CUDA 프로그래밍

CUDA Programming

biztripcru@gmail.com

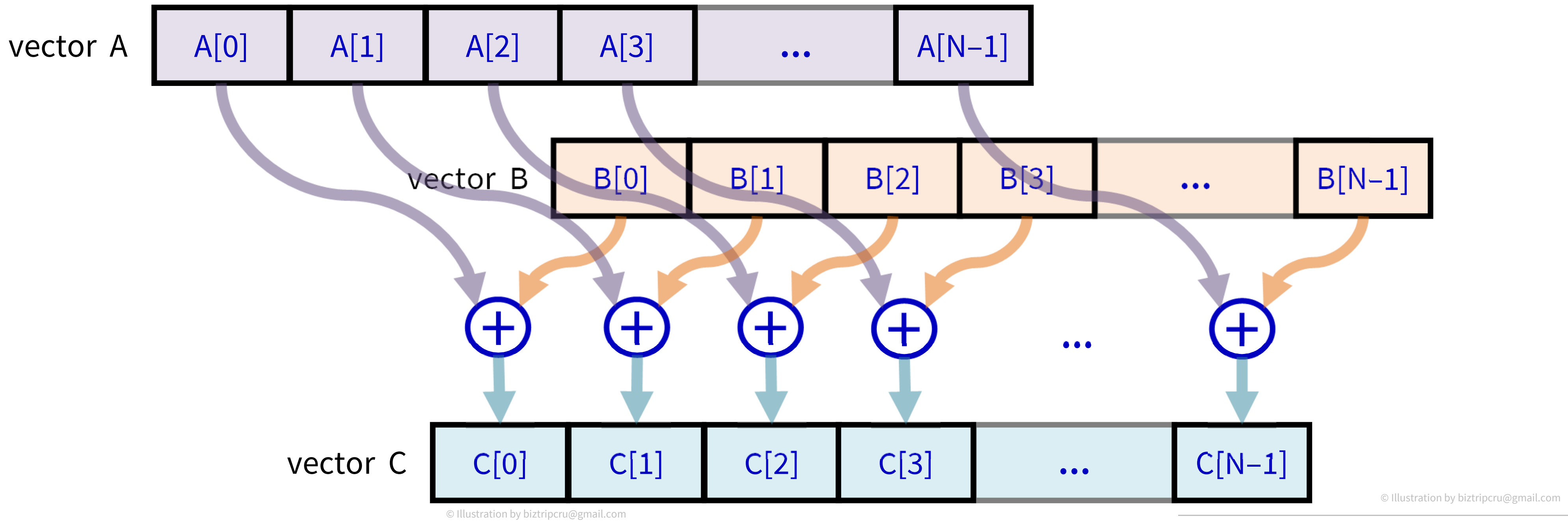
CUDA Kernel

CUDA 커널

내용 contents

Scenario : vector addition

- **vector : represented as 1D array, with *n* elements**
  - **C[i] = A[i] + B[i]**



Vector Addition

- **vector : represented as 1D array**
  - const int a[SIZE];
  - const int b[SIZE];
  - int c[SIZE];

- **vector addition: c[...] = a[...] + b[...]**

- **serial execution: for-loop**
- **CUDA execution: parallel kernel execution**

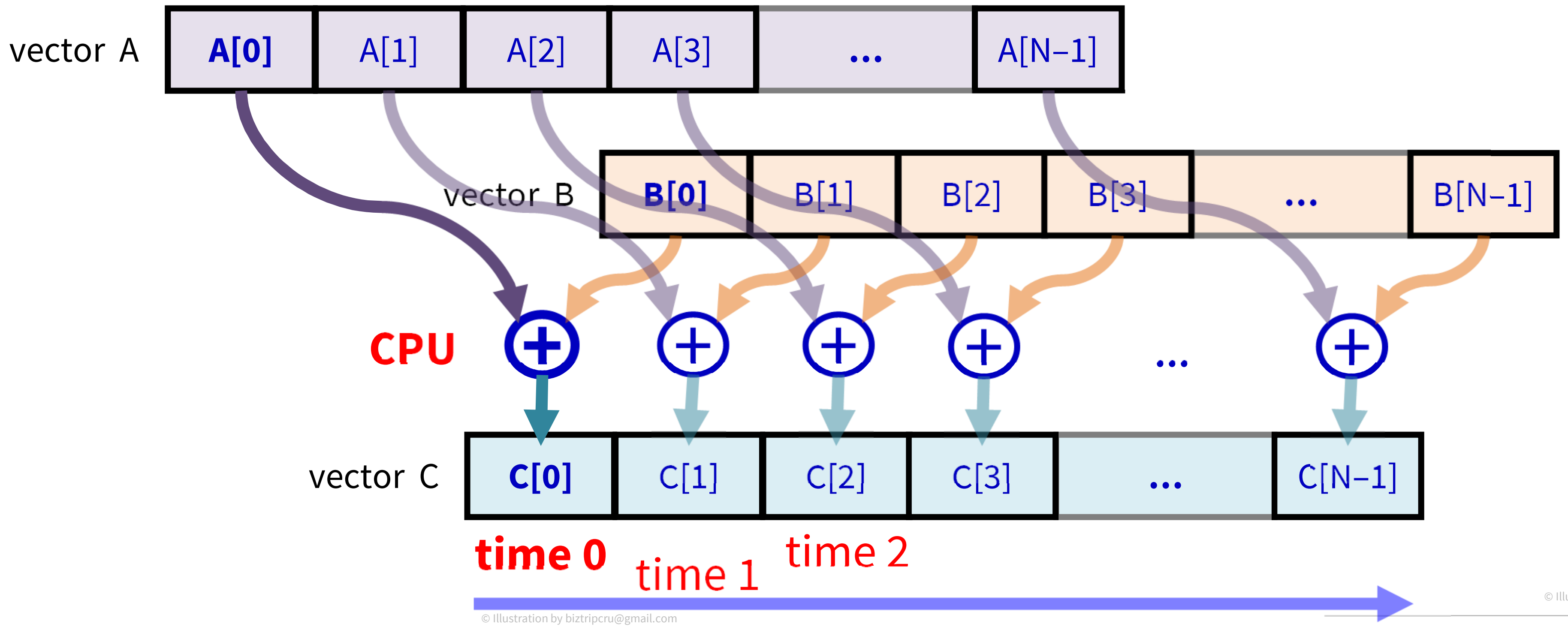
CUDA kernel

- **CPU kernels**
  - with a single CPU core
  - **for-loop**
- **sequential execution**
- **for-loop !**
  - CPU[0]   for time 0
  - CPU[1]   for time 1
  - CPU[2]   for time 2
  - …
  - CPU[n–1]   for time n–1

- **GPU kernels**
  - a set of GPU cores
  - **multiple threads**
- **parallel execution**
- **kernel launch !**
  - GPU[0]   for core #0
  - GPU[1]   for core #1
  - GPU[2]   for core #2
  - …
  - GPU[n–1]   for core #n–1

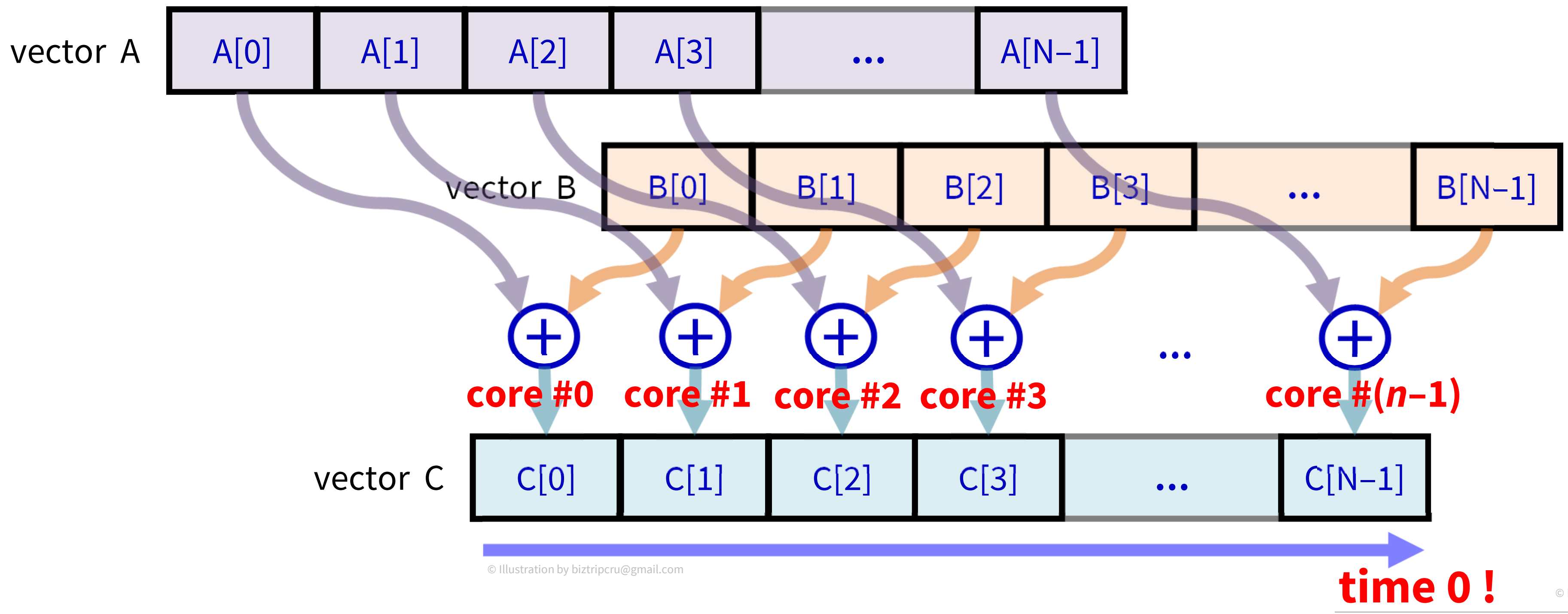
CPU-based vector addition

- **a single CPU core does a single addition**
  - then, the next addition



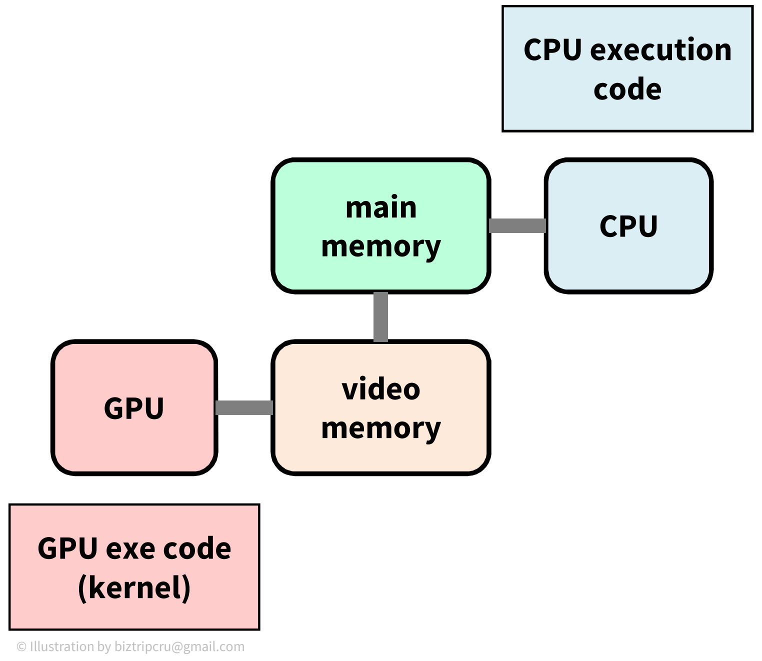
CUDA-based vector addition

- **we have many GPU cores**
  - they do the addition at the same time !



Scenario: CUDA vector addition

- **step 1: host-side**
  - make A, B with source data
  - prepare C for the result
- **step 2: data copy host → device**
  - cudaMemcpy from host to device
- **step 3: addition in CUDA**
  - **kernel launch** for CUDA device
  - result will be stored in device memory
- **step 4: data copy device → host**
  - cudaMemcpy from device to host
- **step 5: host-side**
  - print out



Function call vs Kernel Launch

- **C/C++ function call syntax**

```
void  func_name( int param, … );
for (int i = 0; i < SIZE; ++i) {
   func_name( param, … );
}
```

- **CUDA kernel launch syntax**

```
__global__ void  kernel_name( int param, … );
kernel_name <<< 1, SIZE >>>( param, … );
```

  - <<<, >>> : C/C++ 에서 사용하지 않는 operator

phone call

rocket launch !

CUDA kernel launch

- **prepare a CUDA kernel function,**

```
__global__ void add_kernel( int* c, const int* a, const int* b ) {
   int i = threadIdx.x;   // each thread knows its own index
   c[i] = a[i] + b[i];
}
```

threadIdx

CUDA kernel 에서, index 변수 자동 설정

- **kernel launch syntax**

```
add_kernel<<<1,SIZE>>>( dev_c, dev_a, dev_b );
```

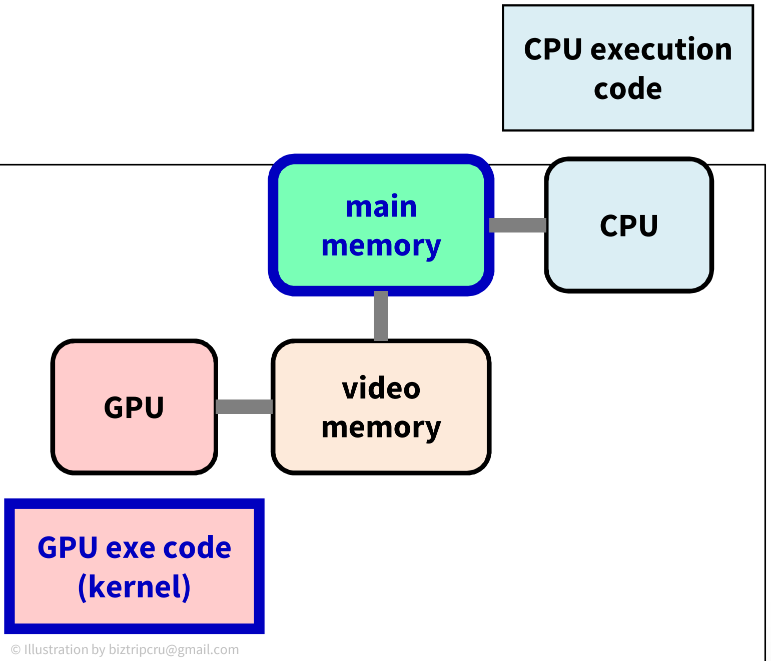
```
for (i = 0; i < SIZE; ++i) {
   ...
}
```

CPU 의 순차 처리 sequential processing

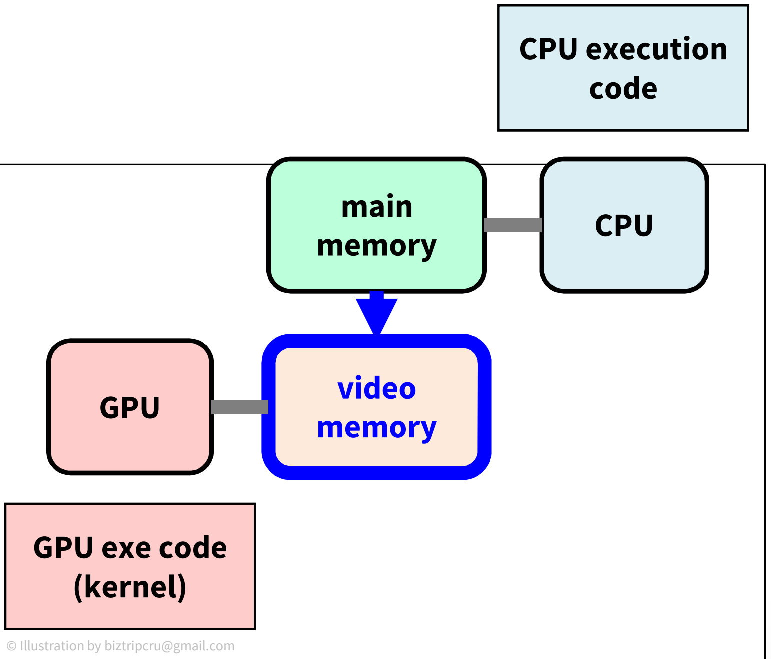
- **CUDA view**
  - a thread executes add_kernel( ) with threadIdx.x = 0
  - a thread executes add_kernel( ) with threadIdx.x = 1
  - ...
  - a thread executes add_kernel( ) with threadIdx.x = SIZE–1

gpu-add.cu

```
#include "./common.cpp"

// kernel program for the device (GPU): compiled by NVCC
__global__ void add_kernel( int* c, const int* a, const int* b ) {
   int i = threadIdx.x;
   c[i] = a[i] + b[i];
}


// main program for the CPU: compiled by MS-VC++
int main(void) {
   // host-side data
   const int SIZE = 5;
   const int a[SIZE] = { 1, 2, 3, 4, 5 };
   const int b[SIZE] = { 10, 20, 30, 40, 50 };
   int c[SIZE] = { 0 };
```

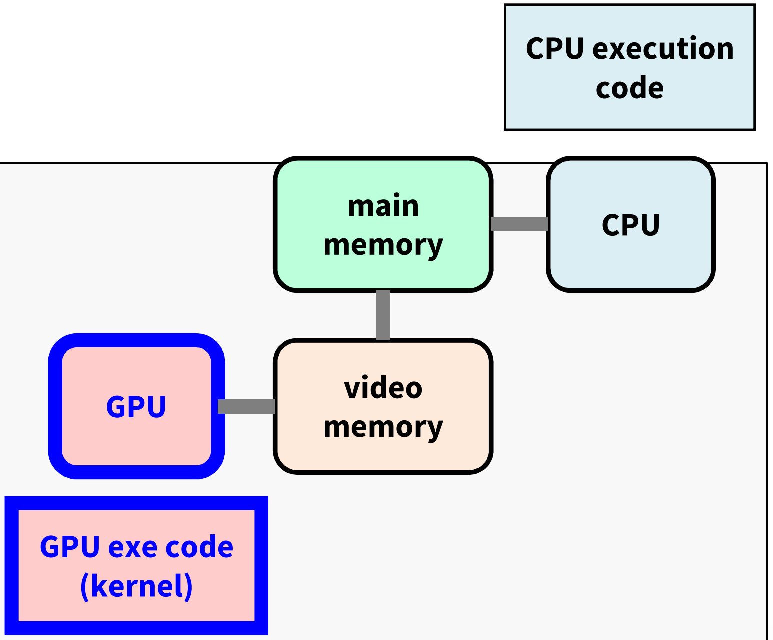


gpu-add.cu 계속



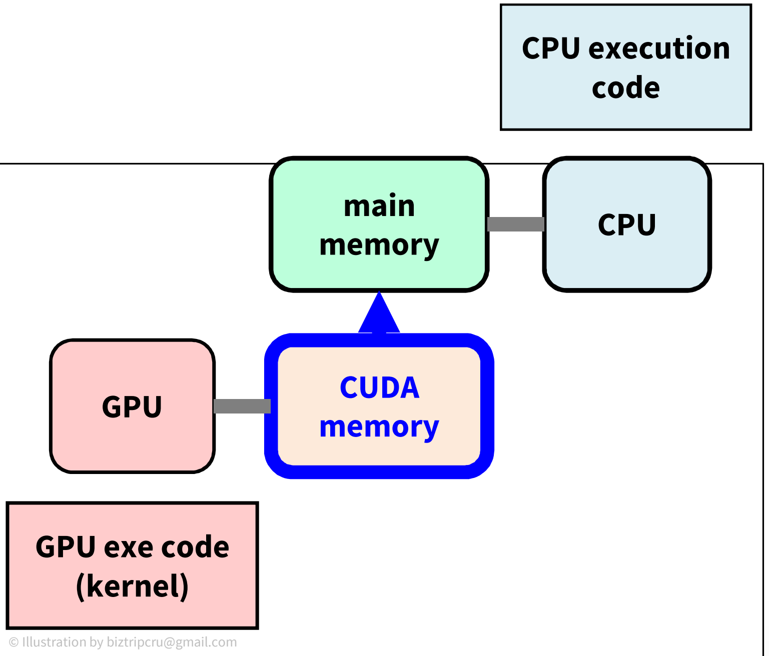
```
// device-side data
int* dev_a = 0;
int* dev_b = 0;
int* dev_c = 0;
// allocate device memory
cudaMalloc( (void**)&dev_a, SIZE * sizeof(int) );
cudaMalloc( (void**)&dev_b, SIZE * sizeof(int) );
cudaMalloc( (void**)&dev_c, SIZE * sizeof(int) );
// copy from host to device
cudaMemcpy( dev_a, a, SIZE * sizeof(int), cudaMemcpyHostToDevice ); // dev_a = a;
cudaMemcpy( dev_b, b, SIZE * sizeof(int), cudaMemcpyHostToDevice ); // dev_b = b;
```

gpu-add.cu 계속



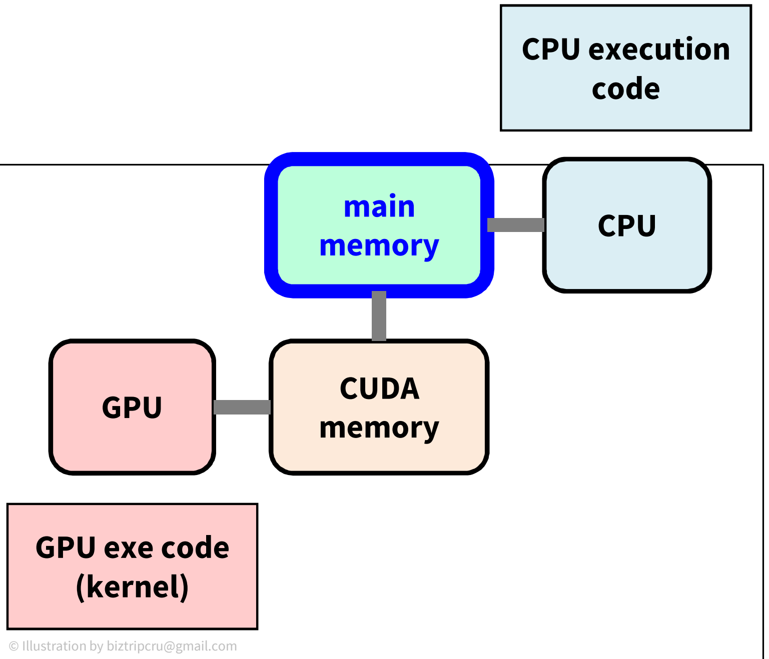
```
// kernel program for the device (GPU): compiled by NVCC
__global__ void add_kernel( int* c, const int* a, const int* b ) {
   int i = threadIdx.x;
   c[i] = a[i] + b[i];
}


int main(void) {
   // host-side data
   const int SIZE = 5;
   . . .

   // launch a kernel on the GPU with one thread for each element.
   add_kernel<<<1,SIZE>>>( dev_c, dev_a, dev_b );     // dev_c = dev_a + dev_b;
   cudaDeviceSynchronize();
```

gpu-add.cu 계속



```
// copy from device to host
cudaMemcpy( c, dev_c, SIZE * sizeof(int), cudaMemcpyDeviceToHost ); // c = dev_c;
// free device memory
cudaFree(dev_a );
cudaFree(dev_b );
cudaFree(dev_c );
```

gpu-add.cu 계속



```
    // print the result
    printf("{%d,%d,%d,%d,%d} + {%d,%d,%d,%d,%d} = {%d,%d,%d,%d,%d}\n",
        a[0], a[1], a[2], a[3], a[4],  b[0], b[1], b[2], b[3], b[4],  c[0], c[1], c[2], c[3], c[4]);
    // done
    fflush( stdout );
    return 0;
}
```

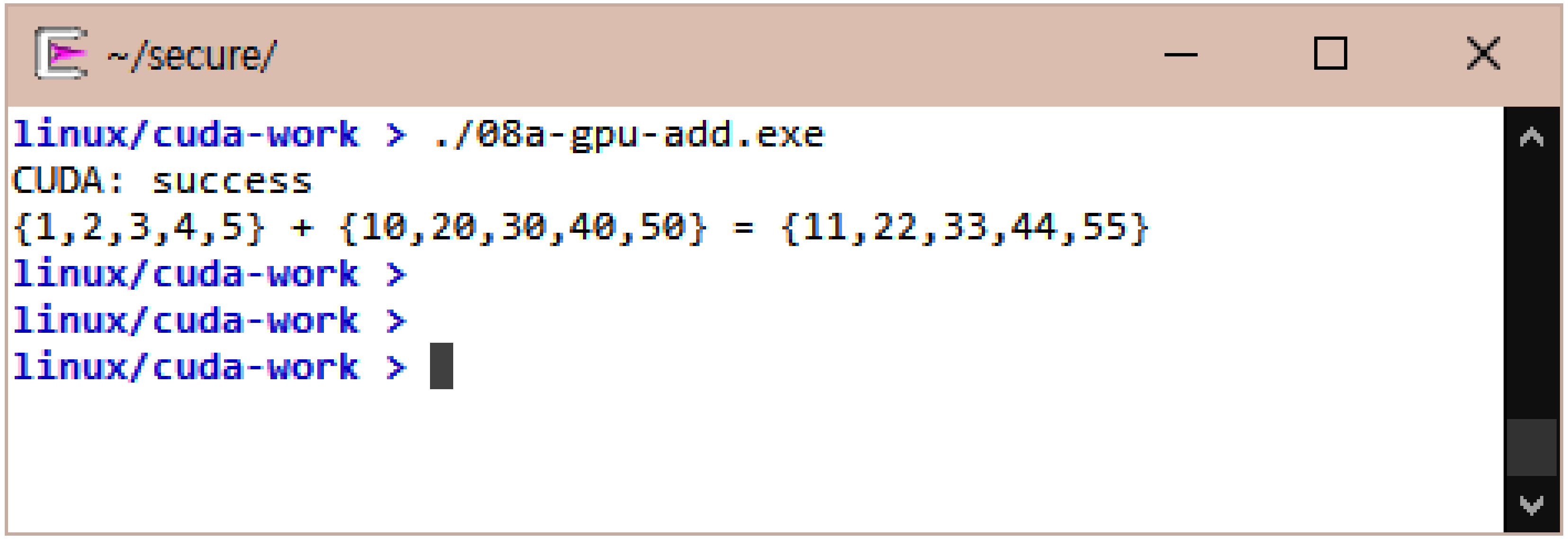
gpu-add.cu

```
#include "./common.cpp"

// kernel program for the device (GPU): compiled by NVCC
__global__ void add_kernel( int* c, const int* a, const int* b ) {
   int i = threadIdx.x;
   c[i] = a[i] + b[i];
}


// main program for the CPU: compiled by MS-VC++
int main(void) {
   // host-side data
   const int SIZE = 5;
   const int a[SIZE] = { 1, 2, 3, 4, 5 };
   const int b[SIZE] = { 10, 20, 30, 40, 50 };
   int c[SIZE] = { 0 };
   // device-side data
   int* dev_a = 0;
   int* dev_b = 0;
   int* dev_c = 0;
   // allocate device memory
   cudaMalloc( (void**)&dev_a, SIZE * sizeof(int) );
   cudaMalloc( (void**)&dev_b, SIZE * sizeof(int) );
   cudaMalloc( (void**)&dev_c, SIZE * sizeof(int) );
```

```
   // copy from host to device
   cudaMemcpy( dev_a, a, SIZE * sizeof(int), cudaMemcpyHostToDevice );
   cudaMemcpy( dev_b, b, SIZE * sizeof(int), cudaMemcpyHostToDevice );
   // launch a kernel on the GPU with one thread for each element.
   add_kernel<<<1,SIZE>>>( dev_c, dev_a, dev_b );
   cudaDeviceSynchronize();
   // copy from device to host
   cudaMemcpy( c, dev_c, SIZE * sizeof(int), cudaMemcpyDeviceToHost );
   // free device memory
   cudaFree(dev_a );
   cudaFree(dev_b );
   cudaFree(dev_c );
   // print the result
   printf("{%d,%d,%d,%d,%d} + {%d,%d,%d,%d,%d} = {%d,%d,%d,%d,%d}\n",
      a[0], a[1], a[2], a[3], a[4], b[0], b[1], b[2], b[3], b[4],
      c[0], c[1], c[2], c[3], c[4]);
   // done
   fflush( stdout );
   return 0;
}
```

gpu-add.cu – result

- **execution result (with error check)**



```
linux/cuda-work > ./08a-gpu-add.exe
CUDA: success
{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}
linux/cuda-work >
linux/cuda-work >
linux/cuda-work >
```

Kernel Error Check

- **kernel launch**
  - it does **NOT** return any error code.
  - But, we can use **cudaPeekAtLastError**( )
- **example**

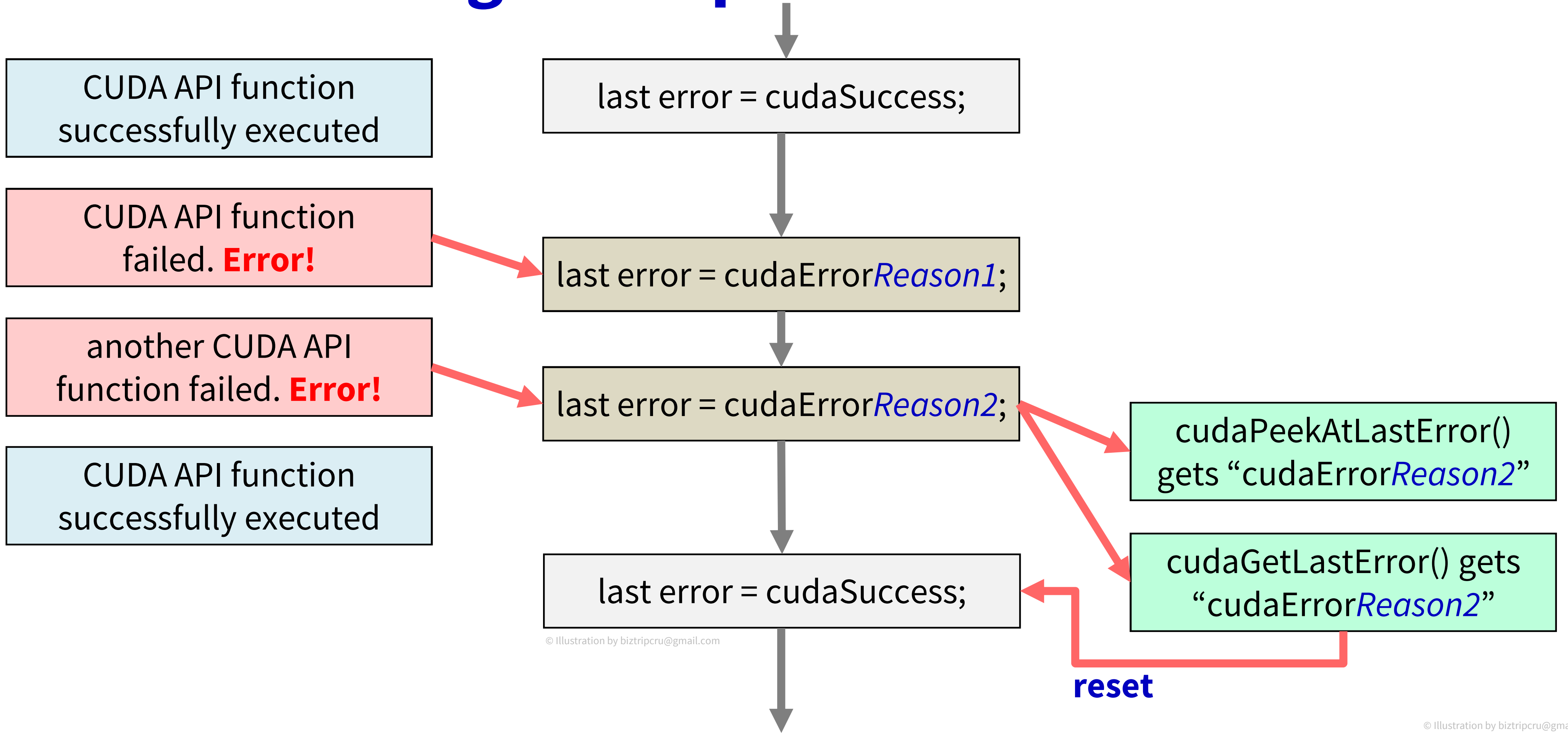
```
add_kernel<<<1,SIZE>>>( dev_c, dev_a, dev_b );     // dev_c = dev_a + dev_b;
cudaDeviceSynchronize();
cudaError_t err = cudaPeekAtLastError();
if (cudaSuccess != err) {
   …
}
```

- **or, use CUDA_CHECK_ERROR() macro in "./common.cpp"**

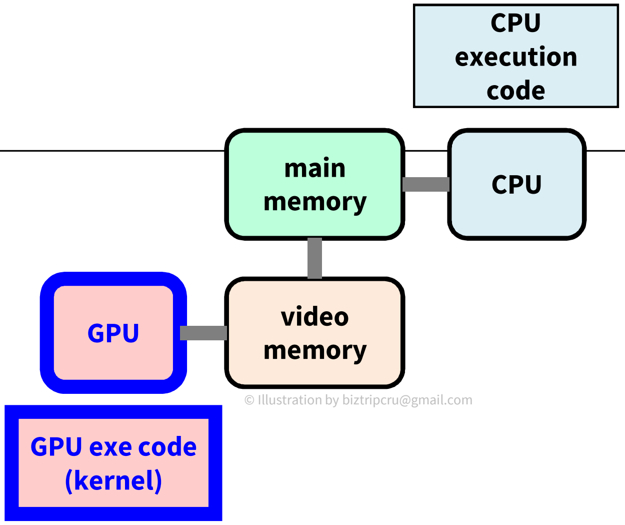
cudaGetLastError( )

- cudaError_t **cudaGetLastError**(void);
  - returns **the last error** due to CUDA runtime calls in the same host thread
  - and **resets** it to **cudaSuccess**
  - So, if no CUDA error since the last call, it returns **cudaSuccess**
  - For multiple errors, it contains **the last error only.**
- cudaError_t **cudaPeekAtLastError**(void);
  - returns the last error
  - Note that this call does **NOT** reset the error to **cudaSuccess**
  - So, the last error code is still available

CUDA Error Flag Concept


CUDA API function successfully executed
CUDA API function failed. Error!
another CUDA API function failed. Error!
CUDA API function successfully executed
last error = cudaSuccess;
last error = cudaErrorReason1;
last error = cudaErrorReason2;
last error = cudaSuccess;
cudaPeekAtLastError() gets "cudaErrorReason2"
cudaGetLastError() gets "cudaErrorReason2"
reset
© Illustration by biztripcru@gmail.com

gpu-err-check.cu



```
// launch a kernel on the GPU with one thread for each element.
add_kernel<<<1,SIZE>>>( dev_c, dev_a, dev_b );     // dev_c = dev_a + dev_b;
cudaDeviceSynchronize();
cudaError_t err = cudaPeekAtLastError();
if (cudaSuccess != err) {
   printf("CUDA: ERROR: cuda failure \"%s\"\n", cudaGetErrorString(err));
   exit(1);
} else {
   printf("CUDA: success\n");
}
```

Kernel Error Check 추가 설명

- **CUDA kernel 함수는 void 만 가능**

- **왜 error 코드를 직접 return 하지 않을까?**
  - CPU는 단 1개의 return 값을 기대
  - CUDA kernel의 100만개 병렬 처리 → 100만개의 return 값 (error code)

- **그러면, (간단한) 계산 결과는 어떻게 알려주나?**
  - CUDA memory 영역의 배열/변수를 직접 update
  - 예: `__global__ void add_kernel( int* outC, const int* inA, const int* inB );`

내용 contents

- **CUDA programming model**
  - CUDA function declarations
  - vector addition example
- **CUDA implementation**
  - multiple thread launch
- **CUDA kernel launch**
  - example source code

CUDA Kernel

CUDA 커널

폰트 끝단 일치 → 큰 교자 타고 혼례 치른 날
정참판 양반댁 규수 큰 교자 타고 혼례 치른 날
정참판 양반댁 규수 큰 교자 타고 혼례 치른 날
본고딕 Noto Sans KR

The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog
Source Sans Pro

Mathematical Notations O(n log n)
Source Serif Pro