CUDA PROGRAMMING

Comp3031 Lab 10
Fall 2014

JIA Xiaoying, SU Zhiyang

Lab content

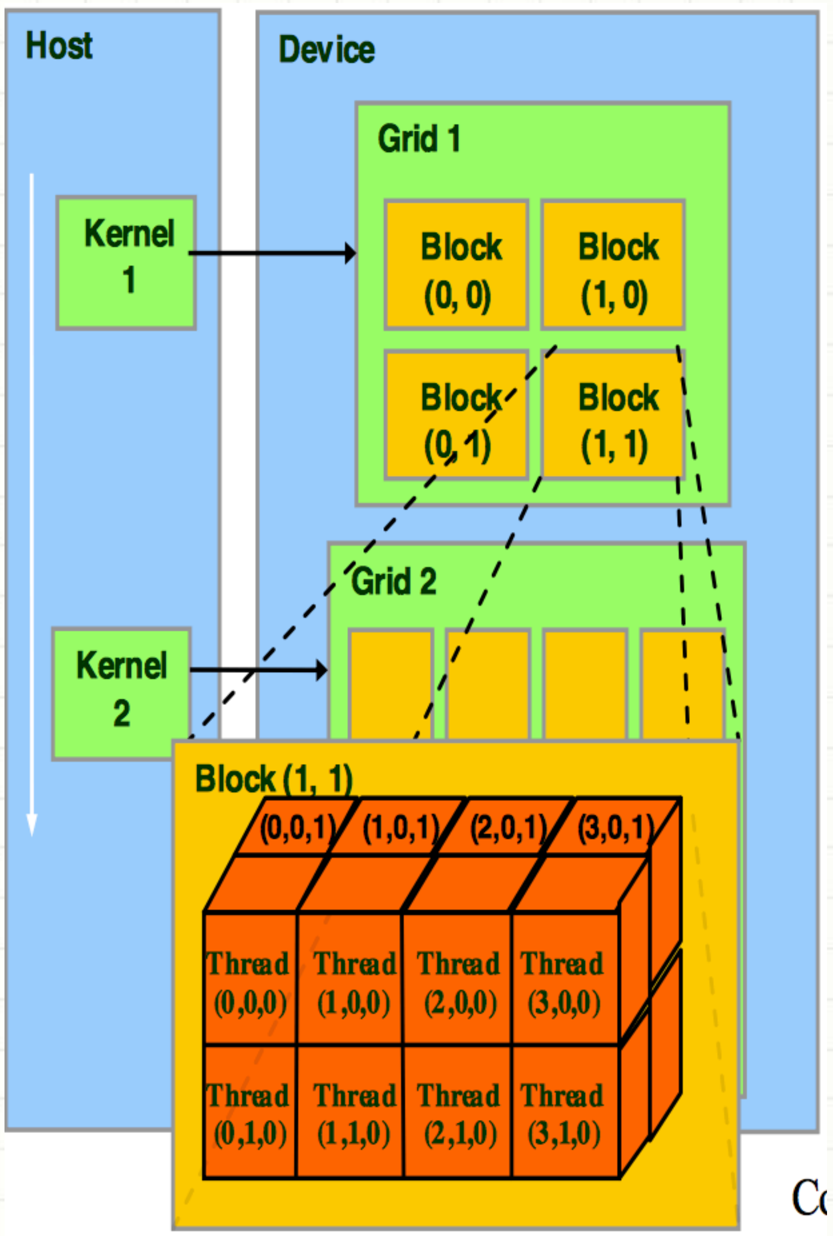
After this lab, you should know:

- How to develop, compile, and test a simple CUDA program in Microsoft Visual Studio

CUDA

- stands for Compute Unified Device Architecture.
- A parallel computing platform on NVIDIA GPUs.
- supports C with NVIDIA extensions and restriction

Thread Structure in a CUDA program (kernel)



Co

CUDA Memory

- Each GPU owns several GBs of device memory, called **global memory**. All threads can access global memory space.
- There is a small piece of on-chip memory shared within each block of threads, called **shared memory**.
- Global memory is large, but slow; whereas shared memory is small but fast

CUDA Environment in Visual Studio

- We have already installed CUDA SDK 6.0 in our lab machine. Its path is

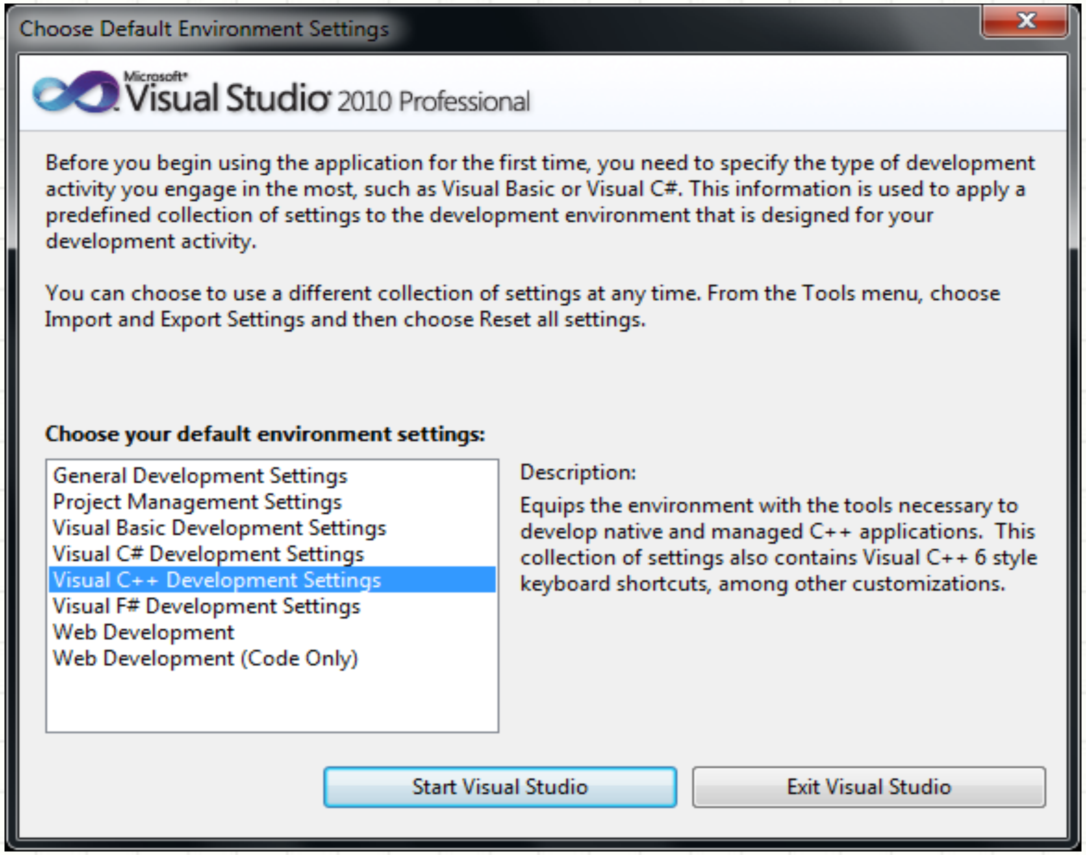
  C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA

- A VS2010 plug-in is provided to help create CUDA project and source files

- Next we will show how to build and run a simple CUDA program example of matrix multiplication in VS2010.

A simple program setup

- Firstly, start Visual Studio 2010 from Start Menu, and select "Visual C++ Development Setting".


Choose Default Environment Settings
Microsoft Visual Studio 2010 Professional

Before you begin using the application for the first time, you need to specify the type of development activity you engage in the most, such as Visual Basic or Visual C#. This information is used to apply a predefined collection of settings to the development environment that is designed for your development activity.

You can choose to use a different collection of settings at any time. From the Tools menu, choose Import and Export Settings and then choose Reset all settings.

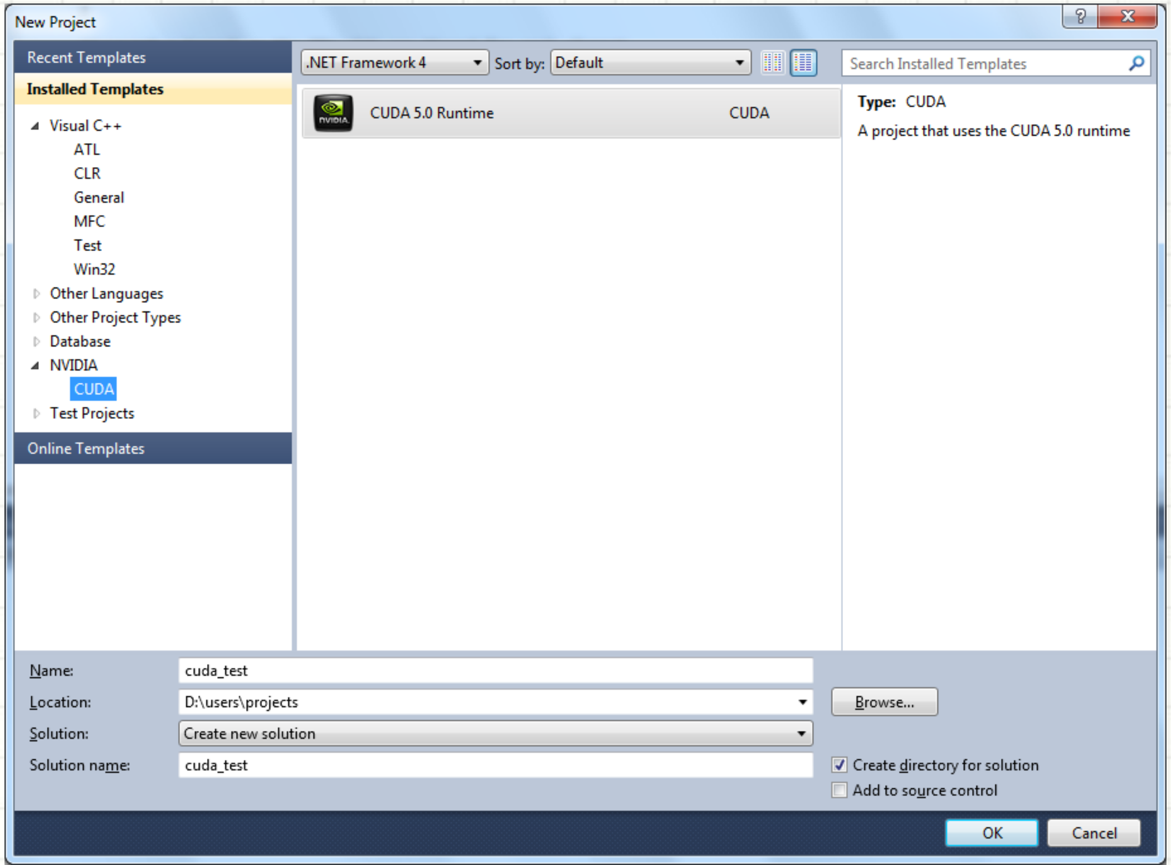
Choose your default environment settings:

- General Development Settings
- Project Management Settings
- Visual Basic Development Settings
- Visual C# Development Settings
- Visual C++ Development Settings
- Visual F# Development Settings
- Web Development
- Web Development (Code Only)

Description:
Equips the environment with the tools necessary to develop native and managed C++ applications. This collection of settings also contains Visual C++ 6 style keyboard shortcuts, among other customizations.

A simple program setup (cont.)

- Next, create an NVIDIA CUDA Runtime project under certain directory, e.g. D:\users\projects.



New Project

Installed Templates: Visual C++, NVIDIA → CUDA

.NET Framework 4 — Sort by: Default

CUDA 5.0 Runtime — CUDA

Type: CUDA

A project that uses the CUDA 5.0 runtime

Name: cuda_test

Location: D:\users\projects

Solution: Create new solution

Solution name: cuda_test

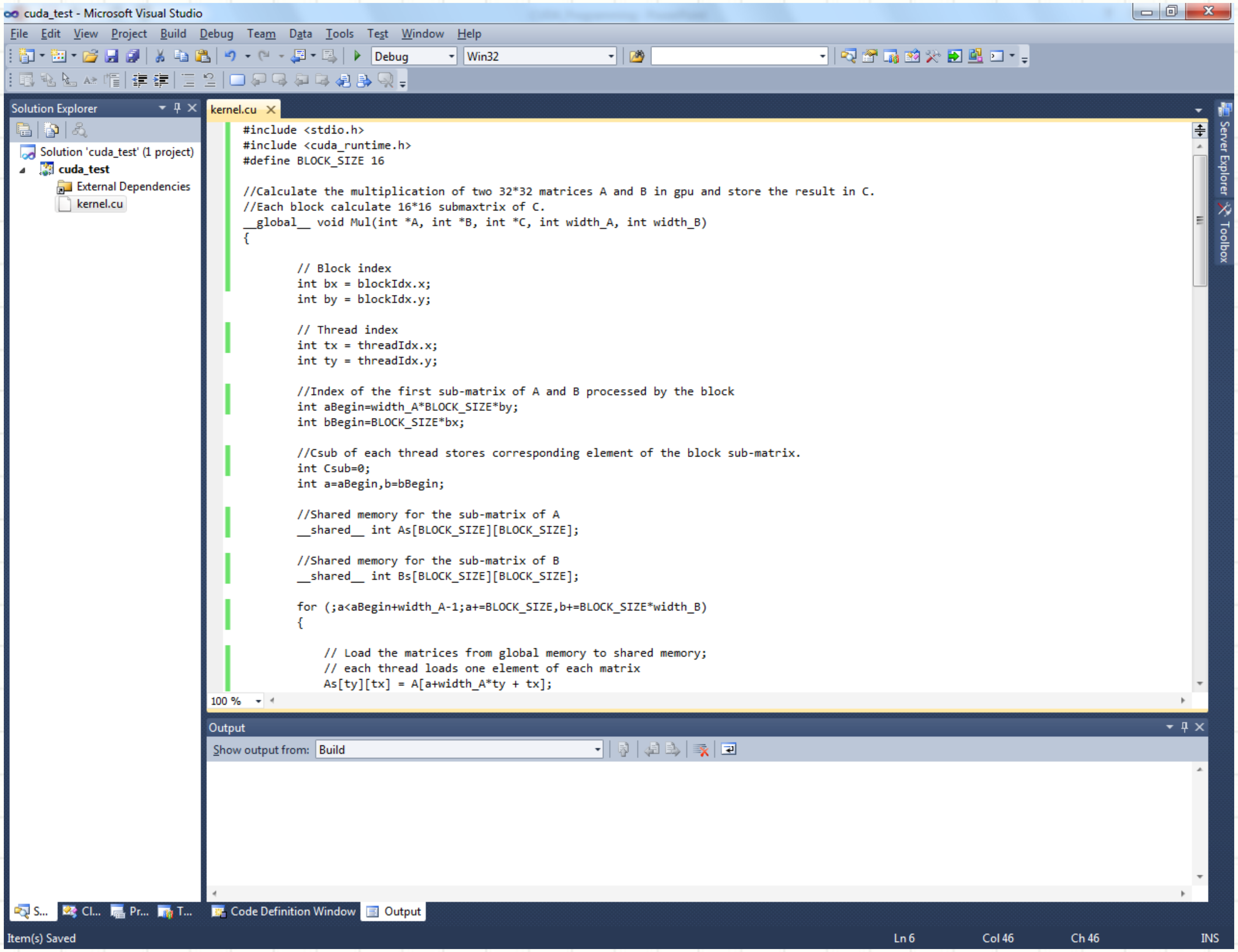
[✓] Create directory for solution

[ ] Add to source control

A simple program setup(cont)

- A default source file, “kernel.cu” is created under the project. Open it and copy the code of file matrix_mul.txt into it (see the figure on the next slide).

A simple program setup(cont)


cuda_test - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Solution Explorer
Solution 'cuda_test' (1 project)
cuda_test
External Dependencies
kernel.cu
Server Explorer
Toolbox
S... Cl... Pr... T...
Code Definition Window
Output

Debug | Win32

kernel.cu

```
#include <stdio.h>
#include <cuda_runtime.h>
#define BLOCK_SIZE 16

//Calculate the multiplication of two 32*32 matrices A and B in gpu and store the result in C.
//Each block calculate 16*16 submaxtrix of C.
__global__ void Mul(int *A, int *B, int *C, int width_A, int width_B)
{

        // Block index
        int bx = blockIdx.x;
        int by = blockIdx.y;

        // Thread index
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        //Index of the first sub-matrix of A and B processed by the block
        int aBegin=width_A*BLOCK_SIZE*by;
        int bBegin=BLOCK_SIZE*bx;

        //Csub of each thread stores corresponding element of the block sub-matrix.
        int Csub=0;
        int a=aBegin,b=bBegin;

        //Shared memory for the sub-matrix of A
        __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];

        //Shared memory for the sub-matrix of B
        __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

        for (;a<aBegin+width_A-1;a+=BLOCK_SIZE,b+=BLOCK_SIZE*width_B)
        {

            // Load the matrices from global memory to shared memory;
            // each thread loads one element of each matrix
            As[ty][tx] = A[a+width_A*ty + tx];
```

100 %

Output

Show output from: Build

Item(s) Saved   Ln 6   Col 46   Ch 46   INS

A simple program example

- Finally you can build the project and run it.
- This example calculates the multiplication of two 32*32 matrices A,B in GPU and stores the result in matrix C.
- Next we will explain the program in detail for you to better understand the basic CUDA programming.

Memory Allocation

- In CUDA, the main memory space of the computer is called **host memory**, and the memory space in GPU device is called **device memory**

- Data usually needs to be transferred between host memory and device memory using cudaMemcpy function.

Host Memory Allocation

Allocate memory space for matrices A, B and C.

```
#define M_SIZE 32
#define BLOCK_SIZE 16

int *h_A = (int*)malloc(M_SIZE*M_SIZE*sizeof(int));
int *h_B = (int*)malloc(M_SIZE*M_SIZE*sizeof(int));
int *h_C = (int*)malloc(M_SIZE*M_SIZE*sizeof(int));
```

Device Memory Allocation

Allocate memory space for matrices A, B and C in GPU, using cudaMalloc function.

```
int *d_A, *d_B, *d_C;
cudaMalloc((void **)&d_A, M_SIZE*M_SIZE*sizeof(int));
cudaMalloc((void **)&d_B, M_SIZE*M_SIZE*sizeof(int));
cudaMalloc((void **)&d_C, M_SIZE*M_SIZE*sizeof(int));
```

Format:

```
cudaMalloc((void **)&val, datasize)
```

It allocates memory for pointer *val* with size of *datasize* in GPU.

Memory copy from CPU to GPU

Then it copies the memory of h_A, h_B in CPU to d_A,d_B in GPU with function cudaMemcpy():

```
cudaMemcpy(d_A, h_A, M_SIZE*M_SIZE*sizeof(int),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, M_SIZE*M_SIZE*sizeof(int),
    cudaMemcpyHostToDevice);
```

Its definition is:

```
cudaMemcpy(&dest, &source, datasize, Type)
```

It copies the memory content of *source* to *dest* with size of *datasize*.

cudaMemcpyHostToDevice: copy from CPU to GPU.

cudaMemcpyDeviceToHost: copy from GPU to CPU.

Create blocks and threads

Then call the function Mul() in GPU and allocate two dimensional grid and block with the following statements:

```
dim3 dimBlock(BLOCK_SIZE1, BLOCK_SIZE2);
dim3 dimGrid(2,2);
Mul<<<dimGrid,dimBlock>>>(d_A, d_B, d_C, M_SIZE, M_SIZE);
```

These statements allocate 2*2 block and 16*16 threads per block in GPU.

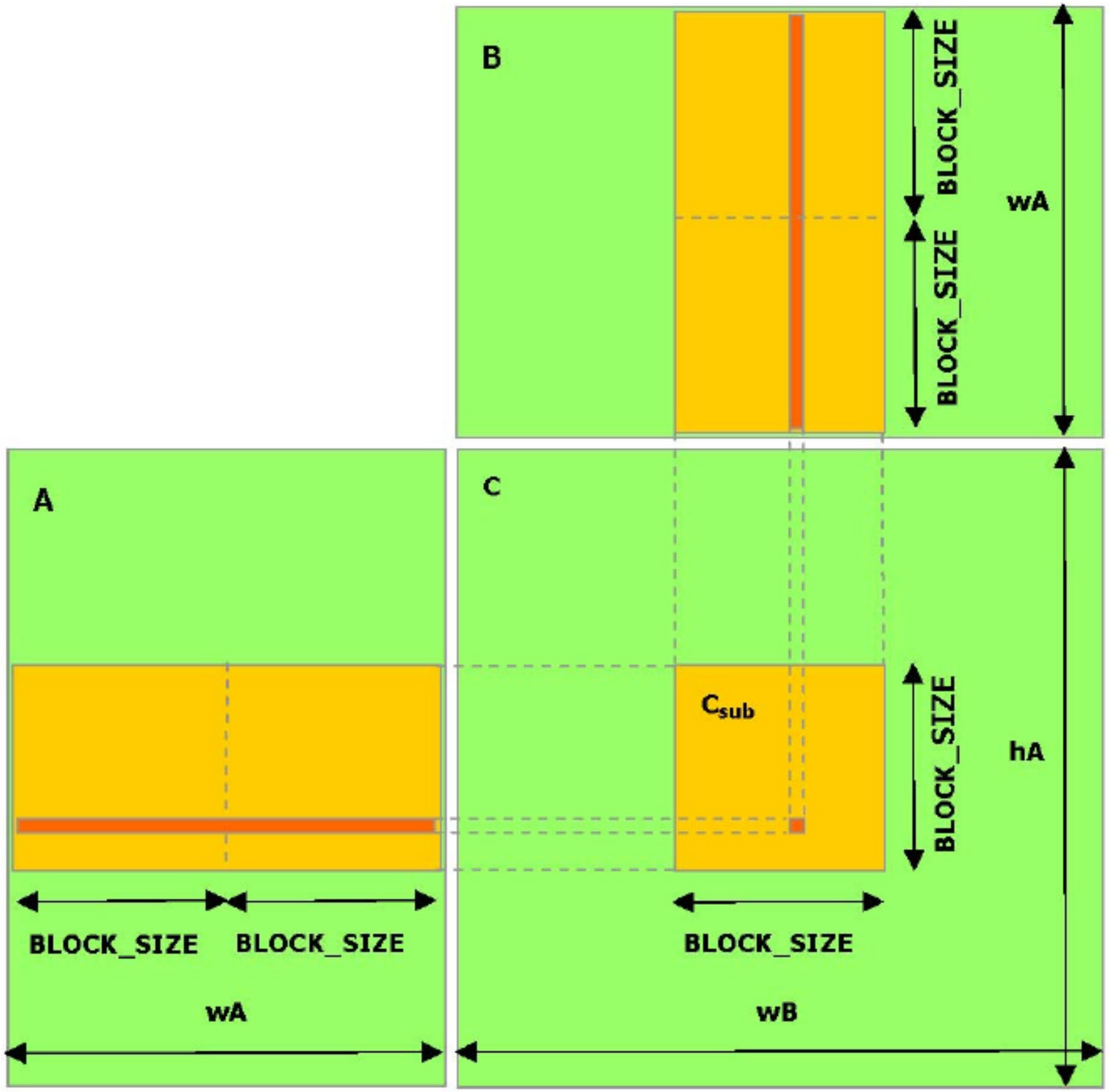
CUDA kernel function

We conduct the parallel computation of matrix multiplication in the function Mul() with identifier __global__ to indicate it is run on GPU:

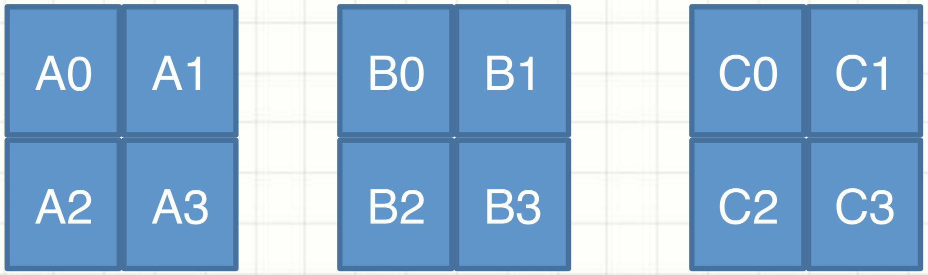
```
__global__ void Mul(int *A,int *B,int *C,int width_A,int width_B)
```

The function divides the original matrices into 4 corresponding sub matrices. Each block calculates one sub-matrix of the result matrix.

The following figure shows the basic idea.


B
BLOCK_SIZE
BLOCK_SIZE
wA
A
C
Csub
BLOCK_SIZE
hA
BLOCK_SIZE
BLOCK_SIZE
BLOCK_SIZE
wA
wB

Basic idea



In GPU, there are 4 blocks calculating sub-matrices C0 to C3 respectively.

$$C0 = A0 \times B0 + A1 \times B2$$

$$C1 = A0 \times B1 + A1 \times B3$$

Thread block 0 stores A0,B0 and A1,B2 while thread block 1 stores A0,B1 and A1,B3. The calculation of C2 and C3 are similar.

Matrix multiplication in GPU

- There are 16*16 threads within each thread block, each thread calculates one element of the result sub-matrix.

- Next we will introduce the calculation in GPU in detail.

Index of each block and thread

In Mul(), we first specify the index of each block and thread(both them are two dimensional):

```
// Block index
int bx = blockIdx.x;
int by = blockIdx.y;
// Thread index
int tx = threadIdx.x;
int ty = threadIdx.y;
```

In this program, bx and by are ranged from 0 to 1 and tx and ty are ranged from 0 to 15.

Index of first sub matrix

Then we indicate the index of the first sub-matrix of A and B processed by the block:

```
int aBegin=width_A*BLOCK_SIZE*by;
int bBegin=BLOCK_SIZE*bx;
```

For example, the first sub matrices of block 0 are A0 and B0 since bx and by of block0 are 0.

Shared memory

Within each block, define two arrays in its shared memory with identifier __shared__ to store the content of corresponding sub matrix:

```
__shared__ int As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];
```

As stores the sub matrix of A while Bs stores the sub matrix of B. For example, in block0, As store A0 and Bs stores B0 at the beginning. These two arrays are stored in the shared memory of each block.

Calculation

- Since one sub matrix of C is calculated by two sub matrices of A and two sub matrices of B, we need to load two sub matrices of A and B two times from global memory for each block and then calculate the multiplication of them in the shared memory.
- In the program, we store the result element of each thread in the value Csub.

Calculation(cont.)

```
for (;a<aBegin+width_A-1; a+=BLOCK_SIZE, b+=BLOCK_SIZE*width_B)
{
        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a+width_A*ty + tx];
        Bs[ty][tx] = B[b+width_B*ty + tx];
        // Synchronize to make sure the matrices are loaded
        __syncthreads();
        for (int k=0;k<BLOCK_SIZE;k++)
        Csub+=As[ty][k]*Bs[k][tx];
        //Make sure computation is done before loading new matrices for global memory.
        __syncthreads();
}
```

This for-loop loads the sub matrices of A and B two times from global memory and calculate the value Csub in each thread. For example, for block0, it loads A0,B0 firstly and calculate Csub, then loads A1,B2 and sum the Csub.

Synchronize

- We use `__syncthreads()` to synchronize all the threads within a block since we should first load the sub matrices and then do the multiplication and then reload after finishing the computation.

Copy result from GPU to CPU

- After calculation, Write the result Csub in each thread to its corresponding position in global memory:

```
//Write the block sub-matrix to global memory.
int c=width_B*BLOCK_SIZE*by+BLOCK_SIZE*bx;
C[c+width_B*ty+tx] = Csub;
```

- Finally copy the result in GPU to CPU(d_C to h_C). The final result is stored in matrix C:

```
cudaMemcpy(d_B, h_B, M_SIZE*M_SIZE*sizeof(int),
        cudaMemcpyDeviceToHost);
```

Exercise

- Given a 32*32 matrix A, apply square calculation to each element of A in GPU and stores the result in A.

Input: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ Output: $\begin{bmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{bmatrix}$

Here is a template file template.txt . You can add your code in it to do the calculation.

Conclusion

In this lab:

- We give a brief overview of CUDA and its architecture.
- We introduce how to build CUDA environment and test a simple CUDA program in Visual Studio 2010.
- More information about CUDA please read
  - CUDA Getting Started Guide
  - CUDA C Programming Guide
  - CUDA C Best Practices Guide