// Multiplication - non-blocking execution: launch and push to device(s)

globalWorkSize[1] = shrRoundUp(BLOCK\_SIZE, workSize[i]);

clEnqueueNDRangeKernel(commandQueue[i], multiplicationKernel[i], 2, 0, globalWorkSize, localWorkSize,

0, NULL, &GPUExecution[i]);

clFlush(commandQueue[i]);

}

#ifdef GPU\_PROFILING

}

#endif

// sync all queues to host

for(unsigned int i = 0; i < ciDeviceCount; i++)

{

clFinish(commandQueue[i]);

}

#ifdef GPU\_PROFILING

// stop and log timer

double dSeconds = shrDeltaT(0)/(double)nIter;

double dNumOps = 2.0 \* (double)uiWA \* (double)uiHA \* (double)uiWB;

double gflops = 1.0e-9 \* dNumOps/dSeconds;

shrLogEx(LOGBOTH | MASTER, 0, "oclMatrixMul, Throughput = %.4f GFlops/s, Time = %.5f s, Size = %.0f, NumDevsUsed = %d, Workgroup = %u\n",

gflops, dSeconds, dNumOps, ciDeviceCount, localWorkSize[0] \* localWorkSize[1]);

// Print kernel timing per GPU

shrLog("\n");

for(unsigned int i = 0; i < ciDeviceCount; i++)

{

shrLog(" Kernel execution time on GPU %d \t: %.5f s\n", i, executionTime(GPUExecution[i]));

}

shrLog("\n");

#endif

for(unsigned int i = 0; i < ciDeviceCount; i++)

{

// Non-blocking copy of result from device to host

clEnqueueReadBuffer(commandQueue[i], d\_C[i], CL\_FALSE, 0, uiWC \* sizeof(float) \* workSize[i],

h\_C + workOffset[i] \* uiWC, 0, NULL, &GPUDone[i]);

}

// CPU sync with GPU

clWaitForEvents(ciDeviceCount, GPUDone);

// Release mem and event objects

for(unsigned int i = 0; i < ciDeviceCount; i++)

{

clReleaseMemObject(d\_A[i]);

clReleaseMemObject(d\_C[i]);

clReleaseMemObject(d\_B[i]);

clReleaseEvent(GPUExecution[i]);

clReleaseEvent(GPUDone[i]);

}

}

////////////////////////////////////////////////////////////////////////////////

//! Run a simple test for

////////////////////////////////////////////////////////////////////////////////

int runTest(int argc, const char\*\* argv)

{

cl\_platform\_id cpPlatform = NULL;

cl\_uint ciDeviceCount = 0;

cl\_device\_id \*cdDevices = NULL;

cl\_int ciErrNum = CL\_SUCCESS;

//Get the NVIDIA platform

ciErrNum = oclGetPlatformID(&cpPlatform);

if (ciErrNum != CL\_SUCCESS)

{

shrLog("Error: Failed to create OpenCL context!\n");

return ciErrNum;

}

//Get the devices

ciErrNum = clGetDeviceIDs(cpPlatform, CL\_DEVICE\_TYPE\_GPU, 0, NULL, &ciDeviceCount);

cdDevices = (cl\_device\_id \*)malloc(ciDeviceCount \* sizeof(cl\_device\_id) );

ciErrNum = clGetDeviceIDs(cpPlatform, CL\_DEVICE\_TYPE\_GPU, ciDeviceCount, cdDevices, NULL);

if (ciErrNum != CL\_SUCCESS)

{

shrLog("Error: Failed to create OpenCL context!\n");

return ciErrNum;

}

//Create the context

cxGPUContext = clCreateContext(0, ciDeviceCount, cdDevices, NULL, NULL, &ciErrNum);

if (ciErrNum != CL\_SUCCESS)

{

shrLog("Error: Failed to create OpenCL context!\n");

return ciErrNum;

}

if(shrCheckCmdLineFlag(argc, (const char\*\*)argv, "device"))

{

// User specified GPUs

char\* deviceList;

char\* deviceStr;

char\* next\_token;

shrGetCmdLineArgumentstr(argc, (const char\*\*)argv, "device", &deviceList);

#ifdef WIN32

deviceStr = strtok\_s (deviceList," ,.-", &next\_token);

#else

deviceStr = strtok (deviceList," ,.-");

#endif

ciDeviceCount = 0;

while(deviceStr != NULL)

{

// get and print the device for this queue

cl\_device\_id device = oclGetDev(cxGPUContext, atoi(deviceStr));

if( device == (cl\_device\_id) -1 ) {

shrLog(" Device %s does not exist!\n", deviceStr);

return -1;

}

shrLog("Device %s: ", deviceStr);

oclPrintDevName(LOGBOTH, device);

shrLog("\n");

// create command queue

commandQueue[ciDeviceCount] = clCreateCommandQueue(cxGPUContext, device, CL\_QUEUE\_PROFILING\_ENABLE, &ciErrNum);

if (ciErrNum != CL\_SUCCESS)

{

shrLog(" Error %i in clCreateCommandQueue call !!!\n\n", ciErrNum);

return ciErrNum;

}

++ciDeviceCount;

#ifdef WIN32

deviceStr = strtok\_s (NULL," ,.-", &next\_token);

#else

deviceStr = strtok (NULL," ,.-");

#endif

}

free(deviceList);

}

else

{

// Find out how many GPU's to compute on all available GPUs

size\_t nDeviceBytes;

ciErrNum |= clGetContextInfo(cxGPUContext, CL\_CONTEXT\_DEVICES, 0, NULL, &nDeviceBytes);

ciDeviceCount = (cl\_uint)nDeviceBytes/sizeof(cl\_device\_id);

if (ciErrNum != CL\_SUCCESS)

{

shrLog(" Error %i in clGetDeviceIDs call !!!\n\n", ciErrNum);

return ciErrNum;

}

else if (ciDeviceCount == 0)

{

shrLog(" There are no devices supporting OpenCL (return code %i)\n\n", ciErrNum);

return -1;

}

// create command-queues

for(unsigned int i = 0; i < ciDeviceCount; ++i)

{

// get and print the device for this queue

cl\_device\_id device = oclGetDev(cxGPUContext, i);

shrLog("Device %d: ", i);

oclPrintDevName(LOGBOTH, device);

shrLog("\n");

// create command queue

commandQueue[i] = clCreateCommandQueue(cxGPUContext, device, CL\_QUEUE\_PROFILING\_ENABLE, &ciErrNum);

if (ciErrNum != CL\_SUCCESS)

{

shrLog(" Error %i in clCreateCommandQueue call !!!\n\n", ciErrNum);

return ciErrNum;

}

}

}

// Optional Command-line multiplier for matrix sizes

shrGetCmdLineArgumenti(argc, (const char\*\*)argv, "sizemult", &iSizeMultiple);

iSizeMultiple = CLAMP(iSizeMultiple, 1, 10);

uiWA = WA \* iSizeMultiple;

uiHA = HA \* iSizeMultiple;

uiWB = WB \* iSizeMultiple;

uiHB = HB \* iSizeMultiple;

uiWC = WC \* iSizeMultiple;

uiHC = HC \* iSizeMultiple;

shrLog("\nUsing Matrix Sizes: A(%u x %u), B(%u x %u), C(%u x %u)\n",

uiWA, uiHA, uiWB, uiHB, uiWC, uiHC);

// allocate host memory for matrices A and B

unsigned int size\_A = uiWA \* uiHA;

unsigned int mem\_size\_A = sizeof(float) \* size\_A;

float\* h\_A\_data = (float\*)malloc(mem\_size\_A);

unsigned int size\_B = uiWB \* uiHB;

unsigned int mem\_size\_B = sizeof(float) \* size\_B;

float\* h\_B\_data = (float\*)malloc(mem\_size\_B);

// initialize host memory

srand(2006);

shrFillArray(h\_A\_data, size\_A);

shrFillArray(h\_B\_data, size\_B);

// allocate host memory for result

unsigned int size\_C = uiWC \* uiHC;

unsigned int mem\_size\_C = sizeof(float) \* size\_C;

float\* h\_C = (float\*) malloc(mem\_size\_C);

// create OpenCL buffer pointing to the host memory

cl\_mem h\_A = clCreateBuffer(cxGPUContext, CL\_MEM\_READ\_ONLY | CL\_MEM\_USE\_HOST\_PTR,

mem\_size\_A, h\_A\_data, &ciErrNum);

if (ciErrNum != CL\_SUCCESS)

{

shrLog("Error: clCreateBuffer\n");

return ciErrNum;

}

// Program Setup

size\_t program\_length;

const char\* header\_path = shrFindFilePath("matrixMul.h", argv[0]);

oclCheckError(header\_path != NULL, shrTRUE);

char\* header = oclLoadProgSource(header\_path, "", &program\_length);

if(!header)

{

shrLog("Error: Failed to load the header %s!\n", header\_path);

return -1000;

}

const char\* source\_path = shrFindFilePath("matrixMul.cl", argv[0]);

oclCheckError(source\_path != NULL, shrTRUE);

char \*source = oclLoadProgSource(source\_path, header, &program\_length);

if(!source)

{

shrLog("Error: Failed to load compute program %s!\n", source\_path);

return -2000;

}

// create the program

cl\_program cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char \*\*)&source,

&program\_length, &ciErrNum);

if (ciErrNum != CL\_SUCCESS)

{

shrLog("Error: Failed to create program\n");

return ciErrNum;

}

free(header);

free(source);

// build the program

ciErrNum = clBuildProgram(cpProgram, 0, NULL, "-cl-fast-relaxed-math", NULL, NULL);

if (ciErrNum != CL\_SUCCESS)

{

// write out standard error, Build Log and PTX, then return error

shrLogEx(LOGBOTH | ERRORMSG, ciErrNum, STDERROR);

oclLogBuildInfo(cpProgram, oclGetFirstDev(cxGPUContext));

oclLogPtx(cpProgram, oclGetFirstDev(cxGPUContext), "oclMatrixMul.ptx");

return ciErrNum;

}

// write out PTX if requested on the command line

if(shrCheckCmdLineFlag(argc, argv, "dump-ptx") )

{

oclLogPtx(cpProgram, oclGetFirstDev(cxGPUContext), "oclMatrixMul.ptx");

}

// Create Kernel

for(unsigned int i = 0; i < ciDeviceCount; ++i) {

multiplicationKernel[i] = clCreateKernel(cpProgram, "matrixMul", &ciErrNum);

if (ciErrNum != CL\_SUCCESS)

{

shrLog("Error: Failed to create kernel\n");

return ciErrNum;

}

}

// Run multiplication on 1..deviceCount GPUs to compare improvement

shrLog("\nRunning Computations on 1 - %d GPU's...\n\n", ciDeviceCount);

for(unsigned int k = 1; k <= ciDeviceCount; ++k)

{

matrixMulGPU(k, h\_A, h\_B\_data, mem\_size\_B, h\_C);

}

// compute reference solution

shrLog("Comparing results with CPU computation... \n\n");

float\* reference = (float\*) malloc(mem\_size\_C);

computeGold(reference, h\_A\_data, h\_B\_data, uiHA, uiWA, uiWB);

// check result

shrBOOL res = shrCompareL2fe(reference, h\_C, size\_C, 1.0e-6f);

if (res != shrTRUE)

{

printDiff(reference, h\_C, uiWC, uiHC, 100, 1.0e-5f);

}

// clean up OCL resources

ciErrNum = clReleaseMemObject(h\_A);

for(unsigned int k = 0; k < ciDeviceCount; ++k)

{

ciErrNum |= clReleaseKernel( multiplicationKernel[k] );

ciErrNum |= clReleaseCommandQueue( commandQueue[k] );

}

ciErrNum |= clReleaseProgram(cpProgram);

ciErrNum |= clReleaseContext(cxGPUContext);

if(ciErrNum != CL\_SUCCESS)

{

shrLog("Error: Failure releasing OpenCL resources: %d\n", ciErrNum);

return ciErrNum;

}

// clean up memory

free(h\_A\_data);

free(h\_B\_data);

free(h\_C);

free(reference);

return ((shrTRUE == res) ? CL\_SUCCESS : -3000);

}

void printDiff(float \*data1, float \*data2, int width, int height, int iListLength, float fListTol)

{

shrLog("Listing first %d Differences > %.6f...\n", iListLength, fListTol);

int i,j,k;

int error\_count=0;

for (j = 0; j < height; j++)

{

if (error\_count < iListLength)

{

shrLog("\n Row %d:\n", j);

}

for (i = 0; i < width; i++)

{

k = j \* width + i;

float fDiff = fabs(data1[k] - data2[k]);

if (fDiff > fListTol)

{

if (error\_count < iListLength)

{

shrLog(" Loc(%d,%d)\tCPU=%.5f\tGPU=%.5f\tDiff=%.6f\n", i, j, data1[k], data2[k], fDiff);

}

error\_count++;

}

}

}

shrLog(" \n Total Errors = %d\n\n", error\_count);

}

Export C interface

extern "C"

void computeGold( float\*, const float\*, const float\*, unsigned int, unsigned int, unsigned int);

////////////////////////////////////////////////////////////////////////////////

//! Compute reference data set

//! C = A \* B

//! @param C reference data, computed but preallocated

//! @param A matrix A as provided to device

//! @param B matrix B as provided to device

//! @param hA height of matrix A

//! @param wB width of matrix B

////////////////////////////////////////////////////////////////////////////////

void

computeGold(float\* C, const float\* A, const float\* B, unsigned int hA, unsigned int wA, unsigned int wB)

{

for (unsigned int i = 0; i < hA; ++i)

for (unsigned int j = 0; j < wB; ++j) {

double sum = 0;

for (unsigned int k = 0; k < wA; ++k) {

double a = A[i \* wA + k];

double b = B[k \* wB + j];

sum += a \* b;

}

C[i \* wB + j] = (float)sum;

}

}

Size setup

#ifndef \_MATRIXMUL\_H\_

#define \_MATRIXMUL\_H\_

// Thread block size

#define BLOCK\_SIZE 16

// Basic Matrix dimensions (can be amplified by command line switch)

// (chosen as multiples of the thread block size for simplicity)

#define WA (5 \* BLOCK\_SIZE) // Matrix A width

#define HA (10 \* BLOCK\_SIZE) // Matrix A height

#define WB (5 \* BLOCK\_SIZE) // Matrix B width

#define HB WA // Matrix B height

#define WC WB // Matrix C width

#define HC HA // Matrix C height

#endif // \_MATRIXMUL\_H\_

# Информационные источники

CUDA

1. Общие положения о технологии:

* <http://www.ixbt.com/>
* <http://steps3d.narod.ru/tutorials/cuda-tutorial.html>

1. Инструменты для работы с технологией:

* <http://www.nvidia.ru/object/cuda_home_new_ru.html>

1. Настройка среды разработки для работы с технологией:

* <http://www.programmerfish.com/how-to-run-cuda-on-visual-studio-2008-vs08/>
* <http://www.youtube.com/watch?v=Cm7W6MaNuF4>

1. Примеры работы с технологией:

* <http://www.gpgpu.ru/>
* <http://gpgpu-computing.blogspot.com/>
* <http://mezhov.blogspot.com/>
* NVIDIA GPU Computing SDK 4.0
* Джейсон Сандерс, Эдвард Кэндрот «CUDA by Example»

OpenMP

1. Общие положения о технологии:

* <http://openmp.org/>

1. Настройка среды разработки для работы с технологией:

* <http://iproc.ru/programming/openmp-visual-studio/>

1. Примеры работы с технологией:

* <http://ru.wikipedia.org/wiki/OpenMP>
* <https://computing.llnl.gov/tutorials/openMP/exercise.html>
* <http://blog.speedgocomputing.com/>
* <http://www.exzec.ru/node/163>
* <http://www.hpcc.unn.ru/>
* <http://parallel.ru/tech/tech_dev/openmp.html>

OpenCL

1. Общие положения о технологии:

* <http://opencl.ru>
* <http://steps3d.narod.ru/>

1. Настройка среды разработки для работы с технологией:

* <http://www.guineacode.com/>

1. Примеры работы с технологией:

* <http://gpgpu-computing.blogspot.com/>
* <http://nvoclc.blogspot.com/>
* <http://vasanthexperiments.wordpress.com/>
* NVIDIA GPU Computing SDK 4.0