Chapter 18 *GPU (CUDA)*

Speaker: Lung-Sheng Chien

Reference: [1] NVIDIA_CUDA_Programming_Guide_2.0.pdf
[2] CudaReferenceManual_2.0.pdf
[3] nvcc_2.0.pdf
[4] NVIDIA forum, http://forums.nvidia.com/index.php?act=idx

OutLine

- CUDA introduction
  - process versus thread
  - SIMD versus SIMT
- Example 1: vector addition, single core
- Example 2: vector addition, multi-core
- Example 3: matrix-matrix product
- Embed nvcc to vc2005

Process versus thread

Reference: http://en.wikipedia.org/wiki/Thread_(computer_science)

- A *process* is the "heaviest" unit of kernel scheduling. Processes own resources allocated by the operating system. Resources include memory, file handles, sockets, device handles, and windows. Processes do not share address spaces or file resources except through explicit methods such as inheriting file handles or shared memory segments, or mapping the same file in a shared way.
- A *thread* (執行緒, 線程) is the "lightest" unit of kernel scheduling. At least one thread (main thread) exists within each process. If multiple threads can exist within a process, then they share the same memory and file resources. Threads do not own resources except for a stack, a copy of the registers including the program counter

Spec [1]

Each multiprocessor is composed of 8 processors, so that a multiprocessor is able to process the 32 threads of a warp in 4 clock cycles

	Number of Multiprocessors	Compute Capability
GeForce GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2x16	1.1
GeForce 9800 GTX	16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 8800 GT	14	1.1
GeForce 9600 GSO, 8800 GS, 8800M GTX	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 9600 GT, 8800M GTS	8	1.1
GeForce 9500 GT, 8600 GTS, 8600 GT, 8700M GT, 8600M GT, 8600M GS	4	1.1
GeForce 8500 GT, 8400 GS, 8400M GT, 8400M GS	2	1.1
GeForce 8400M G	1	1.1
Tesla S1070	4x30	1.3
Tesla C1060	30	1.3
Tesla S870	4x16	1.0

fluid-01 (GeForce GTX 260)

fluid-02 (GeForce 8800 GT)

matrix (GeForce 9600 GT, 8800M GTS)

Support double-precision (GeForce GTX 280, GeForce GTX 260, Tesla S1070, Tesla C1060)

Product information: http://shopping.pchome.com.tw/ and http://www.sunfar.com.tw

Geforce GTX 280

技嘉 GV-N28-1GH-B PCIE顯示卡
《原廠大跌價~狂降$4590~您心動了嗎!》

★GeForce GTX 280晶片
★1GB GDDR3 顯示記憶體
★512位元記憶體管理介面
★PCI Express 2.0介面
★支援最新DirectX 10
★支援PhysX 和 Cuda 技術

建議售價$~~16500~~ 網路價 $15850 貴貴通報

Geforce GTX 260

技嘉 GV-N26-896H-B 顯示卡
《極致效能，捨我其誰》

★NVIDIA GeForce GTX 260晶片
★896MB GDDR3視訊記憶體
★448bit記憶體管理介面
★576MHz核心時脈/1998MHz記憶體時脈
★2xDVI顯示輸出埠
★最新PCI-Express 2.0插槽介面 (介面規格)
★支援最新DirectX 10

建議售價$~~11490~~ 網路價 $10590

Geforce 9600GT

華碩 EN9600GT/HTDI/512M 顯示卡

◆NVIDIA GeForce 9600GT
◆512MB DDR3 視訊記憶體
◆256-bit記憶體介面
◆650MHz核心時脈/1.8GHz記憶體時脈
◆1625MHz Shader Clock時脈
◆2xDVI顯示輸出埠
◆支援 HDMI 輸出
◆PCI-Express2.0插槽介面
◆支援DirectX 10/ShaderModel 4.0
◆華碩獨家Glaciator散熱片
◆支援 NVIDIA SLI 技術

建議售價$~~4750~~ 24h到貨 $4290

Geforce 8800GT

- 廠牌： MSI
- 商品編碼： 202007
- 商品名稱： NX8800GT OC版/T2D/512M 顯示卡
- 價格區間： $7,500 ~ $8,051
- 會員迴轉金： 現金結帳回饋 2% 刷卡結帳回饋 0.2%
- 商品特色： 微星 NX8800GT/16X PCIE/512M
- 評鑑等級：

我要評鑑 發表您對此商品之評鑑

Geforce 9600GT

Spec [2]

GPU Engine Specs:

Processor Cores	64
Graphics Clock (MHz)	650 MHz
Processor Clock (MHz)	1625 MHz
Texture Fill Rate (billion/sec)	20.8

Memory Specs:

Memory Clock (MHz)	900 MHz
Standard Memory Config	512 MB
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	57.6

Geforce 8800GT

GPU Engine Specs:

Processor Cores	112
Graphics Clock (MHz)	600 MHz
Processor Clock (MHz)	1500 MHz
Texture Fill Rate (billion/sec)	33.6

Memory Specs:

Memory Clock (MHz)	900 MHz
Standard Memory Config	512 MB
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	57.6

Spec [3]

Geforce GTX260

GPU Engine Specs:

Processor Cores	192
Graphics Clock (MHz)	576 MHz
Processor Clock (MHz)	1242 MHz
Texture Fill Rate (billion/sec)	36.9

Memory Specs:

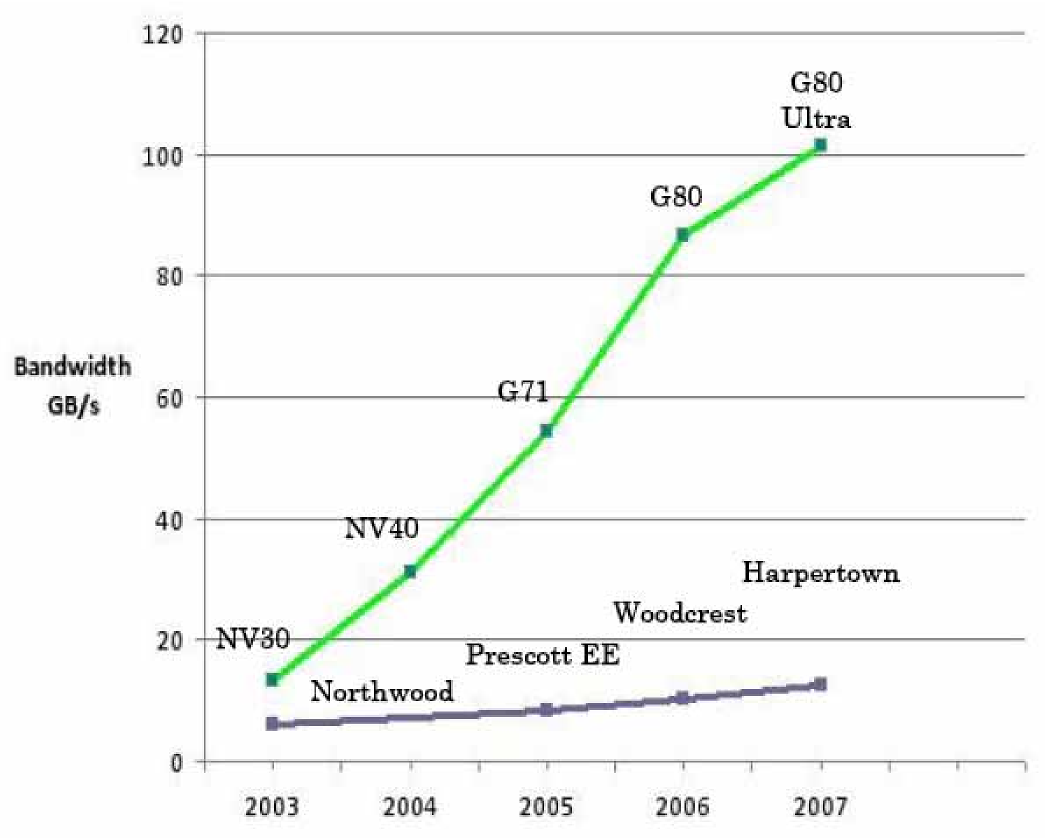
Memory Clock (MHz)	999 MHz
Standard Memory Config	896 MB
Memory Interface Width	448-bit
Memory Bandwidth (GB/sec)	111.9

A very important new addition to the GeForce GTX 200 GPU architecture is double-precision, 64-bit floating point computation support. This benefits various high-end scientific, engineering, and financial computing applications or any computational task requiring very high accuracy of results. Each SM incorporates a double-precision 64-bit floating math unit, for a total of 30 double-precision 64-bit processing cores.

NVIDIA GPU and CUDA

GPU (graphic processor unit): embedded in graphic card (顯示卡)

CUDA is a parallel programming model provided by NVIDIA



GPU has larger memory bandwidth than CPU

GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Data from NVIDIA_CUDA_Programming_Guide_2.0.pdf

Spec for compute capability 1.0

- The maximum number of threads per block is 512
- The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively.
- The maximum size of each dimension of a grid of thread blocks is 65535
- The warp size is 32 threads
- The number of registers per multiprocessor is 8192 (one multiprocessor has 8 processors, one processor has 1024 registers)
- The amount of shared memory available per multiprocessor is 16KB organized into 16 banks.
- The maximum number of active blocks per multiprocessor is 8
- The maximum number of active warps per multiprocessor is 24
- The maximum number of active threads per multiprocessor is 768

cuda

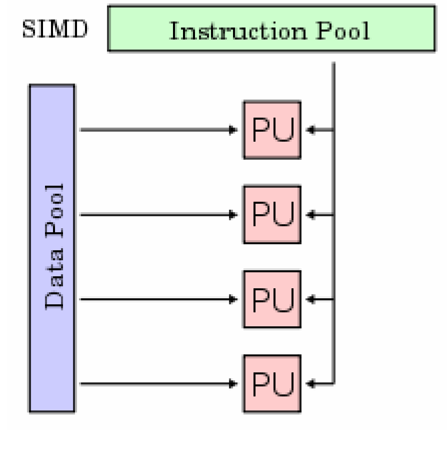
Reference: http://en.wikipedia.org/wiki/CUDA

- **CUDA** (***Compute Unified Device Architecture***) is a compiler and set of development tools that enable programmers to use a variation of C based on the PathScale C compiler to code algorithms for execution on the graphics processing unit (GPU).
- CUDA has been developed by NVIDIA and to use this architecture requires an Nvidia GPU and drivers.
- Unlike CPUs, GPUs have a parallel "many-core" architecture, each core capable of running thousands of threads simultaneously.
- core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization.
- the GPU is well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with high arithmetic intensity – the ratio of arithmetic operations to memory operations.

SIMD (vector machine)

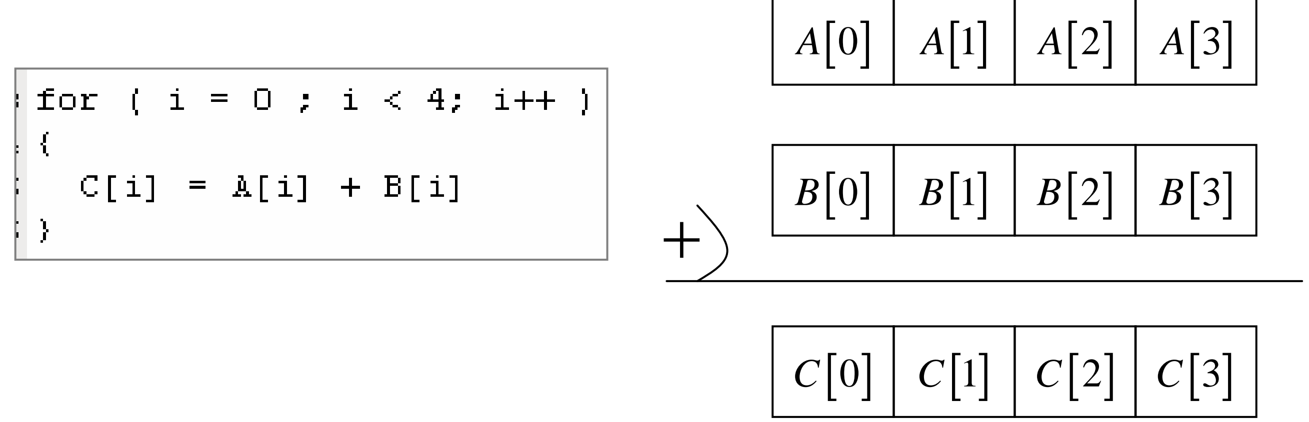
Reference: http://en.wikipedia.org/wiki/SIMD

- **SIMD** (**S**ingle **I**nstruction, **M**ultiple **D**ata) is a technique employed to achieve data level parallelism, as in a vector processor.
  - supercomputers
  - MMX of pentium 4
  - SSE (Streaming SIMD Extensions) of x86 architecture



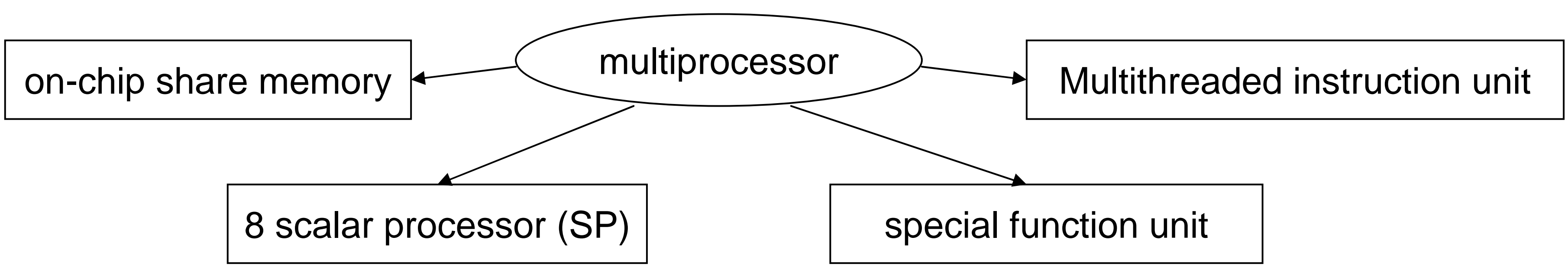
	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

This box: view • talk • edit

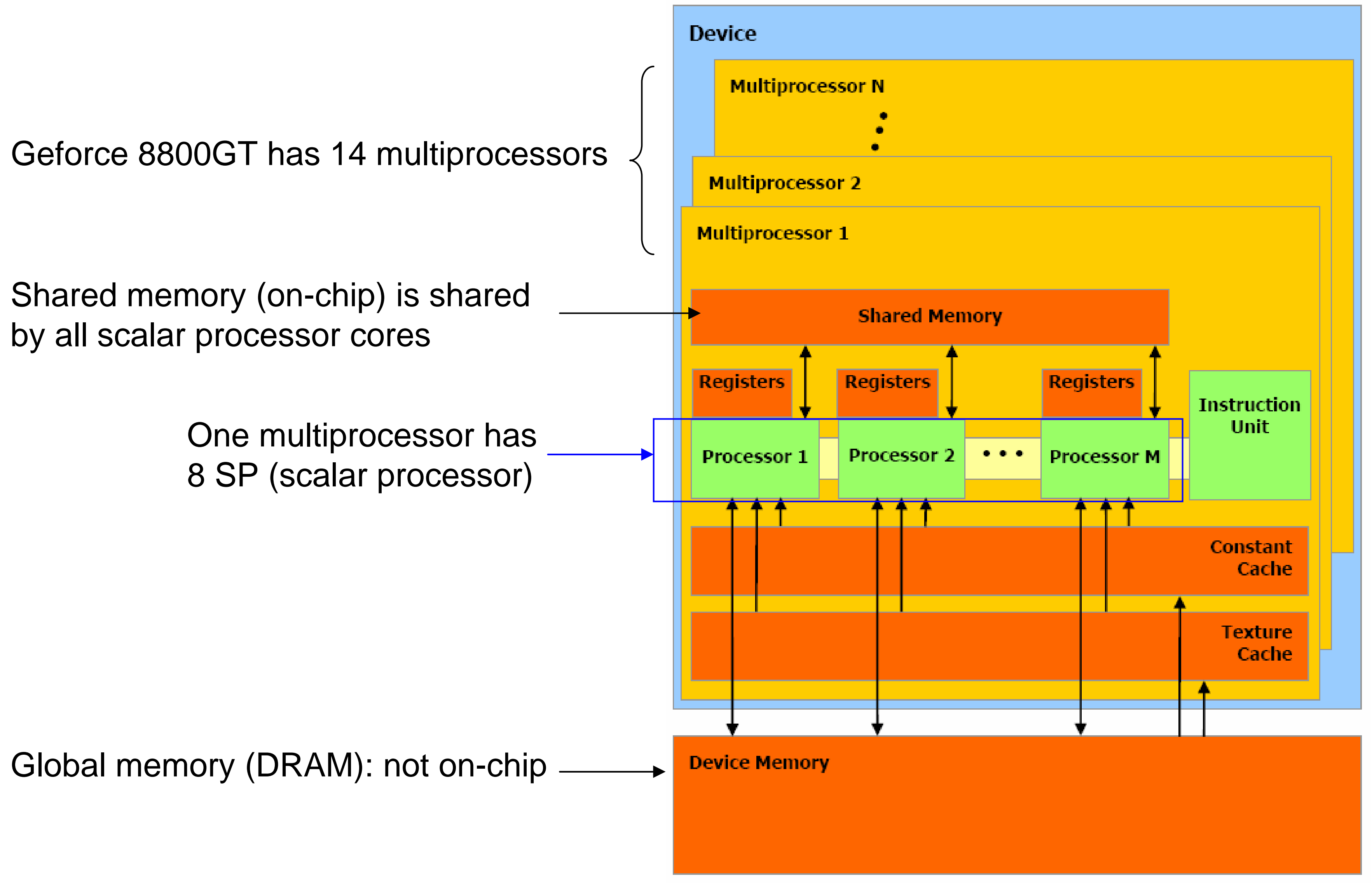


SIMT (CUDA, Tesla architecture)

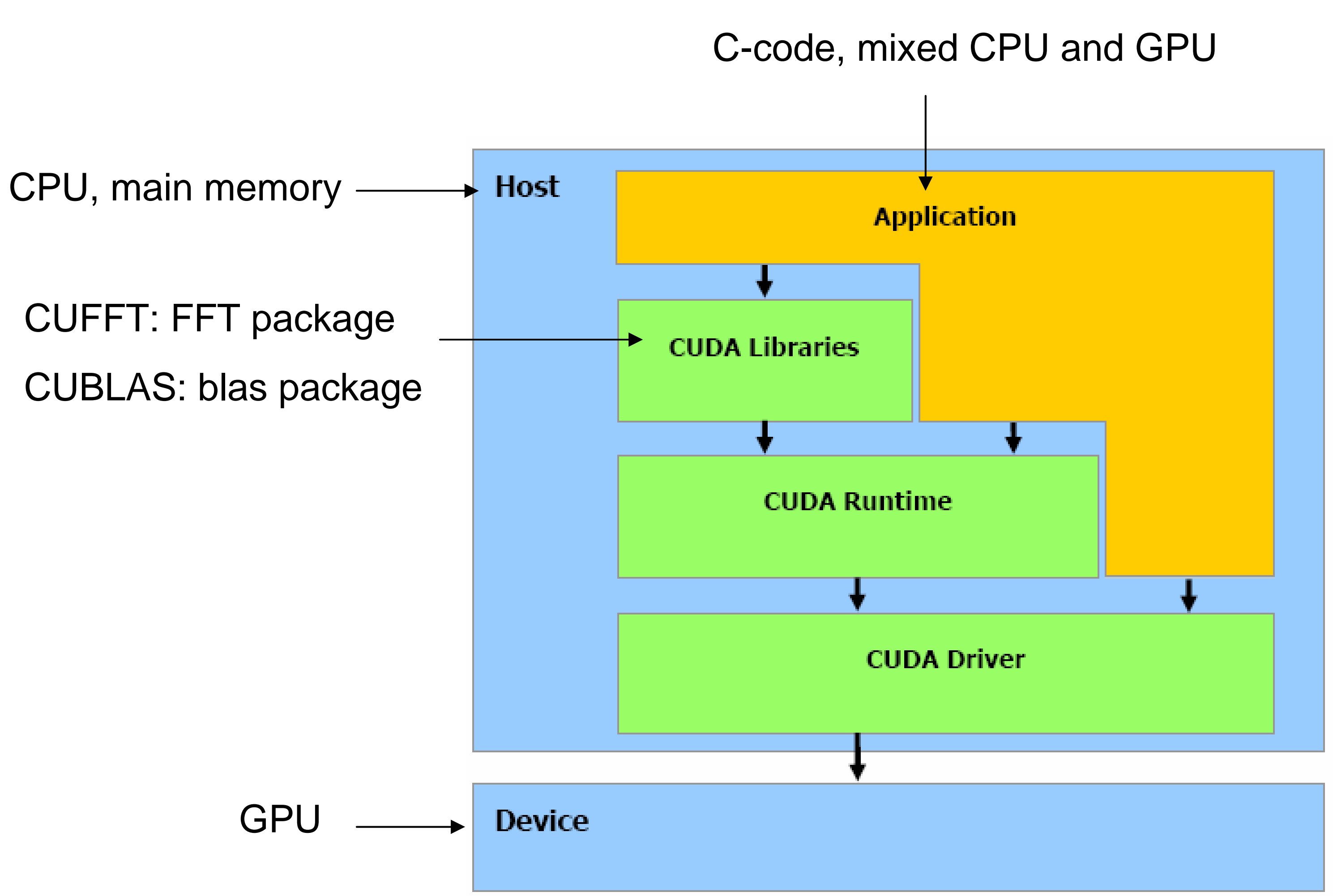
- SIMT (single-instruction, multiple-thread): The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.
- The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps.*
- Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently.
- When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit.
- A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.



A set of multiprocessors with on-chip shared memory



Software stack



OutLine

- CUDA introduction

- Example 1: vector addition, single core
- Example 2: vector addition, multi-core
- Example 3: matrix-matrix product
- Embed nvcc to vc2005

Example 1: vector addition [1]

vecadd_gold.cpp

Tell C++ compiler to compile function *computeGold* as C-function

```
2  #include <stdio.h>
3  #include <time.h>          <- measure time
4
5  extern "C"
6  void computeGold( float*, const float*, const float*, unsigned int );
7
8  void computeGold(float* C, const float* A, const float* B, unsigned int N )
9  {
10    unsigned int i ;
11    clock_t  start, end ;
12
13    start = clock() ;
14    for ( i= 0; i < N ; ++i){
15      C[i] = A[i] + B[i] ;
16    }
17    end = clock() ;
18    double dt = ((double)(end - start))/((double)CLOCKS_PER_SEC) * 1000.0 ;
19    printf("compute gold vector needs %10.4f (ms)\n", dt );
20
21 }
```

$C := A + B$

clcok_t clock(void)

returns the processor time used by the program since the beginning of execution, or -1 if unavailable.
clock()/CLOCKS_PER_SEC is a time in seconds

Question: how to write vector addition in GPU version?

Example 1: vector addition [2]

1 vecadd_GPU.cu

```
 2 #include <stdio.h>
 3 // includes, project
 4 #include <cutil.h>
 5
 6 extern "C" {
 7 void vecadd_GPU(float* h_C, const float* h_A, const float* h_B, unsigned int N) ;
 8 }
 9
10 void vecadd_GPU(float* h_C, const float* h_A, const float* h_B, unsigned int N)
11 {
12   unsigned int mem_size_A = sizeof(float) * N ;
13   unsigned int mem_size_B = sizeof(float) * N ;
14
15   // allocate device memory
16   float* d_A;
17   CUDA_SAFE_CALL(cudaMalloc((void**) &d_A, mem_size_A));
18   float* d_B;
19   CUDA_SAFE_CALL(cudaMalloc((void**) &d_B, mem_size_B));
20
21   // copy host memory to device
22   CUDA_SAFE_CALL(cudaMemcpy(d_A, h_A, mem_size_A,
23                            cudaMemcpyHostToDevice) );
24   CUDA_SAFE_CALL(cudaMemcpy(d_B, h_B, mem_size_B,
25                            cudaMemcpyHostToDevice) );
```

1. extension .cu means cuda file, it cannot be compiled by g++/icpc, we must use cuda compiler nvcc to compile it first, we will discuss this later

2. Header file in directory /usr/local/NVIDIA_CUDA_SDK\common\inc

3. Tell C++ compiler to compile function *vecadd_GPU* as C-function

4. ***cudaMalloc*** allocates device memory block in GPU device, the same as ***malloc***

Example 1: vector addition [3]

```
cudaError_t cudaMalloc( void** devPtr, size_t count )
```

Allocates **count** bytes of linear memory on the device and returns in ***devPtr** a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. **cudaMalloc()** returns **cudaErrorMemoryAllocation** in case of failure.

Relevant return values:
- **cudaSuccess**
- **cudaErrorMemoryAllocation**

5. ***cudaMemcpy*** copies data between GPU and host, the same as ***memcpy***

```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count, enum cudaMemcpyKind kind
```

Copies **count** bytes from the memory area pointed to by **src** to the memory area pointed to by **dst**, where **kind** is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy. The memory areas may not overlap. Calling **cudaMemcpy()** with **dst** and **src** pointers that do not match the direction of the copy results in an undefined behavior.

Relevant return values:
- **cudaSuccess**
- **cudaErrorInvalidValue**
- **cudaErrorInvalidDevicePointer**
- **cudaErrorInvalidMemcpyDirection**

Example 1: vector addition [4]

```
27    // allocate device memory for result
28    unsigned int mem_size_C = sizeof(float) * N ;
29    float* d_C;
30    CUDA_SAFE_CALL(cudaMalloc((void**) &d_C, mem_size_C));
31
32    // create and start timer
33    unsigned int timer = 0;
34    CUT_SAFE_CALL(cutCreateTimer(&timer));
35    CUT_SAFE_CALL(cutStartTimer(timer));
36
37    // execute the kernel
38    vecadd<<< 1, N >>>(d_C, d_A, d_B, N);
39
40    // check if kernel execution generated and error
41    CUT_CHECK_ERROR("Kernel execution failed");
42
43    // copy result from device to host
44    CUDA_SAFE_CALL(cudaMemcpy(h_C, d_C, mem_size_C,
45                              cudaMemcpyDeviceToHost) );
46
47    // stop and destroy timer
48    CUT_SAFE_CALL(cutStopTimer(timer));
49    printf("Processing time: %f (ms) \n", cutGetTimerValue(timer));
50    CUT_SAFE_CALL(cutDeleteTimer(timer));
51
52    CUDA_SAFE_CALL(cudaFree(d_A));
53    CUDA_SAFE_CALL(cudaFree(d_B));
54    CUDA_SAFE_CALL(cudaFree(d_C));
55 }
```

6 (line 30)

7 (line 38)

Measure time (lines 34–49)

In fact, we can use assert() to replace it

Header file *util.h*

```
723 #   define CUDA_SAFE_CALL( call) do {                                          \
724      CUDA_SAFE_CALL_NO_SYNC(call);                                            \
725      cudaError err = cudaThreadSynchronize();                                 \
726      if( cudaSuccess != err) {                                                \
727          fprintf(stderr, "Cuda error in file '%s' in line %i : %s.\n",        \
728                  __FILE__, __LINE__, cudaGetErrorString( err) );              \
729          exit(EXIT_FAILURE);                                                  \
730      } } while (0)
```

Example 1: vector addition [5]

7 vecadd<<< 1, *N* >>>(*d_C, d_A, d_B, N*) ; is called **kernel** function in *vecadd_kernel.cu*

1 thread block

N threads per thread block

vecadd_kernel.cu

```
 3 #include <stdio.h>
 4 #include <assert.h>
 5
 6 __global__  void vecadd( float* C, float* A, float* B, int N)
 7 {
 8 #ifdef __DEVICE_EMULATION__
 9   int bx = blockIdx.x ;
10   assert( 0 == bx) ;
11 #endif
12
13   int i = threadIdx.x ;
14   C[i] = A[i] + B[i] ;
15 }
```

8 **__global__**

The __global__ qualifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host only.

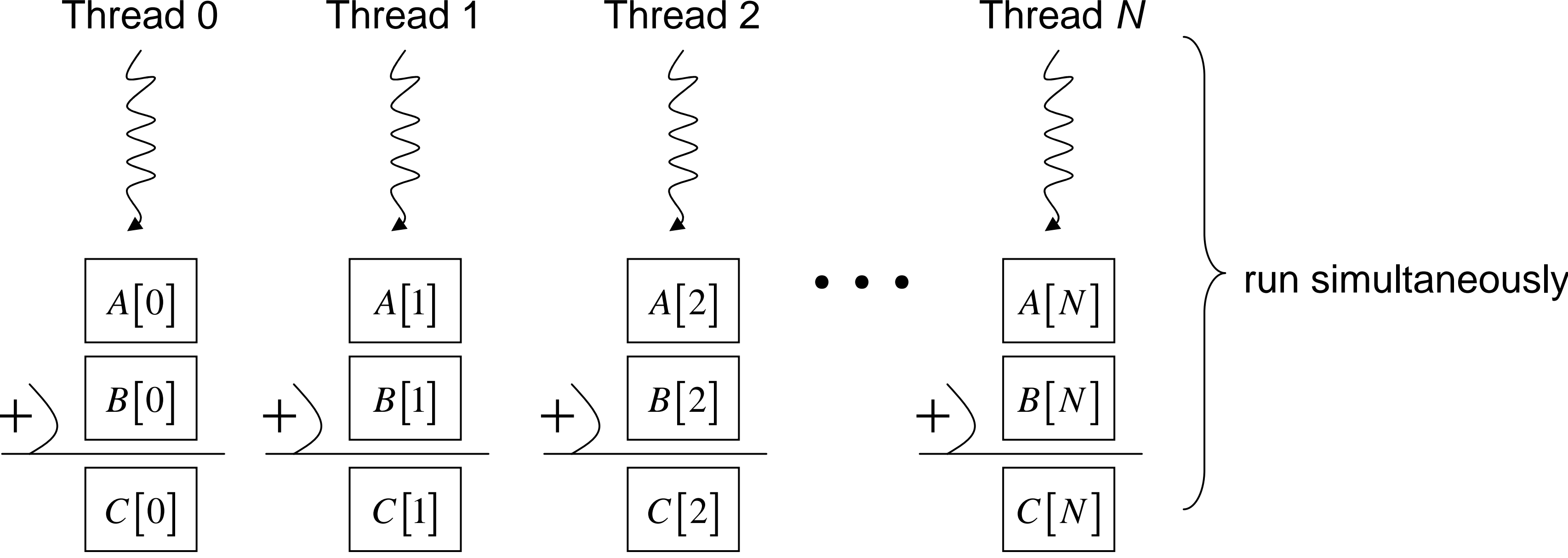
9 If we emulation (仿效) GPU under CPU, then we can use standard I/O, i.e. **printf**, however if we execute on GPU, ***printf*** is forbidden.

In emulation mode, macro __DEVICE_EMULATION__ is set.

Example 1: vector addition [6]

10
```
13    int i = threadIdx.x ;
14    C[i] = A[i] + B[i] ;
```

Each of the threads that execute a kernel is given a unique *thread ID* that is accessible within the kernel through the built-in **threadIdx** variable.



Question 1: how many threads per block, arbitrary?

Question 2: can we use more than two thread blocks?

Example 1: vector addition [7]

Question 1: how many threads per block, arbitrary?

Specifications for Compute Capability 1.0

- The maximum number of threads per block is 512;

Question 3: what happens if we use more than 512 threads in a thread block?

Question 2: can we use more than two thread blocks?

- How many blocks a multiprocessor can process at once depends on how many registers per thread and how much shared memory per block are required for a given kernel.
- If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch. A multiprocessor can execute as many as eight thread blocks concurrently.

Question 4: how to issue more than two thread blocks?

We will answer question 3 and question 4 after we finish this simple example

Example 1: vector addition (driver) [8]

vecadd.cu

```
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5 #include <string.h>
 6 #include <math.h>
 7
 8 // includes, project
 9 #include <cutil.h>
10
11 // includes, kernels
12 #include <vecadd_kernel.cu>
13 #include <vecadd_GPU.cu>
14
15 // declaration, forward
16 void runTest(int argc, char** argv);
17 void randomInit(float*, int);
18 void printDiff(float*, float*, int, int);
19
20 extern "C" {
21 void computeGold(float*, const float*, const float*, unsigned int );
22 void vecadd_GPU(float* h_C, const float* h_A, const float* h_B, unsigned int N) ;
23 }
24
25 int main(int argc, char** argv)
26 {
27     runTest(argc, argv);
28
29     CUT_EXIT(argc, argv);
30 }
```

use macro ***CUT_EXIT***

Include cuda source code such that we only need to compile one file

Tell C++ compiler to compile function *vecadd_GPU* and *computeGold* as C-function

Example 1: vector addition (driver) [9]

```
32 // test  C = A + B
33 void runTest(int argc, char** argv)
34 {
35   unsigned int N = 128 ;
36   CUT_DEVICE_INIT(argc, argv);
37
38   // set seed for rand()
39   srand(2006);
40
41   // allocate host memory for matrices A and B
42   unsigned int size_A = N ;
43   unsigned int mem_size_A = sizeof(float) * size_A;
44   float* h_A = (float*) malloc(mem_size_A);
45
46   unsigned int size_B = N ;
47   unsigned int mem_size_B = sizeof(float) * size_B;
48   float* h_B = (float*) malloc(mem_size_B);
49
50   // allocate host memory for the result
51   unsigned int size_C = N ;
52   unsigned int mem_size_C = sizeof(float) * size_C;
53   float* h_C = (float*) malloc(mem_size_C);
54
55   // initialize host memory
56   randomInit(h_A, size_A);
57   randomInit(h_B, size_B);
58
59   vecadd_GPU( h_C, h_A, h_B, N ) ;
```

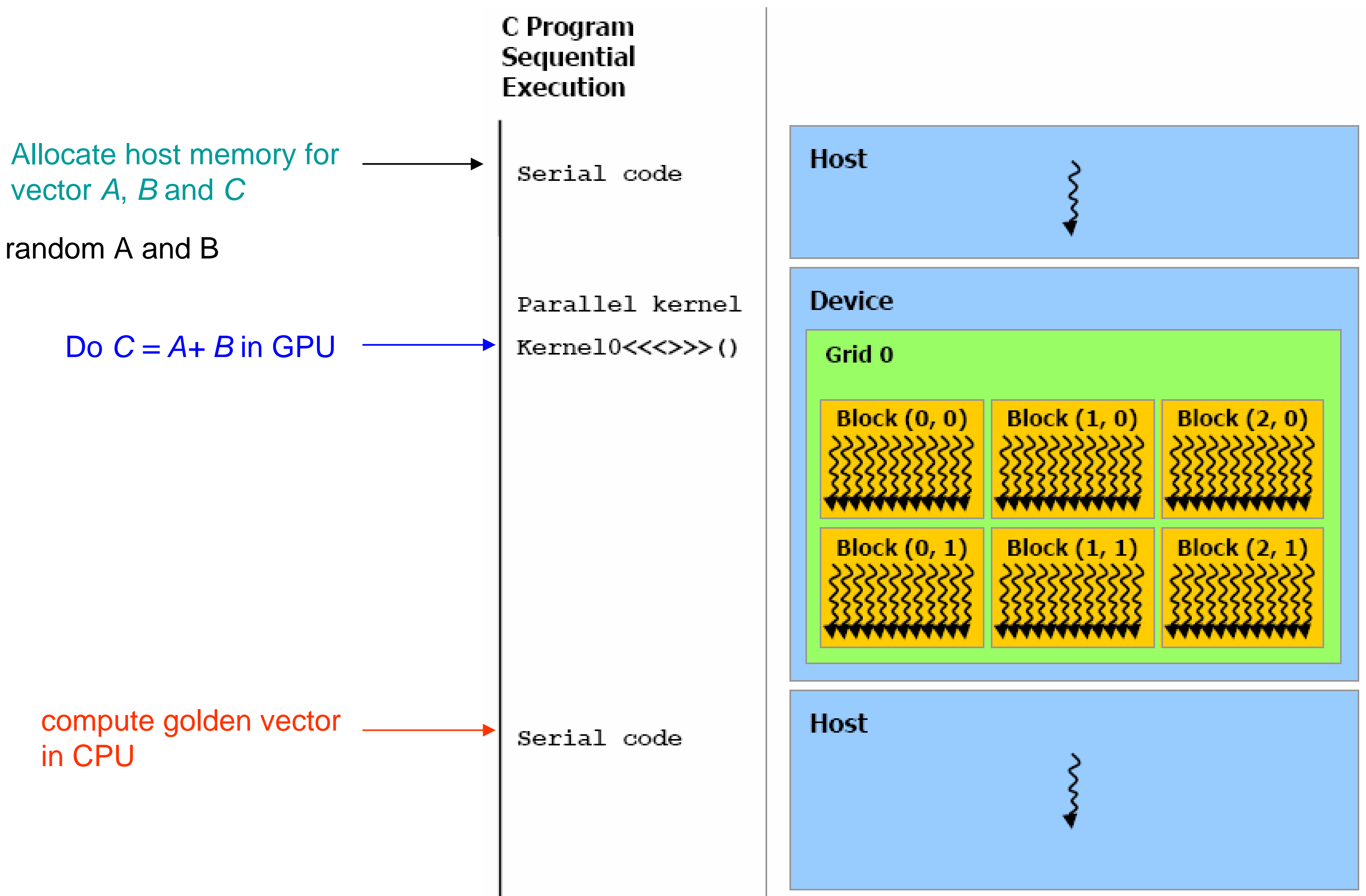
Allocate host memory for vector *A*, *B* and *C*

Do $C = A + B$ in GPU

compute golden vector in CPU

```
61   // compute reference solution
62   float* reference = (float*) malloc(mem_size_C);
63   computeGold(reference, h_A, h_B, N );
64
65   // check result
66   CUTBoolean res = cutCompareL2fe(reference, h_C, size_C, 1e-6f);
67   printf("Test %s \n", (1 == res) ? "PASSED" : "FAILED");
68   if (res!=1) printDiff(reference, h_C, 1, N);
69
70   // clean up memory
71     free(h_A);
72     free(h_B);
73     free(h_C);
74     free(reference);
75 }
```

Example 1: vector addition (driver) [10]



Example 1: vector addition (compile under Linux) [11]

Step 1: upload all source files to workstation, assume you put them in directory *vecadd*

```
[macrold@matrix vecadd]$ ls
Makefile  vecadd.cu  vecadd_GPU.cu  vecadd_gold.cpp  vecadd_kernel.cu
[macrold@matrix vecadd]$
```

Type “man nvcc” to see manual of NVIDIA CUDA compiler

```
NAME
       nvcc - NVIDIA CUDA compiler driver

SYNOPSIS
       nvcc [options] inputfile

OPTIONS
       Options for specifying the compilation phase

       More exactly, this option specifies up to which stage the input files must be com-
       piled, according to the following compilation trajectories for different input file
       types:

               .c/.cc/.cpp/.cxx : preprocess, compile, link
               .cu              : preprocess, cuda frontend, ptxassemble,
                                  merge with host C code, compile, link
               .gpu             : nvopencc compile into cubin
               .ptx             : ptxassemble into cubin.

       --cuda  (-cuda)
           Compile all .cu input files to .cu.c output.


       --compile (-c)
           Compile each .c/.cc/.cpp/.cxx/.cu input file into an object file.

       --run (-run)
           This option compiles and links all inputs into an executable, and executes it.
           Or, when the input is a single executable, it is executed without any compila-
           tion or linking. This step is intended for developers who do not want to be
           bothered with setting the necessary cuda dll search paths (these will be set
           temporarily by nvcc).
```

Example 1: vector addition (compile under Linux) [12]

Step 2: edit Makefile by “vi Makefile”

-L[library path]

-lcuda = libcuda.a

Macro definition

target

```
# In directory /usr/local/cuda/lib
#     libcublas.so
#     libcudart.so    CUDA runtime library
#     libcuda.so
#     libcufft.so
#
# In directory /usr/local/NVIDIA_CUDA_SDK/lib
#     libcutil.a
#     if one use exmaples in /usr/local/NVIDIA_CUDA_SDK/projects
#     then static library libcutil.a (CUDA utility) is necessary
#     also include file collection in /usr/local/NVIDIA_CUDA_SDK/common/inc
#     is important when compile *.cu files
#

INLCUDE = -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include

LIBS = -L/usr/local/NVIDIA_CUDA_SDK/lib -lcutil
LIBS += -L/usr/local/cuda/lib -lcuda -lcudart
LIBS += -L/usr/lib64 -lGL -lGLU

SRC_CU = vecadd.cu

SRC_CXX = vecadd_gold.cpp

CXXFlag = -DCUDA_FLOAT_MATH_FUNCTIONS -DCUDA_NO_SM_11_ATOMIC_INTRINSICS

gxxFlag = -m64 -O2

icpcFlag = -mp -O2

nvcc_run:
        nvcc -run $(INLCUDE) $(LIBS) $(SRC_CU)  $(SRC_CXX)
```

$(SRC_CU) means ***vecadd.cu***

Example 1: vector addition (compile under Linux) [13]

Step 3: type “make nvcc_run”

```
[macrold@matrix vecadd]$ make nvcc_run
nvcc -run -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include -L/usr/local/NVIDIA_CUDA_
SDK/lib -lcutil -L/usr/local/cuda/lib -lcuda -lcudart -L/usr/lib64 -lGL -lGLU vecadd.cu  vecadd_gold.
cpp
Using device 0: GeForce 9600 GT
Processing time: 0.046000 (ms)
compute gold vector needs     0.0000 (ms)
Test PASSED

Press ENTER to exit...
```

1 “Device is Geforce 9600 GT” means GPU is activated correctly.

$N = 128$

2 To execute C = A + B in GPU costs

0.046 ms

3 To execute C = A + B in CPU costs

0.0 ms

```
INLCUDE = -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include

LIBS = -L/usr/local/NVIDIA_CUDA_SDK/lib -lcutil
LIBS += -L/usr/local/cuda/lib -lcuda -lcudart
LIBS += -L/usr/lib64 -lGL -lGLU

SRC_CU = vecadd.cu

SRC_CXX = vecadd_gold.cpp

CXXFlag = -DCUDA_FLOAT_MATH_FUNCTIONS -DCUDA_NO_SM_11_ATOMIC_INTRINSICS

gxxFlag = -m64 -O2

icpcFlag = -mp -O2

nvcc_run:
	nvcc -run $(INLCUDE) $(LIBS) $(SRC_CU)  $(SRC_CXX)
```

Question 5: we know number of threads per block is 512, how to verify this?

Question 6: It seems that CPU is faster than GPU, what’s wrong?

Example 1: vector addition (compile under Linux) [14]

Modify file vecadd.cu, change N to 512, then compile and execute again

```
32 // test  C = A + B
33 void runTest(int argc, char** argv)
34 {
35   unsigned int N = 512 ;
36   printf("N = %d\n", N);
```

```
[macrold@matrix vecadd]$ make nvcc_run
nvcc -run -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include -L/usr/local/NVIDIA_CUDA_
SDK/lib -lcutil -L/usr/local/cuda/lib -lcuda -lcudart -L/usr/lib64 -lGL -lGLU vecadd.cu  vecadd_gold.
cpp
N = 512
Using device 0: GeForce 9600 GT
Processing time: 0.048000 (ms)
compute gold vector needs     0.0000 (ms)
Test PASSED

Press ENTER to exit...
```

Modify file vecadd.cu, change *N* to 513, then compile and execute again, it fails

```
32 // test  C = A + B
33 void runTest(int argc, char** argv)
34 {
35   unsigned int N = 513 ;
36   printf("N = %d\n", N);
```

```
[macrold@matrix vecadd]$ make nvcc_run
nvcc -run -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include -L/usr/local/NVIDIA_CUDA_
SDK/lib -lcutil -L/usr/local/cuda/lib -lcuda -lcudart -L/usr/lib64 -lGL -lGLU vecadd.cu  vecadd_gold.
cpp
N = 513
Using device 0: GeForce 9600 GT
Processing time: 0.133000 (ms)
compute gold vector needs     0.0000 (ms)
Test FAILED
diff(0,0) CPU=0.6031, GPU=1.5329 ndiff(0,1) CPU=0.3403, GPU=0.2968 ndiff(0,2) CPU=0.0919, GPU=0.6766
```

Example 1: vector addition (compile under Linux) [15]

vecadd_GPU.cu

```
// allocate device memory for result
unsigned int mem_size_C = sizeof(float) * N ;
float* d_C;
CUDA_SAFE_CALL(cudaMalloc((void**) &d_C, mem_size_C));

// create and start timer
unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer));
CUT_SAFE_CALL(cutStartTimer(timer));

// execute the kernel
vecadd<<< 1, N >>>(d_C, d_A, d_B, N);^M

// check if kernel execution generated and error
CUT_CHECK_ERROR("Kernel execution failed");

// copy result from device to host
CUDA_SAFE_CALL(cudaMemcpy(h_C, d_C, mem_size_C,
                          cudaMemcpyDeviceToHost) );

// stop and destroy timer
CUT_SAFE_CALL(cutStopTimer(timer));
printf("Processing time: %f (ms) \n",
          cutGetTimerValue(timer));
CUT_SAFE_CALL(cutDeleteTimer(timer));
```

Including C = A + B in GPU and data transformation from device to Host

vecadd_GPU.cu

```
// create and start timer
unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer));
CUT_SAFE_CALL(cutStartTimer(timer));

// execute the kernel
vecadd<<< 1, N >>>(d_C, d_A, d_B, N);

// stop and destroy timer
CUT_SAFE_CALL(cutStopTimer(timer));
printf("in GPU, C = A + B: %f (ms)\n",
   cutGetTimerValue(timer));
CUT_SAFE_CALL(cutDeleteTimer(timer));

//check if kernel execution generated and error
CUT_CHECK_ERROR("Kernel execution failed");

CUT_SAFE_CALL(cutCreateTimer(&timer));
CUT_SAFE_CALL(cutStartTimer(timer));

// copy result from device to host
CUDA_SAFE_CALL(cudaMemcpy(h_C, d_C, mem_size_C,
                       cudaMemcpyDeviceToHost) );
// stop and destroy timer
CUT_SAFE_CALL(cutStopTimer(timer));
printf("device --> Host: %f (ms)\n",
    cutGetTimerValue(timer));
CUT_SAFE_CALL(cutDeleteTimer(timer));
```

```
N = 512
Using device 0: GeForce 9600 GT
in GPU, C = A + B: 0.026000 (ms)
device --> Host: 0.018000 (ms)
compute gold vector needs     0.0000 (ms)
Test PASSED
```

CPU is faster than GPU for small *N*, how about for large *N* ?

Example 1: vector addition (double precision) [16]

Makefile

```
INLCUDE = -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include

LIBS = -L/usr/local/NVIDIA_CUDA_SDK/lib -lcutil
LIBS += -L/usr/local/cuda/lib -lcuda -lcudart
LIBS += -L/usr/lib64 -lGL -lGLU

SRC_CU = vecadd.cu

SRC_CXX = vecadd_gold.cpp

CXXFlag = -DCUDA_FLOAT_MATH_FUNCTIONS -DCUDA_NO_SM_11_ATOMIC_INTRINSICS

gxxFlag = -m64 -O2

icpcFlag = -mp -O2

nvcc_run:
	nvcc -arch sm_13 -run $(INLCUDE) $(LIBS) $(SRC_CU)  $(SRC_CXX)
```

-arch sm_13

enable double precision (on compatible hardware, say Geforce GTX260 in fluid-01.am.nthu.edu.tw)

Remember to replace “float” by “double”

in source code

man nvcc

```
--gpu-name <gpu architecture name> (-arch)
    Specify the name of the nVidia GPU to compile for. This can either be a 'real'
    GPU, or a 'virtual' ptx architecture. Ptx code represents an intermediate for-
    mat that can still be further compiled and optimized for. depending on the ptx
    version, a specific class of actual GPUs.

    The architecture specified with this option is the architecture that is assumed
    by the compilation chain up to the ptx stage, while the architecture(s) speci-
    fied with the -code option are assumed by the last, potentially runtime compi-
    lation stage.

    Allowed values for this option:  'compute_10', 'compute_11', 'compute_13',
    'compute_14', 'compute_20', 'sm_10', 'sm_11', 'sm_13', 'sm_14', 'sm_20'.
    Default value: 'sm_10'.
```

OutLine

- CUDA introduction
- Example 1: vector addition, single core

- Example 2: vector addition, multi-core
- Example 3: matrix-matrix product
- Embed nvcc to vc2005

Example 2: multicore vector addition [1]

vecadd_kernel.cu

```
__global__  void vecadd( float* C, float* A, float* B, int N)
{
#ifdef __DEVICE_EMULATION__
  int bx = blockIdx.x ;
  assert( 0 == bx) ;
#endif

  int i = threadIdx.x ;
  C[i] = A[i] + B[i] ;
}
```

More than two thread blocks, each block has 512 threads

vecadd_kernel.cu

```
__global__  void vecadd_multicore( float* C, float* A, float* B,
          int threads, int N)
{
  int bx = blockIdx.x;
  int i = bx*threads + threadIdx.x ;
  C[i] = A[i] + B[i] ;

#ifdef __DEVICE_EMULATION__
  printf("bx = %d\n", bx ) ;
#endif
}
```

Built-in *blockIdx* variable denotes which block, starting from 0

Built-in *threadIdx* variable denotes which thread, starting from 0

Question 7: how does multi-thread-block work?

Question 8: how to invoke multi-thread-block?

Example 2: multicore vector addition [2]

```
__global__  void vecadd_multicore( float* C, float* A, float* B,
           int threads, int N)
{
  int bx = blockIdx.x;
  int i = bx*threads + threadIdx.x ;
  C[i] = A[i] + B[i] ;

#ifdef __DEVICE_EMULATION__
  printf("bx = %d\n", bx ) ;
#endif
}
```

$SP0$ $SP1$ $SP2$ $SP3$

$block[0]$	$block[1]$	$block[2]$	$block[3]$

$thread[0]$	$thread[1]$	$thread[2]$	$thread[3]$

$threads = 4$

$N = (\#\ of\ \ block) \times threads = 4 \times 4 = 16$

i = bx*threads + threadIdx.x

$block[0]$	$block[1]$	$block[2]$	$block[3]$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$	$A[12]$	$A[13]$	$A[14]$	$A[15]$

Example 2: multicore vector addition [3]

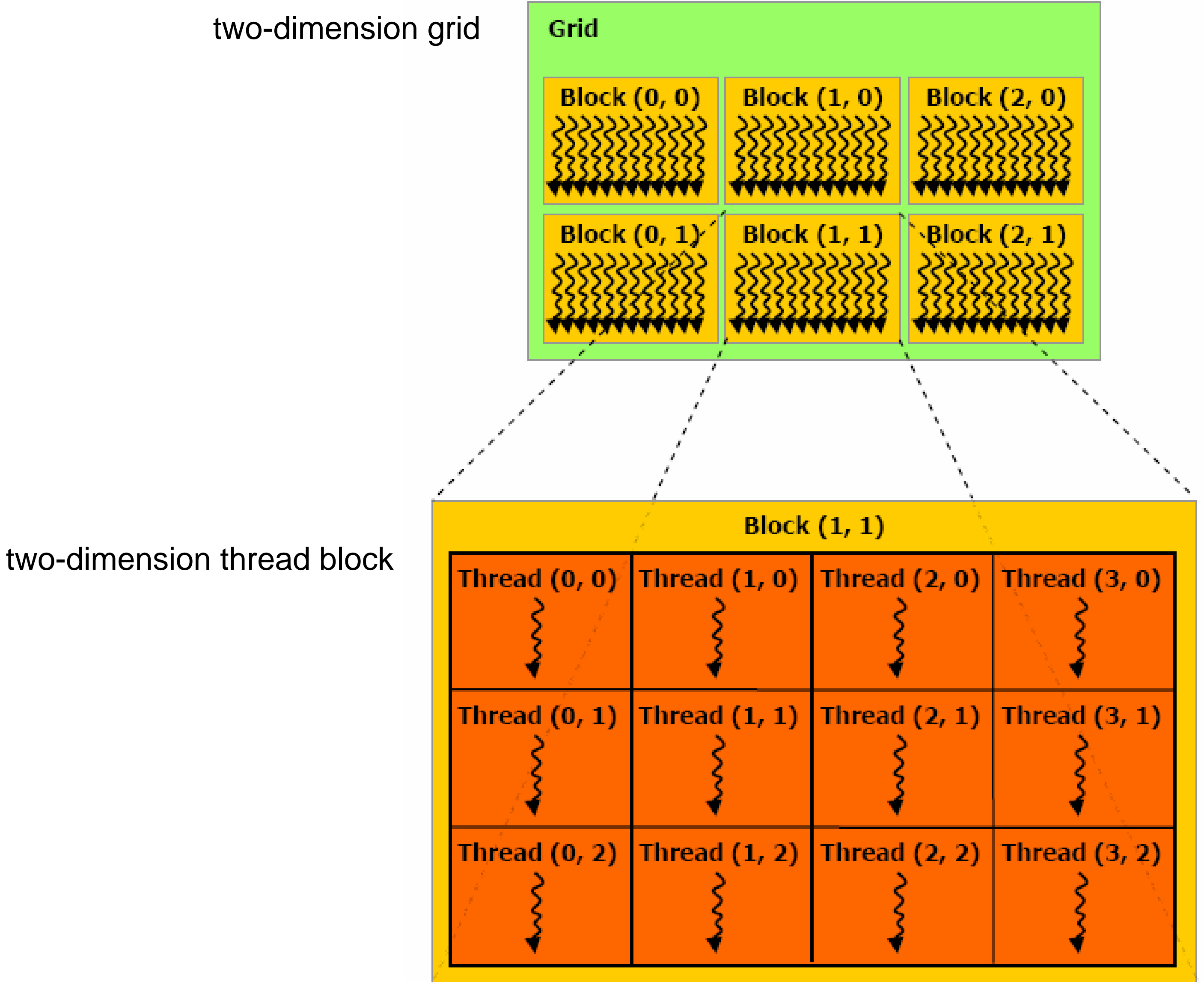
vecadd_GPU.cu

```
11 void vecadd_GPU(float* h_C, const float* h_A, const float* h_B,
12     unsigned int num_block, unsigned int threads )
13 {
14   unsigned int N = num_block*threads ;
15 
16   unsigned int mem_size_A = sizeof(float) * N ;
17   unsigned int mem_size_B = sizeof(float) * N ;
18 
19   // allocate device memory
20   float* d_A;
21   CUDA_SAFE_CALL(cudaMalloc((void**) &d_A, mem_size_A));
22   float* d_B;
23   CUDA_SAFE_CALL(cudaMalloc((void**) &d_B, mem_size_B));
24 
25   // copy host memory to device
26   CUDA_SAFE_CALL(cudaMemcpy(d_A, h_A, mem_size_A,
27                             cudaMemcpyHostToDevice) );
28   CUDA_SAFE_CALL(cudaMemcpy(d_B, h_B, mem_size_B,
29                             cudaMemcpyHostToDevice) );
30 
31   // allocate device memory for result
32   unsigned int mem_size_C = sizeof(float) * N ;
33   float* d_C;
34   CUDA_SAFE_CALL(cudaMalloc((void**) &d_C, mem_size_C));

41   // execute the kernel
42 //   vecadd<<< 1, N >>>(d_C, d_A, d_B, N);
43   vecadd_multicore<<< num_block, threads >>>( d_C, d_A, d_B, threads, N) ;
```

one-dimension thread block

one-dimension grid

Example 2: multicore vector addition [4]



When do matrix – matrix product, we will use two-dimensional index

Example 2: multicore vector addition (driver) [5]

vecadd.cu

```
48 void runTest(int argc, char** argv)
49 {
50   unsigned int num_block = 8 ;
51   unsigned int threads = 512 ;
52   unsigned int N = num_block*threads ;
53
54   printf("num_block = %d, threads = %d, N = %6.2f (KB)\n",
55       num_block, threads, N*4./1024. );
56
57   CUT_DEVICE_INIT(argc, argv);
58
59   // set seed for rand()
60   srand(2006);
61
62   // allocate host memory for matrices A and B
63   unsigned int size_A = N ;
64   unsigned int mem_size_A = sizeof(float) * size_A;
65   float* h_A = (float*) malloc(mem_size_A);
66
67   unsigned int size_B = N ;
68   unsigned int mem_size_B = sizeof(float) * size_B;
69   float* h_B = (float*) malloc(mem_size_B);
70
71   // allocate host memory for the result
72   unsigned int size_C = N ;
73   unsigned int mem_size_C = sizeof(float) * size_C;
74   float* h_C = (float*) malloc(mem_size_C);

76   // initialize host memory
77   randomInit(h_A, size_A);
78   randomInit(h_B, size_B);
79
80   vecadd_GPU( h_C, h_A, h_B, num_block, threads ) ;
```

Maximum size of each dimension of a grid of thread blocks is 65535

Maximum number of threads per block is 512

Example 2: multicore vector addition (result) [6]

$threads = 512 \qquad N = (\#\ of\ \ block) \times threads \qquad size = N \times sizeof(float)\ Byte$

Experimental platform: Geforce 9600 GT

$C = A + B$ ← GPU (ms)

Copy C from device to host ← Device → Host (ms)

Table 1

# of block	size	GPU (ms)	Device → Host (ms)	CPU (ms)
16	32 KB	0.03	0.059	0
32	64 KB	0.032	0.109	0
64	128 KB	0.041	0.235	0
128	256 KB	0.042	0.426	0
256	512 KB	0.044	0.814	0
512	1.024 MB	0.038	1.325	0
1024	2.048 MB	0.04	2.471	0
2048	4.096 MB	0.044	4.818	0
4096	8.192 MB	0.054	9.656	20
8192	16.384 MB	0.054	19.156	30
16384	32.768 MB	0.045	37.75	60
32768	65.536 MB	0.047	75.303	120
65535	131 MB	0.045	149.914	230

Example 2: multicore vector addition [7]

vecadd_GPU.cu

```
void vecadd_GPU(float* h_C, const float* h_A, const float* h_B,
                unsigned int num_block, unsigned int threads )
{
        unsigned int N = num_block*threads ;
        unsigned int mem_size_A = sizeof(float) * N ;
        unsigned int mem_size_B = sizeof(float) * N ;

        // allocate device memory
        float* d_A;
        CUDA_SAFE_CALL(cudaMalloc((void**) &d_A, mem_size_A));
        float* d_B;
        CUDA_SAFE_CALL(cudaMalloc((void**) &d_B, mem_size_B));

        // copy host memory to device
        CUDA_SAFE_CALL(cudaMemcpy(d_A, h_A, mem_size_A,
                                  cudaMemcpyHostToDevice) );
        CUDA_SAFE_CALL(cudaMemcpy(d_B, h_B, mem_size_B,
                                  cudaMemcpyHostToDevice) );

        // allocate device memory for result
        unsigned int mem_size_C = sizeof(float) * N ;
        float* d_C;
        CUDA_SAFE_CALL(cudaMalloc((void**) &d_C, mem_size_C));

        // create and start timer
        unsigned int timer = 0;
        CUT_SAFE_CALL(cutCreateTimer(&timer));
        CUT_SAFE_CALL(cutStartTimer(timer));

        // execute the kernel
        vecadd_multicore<<< num_block, threads >>>( d_C, d_A, d_B, threads, N) ;

        // make sure all threads are done
        cudaThreadSynchronize();

        // stop and destroy timer
        CUT_SAFE_CALL(cutStopTimer(timer));
        printf("in GPU, C = A + B: %f (ms)\n",
           cutGetTimerValue(timer));
        CUT_SAFE_CALL(cutDeleteTimer(timer));
```

All threads work asynchronous

Example 2: multicore vector addition (result, correct timing) [8]

$threads = 512$ $N = (\#\ of\ \ block) \times threads$ $size = N \times sizeof\,(float)\ Byte$

Experimental platform: Geforce 9600 GT

$C = A + B$

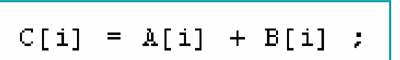
Copy C from device to host

Table 2

# of block	size	GPU (ms)	Device → Host (ms)	CPU (ms)
16	32 KB	0.04	0.059	0
32	64 KB	0.056	0.122	0
64	128 KB	0.057	0.242	0
128	256 KB	0.063	0.381	0
256	512 KB	0.086	0.67	0
512	1.024 MB	0.144	1.513	0
1024	2.048 MB	0.237	2.812	10
2048	4.096 MB	0.404	5.426	10
4096	8.192 MB	0.755	9.079	20
8192	16.384 MB	1.466	17.873	30
16384	32.768 MB	2.86	34.76	60
32768	65.536 MB	5.662	70.286	130
65535	131 MB	11.285	138.793	240

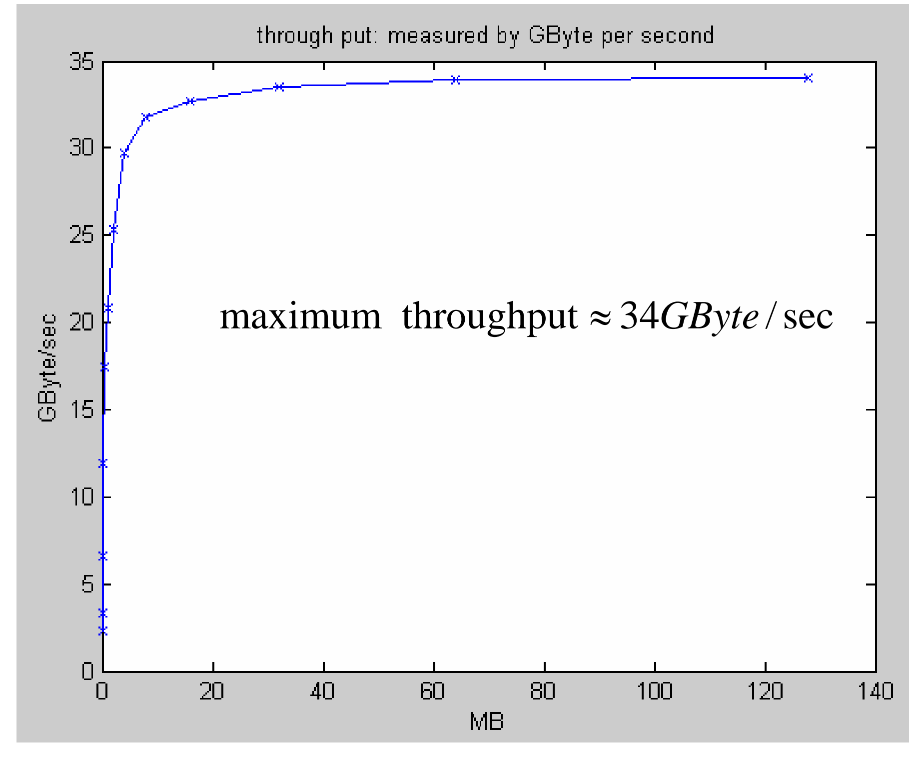
Example 2: multicore vector addition (throughput) [8]

$$\text{define throughput} = \frac{\text{Total data transfer in byte or bit (size)}}{\text{Total time (GPU)}} \times 3$$



```
C[i] = A[i] + B[i] ;
```

1. Load *A[i]*
2. Load *B[i]*
3. store *C[i]*



vectors *A, B, C* are stored in global memory and 3 memory fetch only use a “add” operation, not floating point operation dominanted.

Memory Specs: Geforce 9600GT

Memory Clock (MHz)	900 MHz
Standard Memory Config	512 MB
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	57.6

Exercise

1. So far, one thread is responsible for one data element, can you change this, say one thread takes care of several data entries ?

vecadd_kernel.cu

```
__global__  void vecadd( float* C, float* A, float* B, int N)
{
#ifdef __DEVICE_EMULATION__
  int bx = blockIdx.x ;
  assert( 0 == bx) ;
#endif

  int i = threadIdx.x ;
  C[i] = A[i] + B[i] ;
}
```

vecadd_kernel.cu

```
__global__  void vecadd_multicore( float* C, float* A, float* B,
          int threads, int N)
{
  int bx = blockIdx.x;
  int i = bx*threads + threadIdx.x ;
  C[i] = A[i] + B[i] ;

#ifdef __DEVICE_EMULATION__
  printf("bx = %d\n", bx ) ;
#endif
}
```

2. Maximum number of threads per block is 512, when data set is more than 512, we use multi-thread-block to do parallel computing, however Maximum size of each dimension of a grid of thread blocks is 65535, when data set is more than 131MB, how can we proceed?
3. From table 2, data transfer from device to host is about half of CPU computation, it means that if we can accelerate CPU computation, then GPU has no advantage, right?
4. measure your video card and fill-in table 2, also try double-precision if your hardware supports.

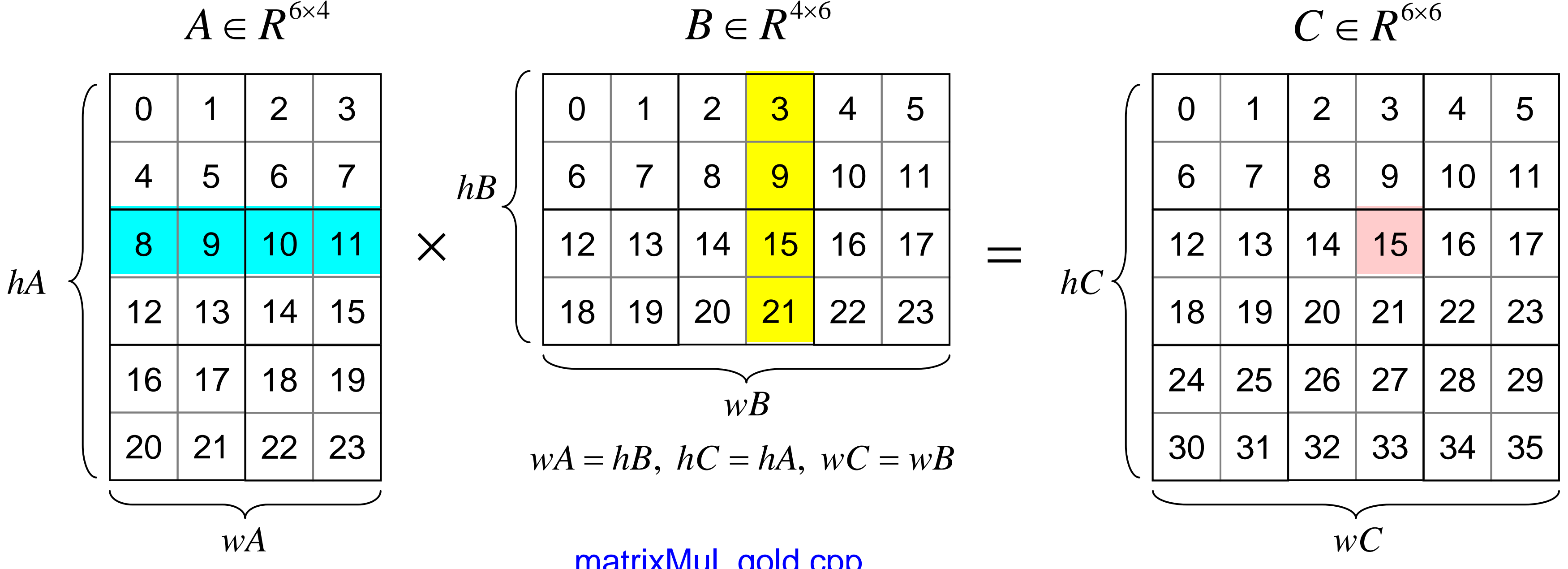
OutLine

- CUDA introduction
- Example 1: vector addition, single core
- Example 2: vector addition, multi-core

- Example 3: matrix-matrix product
  - grid versus thread block
- Embed nvcc to vc2005

Example 3: matrix-matrix product (CPU-version) [1]

Consider matrix-matrix product $C = AB$, all matrices are indexed in row-major and starting from zero (C-like)



matrixMul_gold.cpp

```
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;
        }
}
```

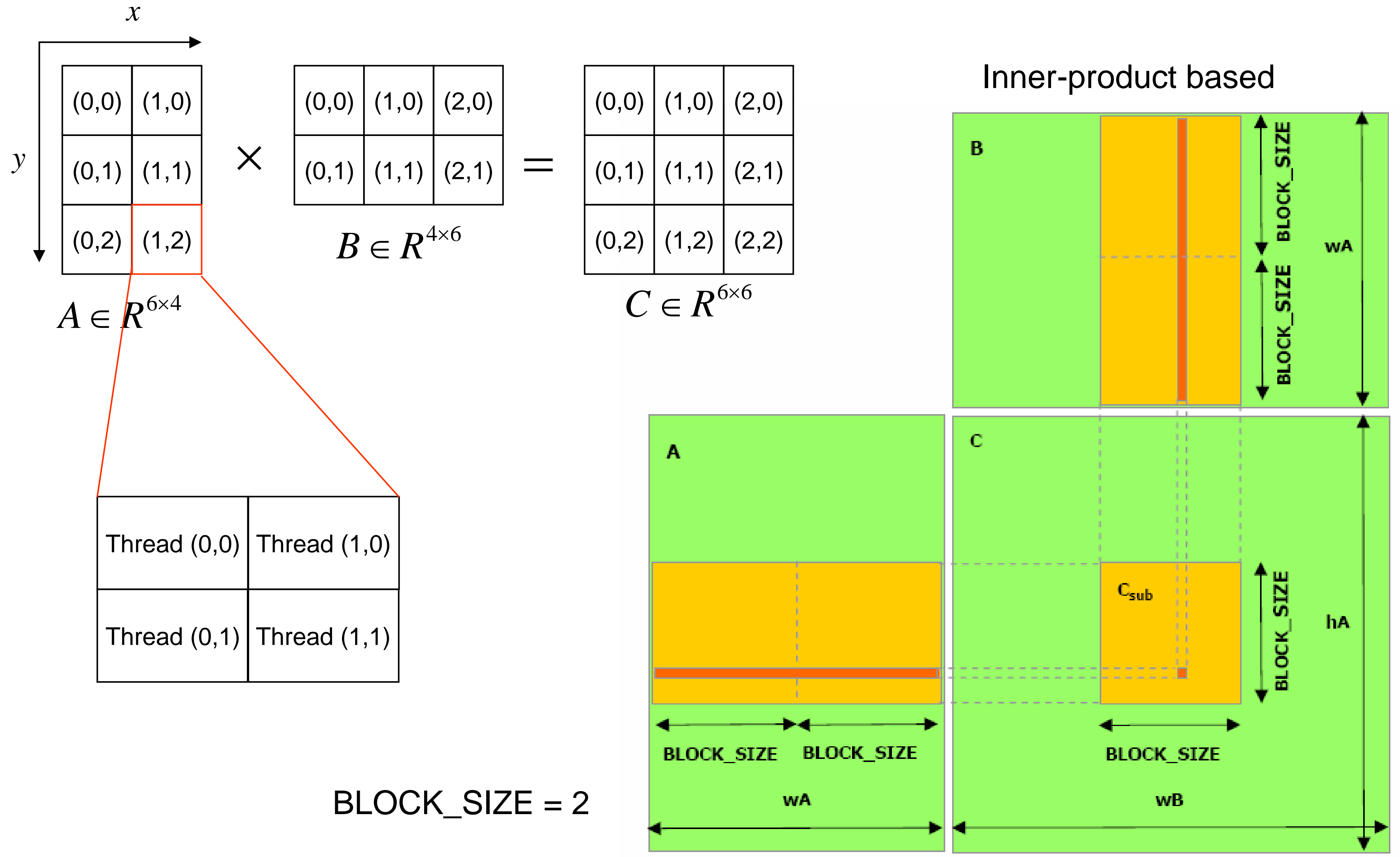
$$A(i,k) = i \times wA + k$$

$$B(k,j) = k \times wB + j$$

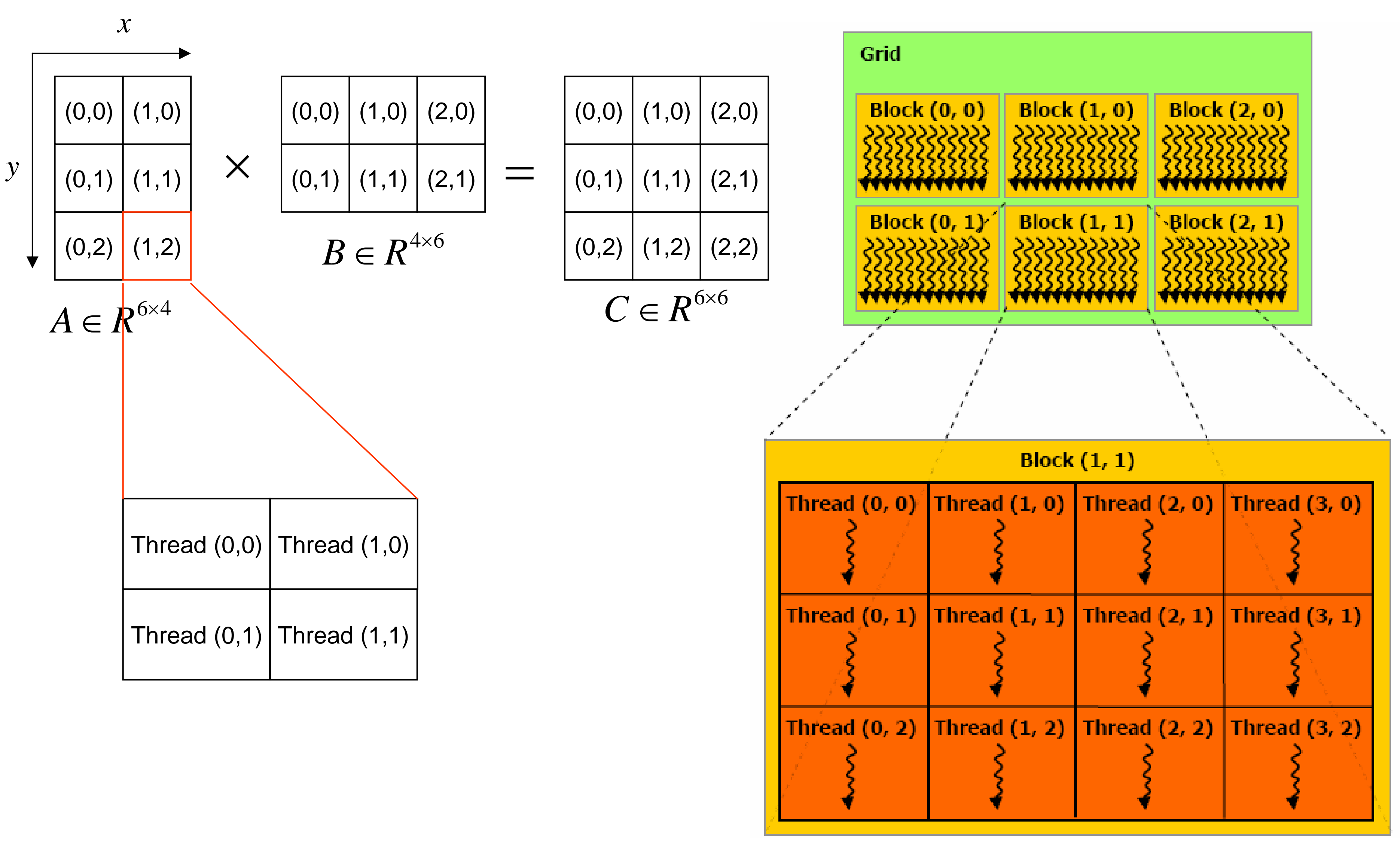
$$C(i,j) = i \times wC + j$$

Example 3: matrix-matrix product (GPU-version) [2]

We use 2x2 block as a unit and divide matrix C into 6 block. Then we plan to deal with each sub-matrix of C with one thread-block.

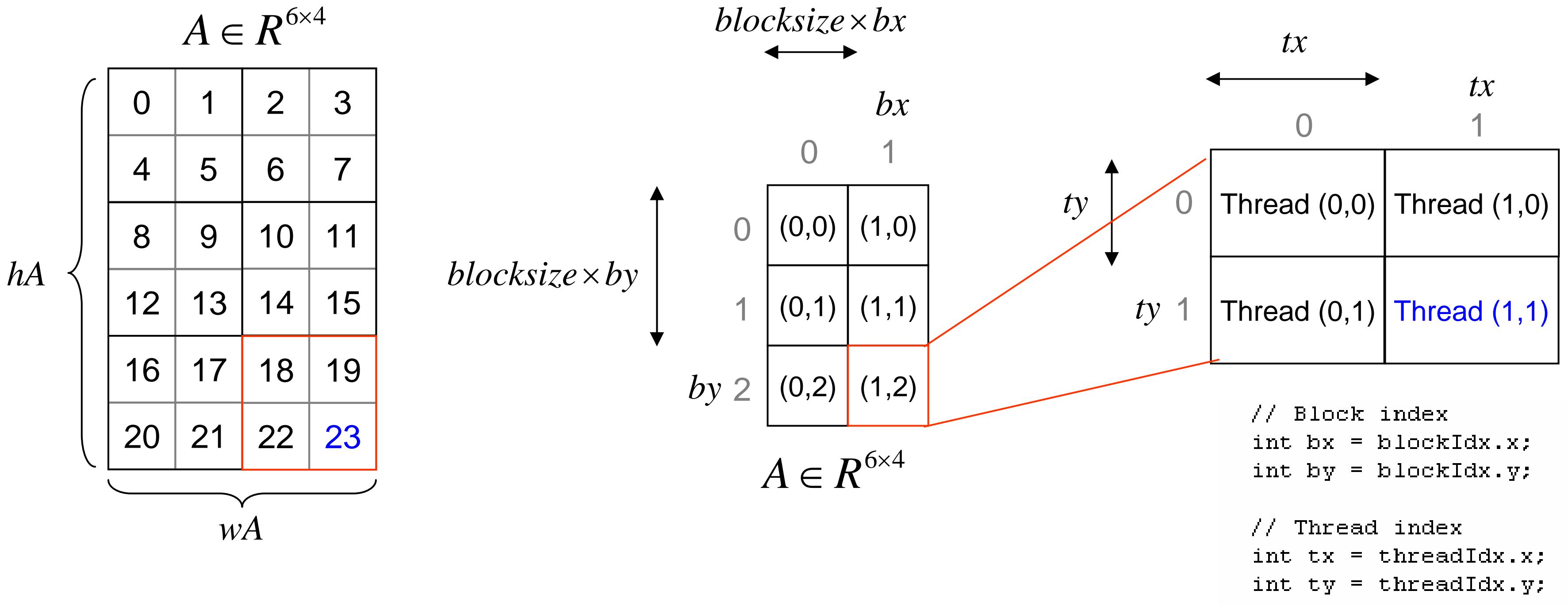


Example 3: matrix-matrix product (GPU-version) [3]



Question 9: how to transform (grid index, thread index) to physical index ?

Example 3: matrix-matrix product (index) [4]



The physical index of first entry in block $(bx, by) = (blocksize \times by) \times wA + blocksize \times bx$

e.g. The physical index of first entry in block $(1,2) = (2\times2)\times4 + 2\times1 = 16 + 2 = 18$

The physical index of (block index, thread index) is $((bx,by),(tx,ty)) = (bx,by) + (wA \times ty) + tx$

e.g. $((bx,by),(tx,ty)) = ((1,2),(1,1)) = 18 + (4\times1) + 1 = 23$

global index

$((bx,by),(tx,ty)) \longrightarrow (blocksize\times bx + tx,\ blocksize \times by + ty) \longrightarrow$ row-major

Example 3: matrix-matrix product [5]

Consider $C(i,j) = \sum_{k=1}^{wA} A(i,k)B(k,j)$ for all $(i,j) \in block(1,1)$ computed simultaneously

(0,0)	(1,0)
(0,1)	(1,1)
(0,2)	(1,2)

$A \in R^{6\times4}$

$\times$

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)

$B \in R^{4\times6}$

$=$

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,2)	(1,2)	(2,2)

$C \in R^{6\times6}$

or equivalently A ((0,1)) B ((1,0)) $+$ A ((1,1)) B ((1,1)) $=$ C ((1,1))

A ((0,1)) B ((1,0)) $+$ A ((1,1)) B ((1,1)) $=$ C ( )

A ((0,1)) B ((1,0)) $+$ A ((1,1)) B ((1,1)) $=$ C ( )

A ((0,1)) B ((1,0)) $+$ A ((1,1)) B ((1,1)) $=$ C ( )

A ((0,1)) B ((1,0)) $+$ A ((1,1)) B ((1,1)) $=$ C ( )

Executed in a thread block, say computed simultaneously.

Clearly we need 4 threads to run at the same time

Example 3: matrix-matrix product [6]

since all 4 threads share the same submatrix of *A* and *B*, we use share memory (on-chip) to store submatrix of *A* and *B* to decrease memory latency.

Step 1: add first product term to submatrix of C

$$C = A\begin{pmatrix}(0,1)\end{pmatrix} B\begin{pmatrix}(1,0)\end{pmatrix} + A\begin{pmatrix}(1,1)\end{pmatrix} B\begin{pmatrix}(1,1)\end{pmatrix}$$

$$C = A\begin{pmatrix}(0,1)\end{pmatrix} B\begin{pmatrix}(1,0)\end{pmatrix} + A\begin{pmatrix}(1,1)\end{pmatrix} B\begin{pmatrix}(1,1)\end{pmatrix}$$

$$C = A\begin{pmatrix}(0,1)\end{pmatrix} B\begin{pmatrix}(1,0)\end{pmatrix} + A\begin{pmatrix}(1,1)\end{pmatrix} B\begin{pmatrix}(1,1)\end{pmatrix}$$

$$C = A\begin{pmatrix}(0,1)\end{pmatrix} B\begin{pmatrix}(1,0)\end{pmatrix} + A\begin{pmatrix}(1,1)\end{pmatrix} B\begin{pmatrix}(1,1)\end{pmatrix}$$

share memory

$$As = A\begin{pmatrix}(0,1)\end{pmatrix}$$

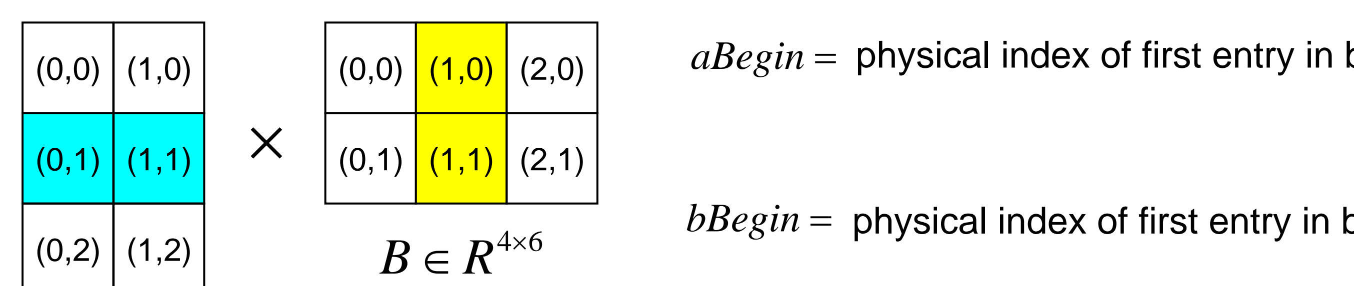
$$Bs = B\begin{pmatrix}(1,0)\end{pmatrix}$$

```
// Declaration of the shared memory array As used to
// store the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

// Declaration of the shared memory array Bs used to
// store the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
```

The __shared__ quantifier declares a variable

- Resides in the shared memory space of a thread block,
- Has the lifetime of the block,
- Is only accessible from all the threads within the block.

Example 3: matrix-matrix product [7]



$A \in R^{6\times4}$ $\times$ $B \in R^{4\times6}$

$aBegin =$ physical index of first entry in block $A\left(\boxed{(0,1)}\right)$

$bBegin =$ physical index of first entry in block $B\left(\boxed{(1,0)}\right)$

The physical index of first entry in block $(bx,by) = (blocksize \times by) \times wA + blocksize \times bx$

$aBegin = (0,by) = (blocksize \times by) \times wA$ $bBegin = (bx,0) = blocksize \times bx$

Step 1: copy $A\left(\boxed{(0,1)}\right)$ to As and $B\left(\boxed{(1,0)}\right)$ to Bs

```
// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix
AS(ty, tx) = A[aBegin + wA * ty + tx];
BS(ty, tx) = B[bBegin + wB * ty + tx];

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

← all threads in this thread block do copy action before submatrix C is computed

The physical index of (block index, thread index) is $((bx,by),(tx,ty)) = (bx,by) + (wA \times ty) + tx$

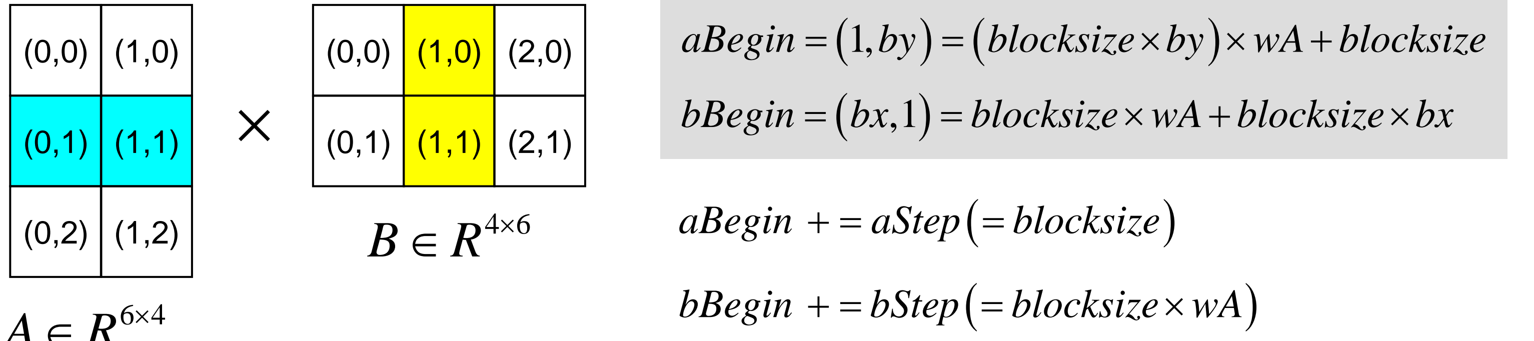
Example 3: matrix-matrix product [8]

Step 2: add first product term to submatrix of C

```
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
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
__syncthreads();
```

Note that each thread in thread block has its ***private*** variable *Csub*

Step 3: move aBegin and bBegin to next block



Example 3: matrix-matrix product [9]

Step 4: copy A (1,1) to As and B (1,1) to Bs

```
// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix
AS(ty, tx) = A[aBegin + wA * ty + tx];
BS(ty, tx) = B[bBegin + wB * ty + tx];

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

Step 5: add second product term to submatrix of C

$C \mathrel{+}= A$ (1,1) B (1,1)

$C \mathrel{+}= A$ (1,1) B (1,1)

$C \mathrel{+}= A$ (1,1) B (1,1)

$C \mathrel{+}= A$ (1,1) B (1,1)

```
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
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
__syncthreads();
```

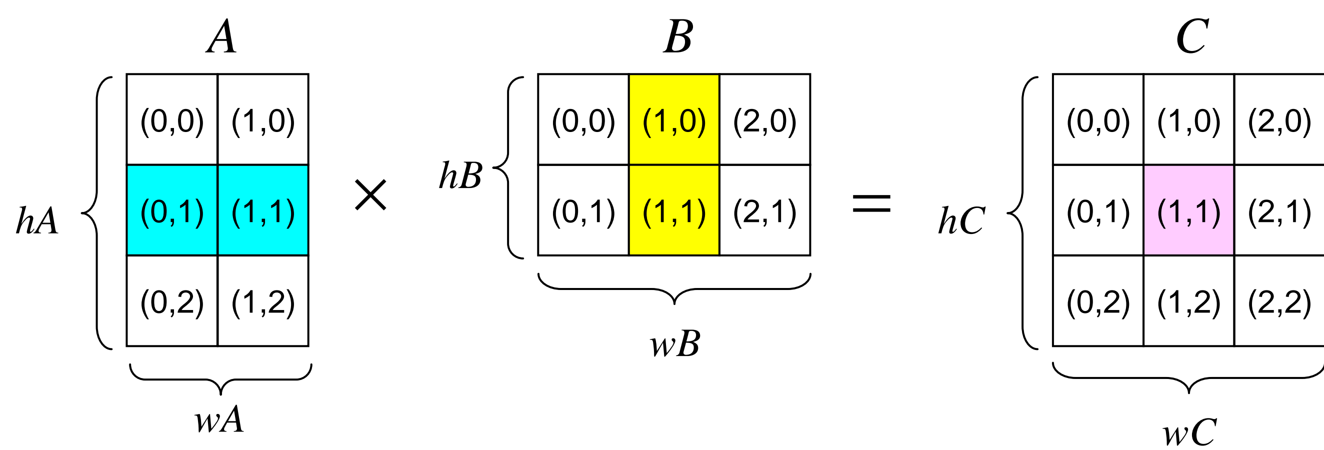
Example 3: matrix-matrix product (source code) [10]

see /usr/local/NVIDIA_CUDA_SDK\projects\matrixMul

matrixMul.h

```
36  #ifndef _MATRIXMUL_H_
37  #define _MATRIXMUL_H_
38
39  // Thread block size
40  #define BLOCK_SIZE 16
41
42  // Matrix dimensions
43  // (chosen as multiples of the thread block size for simplicity)
44  #define WA (2 * BLOCK_SIZE) // Matrix A width
45  #define HA (3 * BLOCK_SIZE) // Matrix A height
46  #define WB (3 * BLOCK_SIZE) // Matrix B width
47  #define HB WA  // Matrix B height
48  #define WC WB  // Matrix C width
49  #define HC HA  // Matrix C height
50
51  #endif // _MATRIXMUL_H_
```

The amount of shared memory available per multiprocessor is 16KB (since multiprocessor has 8 SP, each SP has only 2KB)



Example 3: matrix-matrix product (source code) [11]

matrixMul_kernel.cu

```
71 __global__ void
72 matrixMul( float* C, float* A, float* B, int wA, int wB)
73 {
74     // Block index
75     int bx = blockIdx.x;
76     int by = blockIdx.y;
77
78     // Thread index
79     int tx = threadIdx.x;
80     int ty = threadIdx.y;
81
82     // Index of the first sub-matrix of A processed by the block
83     int aBegin = wA * BLOCK_SIZE * by;
84
85     // Index of the last sub-matrix of A processed by the block
86     int aEnd   = aBegin + wA - 1;
87
88     // Step size used to iterate through the sub-matrices of A
89     int aStep  = BLOCK_SIZE;
90
91     // Index of the first sub-matrix of B processed by the block
92     int bBegin = BLOCK_SIZE * bx;
93
94     // Step size used to iterate through the sub-matrices of B
95     int bStep  = BLOCK_SIZE * wB;
96
97     // Csub is used to store the element of the block sub-matrix
98     // that is computed by the thread
99     float Csub = 0;
```

Each thread has its own index (bx, by) and (tx, ty)

Each thread has its private variable Csub

Example 3: matrix-matrix product (source code) [12]

```
101     // Loop over all the sub-matrices of A and B
102     // required to compute the block sub-matrix
103     for (int a = aBegin, b = bBegin;
104              a <= aEnd;
105              a += aStep, b += bStep) {
106
107         // Declaration of the shared memory array As used to
108         // store the sub-matrix of A
109         __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
110
111         // Declaration of the shared memory array Bs used to
112         // store the sub-matrix of B
113         __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
114
115         // Load the matrices from device memory
116         // to shared memory; each thread loads
117         // one element of each matrix
118         AS(ty, tx) = A[a + wA * ty + tx];
119         BS(ty, tx) = B[b + wB * ty + tx];
120
121         // Synchronize to make sure the matrices are loaded
122         __syncthreads();
123
124         // Multiply the two matrices together;
125         // each thread computes one element
126         // of the block sub-matrix
127         for (int k = 0; k < BLOCK_SIZE; ++k)
128             Csub += AS(ty, k) * BS(k, tx);
130         // Synchronize to make sure that the preceding
131         // computation is done before loading two new
132         // sub-matrices of A and B in the next iteration
133         __syncthreads();
134     }
135
136     // Write the block sub-matrix to device memory;
137     // each thread writes one element
138     int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
139     C[c + wB * ty + tx] = Csub;
140 }
```

1. copy submatrix of A and B to shared memory, this is done by all threads in this thread block (lines 118–119)
2. Add partial result of matrix-matrix product into Csub (line 128)
3. Each thread stores back their computed result into global matrix C (line 139)

$$(bx,by) = (blocksize \times by) \times wA + blocksize \times bx$$

$$((bx,by),(tx,ty)) = (bx,by) + (wA \times ty) + tx$$

Example 3: matrix-matrix product (driver) [13]

matrixMul.cu

```
48 #include <stdlib.h>
49 #include <stdio.h>
50 #include <string.h>
51 #include <math.h>
52 // includes, project
53 #include <cutil.h>
54 // includes, kernels
55 #include <matrixMul_kernel.cu>
56
57 void runTest(int argc, char** argv);
58 void randomInit(float*, int);
59 void printDiff(float*, float*, int, int);
60
61 extern "C" {
62 void computeGold(float*, const float*, const float*,
63     unsigned int, unsigned int, unsigned int);
64 }
65
66 int main(int argc, char** argv)
67 {
68     runTest(argc, argv);
69
70     CUT_EXIT(argc, argv);
71 }
```

vecadd.cu

```
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5 #include <string.h>
 6 #include <math.h>
 7
 8 // includes, project
 9 #include <cutil.h>
10
11 // includes, kernels
12 #include <vecadd_kernel.cu>
13 #include <vecadd_GPU.cu>
14
15 // declaration, forward
16 void runTest(int argc, char** argv);
17 void randomInit(float*, int);
18 void printDiff(float*, float*, int, int);
19
20 extern "C" {
21 void computeGold(float*, const float*, const float*,
22        unsigned int );
23 void vecadd_GPU(float* h_C, const float* h_A,
24        const float* h_B, unsigned int N) ;
25 }
26
27 int main(int argc, char** argv)
28 {
29     runTest(argc, argv);
30
31     CUT_EXIT(argc, argv);
32 }
```

The same structure

Example 3: matrix-matrix product (driver) [14]

matrixMul.h

```
#ifndef _MATRIXMUL_H_
#define _MATRIXMUL_H_

// Thread block size
#define BLOCK_SIZE 16

// Matrix dimensions
#define WA (2 * BLOCK_SIZE) // Matrix A width
#define HA (3 * BLOCK_SIZE) // Matrix A height
#define WB (3 * 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_
```

matrixMul.cu

```
74  void runTest(int argc, char** argv)
75  {
76      CUT_DEVICE_INIT(argc, argv);
77
78      // set seed for rand()
79      srand(2006);
80
81      // allocate host memory for matrices A and B
82      unsigned int size_A = WA * HA;
83      unsigned int mem_size_A = sizeof(float) * size_A;
84      float* h_A = (float*) malloc(mem_size_A);
85      unsigned int size_B = WB * HB;
86      unsigned int mem_size_B = sizeof(float) * size_B;
87      float* h_B = (float*) malloc(mem_size_B);
88
89      // initialize host memory
90      randomInit(h_A, size_A);
91      randomInit(h_B, size_B);
92
93      // allocate device memory
94      float* d_A;
95      CUDA_SAFE_CALL(cudaMalloc((void**) &d_A, mem_size_A));
96      float* d_B;
97      CUDA_SAFE_CALL(cudaMalloc((void**) &d_B, mem_size_B));
98
99      // copy host memory to device
100     CUDA_SAFE_CALL(cudaMemcpy(d_A, h_A, mem_size_A,
101                               cudaMemcpyHostToDevice) );
102     CUDA_SAFE_CALL(cudaMemcpy(d_B, h_B, mem_size_B,
103                               cudaMemcpyHostToDevice) );
```

Allocate host memory for matrix *A*, *B*

Allocate device memory for matrix *A*, *B*

Example 3: matrix-matrix product (driver) [15]

matrixMul.cu

```
105     // allocate device memory for result
106     unsigned int size_C = WC * HC;
107     unsigned int mem_size_C = sizeof(float) * size_C;
108     float* d_C;
109     CUDA_SAFE_CALL(cudaMalloc((void**) &d_C, mem_size_C));
110
111     // allocate host memory for the result
112     float* h_C = (float*) malloc(mem_size_C);
113
114     // create and start timer
115     unsigned int timer = 0;
116     CUT_SAFE_CALL(cutCreateTimer(&timer));
117     CUT_SAFE_CALL(cutStartTimer(timer));
118
119     // setup execution parameters
120     dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
121     dim3 grid(WC / threads.x, HC / threads.y);
122
123     // execute the kernel
124     matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);
125
126     // check if kernel execution generated and error
127     CUT_CHECK_ERROR("Kernel execution failed");
128
129     // copy result from device to host
130     CUDA_SAFE_CALL(cudaMemcpy(h_C, d_C, mem_size_C,
131                               cudaMemcpyDeviceToHost) );
```

matrixMul.h

```
#ifndef _MATRIXMUL_H_
#define _MATRIXMUL_H_

// Thread block size
#define BLOCK_SIZE 16

// Matrix dimensions
#define WA (2 * BLOCK_SIZE) // Matrix A width
#define HA (3 * BLOCK_SIZE) // Matrix A height
#define WB (3 * 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_
```

threads = (16, 16, 1)

grid = (3, 3, 1)

dim3 Type

This type is an integer vector type based on `uint3` that is used to specify dimensions. When defining a variable of type `dim3`, any component left unspecified is initialized to 1.

Example 3: matrix-matrix product (driver) [16]

```
133     // stop and destroy timer
134     CUT_SAFE_CALL(cutStopTimer(timer));
135     printf("Processing time: %f (ms) \n", cutGetTimerValue(timer));
136     CUT_SAFE_CALL(cutDeleteTimer(timer));
137
138     // compute reference solution
139     float* reference = (float*) malloc(mem_size_C);
140     computeGold(reference, h_A, h_B, HA, WA, WB);
141
142     // check result
143     CUTBoolean res = cutCompareL2fe(reference, h_C, size_C, 1e-6f);
144     printf("Test %s \n", (1 == res) ? "PASSED" : "FAILED");
145     if (res!=1) printDiff(reference, h_C, WC, HC);
146
147     // clean up memory
148     free(h_A);
149     free(h_B);
150     free(h_C);
151     free(reference);
152     CUDA_SAFE_CALL(cudaFree(d_A));
153     CUDA_SAFE_CALL(cudaFree(d_B));
154     CUDA_SAFE_CALL(cudaFree(d_C));
155 }
157 // Allocates a matrix with random float entries.
158 void randomInit(float* data, int size)
159 {
160     for (int i = 0; i < size; ++i)
161         data[i] = rand() / (float)RAND_MAX;
162 }
163
164 void printDiff(float *data1, float *data2, int width, int height)
165 {
166   int i,j,k;
167   int error_count=0;
168   for (j=0; j<height; j++) {
169     for (i=0; i<width; i++) {
170       k = j*width+i;
171       if (data1[k] != data2[k]) {
172          printf("diff(%d,%d) CPU=%4.4f, GPU=%4.4f n", i,j, data1[k], data2[k]);
173          error_count++;
174       }
175     }
176   }
177   printf(" nTotal Errors = %d n", error_count);
178 }
```

Example 3: matrix-matrix product (compile on Linux) [17]

Step 1: upload all source files to workstation, assume you put them in directory *matrixMul*

```
[macrold@matrix matrixMul]$ ls
Makefile      matrixMul.cu.cpp  matrixMul_gold.cpp   matrixMul_kernel.cu.cpp
matrixMul.cu  matrixMul.h       matrixMul_kernel.cu
[macrold@matrix matrixMul]$
```

Step 2: edit Makefile by “vi Makefile”

```
INLCUDE = -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include

LIBS = -L/usr/local/NVIDIA_CUDA_SDK/lib -lcutil
LIBS += -L/usr/local/cuda/lib -lcuda -lcudart
LIBS += -L/usr/lib64 -lGL -lGLU

SRC_CU = matrixMul.cu

SRC_CXX = matrixMul_gold.cpp

CXXFlag = -DCUDA_FLOAT_MATH_FUNCTIONS -DCUDA_NO_SM_11_ATOMIC_INTRINSICS

gxxFlag = -m64 -O2

icpcFlag = -mp -O2

nvcc_run:
        nvcc -run $(INLCUDE) $(LIBS) $(SRC_CU)  $(SRC_CXX)
```

Step 3: type “make nvcc_run”

```
[macrold@matrix matrixMul]$ make nvcc_run
nvcc -run -I/usr/local/NVIDIA_CUDA_SDK/common/inc -I/usr/local/cuda/include -L/usr/local/NVIDIA_CUDA_
SDK/lib -lcutil -L/usr/local/cuda/lib -lcuda -lcudart -L/usr/lib64 -lGL -lGLU matrixMul.cu  matrixMul
_gold.cpp
Using device 0: GeForce 9600 GT
Processing time: 0.121000 (ms)
Test PASSED
```

Exercise

- modify code in matrixMul, measure time for computing golden vector , time for C = A*B under GPU and time for data transfer, compare them.

- We have shown you vector addition and matrix-matrix product, which one is better in GPU computation, why?
  (you can compute ratio between floating point operation and memory fetch operation)

- modify source code in matrixMul, use column-major index, be careful indexing rule.

- We have discussed that matrix-vector product has two versions, one is inner-product-based, one is outer-product-based, implement these two methods under GPU

OutLine

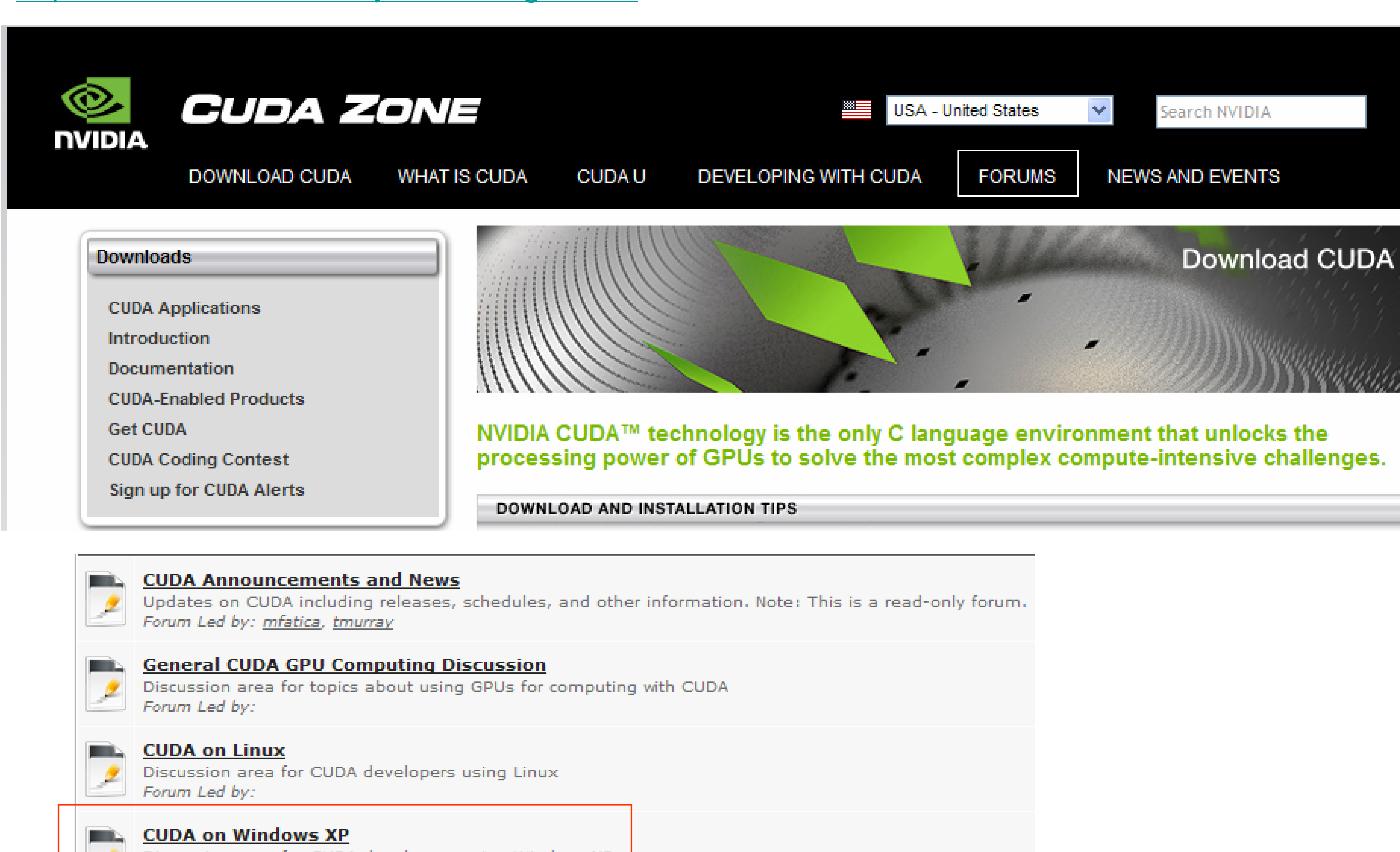
- CUDA introduction
- Example 1: vector addition, single core
- Example 2: vector addition, multi-core

- Example 3: matrix-matrix product

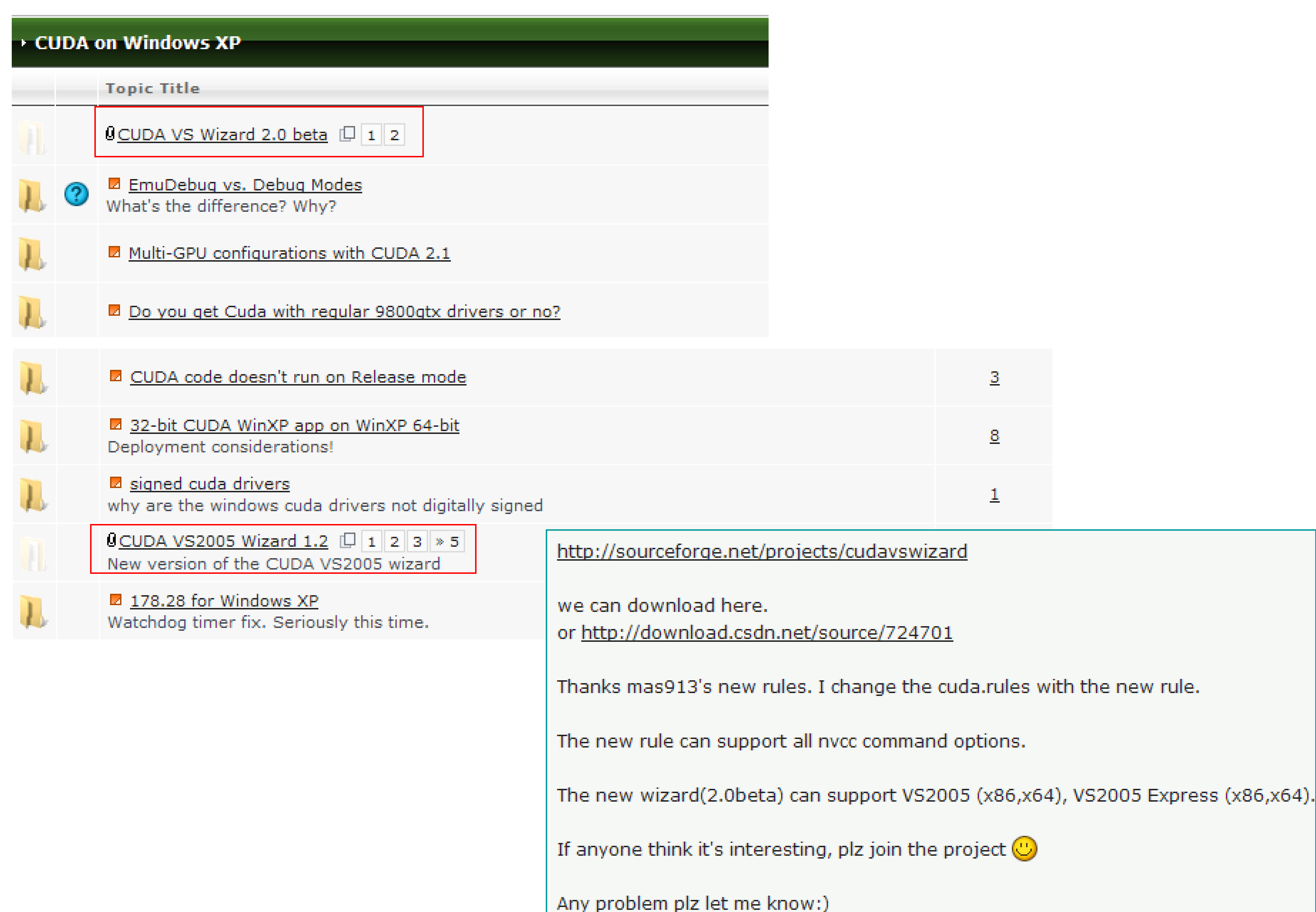
- Embed nvcc to vc2005

Resource: register NVIDIA forum

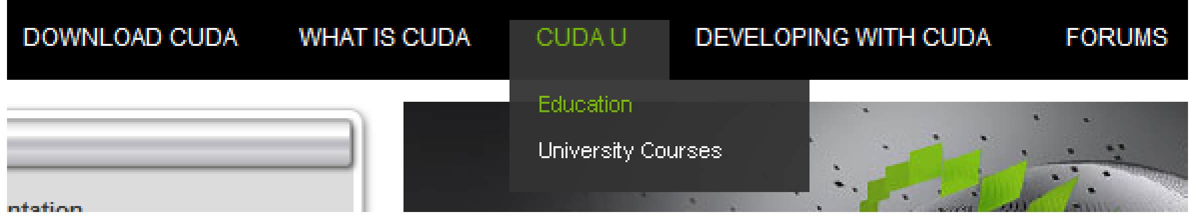
http://www.nvidia.com/object/cuda_get.html



How to embed "nvcc" into VC 2005 [1]



Education: list in NVIDIA website



ITESM Mexico	Ambientes de Programación Avanzada	NA	Isaac Rudomin
Johns Hopkins	General Purpose Computation on the GPU	EN600.407	Matthew Bolitho
Kent State	GPU Computing	CS6/79995	Ye Zhau
McGill	CUDA Programming Environment		Abdelkader Baggag
National Taiwan University	High-performance cryptographic computing, embedded computing		Chen-Mou Cheng
National Taiwan University	Parallel Processing Architectures and Applications (English Chinese)		Chen-Mou Cheng
North Carolina State	Operating System Principles		Frank Mueller
North Carolina State	Graduate Operating Systems	CSC501	Frank Mueller
North Carolina State	Design Automation for VLSI		Xun Liu
University of North Carolina	GPGP: General Purpose computation Using Graphics Processors	COMP790-058	Dinesh Manocha
University of Maryland	Advanced Computer Graphics	CSMC740	Amitabh Varshney
University of Pennsylvania	GPU Programming & Architecture	CIS665	Gary Katz
University of Rochester	Introduction to Programming in CUDA	Workshop	Alice Quillen
University of Southern California	General Purpose Computing Using GPUs on a Linux Cluster		Dan Davis

鄭振牟教授

Education: course website

http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html

University of Illinois at Urbana-Champaign, taught by Prof. Wen-Mei Hwu

http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html 網頁(P) 工具(O)

Fall 2007 Syllabus (Tentative)

Date	Lecture	Material	Assignments
Week 1: Wed, 8/22	Lecture 1 - Introduction	Slides (ppt) Voice (mp3)	MP-0 Released - CUDA installation, run hello world.
Week 2: Tuesday, 8/28 (Make up, **class in 163EL**)	Lecture 2 - GPU Computing and CUDA Intro	Slides (ppt) Voice (mp3)	Read CUDA Programming Guide 1.0.
Wed, 8/29	Lecture 3 - GPU Computing and CUDA Intro	Slides (ppt) Voice (mp3)	MP-1, Simple Matrix Multiplication and Simple Vector Reduction released
Week 3: Tuesday, 9/4 (Make up, **class in 1109 Siebel Center.**)	Lecture 4 - CUDA memory model, tiling	Slides (ppt) Voice (mp3) Joke (mp3)	
Wed, 9/5: (DK in IL)	Lecture 5 - GPU History	Slides (ppt) Voice (mp3)	MP-2, Tiled Matrix Multiplication released MP-1 (both parts) Due: Wednesday, September 5th at 11:59pm.
Week 4: Mon, 9/10	Lecture 6 - CUDA Hardware	Slides (ppt) Voice (mp3)	

How to embed “nvcc” into VC 2005 [2]

1 On desktop, right click the mouse and choose NVIDIA control panel

2 Choose system information

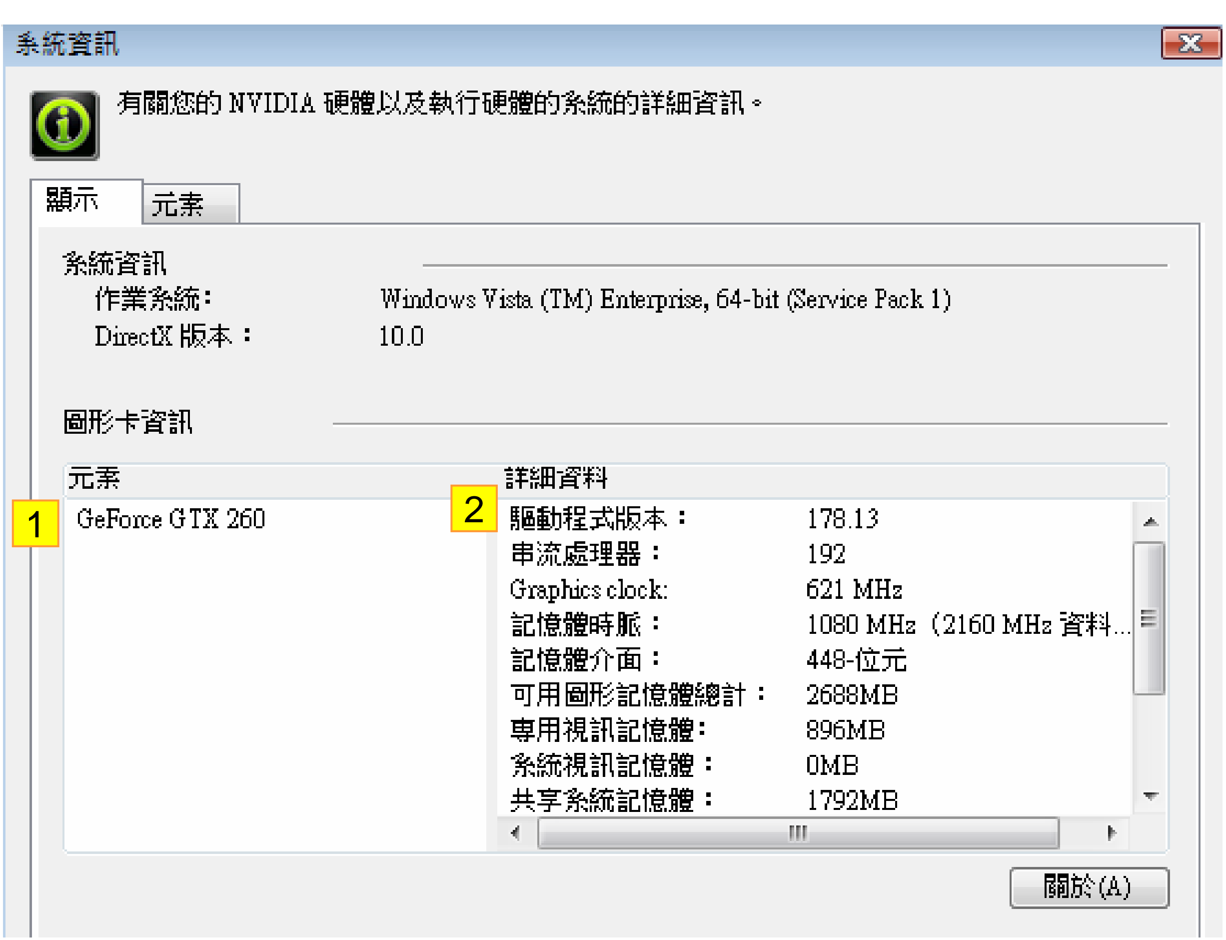
NVIDIA 控制面板
檔案(F) 編輯(E) 檢視(V) 說明(H)
後退 標準設定值
選取一個任務...
3D 設定值
調整影像設定值和預覽
視訊與電視
調整視訊色彩設定值
調整視訊影像設定值
顯示
變更解析度
變更平面顯示器比例縮放
調整桌面色彩設定值
旋轉顯示畫面
設定多台顯示器
2
系統資訊
調整影像設定值和預覽
此頁面可用於預覽您對影像及描繪設定值所做的變更。這些將會
硬體加速 3D 應用程式的預設設定值。
效能：
預覽：
品質：
我要尋找什麼？
移向「效能」一側將會增加圖框速率，但是曲線會出現鋸齒狀。
由 3D 應用程式決定(L)
使用進階 3D 影像設定值(U)
使用我的喜好設定強調(M)： 品質

How to embed “nvcc” into VC 2005 [3]

system information, including

1 chipset

2 driver



How to embed "nvcc" into VC 2005 [4]

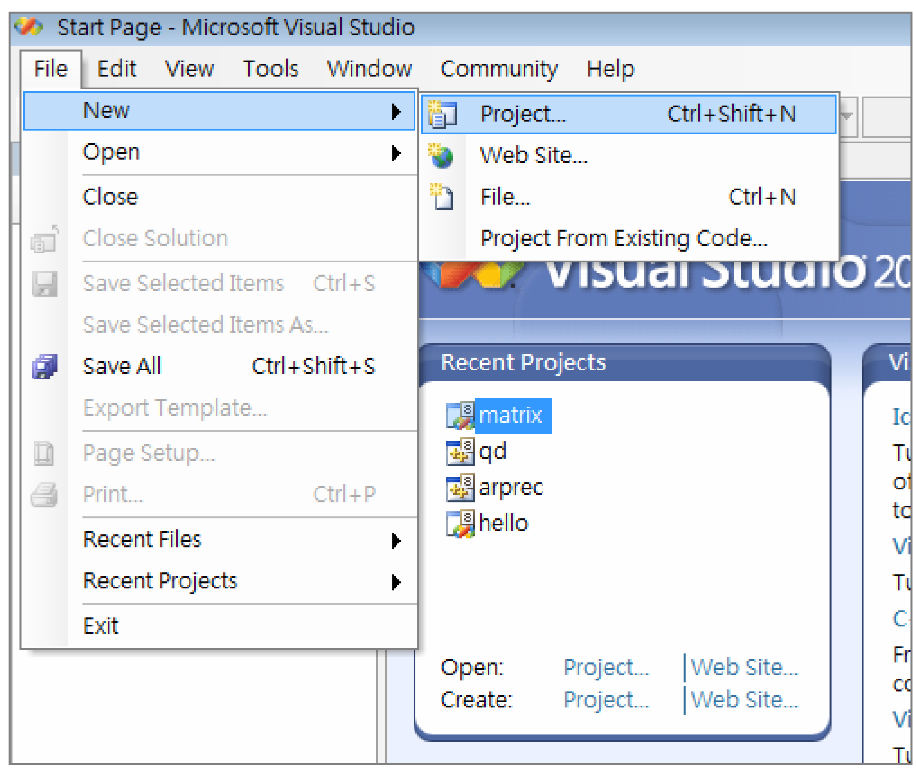
```
系統管理員: 命令提示字元
Microsoft Windows [版本 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserv

C:\Users\root>set
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\root\AppData\Roaming
CC=cl
CNL_COMPILER_VERSION=Microsoft (R) C/C++ Optimizing Compiler
.41 for AMD64
CNL_DIR=C:\Program Files (x86)\VNI\imsl\cnl600
CNL_EXAMPLES=C:\Program Files (x86)\VNI\imsl\cnl600\ms64pc\e
CNL_OS_VERSION=Microsoft Windows Server 2003/XP x64 Edition
CNL_VERSION=6.0.0
CommonProgramFiles=C:\Program Files\Common Files
CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
COMPUTERNAME=FLUID-LAB01
ComSpec=C:\Windows\system32\cmd.exe
CUDA_BIN_PATH=C:\CUDA\bin
CUDA_BIN_PATH_64=C:\CUDA_64\bin
CUDA_INC_PATH=C:\CUDA\include
CUDA_INC_PATH_64=C:\CUDA_64\include
CUDA_LIB_PATH=C:\CUDA\lib
CUDA_LIB_PATH_64=C:\CUDA_64\lib
FP_NO_HOST_CHECK=NO
```

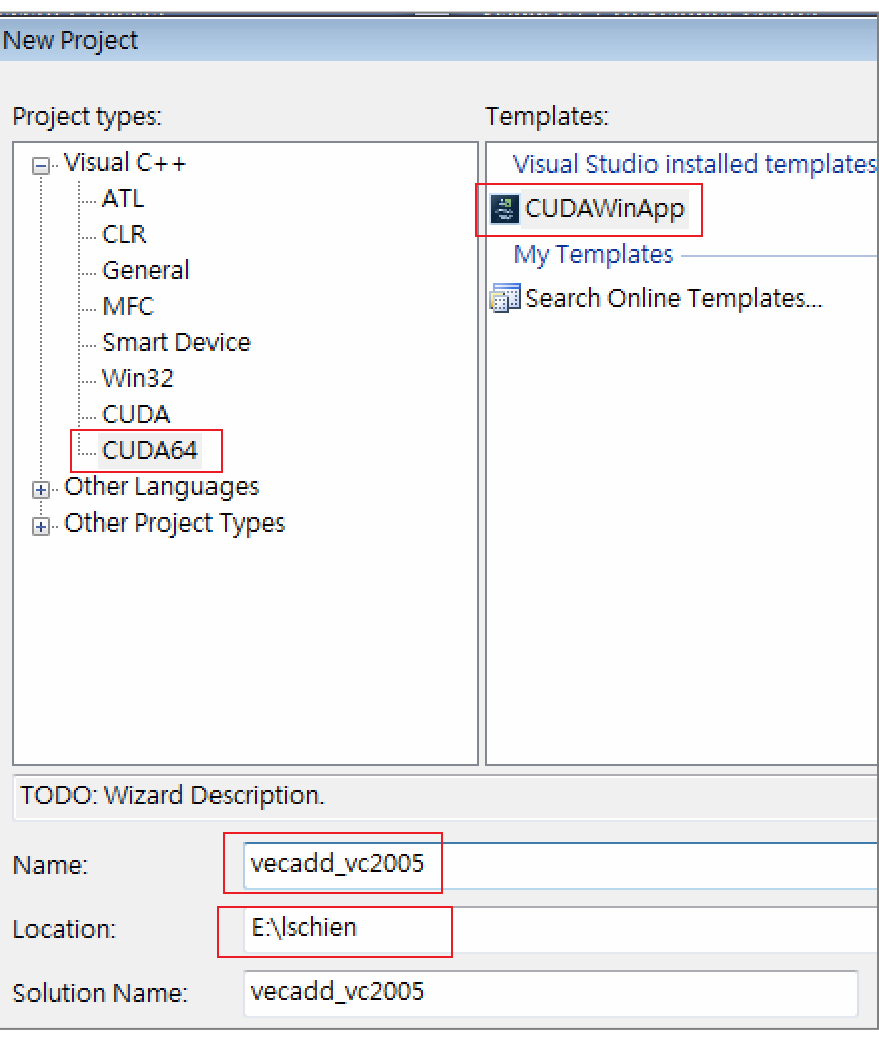
Check environment variables

```
NVSDKCUDA_ROOT=C:\Program Files (x86)\NVIDIA Corporation\NVIDIA CUDA SDK
NVSDKCUDA_ROOT_64=C:\Program Files (x86)\NVIDIA Corporation\NVIDIA CUDA SDK
OMP_NUM_THREADS=1
OS=Windows_NT
Path=C:\Program Files (x86)\VNI\imsl\cnl600\ms64pc\lib;C:\Windows\system32;C:\Wi
ndows;C:\Windows\System32\Wbem;c:\Program Files (x86)\Microsoft SQL Server\90\To
ols\binn\;C:\Program Files\MATLAB\R2008a\bin;C:\Program Files\MATLAB\R2008a\bin\
win64;C:\CUDA\bin;C:\Program Files (x86)\NVIDIA Corporation\NVIDIA CUDA SDK\bin\
win64\Debug;C:\Program Files (x86)\VNI\imsl\cnl600\ms64pc\lib;C:\Program Files (
x86)\SSH Communications Security\SSH Secure Shell
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
```

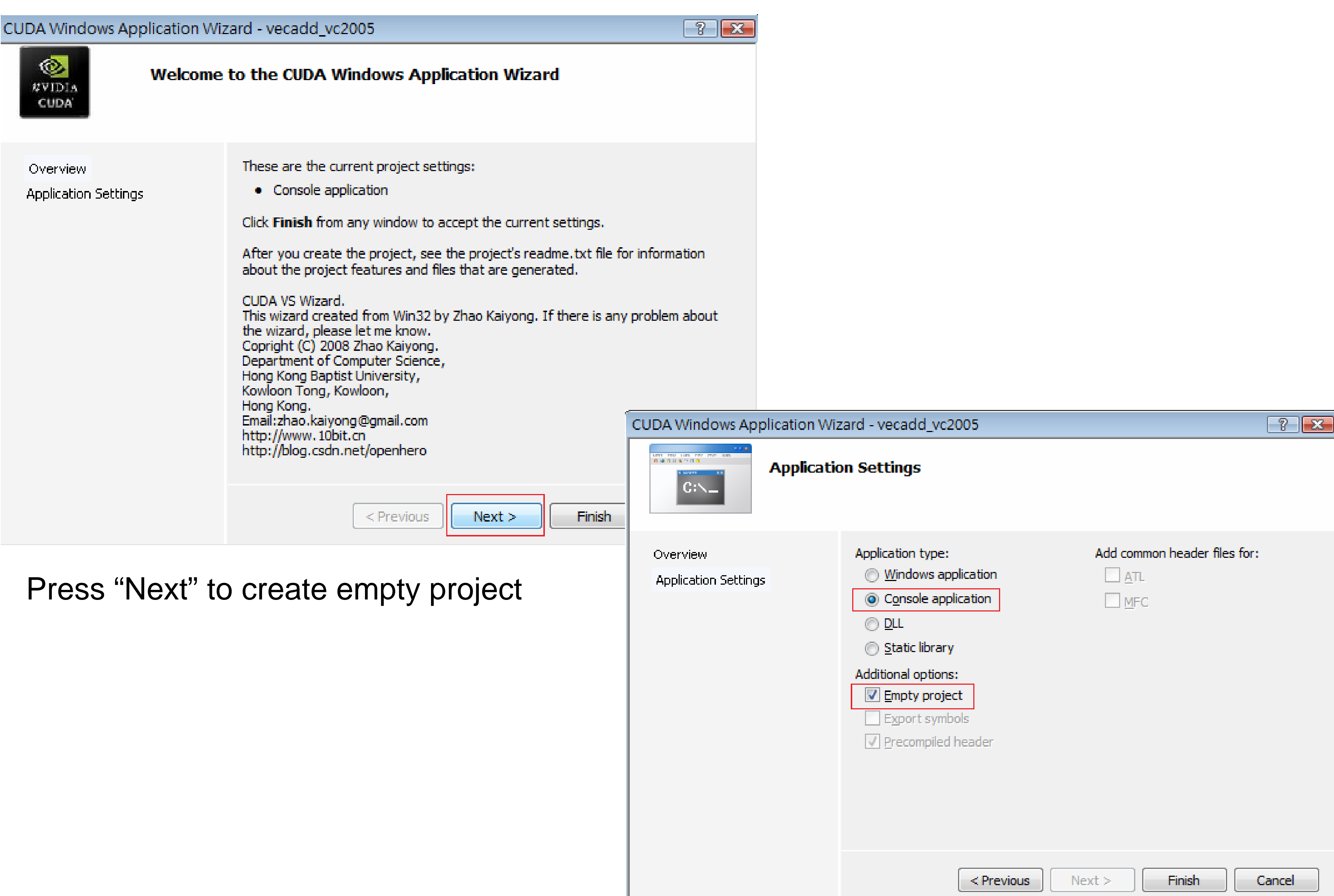
How to embed “nvcc” into VC 2005 [5]

Create a new project: CUDA64 project, this is different from what we do before





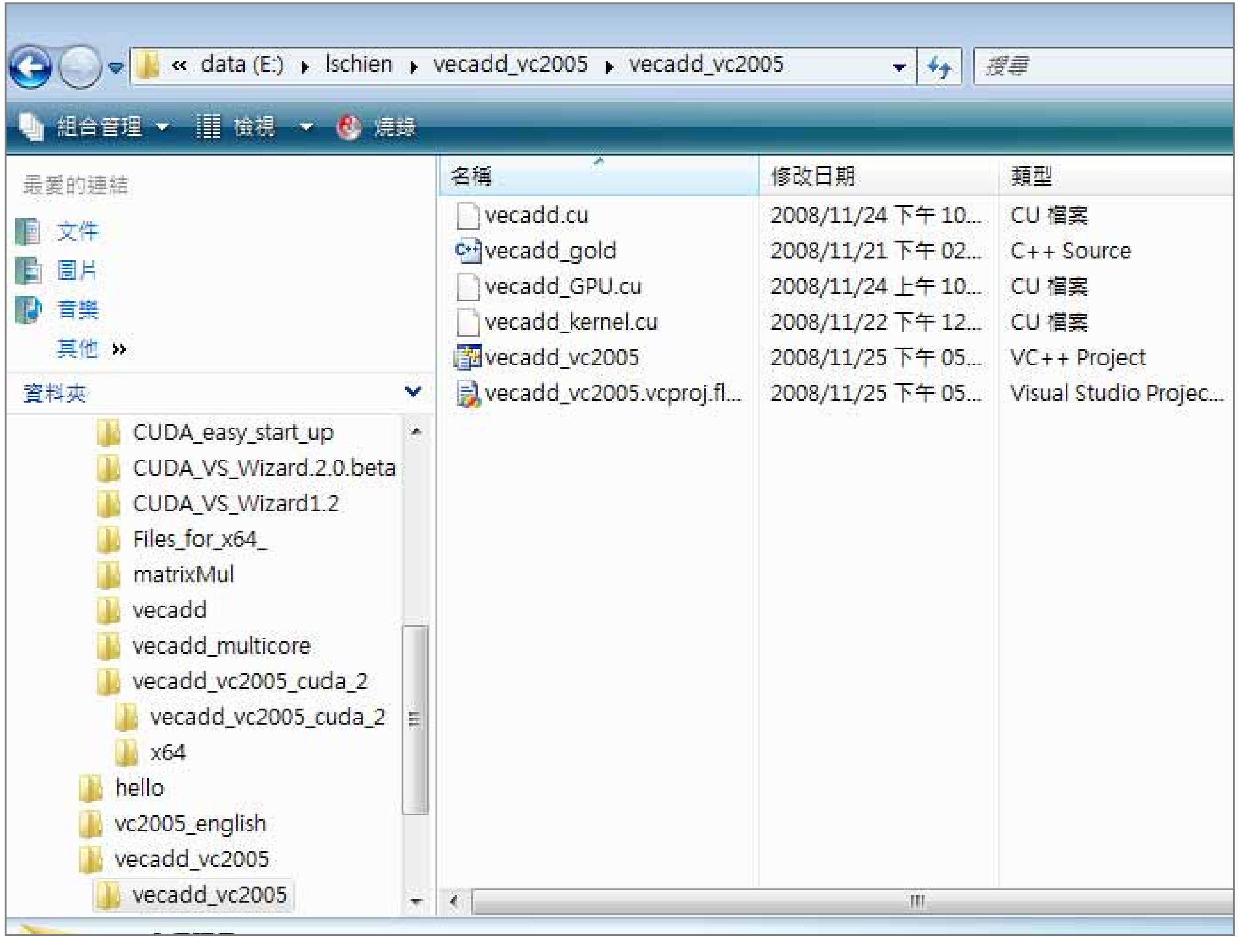
How to embed “nvcc” into VC 2005 [6]



Press “Next” to create empty project

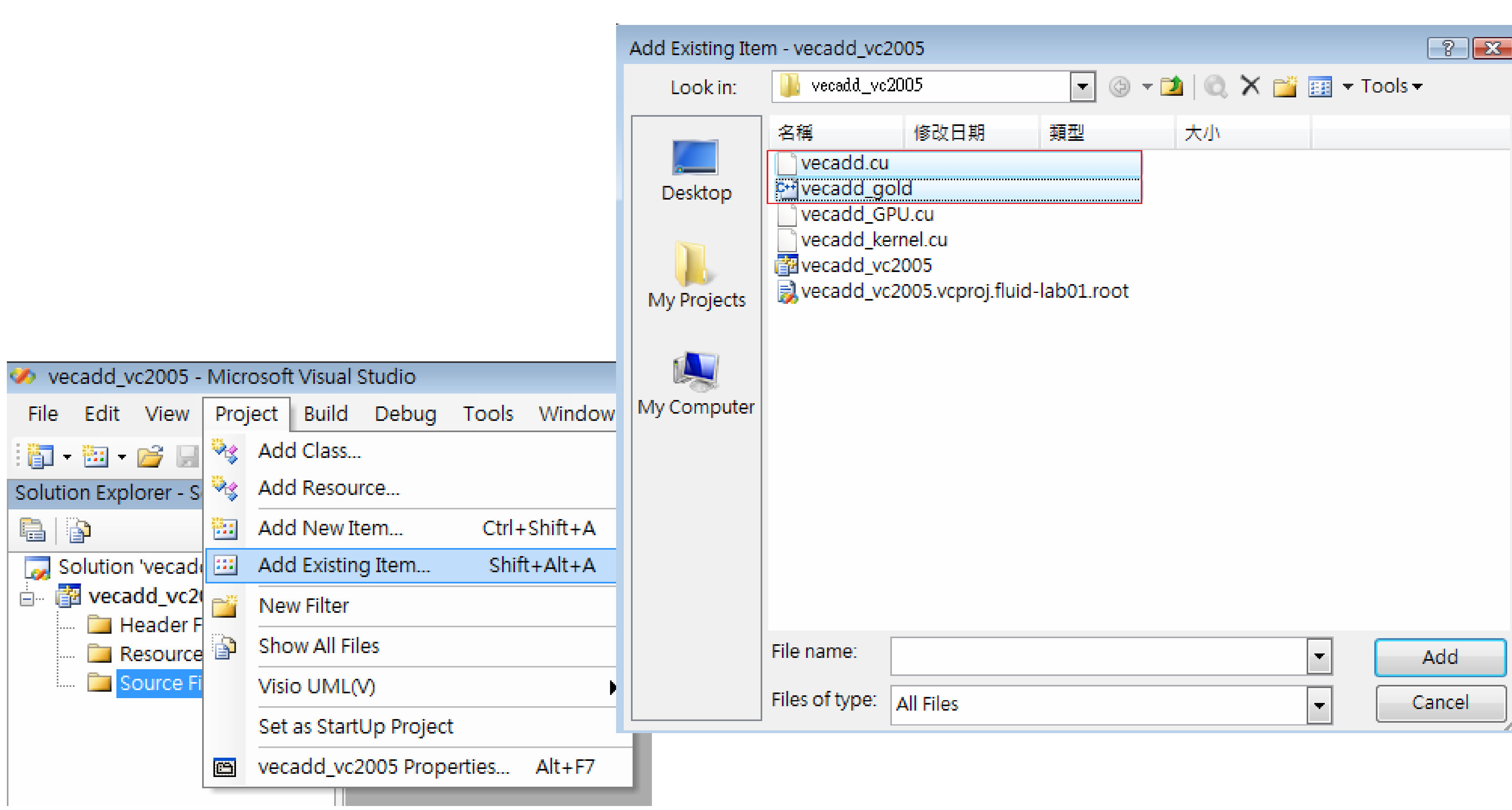
How to embed “nvcc” into VC 2005 [7]

Copy source files, vecadd.cu, vecadd_GPU.cu, vecadd_gold.cpp and vecadd_kernel.cu to directory vecadd_vc2005/vecadd_vc2005



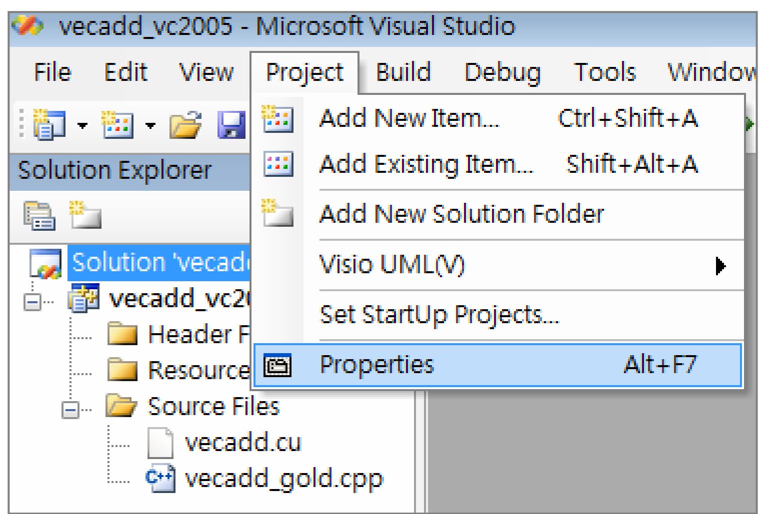
How to embed “nvcc” into VC 2005 [8]

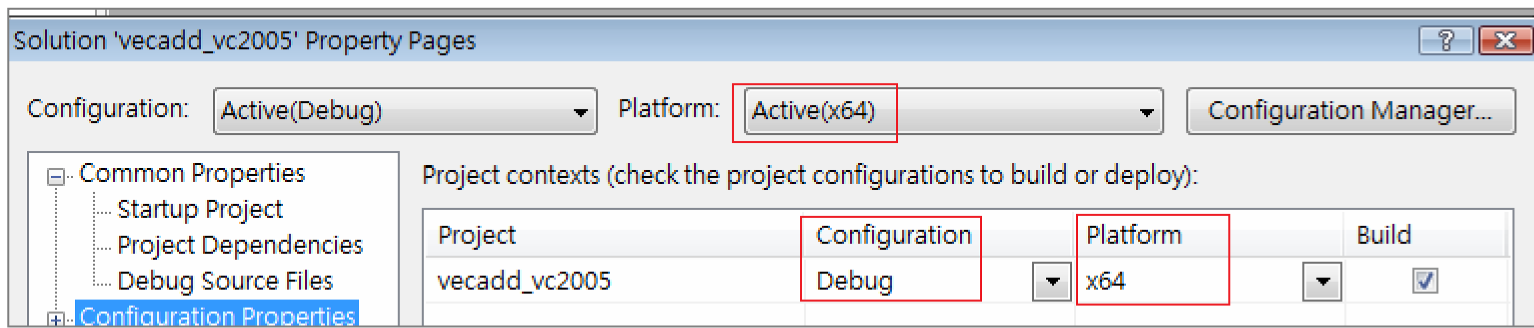
Add source files, vecadd.cu and vecadd_gold.cpp to project



How to embed "nvcc" into VC 2005 [9]

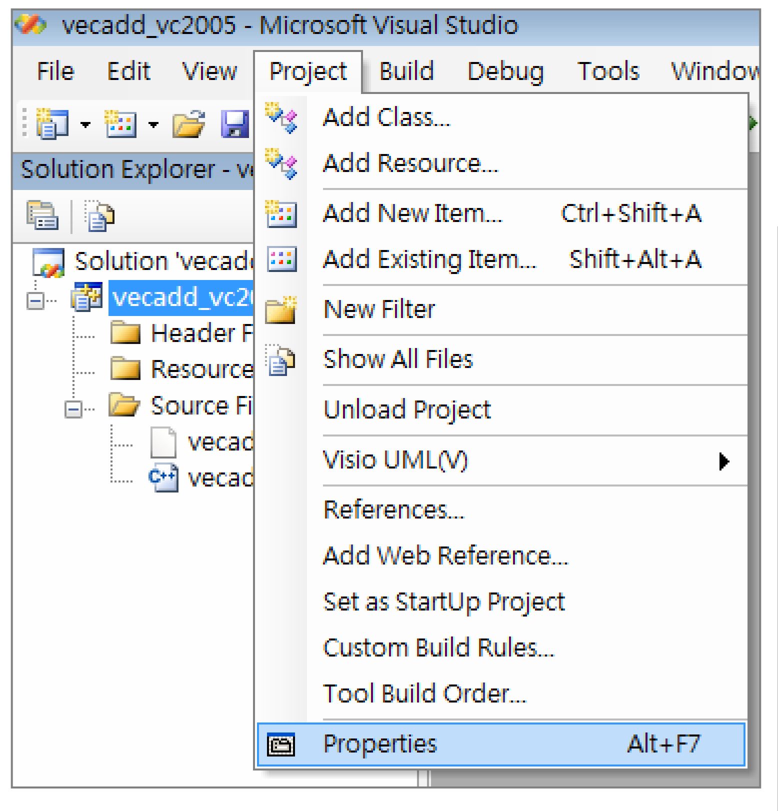
Check solution's property : platform must be x64 (64-bit platform)

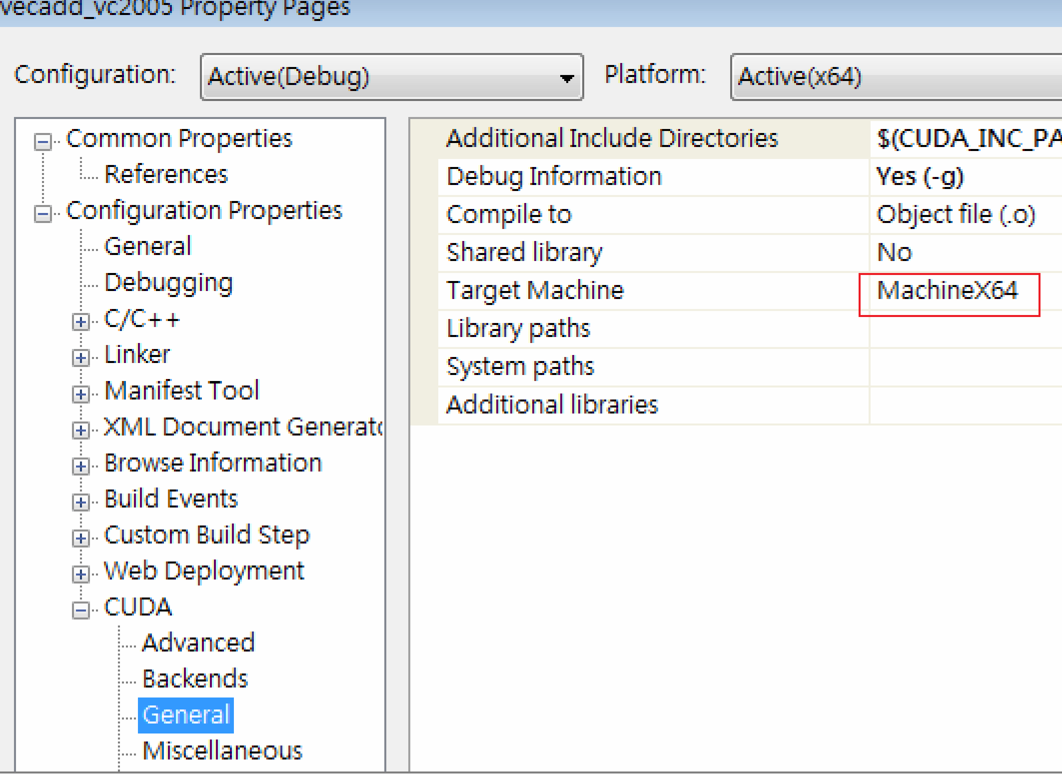




How to embed “nvcc” into VC 2005 [10]

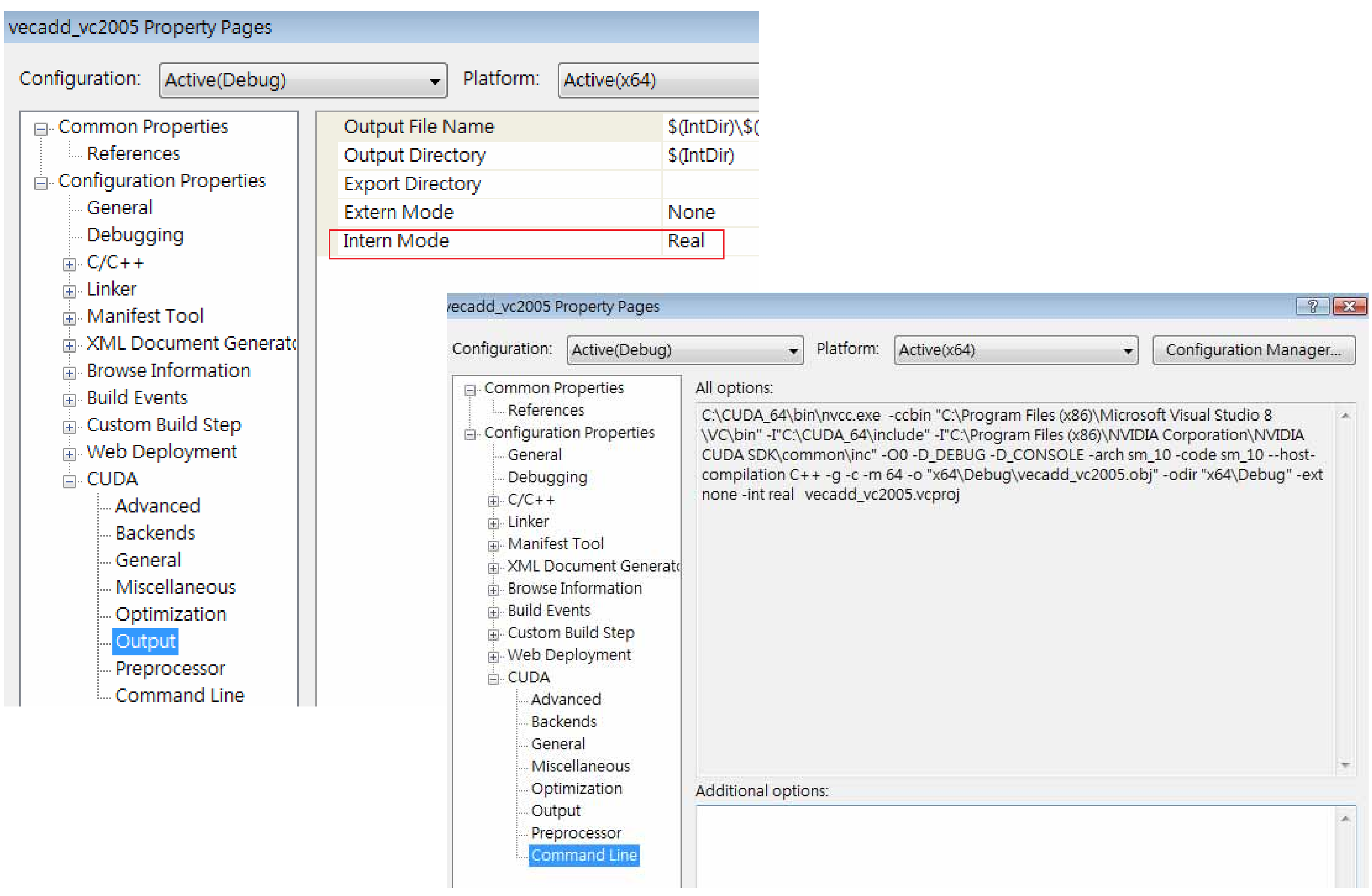
Check solution’s property : CUDA → General → Target Machine → MachineX64 (64-bit platform)



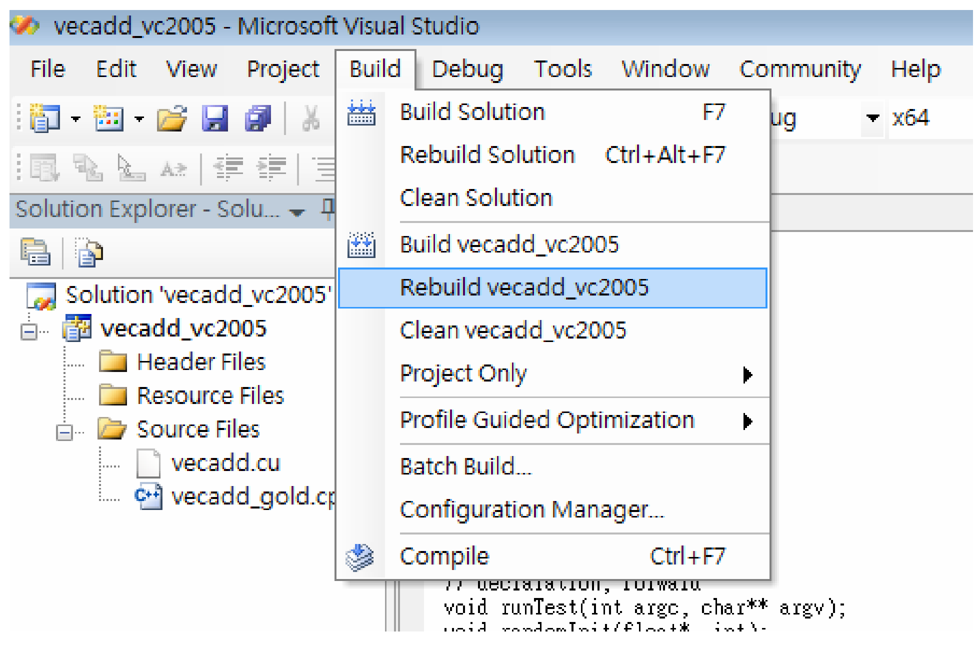


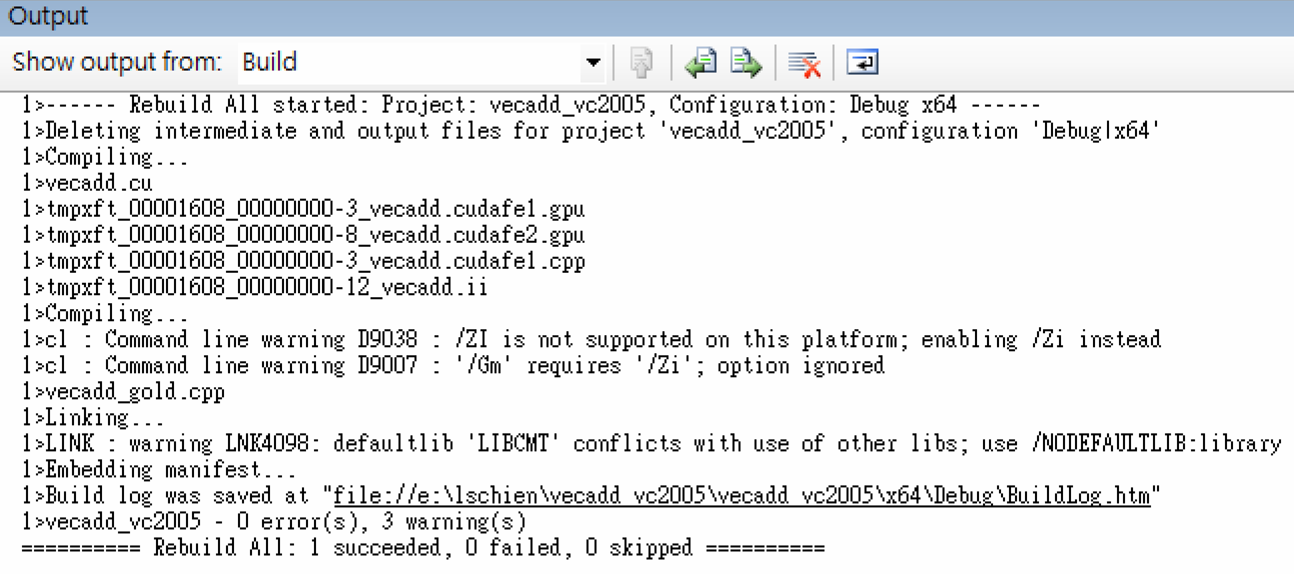
How to embed "nvcc" into VC 2005 [11]

Check solution's property : CUDA → Output → Intern Mode → Real (important)

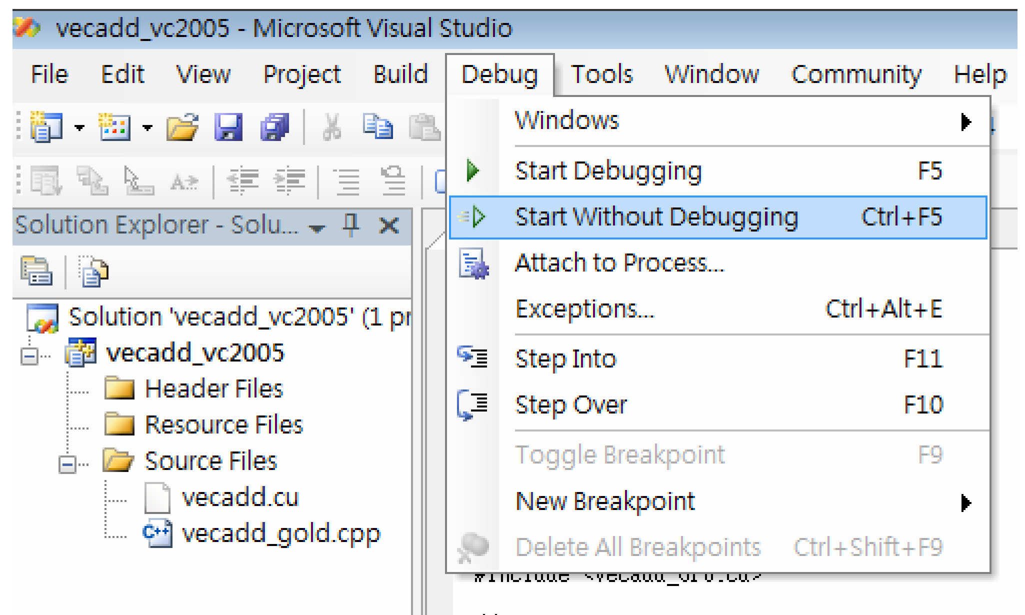


How to embed "nvcc" into VC 2005 (compile) [12]





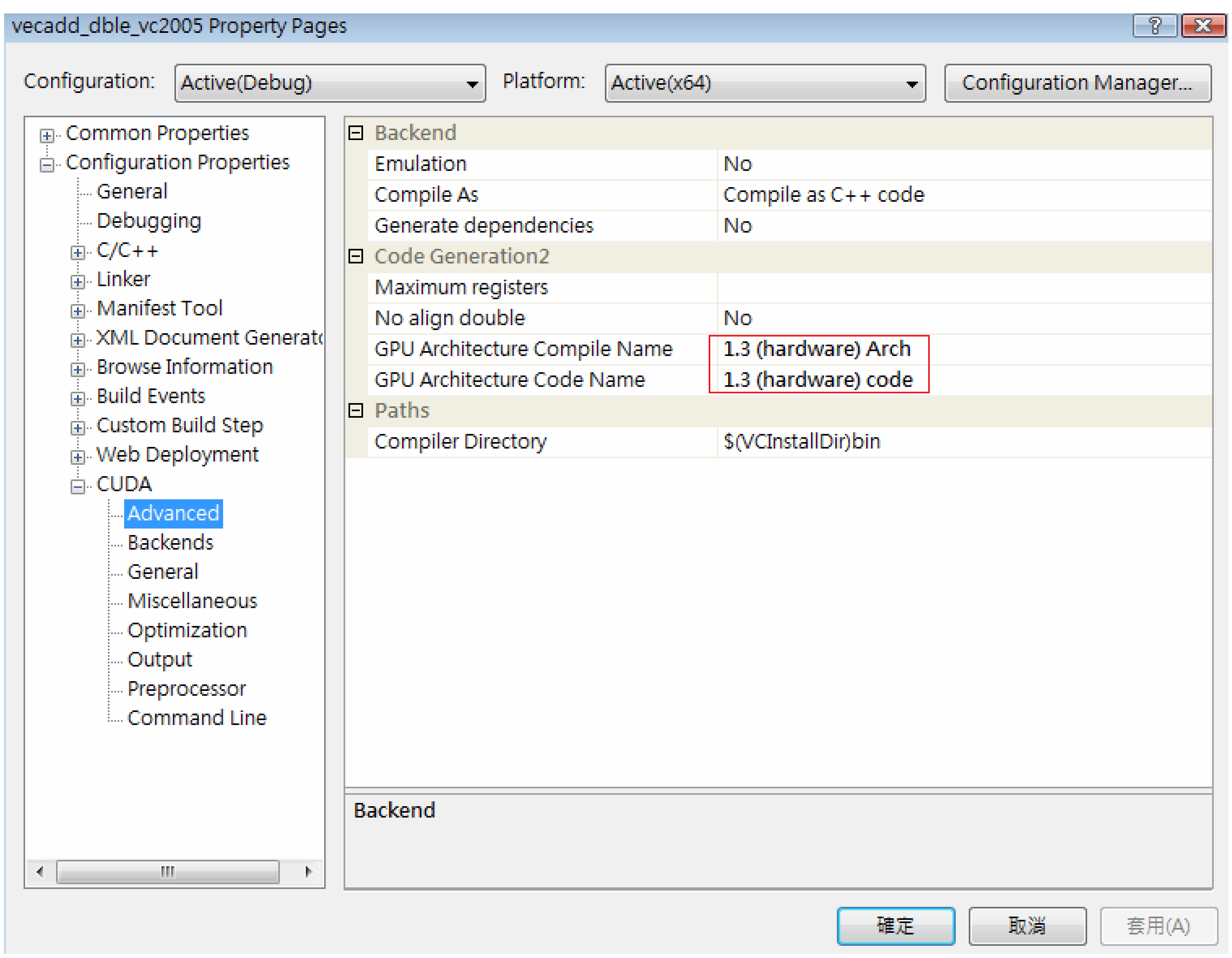
How to embed “nvcc” into VC 2005 (execute) [13]



C:\Windows\system32\cmd.exe

```
N = 512
Using device 0: GeForce GTX 260
in GPU, C = A + B: 18.118784 (ms)
device --> Host: 0.073333 (ms)
compute gold vector needs      0.0000 (ms)
Test PASSED

Press ENTER to exit...
```

How to embed “nvcc” into VC 2005 (double precision) [14]



How to embed “nvcc” into VC 2005 (double precision) [15]

man nvcc

```
--gpu-name <gpu architecture name> (-arch)
    Specify the name of the nVidia GPU to compile for. This can either be a 'real'
    GPU, or a 'virtual' ptx architecture. Ptx code represents an intermediate for-
    mat that can still be further compiled and optimized for. depending on the ptx
    version, a specific class of actual GPUs.

    The architecture specified with this option is the architecture that is assumed
    by the compilation chain up to the ptx stage, while the architecture(s) speci-
    fied with the -code option are assumed by the last, potentially runtime compi-
    lation stage.

    Allowed values for this option:  'compute_10', 'compute_11', 'compute_13',
    'compute_14', 'compute_20', 'sm_10', 'sm_11', 'sm_13', 'sm_14', 'sm_20'.
    Default value: 'sm_10'.


--gpu-code <gpu architecture name>,... (-code)
    Specify the name of nVidia gpu to generate code for.

    Unless option -export-dir is specified (see below), nvcc will embed a compiled
    code image in the executable for each specified 'code' architecture, which is a
    true binary load image for each 'real' architecture (such as a sm_13), and ptx
    code for each virtual architecture (such as compute_10). During runtime, such
    embedded ptx code will be dynamically compiled by the cuda runtime system if no
    binary load image is found for the 'current' GPU, and provided that the ptx
    level is compatible with this current GPU.

    Architectures specified for options -arch and -code may be virtual as well as
    real, but the 'code' architectures must be compatible with the 'arch' architec-
    ture.  For instance, 'arch'=compute_13 is not compatible with 'code'=sm_10,
    because the earlier compilation stages will assume the availability of com-
    pute_13 features that are not present on sm_10.  This option defaults to the
    value of option '-arch'.

    Allowed values for this option:  'compute_10', 'compute_11', 'compute_13',
    'compute_14', 'compute_20', 'sm_10', 'sm_11', 'sm_13', 'sm_14', 'sm_20'.
```

virtual: compute_10,
compute_11,
compute_12,
compute_13

real: sm_10,
sm_11,
sm_12,
sm_13

sm_13: compute capability 1.3