BIGR LAB
big data research laboratory

PyCUDA

蘇育生

Source by 鄭钧輿

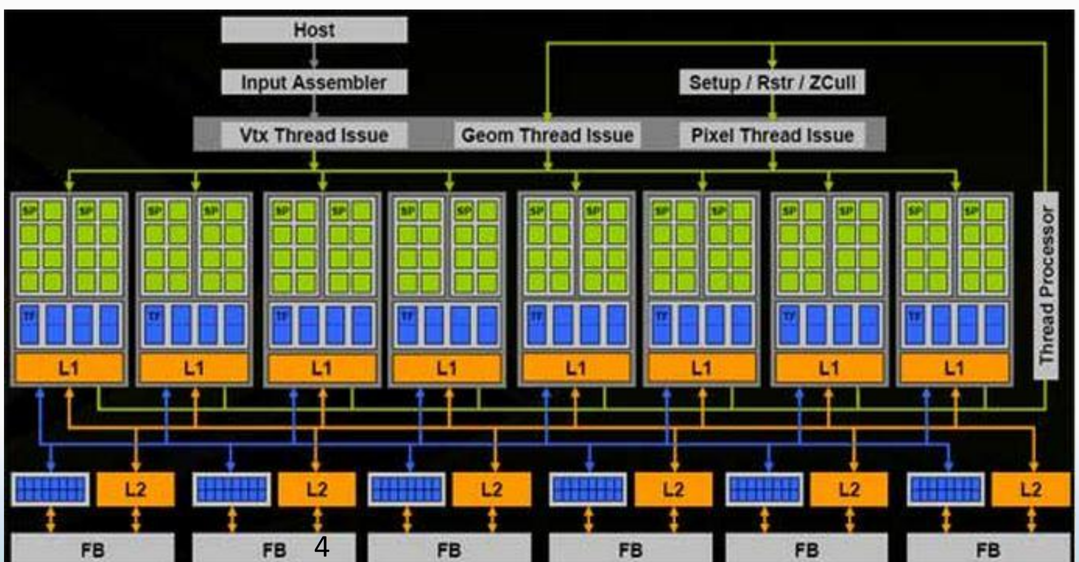
CUDA Introduction

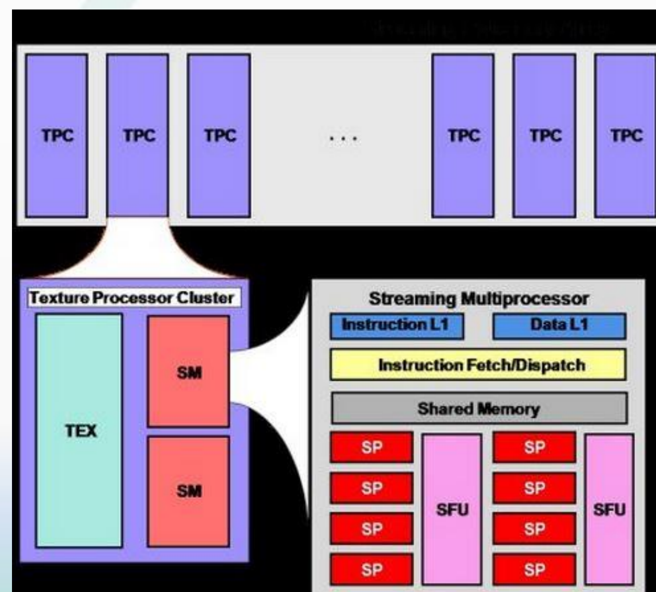
Compute Unified Device Architecture (CUDA)

- CUDA是由NVIDIA所推出的平行運算架構。
- 透過GPU的強大威力，此架構能大幅提昇運算效能。
- CUAD目標:
  - Facilitate heterogeneous computing: CPU + GPU
- 利用C語言結合CUDA的指定延伸語法撰寫程式。
- 應用:影像及視訊處理、計算生化學、流體力學模擬、電腦斷層(CT)影像重建、地震分析、光線追蹤等。

GPU基本構造

- SP(streaming processor)－最基本的處理單元。進行平行運算時也是多個SP在處理
- SM(streaming multiprocessor)－多個sp加上其他資源組成。(其他資源就是儲存資源,共享內存,暫存器…等)



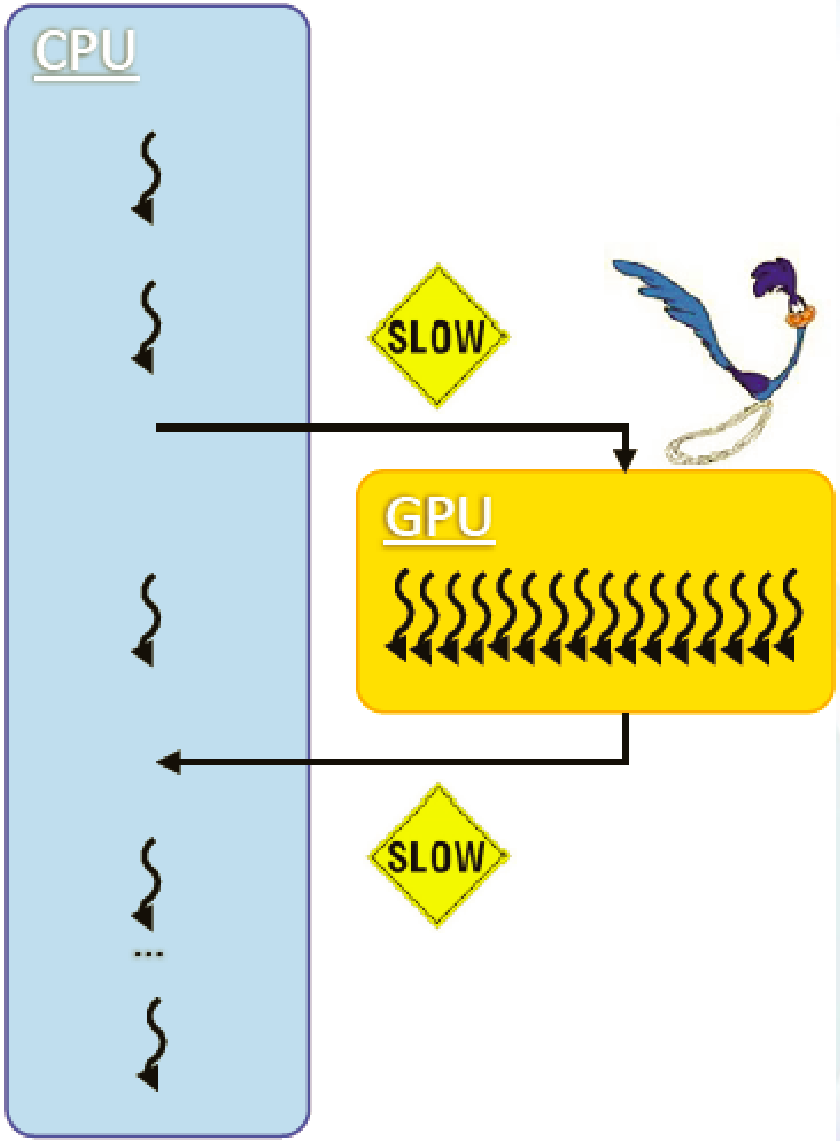


C/CUDA Code

```
// serial code
int main() {
 printf("Hello world!"\n);
// allocate data
 cudaMalloc(...);
// copy data
 cudaMemcpy(...);
// execute kernel

 cudaRun<<<..>>>(...);

  ...

 cudaThreadSynchronize();

// serial code
 printf("Running.."\n);

  ...

 exit (0);
}
```

CUDA Kernel Code

```
// kernel
__global__
void cudaRun(...) {
  ...
}
```



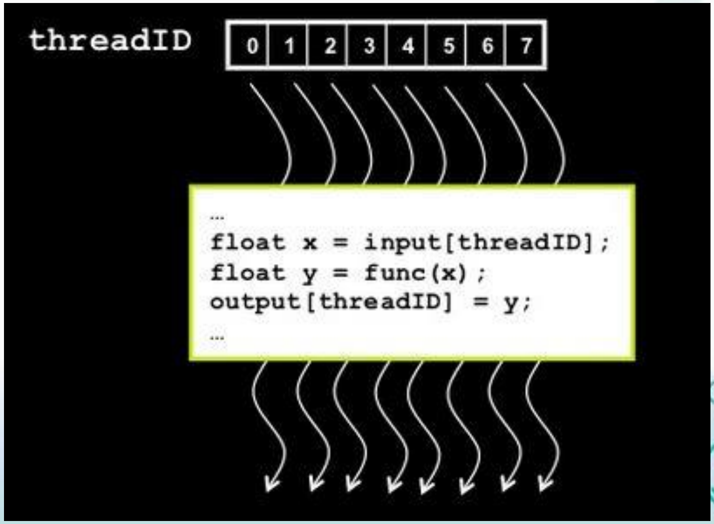
CUDA Kernels and Threads

- KERNEL:把要平行運算的部分寫成FUNCTIONS，然後再DEVICE上執行，就是KERNEL
  - 一次只能同時執行一個KERNEL
  - 會有多個THREADS來執行KERNEL
- CUDA跟CPU之間THREAD的不同
  - CUDA的THREADS非常輕量化(lightweight)
    - Fast switching
    - Very little creation overhead
  - CUDA利用上千個THREADS來提高效率
    - 即使是多核CPU也只能使用一些THREADS

Parallel Threads

CUDA的KERNEL會被array of threads執行

- 所有Threads都會執行同一份code
- 每個Thread都有ID來計算其memory address以及make control decisions

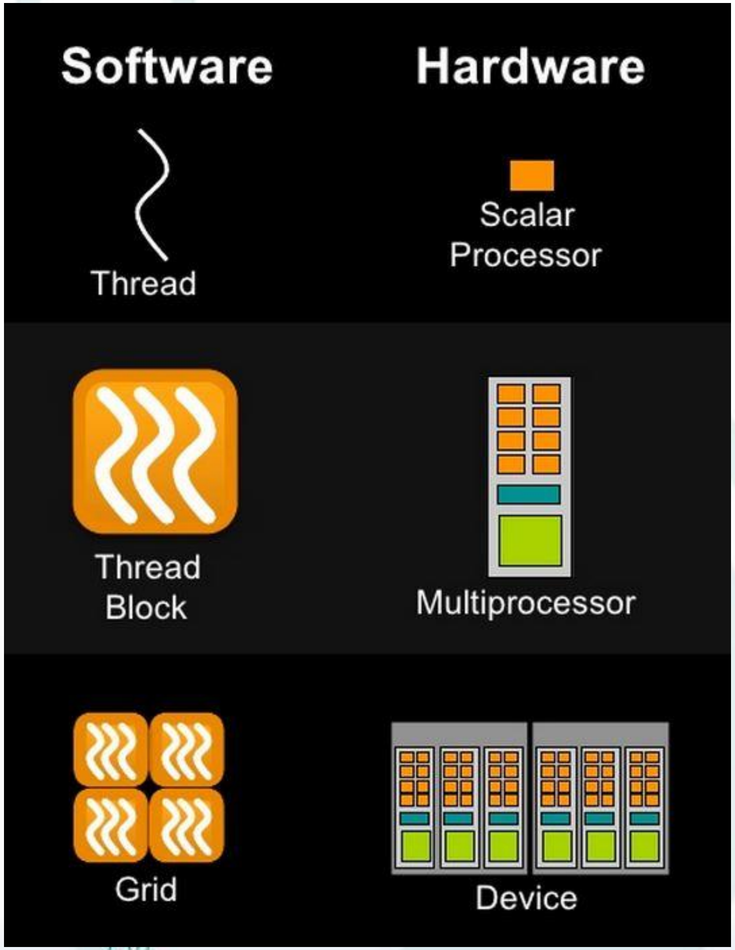


Thread Batching

CUDA的平行化模型是將核心交由一組網格執行，再將網格切成數個區塊，然後每個區塊再分成數個執行緒，依次分發進行平行運算，如果用軍隊來比喻，將核心視為連任務，那網格就是連隊，區塊就是排或班，執行緒就是小兵。

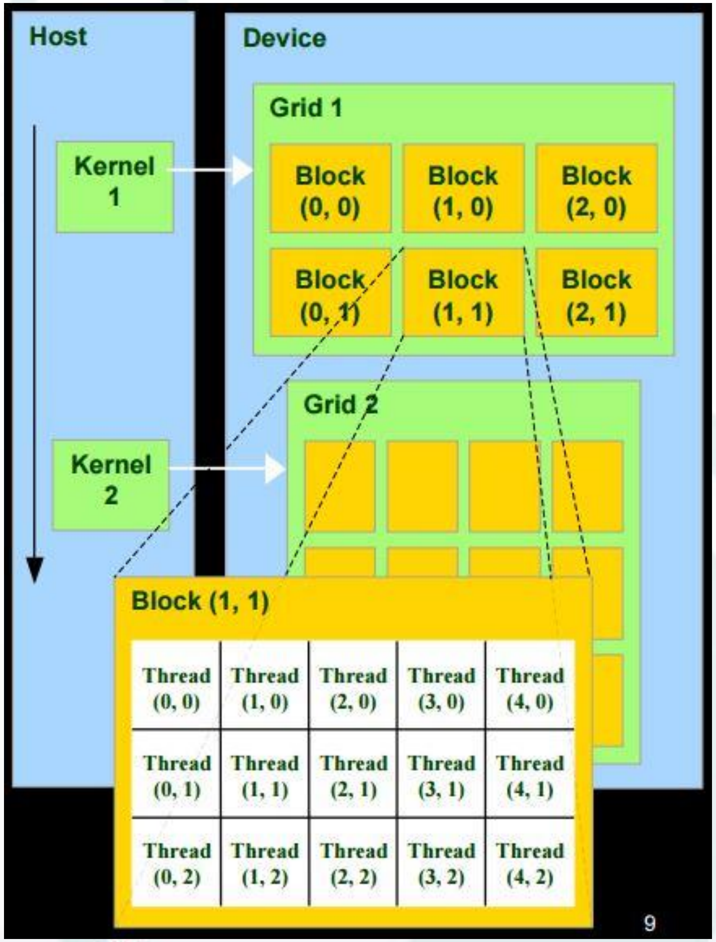
網格(GRID):數個區塊的執行單元
區塊(BLOCK):數個執行緒的執行單元
執行緒(THREAD):最小處理單元(實際寫程式的環境)

CUDA架構對應到硬體架構:
GRID - GPU
BLOCK - Streaming Multiprocessor
Thread - Streaming Processor



CUDA-Programming Model

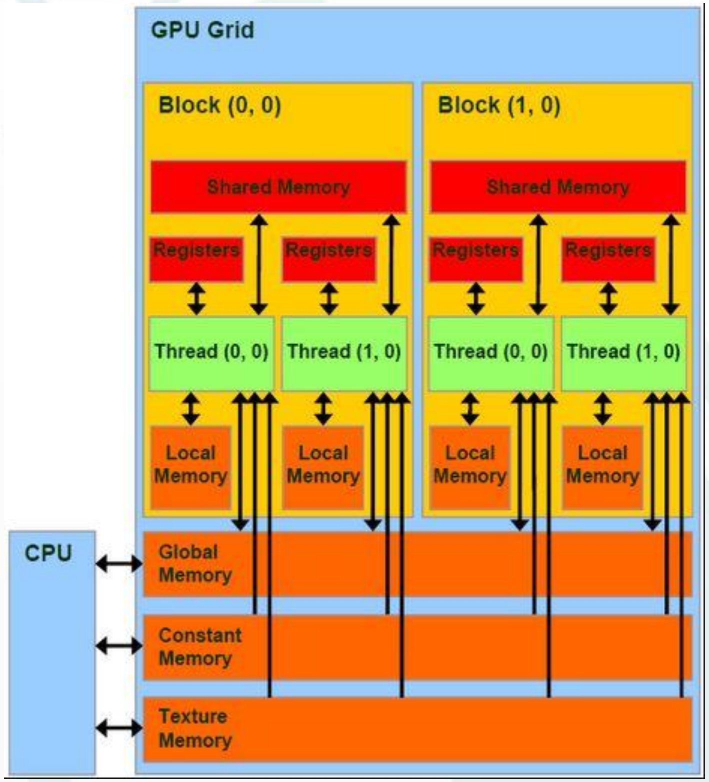
- Kernel會被grid裡面的thread blocks所執行
- Thread blocks是一群threads而彼此間:
  - ➢Sharing data through shared memory
  - ➢Synchronizing their execution
- 不同blocks裡面的threads資料不能共享



高速平行運算暨巨量資料實驗室 BIGRLAB big data research laboratory

Memory model

- Registers
  - Per thread
  - Data lifetime = thread lifetime
- Local memory
  - Per thread off-chip memory (physically in device DRAM)
  - Data lifetime = thread lifetime
- Shared memory
  - Per thread block on-chip memory
  - Data lifetime = block lifetime
- Global(device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation (釋放)
- Host(cpu) memory
  - Not directly accessible by CUDA threads



高速平行運算暨巨量資料實驗室 BIGR LAB big data research laboratory

Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A thread block executes on one multiprocessor
  - Does not migrate (遷移)
- Several blocks can reside concurrently on one multiprocessor
  - Number is limited by multiprocessor resources
    - Registers are partitioned among all resident threads
    - Shared memory is partitioned among all resident threads

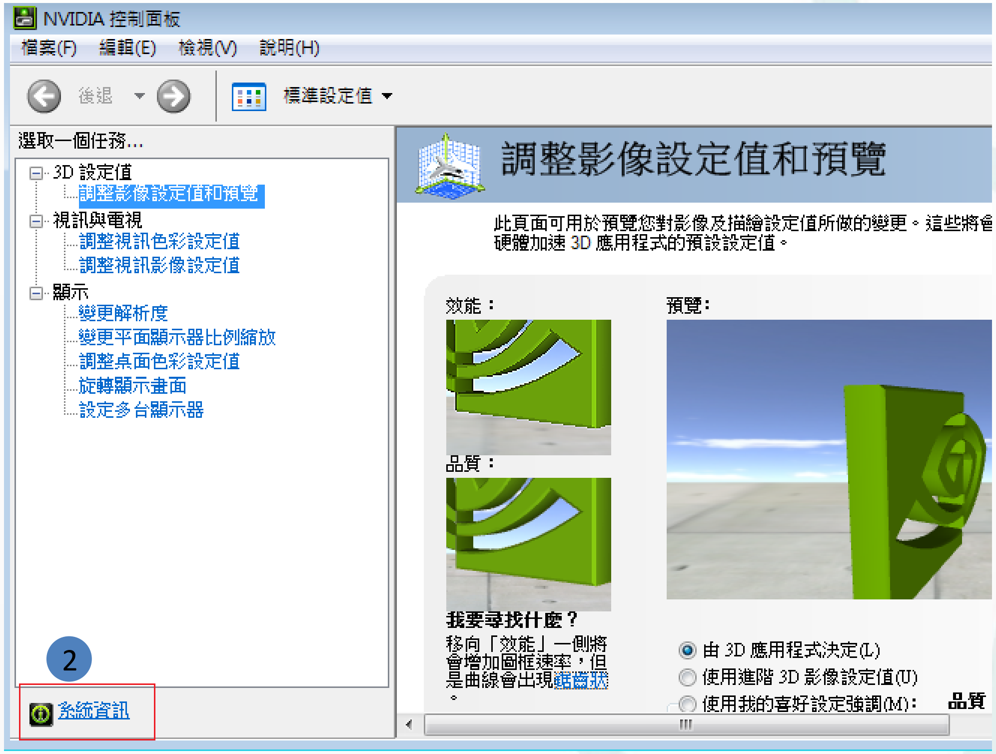
https://www.youtube.com/watch?v=Cm7W6MaNuF4

INSTALL CUDA UNDER VISUAL STUDIO 2008

How to embed "nvcc" into Visual Studio C

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

2 Choose system information



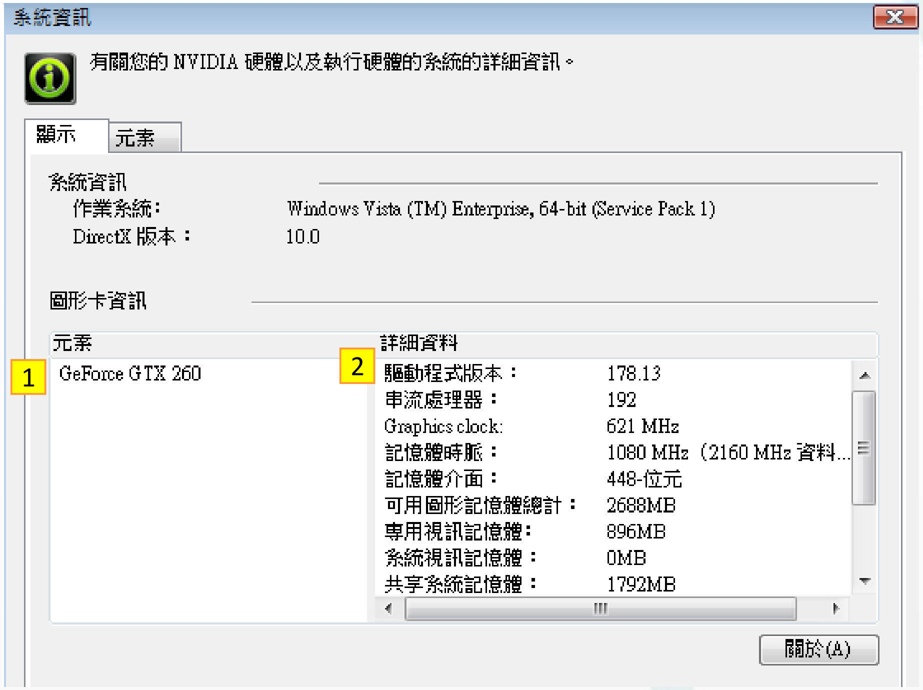
高速平行運算暨巨量資料實驗室 BIGRLAB big data research laboratory

How to embed “nvcc” into Visual Studio C

system information, including

1 chipset

2 driver



How to embed “nvcc” into Visual Studio C

```
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 reserve

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
```

Hello world

- Introduction to programming in CUDA C
  - https://www.youtube.com/watch?v=8RW0oaTJ0YQ

- CUDA Hello world
  - https://www.youtube.com/watch?v=Av6h0hEpv9o

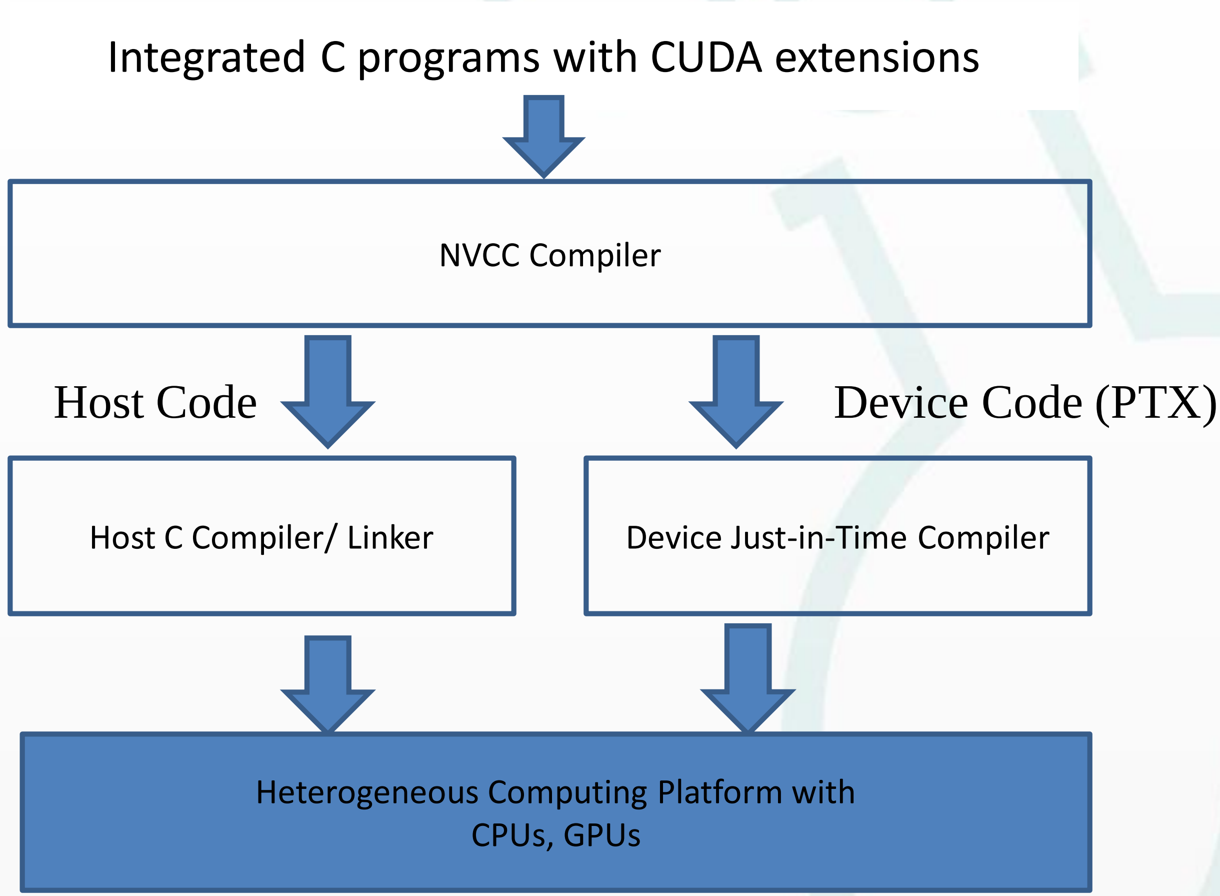
Hello World v.1.0: Basic C Program

```
#include <stdio.h>
int main() {
printf( "Hello world!" \n);
exit (0);
}
```

```
nvcc -o hello hello.cu
```

```
OUTPUT:
Hello World!
```

Compiling A CUDA Program



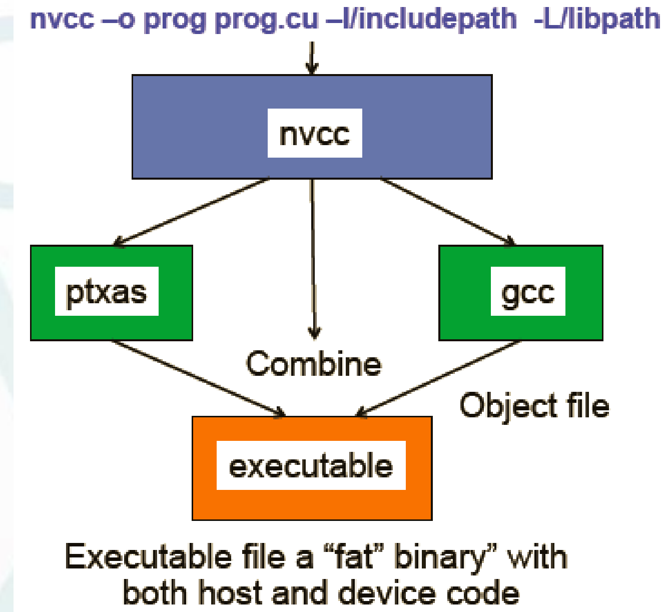
Compiling CUDA Code

directories for the #include files

nvcc –o <exe> <source_file> -I/usr/local/cuda/include

–L/usr/local/cuda/lib –lcuda –lcudart

directories for libraries　　libraries to be linked



- If a Cuda code includes device code, the file must have the extension .cu.
- nvcc separates out code for the CPU and code for the GPU and compiles code.
- It needs regular C compiler installed for the CPU code.

Executing a CUDA Program

- ./a.out
- Host code starts running.
- When first encounter device kernel, GPU code physically sent to GPU and function launched on GPU.

Hello World v.2.0: Kernel Calls

```
#include <stdio.h>


int main() {
 kernel<<<1,1>>>();
 printf("Hello world!\n");
 exit (0);
}

__global__ void kernel () {
  // does nothing
}
```

- An empty function named "kernel" qualified with the specifier __global__ (yes, there are <u>two</u> underscores on each side)
  - Indicates to the compiler that the code should be run on the device, not the host.
- A call to the empty device function with "<<<1,1>>>"
  - Within the "<<<" and ">>>" brackets are **memory arguments** (for the blocks and threads) and within the parentheses are the **parameter arguments** (that you normally use in C).

```
OUTPUT:

Hello World!
```

Hello World v.3.0: Parameter Passing

```
#include <stdio.h>

__global__ void add (int a, int b, int *c);

int main() {
 int c;
 int *dev_c;

 cudaMalloc((void**)&dev_c, sizeof(int));

 add<<<1,1>>>(2,7,dev_c);

 cudaMemcpy(&c, dev_c, sizeof(int),
            cudaMemcpyDeviceToHost);

 printf("Hello world!\n");
 printf("2 + 7 = %d\n", c);

 cudaFree(dev_c);

 exit (0);
}

__global__ void add (int a, int b, int *c) {
 c[0] = a + b;
}
```

- Parameter passing is similar to C.
- There exists a separate set of host + device memory.
- We need to allocate memory to use it on the device.
- We need to copy memory from the host to the device and/or vice versa via cudaMemcpy.
- CudaMalloc (similar to malloc) allocates global memory on the *device*.
- CudaFree (similar to free) deallocates global memory on the *device*.
- Of course, capable of mathematic operations.

```
OUTPUT:

Hello World!
2 + 7 = 9
```

Example 1: vector addition

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

vecadd_gold.cpp

```
#include <stdio.h>
#include <time.h>

extern "C"
void computeGold( float*, const float*, const float*, unsigned int );

void computeGold(float* C, const float* A, const float* B, unsigned int N )
{
  unsigned int i ;
  clock_t  start, end ;

  start = clock() ;
  for ( i= 0; i < N ; ++i){
    C[i] = A[i] + B[i] ;
  }
  end = clock() ;
  double dt = ((double)(end - start))/((double)CLOCKS_PER_SEC) * 1000.0 ;
  printf("compute gold vector needs %10.4f (ms)\n", dt );

}
```

measure time

$C := A + B$

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

Example 1: vector addition

1 vecadd_GPU.cu

```
#include <stdio.h>
// includes, project
#include <cutil.h>

extern "C" {
void vecadd_GPU(float* h_C, const float* h_A, const float* h_B, unsigned int N) ;
}

void vecadd_GPU(float* h_C, const float* h_A, const float* h_B, unsigned int N)
{
  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) );
```

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

```
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

```
// allocate device memory for result
unsigned int mem_size_C = sizeof(float) * N ;
float* d_C;
6 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
7 vecadd<<< 1, N >>>(d_C, d_A, d_B, N);

// 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));

CUDA_SAFE_CALL(cudaFree(d_A));
CUDA_SAFE_CALL(cudaFree(d_B));
CUDA_SAFE_CALL(cudaFree(d_C));
}
```

Measure time

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

Header file ***util.h***

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

Example 1: vector addition

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

```
#include <stdio.h>
#include <assert.h>

__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] ;
}
```

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.

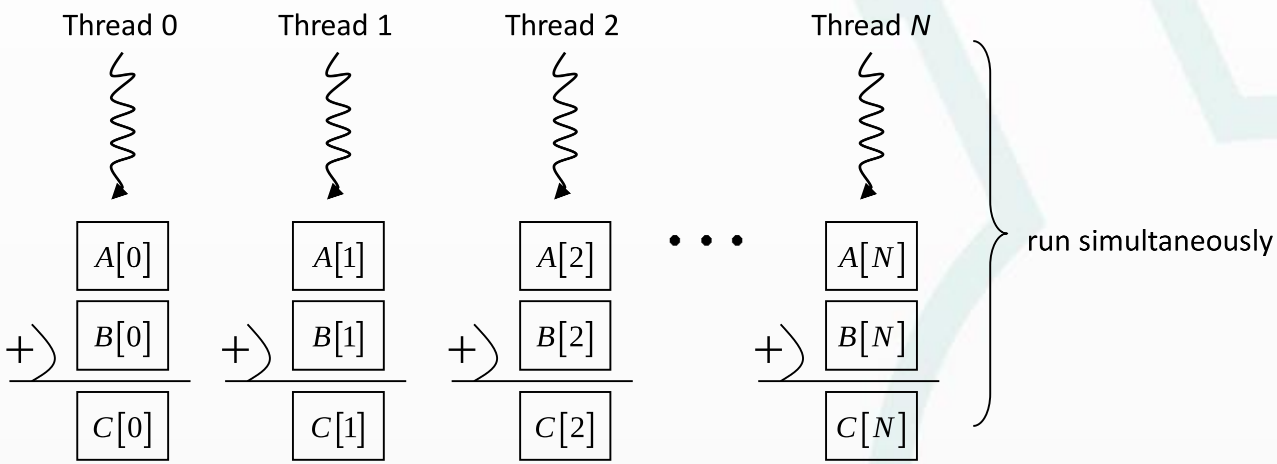
In emulation mode, macro __DEVICE_EMULATION__ is set.

Example 1: vector addition

10

```
int i = threadIdx.x ;
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.



Example 1: vector addition (driver)

vecadd.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes, project
#include <cutil.h>

// includes, kernels
#include <vecadd_kernel.cu>
#include <vecadd_GPU.cu>

// declaration, forward
void runTest(int argc, char** argv);
void randomInit(float*, int);
void printDiff(float*, float*, int, int);

extern "C" {
void computeGold(float*, const float*, const float*, unsigned int );
void vecadd_GPU(float* h_C, const float* h_A, const float* h_B, unsigned int N) ;
}

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

    CUT_EXIT(argc, argv);
}
```

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)

```
// test  C = A + B
void runTest(int argc, char** argv)
{
  unsigned int N = 128 ;
  CUT_DEVICE_INIT(argc, argv);

  // set seed for rand()
  srand(2006);

  // allocate host memory for matrices A and B
  unsigned int size_A = N ;
  unsigned int mem_size_A = sizeof(float) * size_A;
  float* h_A = (float*) malloc(mem_size_A);

  unsigned int size_B = N ;
  unsigned int mem_size_B = sizeof(float) * size_B;
  float* h_B = (float*) malloc(mem_size_B);

  // allocate host memory for the result
  unsigned int size_C = N ;
  unsigned int mem_size_C = sizeof(float) * size_C;
  float* h_C = (float*) malloc(mem_size_C);

  // initialize host memory
  randomInit(h_A, size_A);
  randomInit(h_B, size_B);

  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

```
  // compute reference solution
  float* reference = (float*) malloc(mem_size_C);
  computeGold(reference, h_A, h_B, N );

  // check result
  CUTBoolean res = cutCompareL2fe(reference, h_C, size_C, 1e-6f);
  printf("Test %s \n", (1 == res) ? "PASSED" : "FAILED");
  if (res!=1) printDiff(reference, h_C, 1, N);

  // clean up memory
    free(h_A);
    free(h_B);
    free(h_C);
    free(reference);
}
```

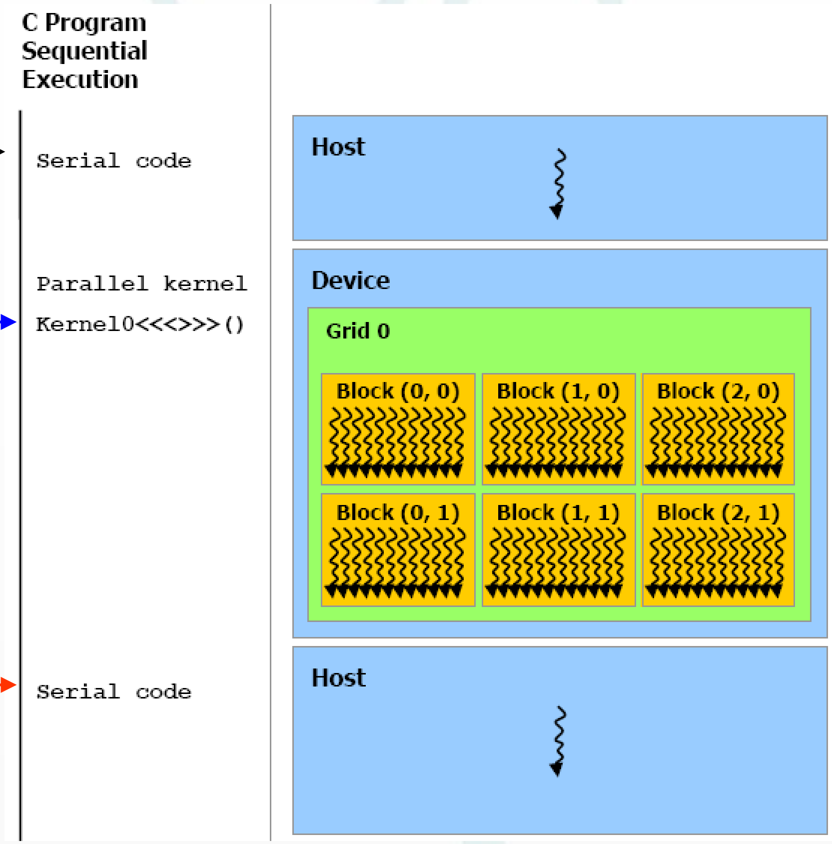
Example 1: vector addition (driver)

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

random A and B

Do *C = A+ B* in GPU

compute golden vector in CPU



Example 1: vector addition (compile under Linux)

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)

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)

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 (1)
Processing time: 0.046000 (ms) (2)
compute gold vector needs     0.0000 (ms) (3)
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)
```

Example 1: vector addition (compile under Linux)

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

```
// test   C = A + B
void runTest(int argc, char** argv)
{
  unsigned int N = 512 ;
  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

```
// test   C = A + B
void runTest(int argc, char** argv)
{
  unsigned int N = 513 ;
  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)

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
```

Example 1: vector addition (compile under Linux)

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)

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'.
```

Example 2: multicore vector addition

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

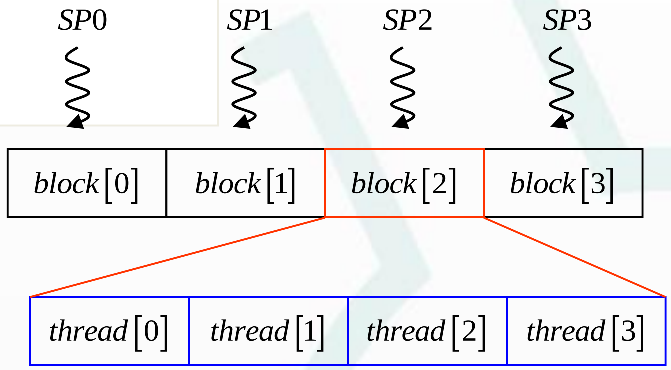
```
__global__  void vecadd_multicore( float* C, float* A, float* B,
          int threads, int N)
{
  int bx = blockIdx.x;
  int i = bx*threads + threadIdx.x ;
  C[i] = A[i] + B[i] ;

#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

Example 2: multicore vector addition

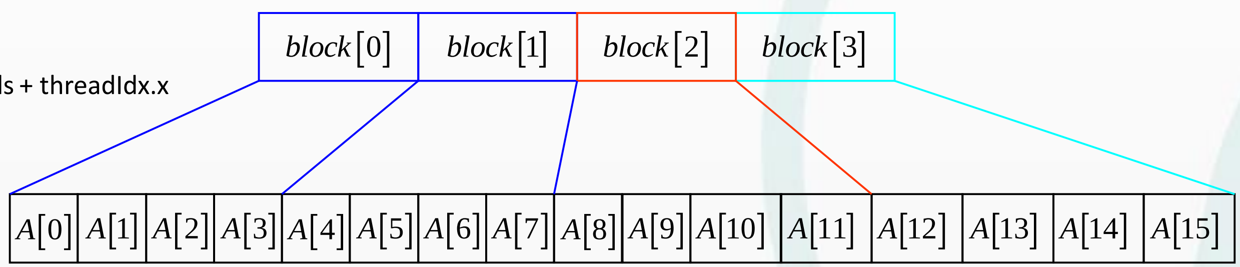
```
__global__  void vecadd_multicore( float* C, float* A, float* B,
          int threads, int N)
{
  int bx = blockIdx.x;
  int i = bx*threads + threadIdx.x ;
  C[i] = A[i] + B[i] ;

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



$threads = 4$

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

i = bx*threads + threadIdx.x



Example 2: multicore vector addition

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));
```

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

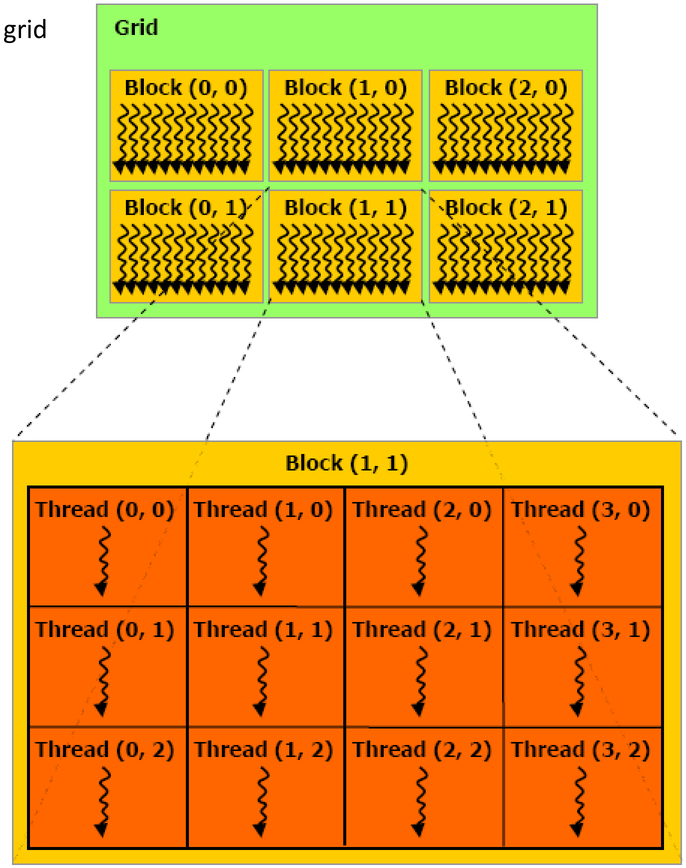
one-dimension thread block

one-dimension grid

Example 2: multicore vector addition

two-dimension grid

two-dimension thread block



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

Example 2: multicore vector addition

vecadd.cu

```
void runTest(int argc, char** argv)
{
  unsigned int num_block = 8 ;
  unsigned int threads = 512 ;
  unsigned int N = num_block*threads ;

  printf("num_block = %d, threads = %d, N = %6.2f (KB)\n",
      num_block, threads, N*4./1024. );

  CUT_DEVICE_INIT(argc, argv);

  // set seed for rand()
  srand(2006);

  // allocate host memory for matrices A and B
  unsigned int size_A = N ;
  unsigned int mem_size_A = sizeof(float) * size_A;
  float* h_A = (float*) malloc(mem_size_A);

  unsigned int size_B = N ;
  unsigned int mem_size_B = sizeof(float) * size_B;
  float* h_B = (float*) malloc(mem_size_B);

  // allocate host memory for the result
  unsigned int size_C = N ;
  unsigned int mem_size_C = sizeof(float) * size_C;
  float* h_C = (float*) malloc(mem_size_C);

  // initialize host memory
  randomInit(h_A, size_A);
  randomInit(h_B, size_B);

  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

高速平行運算暨巨量資料實驗室 BIGRLAB big data research laboratory

Example 2: multicore vector addition

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

Experimental platform: Geforce 9600 GT

Copy C from device to host

$C = A + B$

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

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)

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

Experimental platform: Geforce 9600 GT

Copy C from device to host

$C = A + B$

Table 2

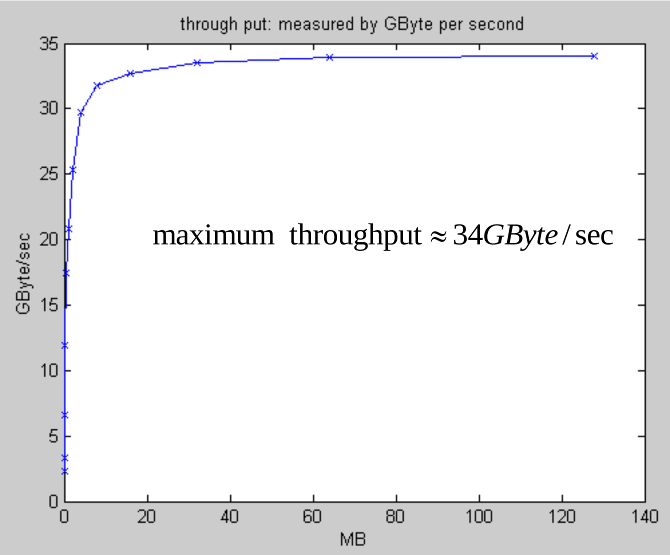
# of block	size	GPU (ms)	Device → Host (ms)	CPU (ms)
16	32 KB	0.04	0.059	0
32	64 KB	0.056	0.122	0
64	128 KB	0.057	0.242	0
128	256 KB	0.063	0.381	0
256	512 KB	0.086	0.67	0
512	1.024 MB	0.144	1.513	0
1024	2.048 MB	0.237	2.812	10
2048	4.096 MB	0.404	5.426	10
4096	8.192 MB	0.755	9.079	20
8192	16.384 MB	1.466	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)

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

```
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 dominated.

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

高速平行運算暨巨量資料實驗室 BIGRLAB big data research laboratory

What is pycuda?

pycuda = python + cuda

can I use python ?

Why pycuda?

- Numpy integration
- Cross platform
- Allow interactive use
- Automatically manage resource
- Full CUDA support and performance

Pycuda Install

- Windows 7/8/8.1 64bit

1. Python 2.7:(http://www.activestate.com/activepython/downloads)
2. Python套件
   - PyCUDA: (http://www.lfd.uci.edu/~gohlke/pythonlibs/#pycuda)
     (pycuda-2014.1+cuda6514-cp27-none-win_amd64.whl)
   - Numpy: (http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy)
     (numpy-1.9.2+mkl-cp27-none-win_amd64.whl)
   - Boost.python : (http://www.lfd.uci.edu/~gohlke/pythonlibs/#boost.python)
     (boost_python-1.55-cp27-none-win_amd64.whl)

   安裝方法:使用pip install(需到下載好的套件.whl檔目錄)

   (ex: >pip install "numpy-1.9.2+mkl-cp27-none-win_amd64.whl")
3. CUDA Toolkit: (https://developer.nvidia.com/cuda-toolkit-)

   (CUDA Toolkit 6.5 需跟上面PyCUDA配合才可以)
4. Nvidia driver : (http://www.nvidia.com.tw/Download/index.aspx?lang=tw)

   按照步驟輸入自己的顯示卡型號即可以找到driver，再把driver下載後安裝即可。

Pycuda Install(Cont.)

5. Visual Studio 2010 or Visual Studio 2012(除了express版)

   (這裡是用Visual Studio 2010 Ultimate)

   在環境變數的Path中增加(看你的VC資料夾放哪):

   C:\Program Files\Microsoft Visual Studio 10.0\VC\bin;C:\Program Files\Microsoft Visual Studio 10.0\VC\bin\amd64;C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE;

PycudaTutorial

Example 1 : Double Vector (Hello World)

Example-double vector

- The "hello world" program of CUDA is a program to double vector

$$C[i] = 2 * C[i] \text{ for } i=0 \text{ to } N-1$$

- For the CUDA solution, there are two parts
  - Kernel code
  - Host code

double vector- kernel

```
__global__ void double_vec( float * a )
{
    int idx = threadIdx.x + threadIdx.y * 4;
    a[idx] *= 2;
}
```

double vector- host

- Host program － 在host 執行:
  - –建立CUDA的基本環境
  - –建立與管理kernels
- 2個步驟在Host program中建置CUDA環境
  - –1. Transferring Data
  - –2. Executing a Kernel

pyCUDA workflow

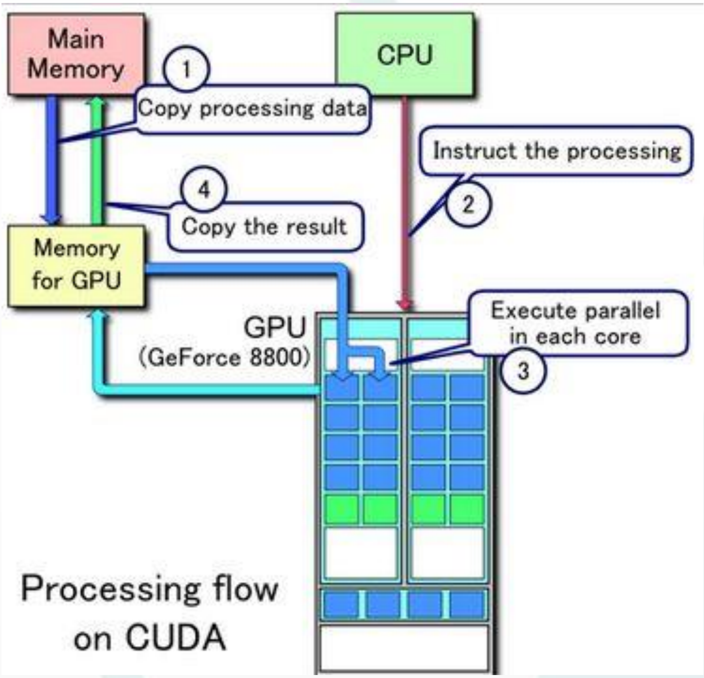
PYCUDA基本流程:

1.複製記憶體至GPU

2.CPU指令驅動GPU(建置環境)

3.GPU每一核心平行處理(平行運算)

4.GPU將結果傳回主記憶體(回傳資料)



1.Transferring data

- 建立資料:

```
import numpy
a=numpy.random.randn(4,4).astype(numpy.float32)
```

4x4 array of random numbers

- 在device上分配記憶體:

```
a_gpu = cuda.mem_alloc(a.nbytes)
```

- 將資料傳進gpu中:

```
cuda.memcpy_htod(a_gpu,a)
```

2. Executing a kernel

- 寫CUDA C code(Kernel code):

```
kernel =SourceModule("""
    __global__ void double_vec( float * a ) {
        int idx = threadIdx.x + threadIdx.y * 4;
        a[idx] *= 2;
    }""")
```

- 定義並呼叫 Kernel function:

```
double_vec = kernel.get_function( "double_vec" )
double_vec(a_gpu,block=(4,4,1))
```

- 將資料從GPU中取回來並輸出:

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled,a_gpu)
print a_doubled
print a
```

double vector – host program

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy
def main:
        #create data
a=numpy.random.randn(4,4).astype(numpy.float32)
        #allocate memory on device
        a_gpu = cuda.mem_alloc(a.nbytes)
        #transfer data to GPU
        cuda.memcpy_htod(a_gpu,a)
        #kernel code
        kernel =SourceModule( “” ”
               __global__ void double_vec( float *
a ) {
                      int idx = threadIdx.x +
threadIdx.y * 4;
                      a[idx] *= 2;
                }” ” ” )
        #define kernel function and call it
        double_vec =
kernel.get_function( “double_vec” )
        double_vec(a_gpu,block=(4,4,1))
        #fetch the data back from GPU
        a_doubled = numpy.empty_like(a)
        cuda.memcpy_dtoh(a_doubled,a_gpu)
        #output result
        print a_doubled
        print a
if __name__ == ’__main__’:
    main()
```

```
[[ 0.51360393 1.40589952 2.25009012 3.02563429]
[-0.75841576 -1.18757617 2.72269917 3.12156057]
[ 0.28826082 -2.92448163 1.21624792 2.86353827]
[ 1.57651746 0.63500965 2.21570683 -0.44537592]]
```

Source by http://documen.tician.de/pycuda/tutorial.html

Example 2 : Vector Add (Demo)

We help you to understand new technologies and trends!