INTRODUCTION TO CUDA PROGRAMING

Đỗ Như Tài
Khoa Công nghệ Thông tin
Trường Đại học Ngoại ngữ - Tin học TP.HCM

NỘI DUNG TRÌNH BÀY

- Giới thiệu tính toán song song
- Lập trình song song trên GPUs
- CUDA
- Mô hình lập trình CUDA
- Mô hình bộ nhớ CUDA
- Lập trình C/C++ với CUDA
- Ví dụ minh họa

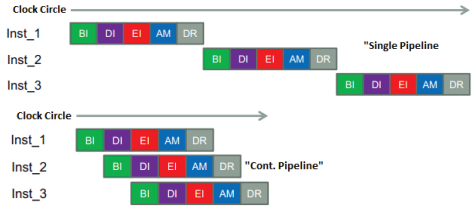
TÍNH TOÁN SONG SONG LÀ GÌ?

- Chạy trên nhiều máy hoặc sử dụng nhiều bộ vi xử lý chạy song song
- Các kiến trúc cho tính toán song song hiện nay:

Tính toán trên nhiều vi xử lý (Cores) | Cluster | Graphics Cards (GPUs)

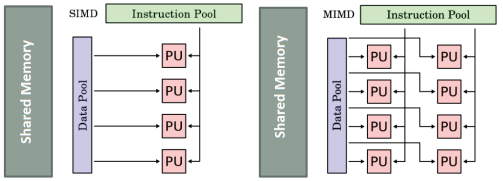
CÁC LOẠI TÍNH TOÁN SONG SONG (1)

- Mức độ bit
  - Dựa trên việc tăng số lượng bit dữ liệu xử lý
  - Các loại bộ vi xử lý 8 bits, 16 bits, 32 bits, 64 bits
- Mức độ lệnh
  - Khả năng xử lý các lệnh trùng lặp, phụ thuộc vào processor pipeline



CÁC LOẠI TÍNH TOÁN SONG SONG (2)

- Mức độ dữ liệu
  - Mỗi vi xử lý thực hiện tác vụ giống nhau tại các phần khác nhau của dữ liệu (SIMD - ***Single Instruction Multiple Data***)
- Mức độ nhiệm vụ
  - Các vi xử lý khác nhau chạy các lệnh khác nhau ở các phần khác nhau của dữ liệu (MIMD - ***Multiple Instruction Multiple Data***)



So sánh giữa SIMD và MIMD

CÁC LOẠI NGÔN NGỮ LẬP TRÌNH SONG SONG

OpenMP (Shared Memory) | **OpenMPI** (Distributed Memory) | **CUDA** (**C**ompute **U**nified **D**evice **A**rchitecture) on GPUs

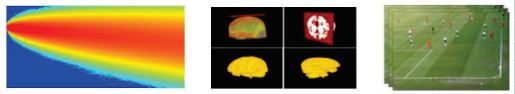
OpenCL (**Open C**omputing **L**anguage) on GPUs, CPUs

Kiến trúc hỗn hợp
+ Máy tính (nhiều nhân) và một GPU (hoặc nhiều GPUs)
+ Nhiều cụm máy tính nhiều nhân kết hợp và trên nó có một GPU (hoặc nhiều GPUs)

Kiến trúc hỗn hợp dẫn đến pha trộn nhiều ngôn ngữ:
+ CUDA (OpenMP, OpenMPI)
+ OpenCL (OpenMP, OpenMPI)
+ PGI Fortan
+ OpenACC

GPUs

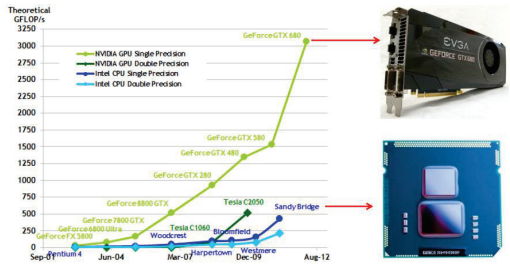
- Bo mạch đồ họa
- Các vi xử lý mềm dẻo, mạnh mẽ trong tính toán song song
- Giải quyết các vấn đề trong nhiều lĩnh vực khác nhau: tài chính, đồ họa, xử lý ảnh và video, toán học, vật lý, sinh học, …



Phương trình vi phân | Y học | Phát hiện đối tượng

Sinh học phân tử

GPUs (1)

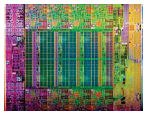


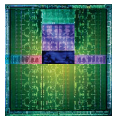
CPU vs. GPU

- Kiến trúc GPU vs CPU

CPU vs. GPU

	Xeon E5-2687W	Kepler GTX 680
Cores	8 (superscalar)	1536 (simple)
Active threads	2 per core	~11 per core
Frequency	3.1 GHz	1.0 GHz
Peak performance (SP)	397 GFlop/s	3090 GFlop/s
Peak mem. bandwidth	51 GB/s	192 GB/s
Maximum power	150 W	195 W (toàn bộ card)
Price	$1900	$500 (toàn bộ card)
Release dates	03/2012	03/2012





ƯU ĐIỂM GPU

- Tốc độ
  - Hơn 8 lần về số phép toán trên giây
- Nhiều băng thông bộ nhớ chính
  - Hơn 4 lần về tốc độ truyền bộ nhớ
- Hiệu quả về chi phí, năng lượng và kích thước
  - Hơn 29 lần về chi phí
  - Hơn 6 lần về số watt tiêu thụ
  - Hơn 11 lần về diện tích

NHƯỢC ĐIỂM GPU

- Tại sao máy tính không sử dụng GPU trong toàn bộ công việc?
- GPUs chỉ có thể thực hiện trong một số mã chương trình cần tăng tốc thực hiện
  - Cần cơ chế song song, sử dụng lại dữ liệu và có tính đều đặn
- GPUs khó lập trình và điều chỉnh hơn so với CPU
  - Các công cụ lập trình thiếu
  - Cơ chế kiến trúc
  - Thiếu hỗ trợ về các mã nguồn đặc trưng

CUDA

- Một công nghệ cho phép thực thi mã trên GPU cho xử lý song song
  - Sử dụng chipset NVIDIA trong năm 2006
  - Để sử dụng kiến trúc này là cần thiết để có một GeForce 8 series (hoặc tương đương Quadro)
- Các phần mềm hỗ trợ CUDA

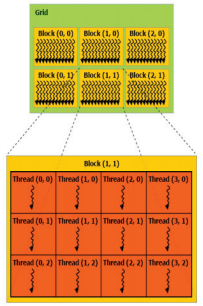
Adobe, ANSYS, Autodesk, DASSAULT SYSTEMES, NATIONAL INSTRUMENTS, MathWorks, Microsoft, Wolfram Mathematica

TÍNH NĂNG CUDA

- Hỗ trợ ngôn ngữ lập trình C, C++, Fortan, Matlab, Python, LabView…
- Hỗ trợ quản lý thread và dữ liệu song song
- Các thư viện:
  - FFT (Fast Fourier Transform)
  - BLAS (Basic linear algebra subrountines)
  - CURAND (Generate random numbers)
  - CUSPARSE (Linear algebra subrountines to operate sparse matrices)
  - NPP (NVIDIA performance primitives)
- Hỗ trợ các thư viện: OpenGL, DirectX
- Hỗ trợ nhiều hệ điều hành:
  - Windows, Linux, MacOS, …

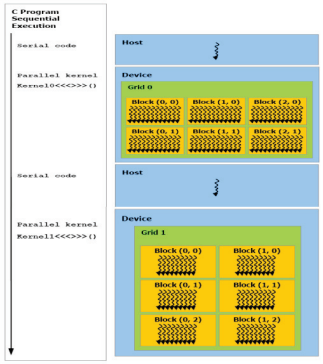
MÔ HÌNH LẬP TRÌNH CUDA (1)

- Một chương trình chạy trên card đồ họa gọi là **kernel**
- Một kernel tổ chức thành một ***grid*** có các ***blocks*** với các ***thread*** chạy song song cùng xử lý tập lệnh
- Một block bao gồm nhiều thread tương tác nhau:
  - Dễ dàng xử lý Shared Memory
  - Đồng bộ giữa các thread
  - Mỗi thread có một định danh
  - Tổ chức thành mảng 1, 2, hay 3 chiều
- Một Grid chứa nhiều blocks:
  - Có số lượng thread giới hạn trong mỗi block
  - Mỗi block có một định danh
  - Tổ chức thành dạng mảng:
    - 1 hay 2 chiều (capacity <2.0)
    - 1, 2 hay 3 chiều (capacity >=2.0)



MÔ HÌNH LẬP TRÌNH CUDA (2)

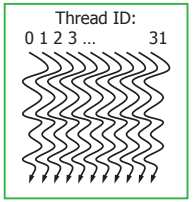
- Chạy trên Host và Device
  - Host: CPU
  - Device: GPU
  - Kernel: tập hợp các lệnh chạy trên Device



THREAD PHÂN KỲ

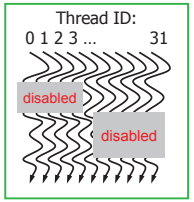
- Mã không phân kỳ

```
if (threadID >= 32) {
  some_code;
} else {
  other_code;
}
```

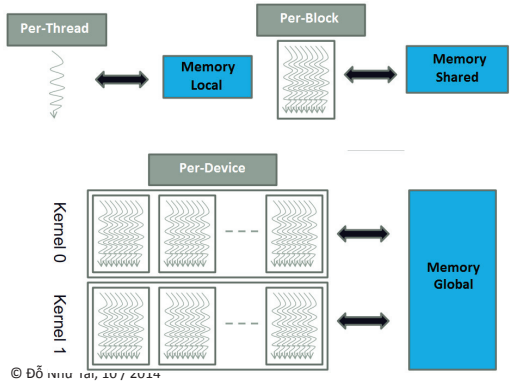


- Mã phân kỳ

```
if (threadID >= 13) {
  some_code;
} else {
  other_code;
}
```

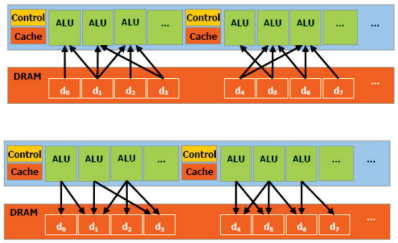


MÔ HÌNH BỘ NHỚ CUDA (1)



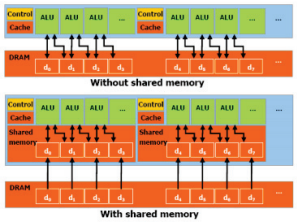
MÔ HÌNH BỘ NHỚ CUDA (2)

- CUDA cung cấp một địa chỉ bộ nhớ đọc và ghi chứa trong DRAM (bộ nhớ Device)

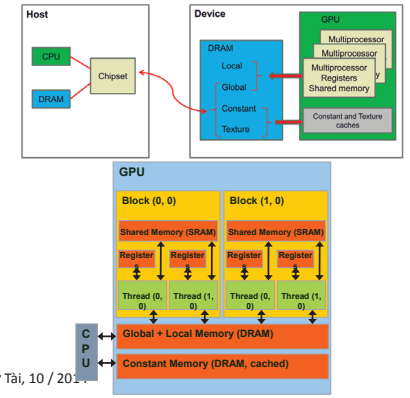


MÔ HÌNH BỘ NHỚ CUDA (3)

- CUDA cung cấp Shared Memory giúp cho việc đọc và ghi nhanh hơn
- Cho phép nhiều thread chia sẽ dữ liệu với nhau
- Tối thiểu việc truy xuất DRAM



MÔ HÌNH BỘ NHỚ CUDA (4)



CÁC LỆNH QUẢN LÝ BỘ NHỚ

- Tạo bộ nhớ
  - cudaMalloc ((void**) devPtr, size_t size)
  - cudaMallocHost ((void**) hostPtr, size_t size)
  - cudaFree (void *devPtr)
  - cudaFreeHost (void *hostPtr)
- Sao chép bộ nhớ
  - cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - cudaMemcpy2D(void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)
  - cudaMemcpyToSymbol(const char *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind) H→D, D→D
  - cudaMemcpyFromSymbol(void *dst, const char *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind) D→H D→D

  Kind = cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice.

QUALIFIERS CỦA HÀM

- __device__
  - Chạy trên Device,
  - Chỉ gọi từ Device
- __global__
  - Chạy trên Device
  - Chỉ gọi từ Host
- __host__
  - Chạy trên Host
  - Chỉ gọi từ Host

QUALIFIERS CỦA BIẾN

- __device__
  - Nằm trong Global Memory
  - Tồn tại suốt chương trình ứng dụng
  - Xử lý từ tất cả thread cùng grid và host
- __constant__ (có thể đi cùng với __device__)
  - Nằm trong Constant Memory
  - Tồn tại suốt chương trình ứng dụng
  - Xử lý từ tất cả thread cùng grid và host
- __shared__ (có thể đi cùng với __device__)
  - Nằm trong Shared Memory của thread block
  - Tồn tại suốt thread block
  - Xử lý từ trong thread cùng block

HÀM KERNEL

- Gọi hàm Kernel
  - Khai báo hàm
    - __global__ void NameFunc(tham số);
  - Lời gọi hàm
    - NameFunc <<< Dg, Db, Ns, St >>> (tham số);
  - Dg: kiểu dim3, qui định kích thước grid
  - Db: kiểu dim3, qui định kích thước block
  - Ns: số bytes trong Shared Memory
  - St: kiểu cudaStream_t xác định stream sử dụng hàm kernel
- Các biến định nghĩa trước: các hàm __global__ và __device__ có sẵn các biến sau:
  - gridDim: loại dim3, xác định kích thước grid
  - blockIdx: loại uint3, xác định chỉ số block trong grid
  - blockDim: loại dim3, xác định kích thước block
  - threadIdx: loại uint3, xác định chỉ số thread trong block

CÁC KIỂU DỮ LIỆU

- char1, uchar1, char2, uchar2, char3, char3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, int4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, longlong1, longlong2, float1, float2, float3, float4, double1,double2
- Các thành phần 1, 2, 3, 4 tương ứng với x, y, z, w
- Ví dụ:

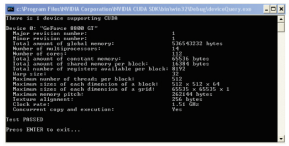
```
float3 temp[10];
.....
temp[i].x=0.0; temp[i].y=0.0; temp[i].z=0.0;
```


CÁC HÀM TOÁN HỌC

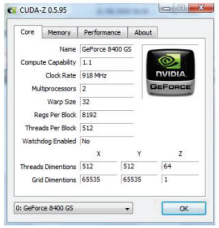
- __FuncName(): chạy ở mức độ phần cứng, ít chính xác, bao gồm: __sinf(x), __expf(x), __logf(x),...
- FuncName(): chậm hơn nhưng chính xác, bao gồm: sinf(x), expf(x), logf(x),...
- -use_fast_math: Nvcc compiler option

KIỂM TRA THÔNG TIN GPUs

- deviceQueryDrv



- CUDA-Z



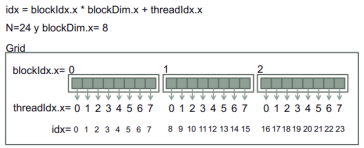
Ví dụ 1: Hello World

```
#include <stdio.h>

// printf() is only supported
// for devices of compute capability 2.0 and higher

__global__ void helloCUDA(float e){
    printf("Hello, I am thread %d of block %d with value e=%f\n", threadIdx.x,blockIdx.x, e);
}

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

    helloCUDA<<<3, 4>>>(2.71828f);

    cudaDeviceReset();//is called to reinitialize the device.
    system("pause");
    return(0);
}
```

```
C:\Users\lab-h dr8s6v1\Documents\FRANCO_XPS\DOCTORADO\Taller\CUDA_2012\CUDA_HelloW...
Hello, I am thread 0 of block 1 with value e=2.718280
Hello, I am thread 1 of block 1 with value e=2.718280
Hello, I am thread 2 of block 1 with value e=2.718280
Hello, I am thread 3 of block 1 with value e=2.718280
Hello, I am thread 0 of block 0 with value e=2.718280
Hello, I am thread 1 of block 0 with value e=2.718280
Hello, I am thread 2 of block 0 with value e=2.718280
Hello, I am thread 3 of block 0 with value e=2.718280
Hello, I am thread 0 of block 2 with value e=2.718280
Hello, I am thread 1 of block 2 with value e=2.718280
Hello, I am thread 2 of block 2 with value e=2.718280
Hello, I am thread 3 of block 2 with value e=2.718280
Presione una tecla para continuar . . .
```


Ví dụ 2: Tính tổng 2 Vector

- Vector sum (c = a + b)
  - Create vectors a, b, c on the host and load data a, b
  - Create vectors a, b, c in the device
  - Copy the contents of the vectors a and b on host to device
  - Initialize the grid and block dimensions, and launch kernel to add a and b on device
  - Copy the result on device to the host
  - Write result vector c on host
  - Free Device and Host Memory



Ví dụ 2: Tính tổng 2 Vector

```
// Código principal que se ejecuta en el Host
int main(void){
    float *a_h,*b_h,*c_h; //Punteros a arreglos en el Host
    float *a_d,*b_d,*c_d;  //Punteros a arreglos en el Device
    const int N = 24;  //Número de elementos en los arreglos  (probar 1000000)

    size_t size=N * sizeof(float);

    a_h = (float *)malloc(size); // Pedimos memoria en el Host
    b_h = (float *)malloc(size);
    c_h = (float *)malloc(size);//También se puede con cudaMallocHost

    //Inicializamos los arreglos a,b en el Host
    for (int i=0; i<N; i++){
        a_h[i] = (float)i;
        b_h[i] = (float)(i+1);
    }

    printf("\nArreglo a:\n");
    for (int i=0; i<N; i++) printf("%f ", a_h[i]);
    printf("\n\nArreglo b:\n");
    for (int i=0; i<N; i++) printf("%f ", b_h[i]);

    cudaMalloc((void **) &a_d,size);   // Pedimos memoria en el Device
    cudaMalloc((void **) &b_d,size);
    cudaMalloc((void **) &c_d,size);

    //Pasamos los arreglos a y b del Host al Device
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
```

Ví dụ 2: Tính tổng 2 Vector

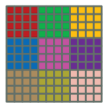
```
    //Realizamos el cálculo en el Device
    int block_size =8;
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);

    Suma_vectores <<< n_blocks, block_size >>> (c_d,a_d,b_d,N);

    //Pasamos el resultado del Device al Host
    cudaMemcpy(c_h, c_d, size,cudaMemcpyDeviceToHost);

    //Resultado
    printf("\n\nArreglo c:\n");
    for (int i=0; i<N; i++) printf("%f ", c_h[i]);

    _getche();

    // Liberamos la memoria del Host
    free(a_h);
    free(b_h);
    free(c_h);

    // Liberamos l
    cudaFree(a_d);
    cudaFree(b_d);
    cudaFree(c_d);
    return(0);
}
```

```
// Función Kernel que se ejecuta en el Device.
__global__ void Suma_vectores(float *c,float *a,float *b, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N){
      c[idx] = a[idx] + b[idx];
  }
}
```

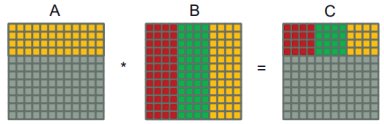

Ví dụ 3: Nhân 2 ma trận

```
idx = blockIdx.x * blockDim.x + threadIdx.x
idy = blockIdx.y * blockDim.y + threadIdx.y

nfil=12, ncol=12, BLOCK_SIZE=4
```

Grid



blockIdx.x={0,1,2}
blockIdx.y={0,1,2}
threadIdx.x={0,1,2,3}
threadIdx.y={0,1,2,3}
idx={0,1,2,…,11}
idy={0,1,2,…,11}



Ví dụ 3: Nhân 2 ma trận

```
//Multiplicacion de Matrices en Memoria Global (GM)
__global__ void Multiplica_Matrices_GM(float *C,float *A,float *B,
                                       int nfil,int ncol)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int index=idy*ncol+idx;
    if (idy<nfil && idx<ncol){
        float sum=0.0f;
        for(int k=0;k<ncol;k++){
            sum+=A[idy*ncol+k]*B[k*ncol+idx];
        }
        C[index] = sum;
    }
}
```


Ví dụ 3: Nhân 2 ma trận

- Create matrix a, b, c on the host and load data a, b
- Create matrix a, b, c in the device
- Copy the contents of the matrix a and b on host to device
- Initialize the grid and block dimensions, and launch kernel to multiply a and b on device
- Copy the result on device to the host
- Write result matrix c on host
- Free Device and Host Memory

TÀI LIỆU THAM KHẢO

- Francisco Javier Hernández López, ***EJEMPLOS DE PROGRAMACIÓN EN CUDA***, http://www.cimat.mx/~fcoj23
- Martin Burtscher, ***CUDA Optimization Tutorial***, http://www.cs.txstate.edu/~burtscher/tutorials/COT5/slides.pptx

XIN CHÂN THÀNH CÁM ƠN QUÝ THẦY CÔ ĐÃ LẮNG NGHE!

1	2
3	4
5	6
7	8
9	10
11	12
13	14
15	16
17	18
19	20
21	22
23	24
25	26
27	28
29	30
31	32
33	34
35	36

LAB FOR INTRODUCTION TO CUDA

SETUP ENVIRONMENT WITH CUDA INTEGRATED IN VISUAL STUDIO 2010

Ensure you have Microsoft Visual Studio 2010 installed before you install the CUDA toolkit, if you have already installed the CUDA toolkit and have not yet installed Visual Studio then un-install it before continuing (not doing so will result in Visual Studio integration not functioning)

Setup your environment

- Install Microsoft Visual Studio 2010 (***optional***)

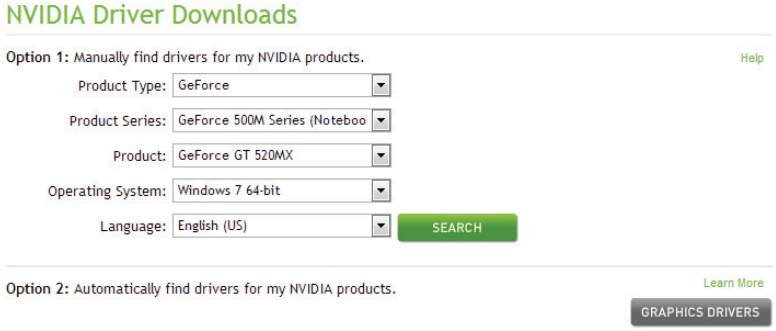
You can download Microsoft Visual C++ 2010 Express for free from http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express.

- Verify a CUDA-Capable GPU

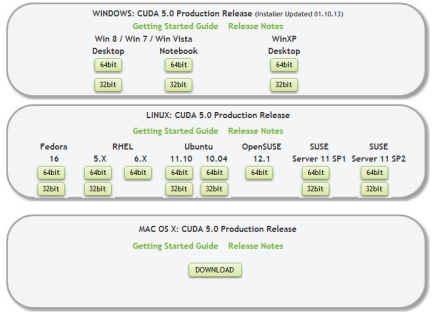
Find the vendor name and model of your graphics card. If it is an NVIDIA card that is listed in ***http://www.nvidia.com/object/cuda_gpus.html***, your GPU is CUDA-capable.

- Install the latest Nvidia driver

You can download from ***http://www.nvidia.com/Download/index.aspx?lang=en-us***

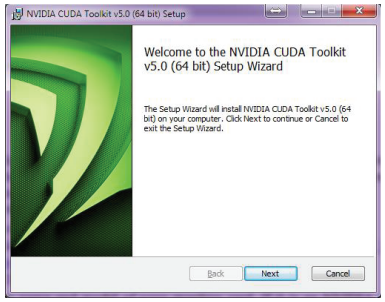


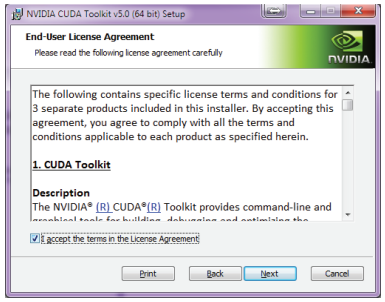
- Install CUDA Toolkit and CUDA SDK
  - Download the NVIDIA CUDA Toolkit from ***http://www.nvidia.com/content/cuda/cuda-downloads.html***. The NVIDIA CUDA Toolkit contains ***the driver*** and ***tools*** needed to create, build and run a CUDA application as well as libraries, header files, CUDA ***samples*** source code, and other resources.

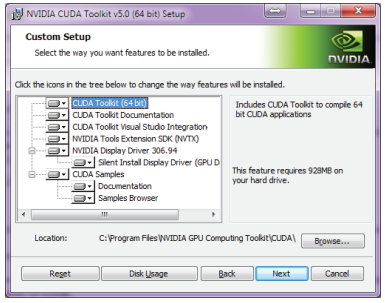


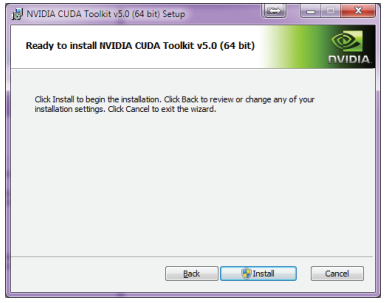
Choose the platform you are using and download the NVIDIA CUDA Toolkit. Example: ***Notebook – 64bit***

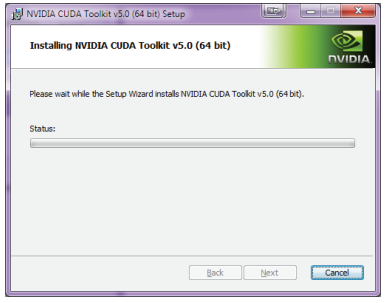
- Install the CUDA Toolkit by executing the Toolkit installer and following the on-screen prompts.

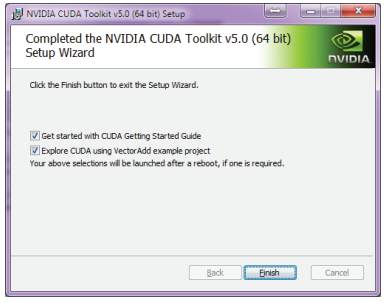












- CUDA Toolkit:
  - The CUDA Toolkit installation defaults to ***C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v#.#***, where #.# is version number 3.2 or higher.

Bin\ → the compiler executables and runtime libraries
Include\ →the header files needed to compile CUDA programs
Lib\ → the library files needed to link CUDA programs
Doc\ →the CUDA C Programming Guide, CUDA C Best Practices Guide, documentation for the CUDA libraries, and other CUDA Toolkit-related documentation

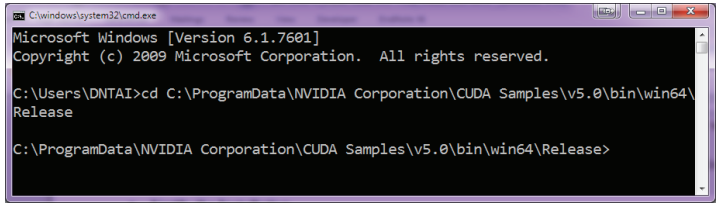
  - CUDA Samples

The CUDA Samples contain source code for many example problems and templates with Microsoft Visual Studio 2008 and 2010 projects.
For Windows Vista, Windows 7, and Windows Server 2008, the samples can be found here: ***C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0***

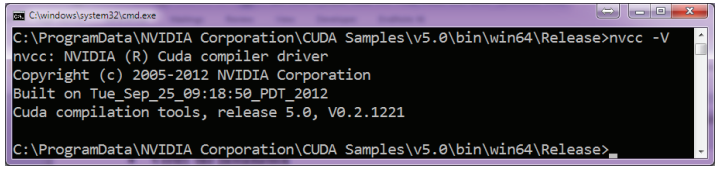
Verify the Installation

Before continuing, it is important to verify that the CUDA programs can find and communicate correctly with the CUDA-capable hardware. To do this, you need to compile and run some of the included sample programs.

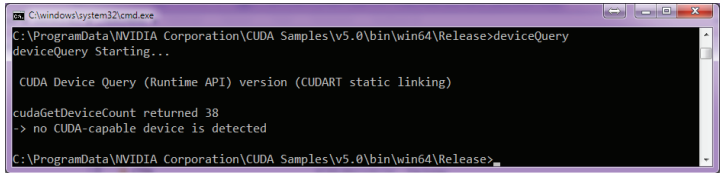
- Start Command Prompt:
  Start > All Programs > Accessories > Command Prompt
- Type cd command to move to directory ***\bin\win[64,32]\release*** in Cuda Sapmles
  For Windows Vista, Windows 7, Windows 8, Windows Server 2003, and Windows Server 2008 64 bits:
  cd C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\bin\win64\Release



- Check version of the CUDA Toolkit by running ***nvcc -V***



- Running the ***queryDevice*** program ensures that ***CUDA-capable devices are present***



No CUDA-capable devices

  - If a CUDA-capable device and the CUDA Driver are installed but deviceQuery reports that no CUDA-capable devices are present, ensure the deivce and driver are properly installed.

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\bin\win64\Release>deviceQuery.exe
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 520MX"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 512 MBytes (536870912 bytes)
  ( 1) Multiprocessors x ( 48) CUDA Cores/MP:    48 CUDA Cores
  GPU Clock rate:                                1800 MHz (1.80 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 65536 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.0, CUDA Runtime Version = 5.0, NumDevs =
1, Device0 = GeForce GT 520MX
```

Valid Results from Sample CUDA Device Query Program

- Running the ***bandwidthTest*** program ensures that the system and the CUDA-capable device are able to communicate correctly.

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\bin\win64\Release>bandwidthTest.exe
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GeForce GT 520MX
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     6304.2

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     6327.1

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     12652.7
```

Valid Results from Sample CUDA Bandwidth Test Program

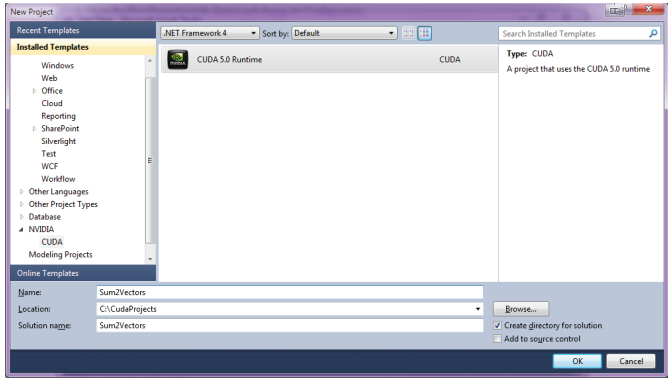
Create Cuda project from template

Step 1. Open Visual Studio 2010

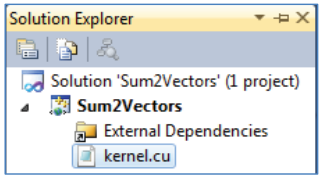
Step 2. Menu File → New → Project

- Choose NVIDIA → CUDA
- CUDA 5.0 Runtime Templates

- Fill other information:
  - Name: Sum2Vectors
  - Location: CudaProjects
  - Solution name: Sum2Vectors

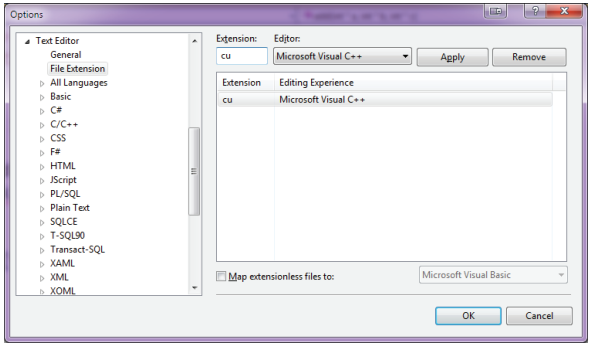


Step 3. Replace content of ***kernel.cu*** with source code below:



Step 4. To get syntax highlighting working for .cu files:

- Tools → Options → Text editor → File extension
- Type .cu in input box → Choose Microsoft Visual C++
- Click Add → Press Ok → Restart Visual Studio.



Step 5. Press Ctrl + F5 to build and run the program.

Source Code

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

#define N 10
```

```
#define CUDA_ERROR 1

__global__ void add( int *a, int *b, int *c ) {
	int tid = blockIdx.x;

	if(tid < N)
		c[tid] = a[tid] + b[tid];
}

int main( void ) {
	int count;
	cudaDeviceProp prop;

	int a[N], b[N], c[N];
	int *dev_a, *dev_b, *dev_c;

	if( cudaGetDeviceCount(&count) != cudaSuccess)
		return CUDA_ERROR;

	for(int i = 0; i < count; i++) {
		if(cudaGetDeviceProperties(&prop, i) != cudaSuccess)
			return CUDA_ERROR;

		printf("--- General Information for device %d ---\n", i);
		printf("Name: %s\n", prop.name);
		printf("Compute capability: %d.%d\n", prop.major, prop.minor);
		printf("Max threads per block: %d\n", prop.maxThreadsPerBlock);
		printf("Max thread dimensions: (%d, %d, %d)\n",
			prop.maxThreadsDim[0],
			prop.maxThreadsDim[1],
			prop.maxThreadsDim[2]);
		printf("Max grid dimensions; (%d, %d, %d)\n",
			prop.maxGridSize[0],
			prop.maxGridSize[1],
			prop.maxGridSize[2]);
	}

	cudaMalloc((void**)&dev_a, N * sizeof(int));
	cudaMalloc((void**)&dev_b, N * sizeof(int));
	cudaMalloc((void**)&dev_c, N * sizeof(int));

	printf("\n\n--- Adding 2 vectors on the GPU ---\n");
	for(int i = 0; i < N; i++) {
		a[i] = i * 2;
		b[i] = i * i;
	}

	cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
	cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice);

	add<<<N, 1>>>(dev_a, dev_b, dev_c);

	cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );

	for(int i = 0; i < N; i++) {
		printf("%d + %d = %d\n", a[i], b[i], c[i]);
	}

	cudaFree( dev_a );
	cudaFree( dev_b );
	cudaFree( dev_c );

	return 0;
}
```

Output:

```
--- General Information for device 0 ---
Name: GeForce GT 520MX
Compute capability: 2.1
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions; (65535, 65535, 65535)

--- Adding 2 vectors on the GPU ---
0 + 0 = 0
2 + 1 = 3
```



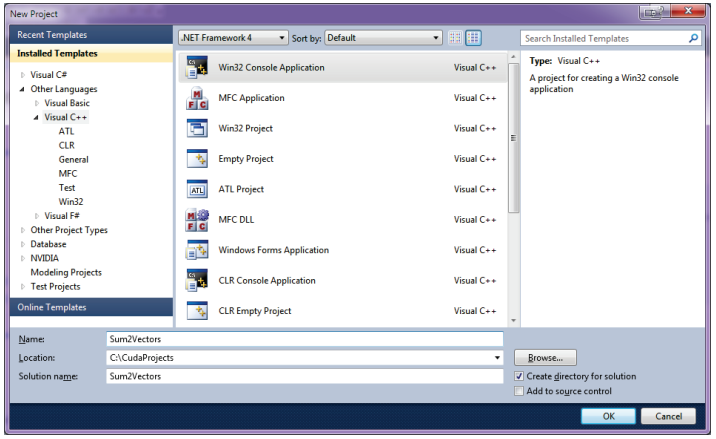
```
4 + 4 = 8
6 + 9 = 15
8 + 16 = 24
10 + 25 = 35
12 + 36 = 48
14 + 49 = 63
16 + 64 = 80
18 + 81 = 99
```

Create Cuda project from Empty Project

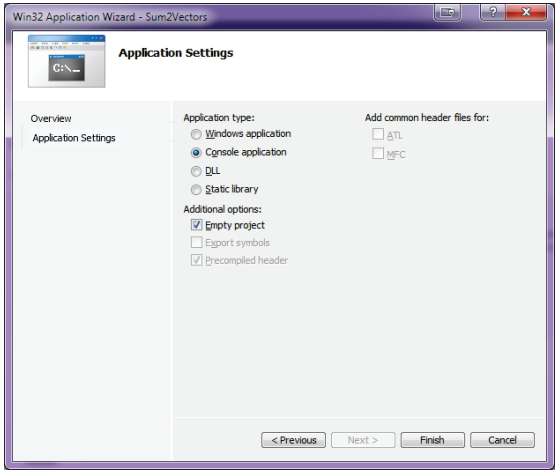
Step 1. Open Microsoft Visual Studio 2010

Step 2. Create a New Empty project:

- File → New → Project
- Select Win32 Console Application.

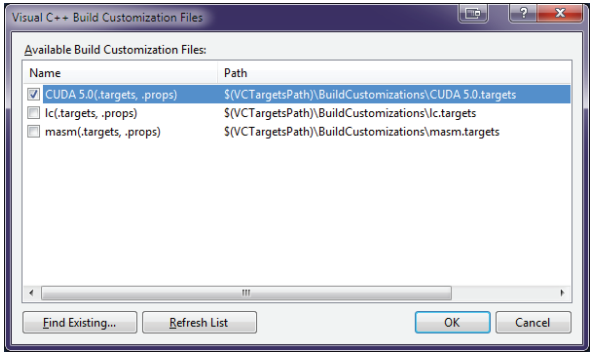


Step 3. In Win32 Application Wizard, choose Console application and Empty project. After that, press Finish button:



Step 4. Right Click on the Project at the Solution Explorer and Select ***Build Customizations…***
You should now see a list of configuration files but none of them are for CUDA. So you must click on Find Existing button and add the CUDA target files located at ***C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\BuildCustomizations***.

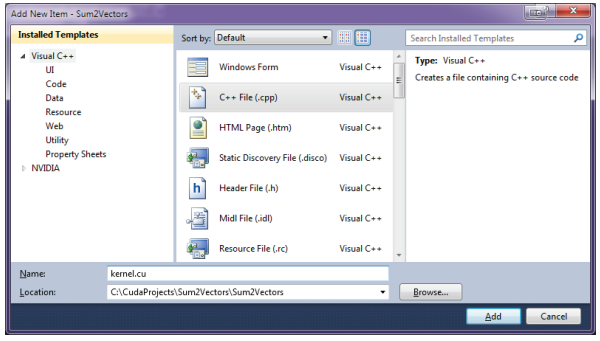
Step 5. Select one of the CUDA configuration files and click Ok.

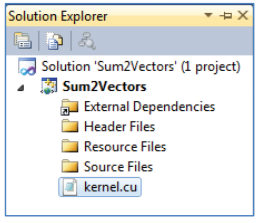


Step 6. To get syntax highlighting working for .cu files:

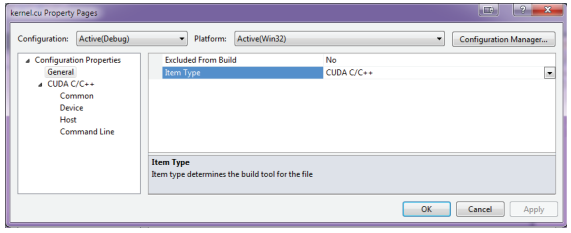
- Tools → Options → Text editor → File extension
- Type .cu in input box → Choose Microsoft Visual C++
- Click Add → Press Ok → Restart Visual Studio.

Step 7. Add a new cpp source file and change it's extension to .cu. In case, its name is ***kernel.cu***.

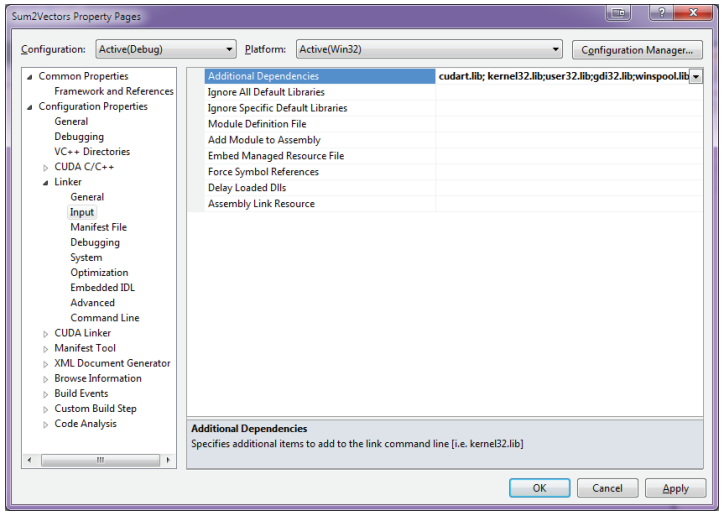




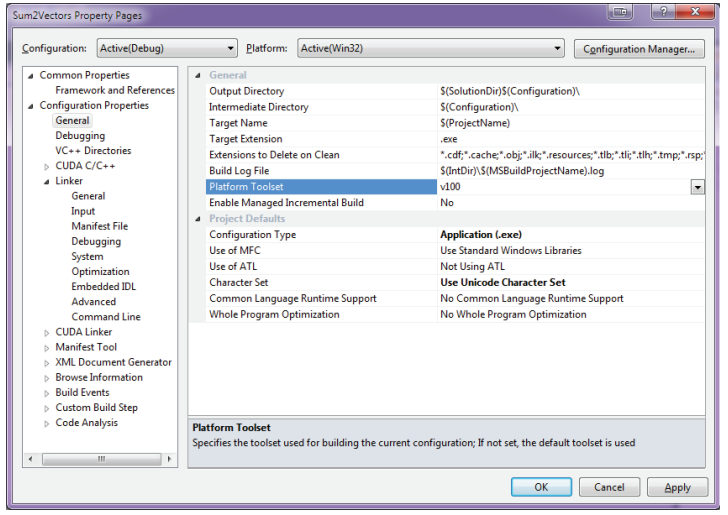
Step 8. Open the file's Properties (Alt + Enter) and change its Configuration Properties → General → Item type to CUDA C/C++.



Step 9. Now, go to the project's property page and add the following Additional Dependency cudart.lib to the Linker's Input. (Configuration Properties →Linker → Input → Additional Dependency)



Step 10. Still at the project's properties page change Configuration Properties → General → Platform Toolset to Visual Studio 2010 (v100)



Step 11. Type source code kernel.cu above, and press Ctrl + F5 to build and run programs.

LAB 01: INITIAL LAB TOUR

Objective

The purpose of this lab is to get you familiar with basic cuda function, debug cuda program, outline cuda code and program benchmark.

Exercises

Exercise 1. CudaInfo

Write a program to show the hardware information used:

- GPU card's name
- GPU computation capabilities
- Maximum number of block dimensions
- Maximum number of grid dimensions
- Maximum size of GPU memory
- Amount of constant and share memory
- Warp size

Example Console Outputs:

```
There is 1 device supporting CUDA
Device 0 name: GeForce GT 520MX
 Computational Capabilities: 2.1
 Maximum global memory size: 536870912
 Maximum constant memory size: 65536
 Maximum shared memory size per block: 49152
 Maximum block dimensions: 1024 x 1024 x 64
 Maximum grid dimensions: 65535 x 65535 x 65535
 Warp size: 32
Press any key to continue . . .
```

Ex1_CudaInfo.cpp

```
#include<iostream>
#include<cuda_runtime.h>
using namespace std;
int main(int argc, char ** argv)
{
	int deviceCount;
	cudaGetDeviceCount(&deviceCount);
	for (int dev = 0; dev < deviceCount; dev++)
	{
		cudaDeviceProp deviceProp;
		cudaGetDeviceProperties(&deviceProp, dev);
		if (dev == 0)
		{
			if (deviceProp.major == 9999 && deviceProp.minor == 9999)
			{
				cout << "No CUDA GPU has been detected"
					<< endl;
				return -1;
			}
			else if (deviceCount == 1)
			{
				cout << "There is 1 device supporting CUDA"
					<< endl;
			}
			else
			{
				cout << "There are " << deviceCount
```



```
					<< " devices supporting CUDA" << endl;
			}
		}
		cout << "Device " << dev << " name: "
			<< deviceProp.name << endl;
		cout << " Computational Capabilities: "
			<< deviceProp.major << "."
			<< deviceProp.minor << endl;
		cout << " Maximum global memory size: "
			<< deviceProp.totalGlobalMem << endl;
		cout << " Maximum constant memory size: "
			<< deviceProp.totalConstMem << endl;
		cout << " Maximum shared memory size per block: "
			<< deviceProp.sharedMemPerBlock << endl;
		cout << " Maximum block dimensions: "
			<< deviceProp.maxThreadsDim[0] << " x "
			<< deviceProp.maxThreadsDim[1] << " x "
			<< deviceProp.maxThreadsDim[2] << endl;
		cout << " Maximum grid dimensions: "
			<< deviceProp.maxGridSize[0] << " x "
			<< deviceProp.maxGridSize[1] << " x "
			<< deviceProp.maxGridSize[2] << endl;
		cout << " Warp size: " << deviceProp.warpSize << endl;
	}
	cudaDeviceReset();// called to reinitialized the device
	system("PAUSE");
	return 0;
}
```

Exercise 2. HelloWorld

Write a program to launch a kernel with grid size 3 and block size 4. The kernel prints the message "Hello, I am a thread [thread id] of block [block id] with value e=[e]\n" where a input parameter e, current thread id and current block id.

Example Console Outputs:

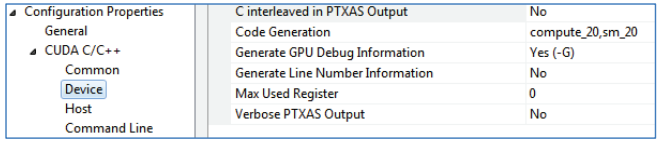
```
Hello, I am a thread 0 of block 0 with value e=2.718280
Hello, I am a thread 1 of block 0 with value e=2.718280
Hello, I am a thread 2 of block 0 with value e=2.718280
Hello, I am a thread 3 of block 0 with value e=2.718280
Hello, I am a thread 0 of block 2 with value e=2.718280
Hello, I am a thread 1 of block 2 with value e=2.718280
Hello, I am a thread 2 of block 2 with value e=2.718280
Hello, I am a thread 3 of block 2 with value e=2.718280
Hello, I am a thread 0 of block 1 with value e=2.718280
Hello, I am a thread 1 of block 1 with value e=2.718280
Hello, I am a thread 2 of block 1 with value e=2.718280
Hello, I am a thread 3 of block 1 with value e=2.718280
Press any key to continue . . .
```

Ex1_HelloWorld.cu

```
#include<cstdio>
#include<cstdlib>
#include<cuda_runtime.h>

__global__ void helloCuda(float e)
{
	printf("Hello, I am a thread %d of block %d with value e=%f\n",
		threadIdx.x, blockIdx.x, e);
}

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

```
{
	helloCuda<<<3,4>>>(2.71828f);

	cudaDeviceReset();// called to reinitialized the device
	system("PAUSE");
	return 0;
}// main
```

**** printf() is only supported for devices of compute capability 2.0 and higher. So do not forget to put -arch = compute_20 the nvcc compiler.***



Exercise 3. Utils

Write library ***utility.h*** to:

- Check cuda errors
- Benchmark time for program

utils.h

```
#include <windows.h>
#include <iostream>
#include <iomanip>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

#ifndef __UTILS_H__
#define __UTILS_H__

#define checkCudaErrors(val) check( (val), #val, __FILE__, __LINE__)

template<typename T>
void check(T err, const char* const func, const char* const file, const int line) {
	if (err != cudaSuccess) {
		std::cerr << "CUDA error at: " << file << ":" << line << std::endl;
		std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
		exit(1);
	}
}

struct GpuTimer {
	cudaEvent_t start;
	cudaEvent_t stop;

	GpuTimer() {
		cudaEventCreate(&start);
		cudaEventCreate(&stop);
	}

	~GpuTimer() {
		cudaEventDestroy(start);
		cudaEventDestroy(stop);
	}

	void Start(){
```



```
		cudaEventRecord(start, 0);
	}

	void Stop() {
		cudaEventRecord(stop, 0);
	}

    float Elapsed() {
		float elapsed;
		cudaEventSynchronize(stop);
		cudaEventElapsedTime(&elapsed, start, stop);
		return elapsed;
	}

	void StopAndPrint(std::ostream &os, const char *content) {
		this->Stop();
		os << "["<< content <<"]="
			<< std::fixed << std::setprecision(10) <<  this->Elapsed() << "
msecs. " << std::endl;
	}
};

// CudaTimer class from: https://bitbucket.org/ashwin/cudatimer
class CudaTimer {
private:
	double        _freq;
	LARGE_INTEGER _time1;
	LARGE_INTEGER _time2;

public:
	CudaTimer::CudaTimer() {
		LARGE_INTEGER freq;
		QueryPerformanceFrequency(&freq);
		_freq = 1.0 / freq.QuadPart;
		return;
	}

	void Start() {
		cudaDeviceSynchronize();
		QueryPerformanceCounter(&_time1);
		return;
	}

    void Stop() {
		cudaDeviceSynchronize();
		QueryPerformanceCounter(&_time2);
		return;
	}

    double Elapsed() const {
		return (_time2.QuadPart - _time1.QuadPart) * _freq * 1000;
	}

	void StopAndPrint(std::ostream &os, const char *content) {
		this->Stop();
		os << "["<< content <<"]="
			<< std::fixed << std::setprecision(10)
			<<  this->Elapsed() << " msecs. " << std::endl;
	}
};
#endif /* __UTILS_H__ */
```

Exercise 4. VectorAdd_11

Write a program to add two vectors.

- The host launches kernel [1,1] to solve problem.
- The program logs time of host and cuda code segment: initialize, copy, malloc, launch, free, …
- After that, write the analysis data table and graph for Total Run Time

Dataset #	Description	Result	Total Run Time	Gpu Time
1	1000	Correct	??	??
…	…	…	…	…

Example Console Outputs:

```
[Initialize data and creating memory on host]=0.5579472021 msecs.
[The number of input elements in the input is]=33792
[Allocating GPU memory]=0.4394533983 msecs.
[Copying input memory to the GPU]=0.3144284400 msecs.
[Performing CUDA computation]=8.3080950861 msecs.
[Copying output memory to the CPU]=0.3722758088 msecs.
[RET]=We did it!
[Freeing GPU Memory]=0.1740086174 msecs.
[Freeing Host Memory]=0.1866044154 msecs.
Press any key to continue . . .
```

Ex1_VectorAdd_11.cu

```
#include <iostream>
#include "utils/utils.h"
#include <cuda_runtime.h>

using namespace std;

#define N 1024 * 33

__global__ void add( int *a, int *b, int *c ) {
    int tid = 0;
    // loop over all the element in the vector
    while (tid < N){
        c[tid] = a[tid] + b[tid];
        tid += 1; // we are using one thread in one block
    }
}

int main( void ) {

    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

      CudaTimer timer;

      /* Initialize data and creating memory on host */
      timer.Start();

    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
```



```
    }

    timer.StopAndPrint(cout, "Initialize data and creating memory on host");
    cout << "[The number of input elements in the input is]=" << N << endl;

    /* Allocating GPU memory */
    timer.Start();

    checkCudaErrors( cudaMalloc( (void**)&dev_a, N * sizeof(int))  );
    checkCudaErrors(  cudaMalloc( (void**)&dev_b, N * sizeof(int)) );
    checkCudaErrors(  cudaMalloc( (void**)&dev_c, N * sizeof(int)) );

      timer.StopAndPrint(cout, "Allocating GPU memory");

    /* Copying input memory to the GPU */
    timer.Start();

    checkCudaErrors( cudaMemcpy(dev_a, a, N * sizeof(int),cudaMemcpyHostToDevice) );
    checkCudaErrors( cudaMemcpy(dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice) );

    timer.StopAndPrint(cout, "Copying input memory to the GPU");

    /* Launch the GPU Kernel */
    timer.Start();

    add<<<1,1>>>( dev_a, dev_b, dev_c );

    timer.StopAndPrint(cout, "Performing CUDA computation");

    /* Copying output memory to the CPU */
    timer.Start();

    checkCudaErrors( cudaMemcpy(c, dev_c,N*sizeof(int),cudaMemcpyDeviceToHost) );

    timer.StopAndPrint(cout, "Copying output memory to the CPU");

    // verify that the GPU did the work we requested
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "[RET]=Error:  %d + %d != %d\n", a[i], b[i], c[i] );
            success = false;
        }
    }
    if (success)  printf( "[RET]=We did it!\n" );

    /* Freeing GPU Memory */
    timer.Start();

    checkCudaErrors( cudaFree(dev_a) );
    checkCudaErrors( cudaFree(dev_b) );
    checkCudaErrors( cudaFree(dev_c) );

    timer.StopAndPrint(cout, "Freeing GPU Memory");

      /* Freeing Host Memory */
      timer.Start();

    free(a);
    free(b);
    free(c);
```

```
    timer.StopAndPrint(cout, "Freeing Host Memory");
    system("PAUSE");

    return 0;
}
```


LAB 02: VECTOR ADDITION

Objective

The purpose of this lab is to get you familiar with using the CUDA API by implementing a simple vector addition kernel and its associated setup code.

Exercise

Vector sum (c = a + b)

- Create vectors a, b, c on the host and load data a, b from file "LAB02.INP"
- Create vectors a, b, c in the device
- Copy the contents of the vectors a and b on host to device
- Initialize the grid and block dimensions, and launch kernel to add a and b on device
- Copy the result on device to the host
- Write result vector c on host into file "LAB02.OUT"
- Free Device and Host Memory

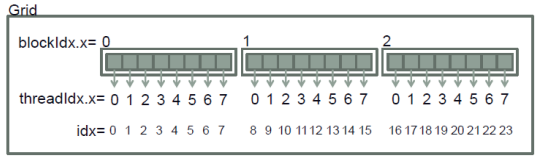
Sample

LAB02.INP	LAB02.OUT
4 1 2 3 4 5 6 7 8	4 6 8 10 12

Report

Dataset #	Description	Result	Total Run Time	Gpu Time
…	…	…	…	…

Idea



Console Output Samples

```
[Importing data and creating memory on host]=3803.5237444788 msecs.
[The input length]=100000
[Allocating GPU memory]=1.6509825655 msecs.
[Copying input memory to the GPU]=0.4613794172 msecs.
[Performing CUDA computation]=0.2290239930 msecs.
[Copying output memory to the CPU]=0.7622790370 msecs.
[Freeing GPU Memory]=0.2234587875 msecs.
[Output results]=649.8741353218 msecs.
[Freeing Host Memory]=0.4263910893 msecs.
Press any key to continue . . .
```

Exercise 1. GenVectorAdd

Write a program to generate data set for problem "vector addition"

Ex2_GenVectorAdd.cpp

```
/*
ex2_gen_vector_add.cpp:
```

```
Generate data for assignment ex2_vector_add.cu
*/
#include <cassert>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <vector>

typedef std::vector< float > FloatVec;

float genRandomFloat()
{
    return ( (float) rand() / RAND_MAX );
}

void genVector( FloatVec& vec, int vecLen )
{
    for ( int i = 0; i < vecLen; ++i )
        vec.push_back( genRandomFloat() );
}

void addVector( const FloatVec& vecA, const FloatVec& vecB, FloatVec& vecC )
{
    assert( vecA.size() == vecB.size() );

    for ( int i = 0; i < (int) vecA.size(); ++i )
        vecC.push_back( vecA[i] + vecB[i] );
}

void writeVector( const FloatVec& vec, const char* fname )
{
    std::ofstream outFile( fname );

    if ( !outFile )
    {
        std::cout << "Error! Opening file: " << fname << " for writing vector.\n";
        exit(1);
    }

    std::cout << "Writing vector to file: " << fname << std::endl;

    const int vecLen = (int) vec.size();

    outFile << vecLen << std::endl;

    for ( int i = 0; i < vecLen; ++i )
        outFile << vec[i] << " ";
       outFile << std::endl;
}

void writeResults( const FloatVec& vec1, const FloatVec& vec2, const char* fname )
{
    std::ofstream outFile( fname );

    if ( !outFile )
    {
        std::cout << "Error! Opening file: " << fname << " for writing vector.\n";
        exit(1);
    }

    std::cout << "Writing vector to file: " << fname << std::endl;
```



```
    const int vecLen = (int) vec1.size();

    outFile << vecLen << std::endl;

    for ( int i = 0; i < vecLen; ++i )
        outFile << vec1[i] << " ";
       outFile << std::endl;
       for ( int i = 0; i < vecLen; ++i )
        outFile << vec2[i] << " ";
       outFile << std::endl;
}

int main( int argc, const char** argv )
{
    // Info for user

    std::cout << "ex2_gen_vector_add: Generates data files to use as input for assignment
ex2_vector_add.\n";
    std::cout << "Invoke as: ex2_gen_vector_add [VectorLength]\n\n";

    // Read input

    if ( 2 != argc )
    {
        std::cout << "Error! Wrong number of arguments to program.\n";
        return 0;
    }

    // Create vectors

    const int vecLen = atoi( argv[1] );

    FloatVec vecA;
    FloatVec vecB;
    FloatVec vecC;

    genVector( vecA, vecLen );
    genVector( vecB, vecLen );
    addVector( vecA, vecB, vecC );

    // Write to files

    writeResults( vecA, vecB, "LAB02.INP" );
       writeVector ( vecC, "LAB02.TXT" );
    return 0;
}
```

Exercise 2. VectorAdd_NoCuda

Write a program to calculate sum of two vector without Cuda

Ex2_VectorAdd_NoCuda

```
#include <iostream>
#include <fstream>
#include "utils/utils.h"

#define INPUT_FILE "LAB02.INP"
#define OUTPUT_FILE "LAB02.OUT"

using namespace std;

void vecAdd(float * in1, float * in2, float * out, int len) {
       // TODO
       int idx = 0;
```

```
	while (idx<len)
	{
		out[idx] = in1[idx] + in2[idx];
		idx++;
	}
}

int main(int argc, char ** argv)
{
	int inputLength;
	float *hostInput1, *hostInput2, *hostOutput;

	CudaTimer timer;

	/* Importing data and creating memory on host */
	ifstream finp(INPUT_FILE);
	timer.Start();

	// TODO
	finp >> inputLength;
	int byteSize = sizeof(float) * inputLength;
	hostInput1 = (float *)malloc(byteSize);
	hostInput2 = (float *)malloc(byteSize);
	hostOutput = (float *)malloc(byteSize);
	for (int i = 0; i< inputLength; i++)
	{
		finp >> hostInput1[i];
	}
	for (int i = 0; i< inputLength; i++)
	{
		finp >> hostInput2[i];
	}
	timer.StopAndPrint(cout, "Importing data and creating memory on host");
	cout << "[The input length]=" << inputLength << endl;
	finp.close();

	/* Launch the GPU Kernel */
	timer.Start();
	// TODO
	vecAdd(hostInput1, hostInput2, hostOutput, inputLength);

	timer.StopAndPrint(cout, "Performing CPU computation");

	/* Output results */
	ofstream fout(OUTPUT_FILE);
	timer.Start();
	// TODO
	fout << inputLength << endl;
	for (int i = 0; i<inputLength; i++)
	{
		fout << hostOutput[i] << " ";
	}
	fout << endl;

	timer.StopAndPrint(cout, "Output results");
	fout.close();

	/* Freeing Host Memory */
	timer.Start();

	free(hostInput1);
	free(hostInput2);
	free(hostOutput);
```



```
	timer.StopAndPrint(cout, "Freeing Host Memory");

	system("PAUSE");

	return 0;
}
```

Output

```
[Importing data and creating memory on host]=3852.7010056112 msecs.
[The input length]=100000
[Performing CPU computation]=0.5052314548 msecs.
[Output results]=666.8961900977 msecs.
[Freeing Host Memory]=0.3657446543 msecs.
Press any key to continue . . .
```

Exercise 3. VectorAdd

Write a program to calculate sum of two vector using Cuda

Ex2_VectorAdd.cu

```
#include <iostream>
#include <fstream>
#include "utils/utils.h"
#include <cuda_runtime.h>

#define INPUT_FILE "LAB02.INP"
#define OUTPUT_FILE "LAB02.OUT"

using namespace std;

__global__ void vecAdd(float * in1, float * in2, float * out, int len) {
    // TODO
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx<len)
	{
		out[idx] = in1[idx] + in2[idx];
		// DEBUG
		// printf("[gsize=%d,bsize=%d,b_id=%d,th_id=%d]: %d: [%g]+[%g]=%g\n",
		//		gridDim.x, blockDim.x, blockIdx.x, threadIdx.x,
		//		idx, in1[idx], in2[idx], out[idx]);
	}
}

int main(int argc, char ** argv)
{
	int inputLength;
    float *hostInput1, *hostInput2, *hostOutput;
    float *deviceInput1, *deviceInput2, *deviceOutput;

	CudaTimer timer;
	GpuTimer gpuTimer;

	/* Importing data and creating memory on host */
	ifstream finp(INPUT_FILE);
	timer.Start();

	// TODO
	finp >> inputLength;
	int byteSize = sizeof(float) * inputLength;
	hostInput1 = (float *)malloc(byteSize);
	hostInput2 = (float *)malloc(byteSize);
	hostOutput = (float *)malloc(byteSize);
```

```
	for(int i=0; i< inputLength; i++)
	{
		finp >> hostInput1[i];
	}
	for(int i=0; i< inputLength; i++)
	{
		finp >> hostInput2[i];
	}
	timer.StopAndPrint(cout, "Importing data and creating memory on host");
	cout << "[The input length]=" << inputLength << endl;
	finp.close();


	/* Allocating GPU memory */
	timer.Start();
	// TODO
	cudaMalloc((void **) &deviceInput1, byteSize);
cudaMalloc((void **) &deviceInput2, byteSize);
	cudaMalloc((void **) &deviceOutput, byteSize);

	timer.StopAndPrint(cout,"Allocating GPU memory");


	/* Copying input memory to the GPU */
	timer.Start();
	// TODO
	cudaMemcpy(deviceInput1, hostInput1, byteSize,cudaMemcpyHostToDevice);
cudaMemcpy(deviceInput2, hostInput2, byteSize,cudaMemcpyHostToDevice);

	timer.StopAndPrint(cout, "Copying input memory to the GPU");

	/* Initialize the grid and block dimensions */
// TODO
	int block_size = 1024;
int n_blocks = inputLength /block_size + (inputLength%block_size == 0 ? 0:1);

	/* Launch the GPU Kernel */
	gpuTimer.Start();
	// TODO
	vecAdd<<<n_blocks, block_size >>>(deviceInput1, deviceInput2, deviceOutput, inputLength);

	cudaThreadSynchronize();
	gpuTimer.StopAndPrint(cout, "Performing CUDA computation");

	/* Copying output memory to the CPU */
	timer.Start();
	// TODO
	cudaMemcpy(hostOutput, deviceOutput, byteSize,cudaMemcpyDeviceToHost);

	timer.StopAndPrint(cout, "Copying output memory to the CPU");

	/* Freeing GPU Memory */
	timer.Start();
	// TODO
	cudaFree(deviceInput1);
	cudaFree(deviceInput2);
	cudaFree(deviceOutput);

	timer.StopAndPrint(cout, "Freeing GPU Memory");

	/* Output results */
	ofstream fout(OUTPUT_FILE);
	timer.Start();
```



```
	// TODO
	fout << inputLength << endl;
	for(int i=0; i<inputLength; i++)
	{
		fout << hostOutput[i] << " ";
	}
	fout << endl;

	timer.StopAndPrint(cout, "Output results");
	fout.close();

	/* Freeing Host Memory */
	timer.Start();

	free(hostInput1);
	free(hostInput2);
	free(hostOutput);

	timer.StopAndPrint(cout, "Freeing Host Memory");

	system("PAUSE");

    return 0;
}
```

Output

```
[Importing data and creating memory on host]=3803.5237444788 msecs.
[The input length]=100000
[Allocating GPU memory]=1.6509825655 msecs.
[Copying input memory to the GPU]=0.4613794172 msecs.
[Performing CUDA computation]=0.2290239930 msecs.
[Copying output memory to the CPU]=0.7622790370 msecs.
[Freeing GPU Memory]=0.2234587875 msecs.
[Output results]=649.8741353218 msecs.
[Freeing Host Memory]=0.4263910893 msecs.
Press any key to continue . . .
```

LAB 03: BASIC MATRIX-MATRIX MULTIPLICATION

Objective

Implement a basic dense matrix multiplication routine.

Exercise

Dense Matrix Multiplication (c = a * b)

- Create matrix a, b, c on the host and load data a, b from file "LAB03.INP"
- Create matrix a, b, c in the device
- Copy the contents of the matrix a and b on host to device
- Initialize the grid and block dimensions, and launch kernel to multiply a and b on device
- Copy the result on device to the host
- Write result matrix c on host into file "LAB03.OUT"
- Free Device and Host Memory

Sample

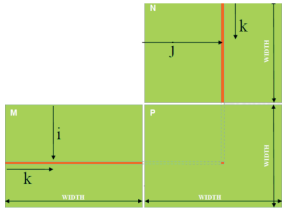
LAB03.INP	LAB03.OUT
2 3 0.00125126 0.563585 0.193304 0.80874 0.585009 0.479873 3 4 0.350291 0.895962 0.82284 0.746605 0.174108 0.858943 0.710501 0.513535 0.303995 0.0149846 0.0914029 0.364452	2 4 0.157327 0.488106 0.419126 0.360805 0.531029 1.23428 1.12498 1.07912

Report

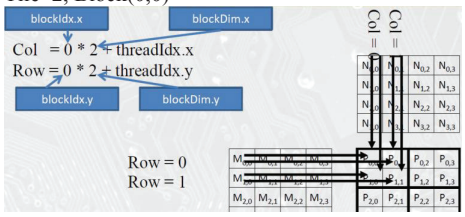
Dataset #	Description	Result	Total Run Time	Gpu Time
…	…	…	…	…

Idea

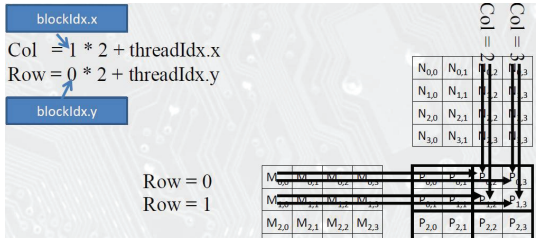
Matrix multiplication



Tile=2, Block(0,0)



Tile=2, Block(0,1)



Console Output Samples

```
[Importing data and creating memory on host]=884.91452584750 msecs.
[The dimensions of A]=200 x 100
[The dimensions of B]=100 x 256
[The dimensions of C]=200 x 256
[Allocating GPU memory]=1.2577137600 msecs.
[Copying input memory to the GPU]=0.1651449077 msecs.
[Performing CUDA computation]=3.6639039516 msecs.
[Copying output memory to the CPU]=0.1506830655 msecs.
[Freeing GPU Memory]=0.1768076836 msecs.
[Output results]=348.1735159817 msecs.
[Freeing Host Memory]=0.4119292471 msecs.
Press any key to continue . . .
```

Exercise 1. GenMatrixDenseMul

Write a program to generate data set for problem "dense matrix multiplication"

Ex3_GenMatrixDenseMul.cpp

```
/*
ex3_gen_matrix_dense_mul.cpp:
Generate data for assignment ex3_matrix_dense_mul.cu
*/
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>

float genRandomFloat()
{
    return ( (float) rand() / RAND_MAX );
}

void genMatrix( float* mat, int rows, int cols )
{
    for ( int r = 0; r < rows; ++r )
        for ( int c = 0; c < cols; ++c )
            mat[ cols * r + c ] = genRandomFloat();
}

void mulMatrices
(
       const float* matA,
       const float* matB,
       float*       matC,
       int ARows,
       int ACols,
       int BCols
)
{
    const int CRows = ARows;
    const int CCols = BCols;

    for ( int r = 0; r < CRows; ++r )
    {
        for ( int c = 0; c < CCols; ++c )
        {
            float sum = 0.0;

            for ( int z = 0; z < ACols; ++z )
                sum += matA[ ACols * r + z ] * matB[ BCols * z + c ];

            matC[ CCols * r + c ] = sum;
```

```
        }
    }
}

void writeMatrix( const float* mat, int rows, int cols, const char* fname )
{
    std::ofstream outFile( fname );

    if ( !outFile )
    {
        std::cout << "Error! Opening file: " << fname << " for writing matrix\n";
        exit(1);
    }

    std::cout << "Writing matrix to file: " << fname << std::endl;

    outFile << rows << " " << cols << std::endl;

       int idx = 0;

    for ( int r = 0; r < rows; ++r )
    {
        for ( int c = 0; c < cols; ++c )
        {
            outFile << mat[ idx++ ] << " ";
        }

        outFile << std::endl;
    }
}

void writeInput(
       const float* matA, int rowsA, int colsA,
       const float* matB, int rowsB, int colsB,
       const char* fname
)
{
       std::ofstream outFile( fname );

    if ( !outFile )
    {
        std::cout << "Error! Opening file: " << fname << " for writing matrix\n";
        exit(1);
    }

    std::cout << "Writing matrix to file: " << fname << std::endl;

    outFile << rowsA << " " << colsA << std::endl;
    int idx = 0;
    for ( int r = 0; r < rowsA; ++r )
    {
        for ( int c = 0; c < colsA; ++c )
        {
            outFile << matA[ idx++ ] << " ";
        }
        outFile << std::endl;
    }

       outFile << rowsB << " " << colsB << std::endl;
       idx = 0;
    for ( int r = 0; r < rowsB; ++r )
    {
        for ( int c = 0; c < colsB; ++c )
```



```
        {
            outFile << matB[ idx++ ] << " ";
        }
        outFile << std::endl;
    }
}

int main( int argc, const char** argv )
{
    // Info for user

    std::cout << "GenDataMP2: Generates data files to use as input for assignment MP2.\n";
    std::cout << "Invoke as: GenDataMP2 [MatrixARows] [MatrixAColumns] [MatrixBColumns]\n\n";

    std::cout << "Datasets used in online submission are ...\n";
    std::cout << "Dataset0:  64  64  64\n";
    std::cout << "Dataset1: 128  64 128\n";
    std::cout << "Dataset2: 100 128  56\n";
    std::cout << "Dataset3: 256 128 256\n";
    std::cout << "Dataset4:  32 128  32\n";
    std::cout << "Dataset5: 200 100 256\n\n";

    // Read input

    if ( 4 != argc )
    {
        std::cout << "Error! Wrong number of arguments to program.\n";
        return 0;
    }

    const int ARows = atoi( argv[1] );
    const int ACols = atoi( argv[2] );
    const int BRows = ACols;
    const int BCols = atoi( argv[3] );
    const int CRows = ARows;
    const int CCols = BCols;

    std::cout << "Dimensions of matrix A = [" << ARows << " x " << ACols << "]\n";
    std::cout << "Dimensions of matrix B = [" << BRows << " x " << BCols << "]\n";
    std::cout << "Dimensions of matrix C = [" << CRows << " x " << CCols << "]\n";

    // Memory for matrices

    float* matA = new float[ ARows * ACols ];
    float* matB = new float[ BRows * BCols ];
    float* matC = new float[ CRows * CCols ];

    // Create matrices

    genMatrix( matA, ARows, ACols );
    genMatrix( matB, BRows, BCols );
    mulMatrices( matA, matB, matC, ARows, ACols, BCols );

    // Write to files

    writeInput( matA, ARows, ACols, matB, BRows, BCols, "LAB03.INP" );
    writeMatrix( matC, CRows, CCols, "LAB03.TXT" );

    return 0;
}
```

Exercise 2. MatrixDenseMul_NoCuda

Write a program to calculate multiplication of two dense matrixes without Cuda

Ex3_MatrixDenseMul_NoCuda.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "utils/utils.h"
#include <cuda_runtime.h>

#define INPUT_FILE "LAB03.INP"
#define OUTPUT_FILE "LAB03.OUT"

using namespace std;

#define THREADS 32
#define MAX_BLOCKS(size) \
	((size-1)/THREADS + 1)

// Compute C = A * B
void matrixMultiply(float * A, float * B, float * C,
	int numARows, int numAColumns,
	int numBRows, int numBColumns,
	int numCRows, int numCColumns)
{
	// TODO
	int row, col, k;

	for (row = 0; row < numCRows; row++)
	{
		for (col = 0; col < numCColumns; col++)
		{
			float value = 0;
			for (k = 0; k < numAColumns; ++k)
			{
				value += A[row * numAColumns + k] * B[k * numBColumns + col];
			}
			C[row * numCColumns + col] = value;
		}
	}
}

int main(int argc, char ** argv)
{
	float *hostA, *hostB, *hostC; // The A, B, and C (output) matrix
	int numARows; // number of rows in the matrix A
	int numAColumns; // number of columns in the matrix A
	int numBRows; // number of rows in the matrix B
	int numBColumns; // number of columns in the matrix B
	int numCRows; // number of rows in the matrix C (you have to set this)
	int numCColumns; // number of columns in the matrix C (you have to set this)

	int i, j;
	CudaTimer timer;

	/* Importing data and creating memory on host */
	ifstream finp(INPUT_FILE);
	timer.Start();
	// TODO
	finp >> numARows >> numAColumns;

	int byteSizeA = sizeof(float) * numARows * numAColumns;
	hostA = (float *)malloc(byteSizeA);
	for (i = 0; i < numARows; i++)
	{
		for (j = 0; j < numAColumns; j++)
```



```
		{
			finp >> hostA[i*numAColumns + j];
		}
	}

	finp >> numBRows >> numBColumns;
	int byteSizeB = sizeof(float) * numBRows * numBColumns;
	hostB = (float *)malloc(byteSizeB);
	for (i = 0; i < numBRows; i++)
	{
		for (j = 0; j < numBColumns; j++)
		{
			finp >> hostB[i*numBColumns + j];
		}
	}
	// Set numCRows and numCColumns
	// TODO
	numCRows = numARows;
	numCColumns = numBColumns;

	// Allocate the hostC matrix
	// TODO
	int byteSizeC = sizeof(float) * numCRows * numCColumns;
	hostC = (float *)malloc(byteSizeC);

	timer.StopAndPrint(cout, "Importing data and creating memory on host");

	cout << "[The dimensions of A]=" << numARows << " x " << numAColumns << endl;
	cout << "[The dimensions of B]=" << numBRows << " x " << numBColumns << endl;
	cout << "[The dimensions of C]=" << numCRows << " x " << numCColumns << endl;

	/* Launch the CPU Kernel */
	timer.Start();
	// TODO
	matrixMultiply(hostA, hostB, hostC,
		numARows, numAColumns,
		numBRows, numBColumns,
		numCRows, numCColumns);

	timer.StopAndPrint(cout, "Performing CPU computation");

	/* Output results */
	ofstream fout(OUTPUT_FILE);
	timer.Start();
	// TODO
	fout << numCRows << " " << numCColumns << endl;
	for (i = 0; i < numCRows; i++)
	{
		for (j = 0; j < numCColumns; j++)
		{
			fout << hostC[i * numCColumns + j] << " ";
		}
		fout << endl;
	}

	timer.StopAndPrint(cout, "Output results");
	fout.close();

	/* Freeing Host Memory */
	timer.Start();

	free(hostA);
	free(hostB);
```

```
	free(hostC);

	timer.StopAndPrint(cout, "Freeing Host Memory");

	system("PAUSE");

	return 0;
}
```

Output

```
[Importing data and creating memory on host]=899.7868977576 msecs.
[The dimensions of A]=200 x 100
[The dimensions of B]=100 x 256
[The dimensions of C]=200 x 256
[Performing CPU computation]=27.4121886272 msecs.
[Output results]=386.2818697016 msecs.
[Freeing Host Memory]=0.2537820050 msecs.
Press any key to continue . . .
```

Exercise 3. MatrixDenseMul

Write a program to calculate multiplication of two dense matrixes using Cuda

Ex3_MatrixDenseMul.cu

```
#include <iostream>
#include <fstream>
#include <string>
#include "utils/utils.h"
#include <cuda_runtime.h>

#define INPUT_FILE "LAB03.INP"
#define OUTPUT_FILE "LAB03.OUT"

using namespace std;

#define THREADS 32
#define MAX_BLOCKS(size) \
	((size-1)/THREADS + 1)

// Compute C = A * B
__global__ void matrixMultiply(float * A, float * B, float * C,
                               int numARows, int numAColumns,
                               int numBRows, int numBColumns,
                               int numCRows, int numCColumns)
{
    // TODO
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;

	if ((row < numCRows) && (col < numCColumns))
	{
		float value = 0;
#pragma unroll
		for (int k = 0; k < numAColumns; ++k)
			value += A[row * numAColumns + k] * B[k * numBColumns + col];
		C[row * numCColumns + col] = value;
	}
}

int main(int argc, char ** argv)
{
    float *hostA, *hostB, *hostC; // The A, B, and C (output) matrix
    float *deviceA, *deviceB, *deviceC;
```



```
    int numARows; // number of rows in the matrix A
    int numAColumns; // number of columns in the matrix A
    int numBRows; // number of rows in the matrix B
    int numBColumns; // number of columns in the matrix B
    int numCRows; // number of rows in the matrix C (you have to set this)
    int numCColumns; // number of columns in the matrix C (you have to set this)

	int i, j;
	CudaTimer timer;
	GpuTimer gpuTimer;

	/* Importing data and creating memory on host */
	ifstream finp(INPUT_FILE);
	timer.Start();
	// TODO
	finp >> numARows >> numAColumns;

	int byteSizeA = sizeof(float) * numARows * numAColumns;
	checkCudaErrors( cudaMallocHost(&hostA, byteSizeA) );
	for(i=0; i < numARows; i++)
	{
		for(j=0; j < numAColumns; j++)
		{
			finp >> hostA[i*numAColumns + j];
		}
	}

	finp >> numBRows >> numBColumns;
	int byteSizeB = sizeof(float) * numBRows * numBColumns;
	checkCudaErrors( cudaMallocHost(&hostB, byteSizeB) );
	for(i=0; i < numBRows; i++)
	{
		for(j=0; j < numBColumns; j++)
		{
			finp >> hostB[i*numBColumns + j];
		}
	}
	// Set numCRows and numCColumns
	// TODO
	numCRows = numARows;
    numCColumns = numBColumns;

	// Allocate the hostC matrix
	// TODO
	int byteSizeC = sizeof(float) * numCRows * numCColumns;
	checkCudaErrors( cudaMallocHost(&hostC,byteSizeC) );

	timer.StopAndPrint(cout, "Importing data and creating memory on host");

	cout << "[The dimensions of A]=" << numARows << " x " << numAColumns << endl;
    cout << "[The dimensions of B]=" << numBRows << " x " << numBColumns << endl;
	cout << "[The dimensions of C]=" << numCRows << " x " << numCColumns << endl;

	/* Allocating GPU memory */
	timer.Start();
    // TODO
	checkCudaErrors( cudaMalloc(&deviceA, byteSizeA) );
    checkCudaErrors( cudaMalloc(&deviceB, byteSizeB) );
    checkCudaErrors( cudaMalloc(&deviceC, byteSizeC) );

	timer.StopAndPrint(cout, "Allocating GPU memory");

	/* Copying input memory to the GPU */
```

```
	timer.Start();
// TODO
	checkCudaErrors( cudaMemcpy(deviceA, hostA, byteSizeA, cudaMemcpyHostToDevice) );
checkCudaErrors( cudaMemcpy(deviceB, hostB, byteSizeB, cudaMemcpyHostToDevice) );

	timer.StopAndPrint(cout, "Copying input memory to the GPU");

/* Initialize the grid and block dimensions */
// TODO
dim3 block_size(THREADS, THREADS);
	dim3 grid_size(MAX_BLOCKS(numCColumns), MAX_BLOCKS(numCRows));

	/* Launch the GPU Kernel */
	gpuTimer.Start();
	// TODO
	matrixMultiply<<<grid_size, block_size>>>(deviceA, deviceB, deviceC,
	                                numARows, numAColumns,
	                                numBRows, numBColumns,
	                                numCRows, numCColumns);


cudaThreadSynchronize();
	gpuTimer.StopAndPrint(cout, "Performing CUDA computation");

	/* Copying output memory to the CPU */
	timer.Start();
// TODO
	checkCudaErrors( cudaMemcpy(hostC, deviceC, byteSizeC, cudaMemcpyDeviceToHost) );

	timer.StopAndPrint(cout, "Copying output memory to the CPU");

	/* Freeing GPU Memory */
	timer.Start();
// TODO
	checkCudaErrors( cudaFree(deviceA) );
checkCudaErrors( cudaFree(deviceB) );
checkCudaErrors( cudaFree(deviceC) );


	timer.StopAndPrint(cout, "Freeing GPU Memory");

	/* Output results */
	ofstream fout(OUTPUT_FILE);
	timer.Start();
	// TODO
	fout << numCRows << " " << numCColumns << endl;
	for(i=0;i < numCRows; i++)
	{
		for(j=0;j < numCColumns; j++)
		{
			fout << hostC[i * numCColumns + j] << " ";
		}
		fout << endl;
	}

	timer.StopAndPrint(cout, "Output results");
	fout.close();

	/* Freeing Host Memory */
	timer.Start();

	cudaFreeHost(hostA);
	cudaFreeHost(hostB);
```



```
	cudaFreeHost(hostC);

	timer.StopAndPrint(cout, "Freeing Host Memory");

	system("PAUSE");

    return 0;
}
```

Output

```
[Importing data and creating memory on host]=884.9145258475 msecs.
[The dimensions of A]=200 x 100
[The dimensions of B]=100 x 256
[The dimensions of C]=200 x 256
[Allocating GPU memory]=1.2577137600 msecs.
[Copying input memory to the GPU]=0.1651449077 msecs.
[Performing CUDA computation]=3.6639039516 msecs.
[Copying output memory to the CPU]=0.1506830655 msecs.
[Freeing GPU Memory]=0.1768076836 msecs.
[Output results]=348.1735159817 msecs.
[Freeing Host Memory]=0.4119292471 msecs.
Press any key to continue . . .
```

--- The End ---