CUDA 프로그래밍

CUDA Programming
An Massively Parallel Computing Approach

biztripcru@gmail.com

© 2021–2022. biztripcru@gmail.com. All rights reserved.
모든 저작권은 biztripcru@gmail.com 에게 있습니다.

CPU Kernel

CPU 커널

본 동영상과, 본 동영상 촬영에 사용된 발표 자료는 저작권법의 보호를 받습니다.
본 동영상과 발표 자료는 공개/공유/복제/상업적 이용 등, **개인 수강 이외의 다른 목적으로 사용하지 못합니다.**

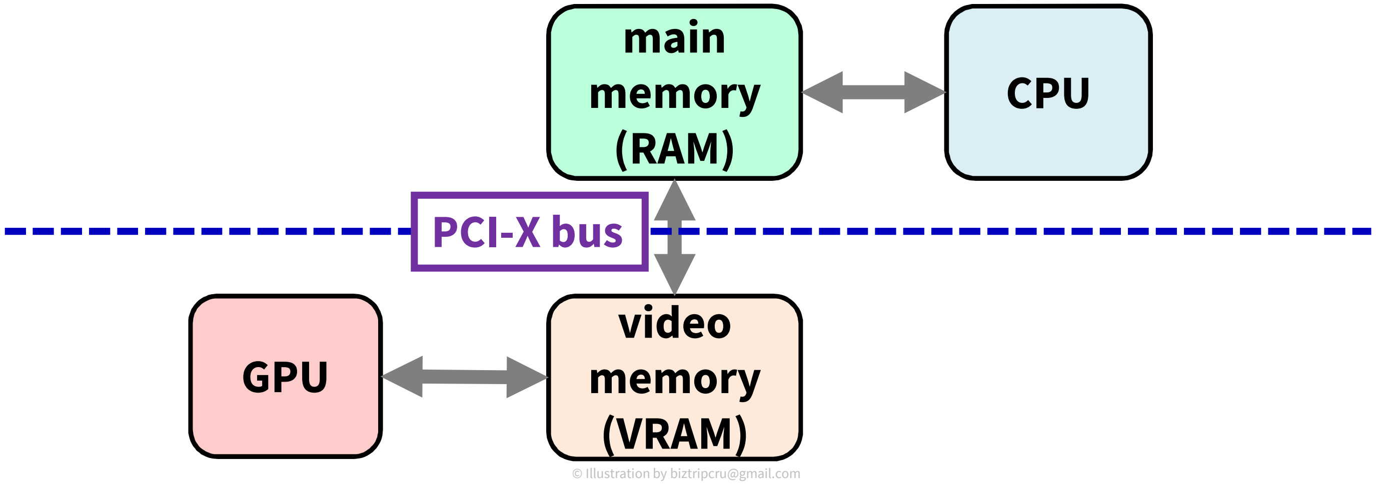
© 2021–2022. biztripcru@gmail.com. All rights reserved.
모든 저작권은 biztripcru@gmail.com 에게 있습니다.

내용 contents

- **CUDA programming model**
  - CUDA function declarations
  - vector addition example
- **CPU implementation**
  - for loop
- **kernel concept**
  - function, as loop-body

CUDA 프로그래밍 모델

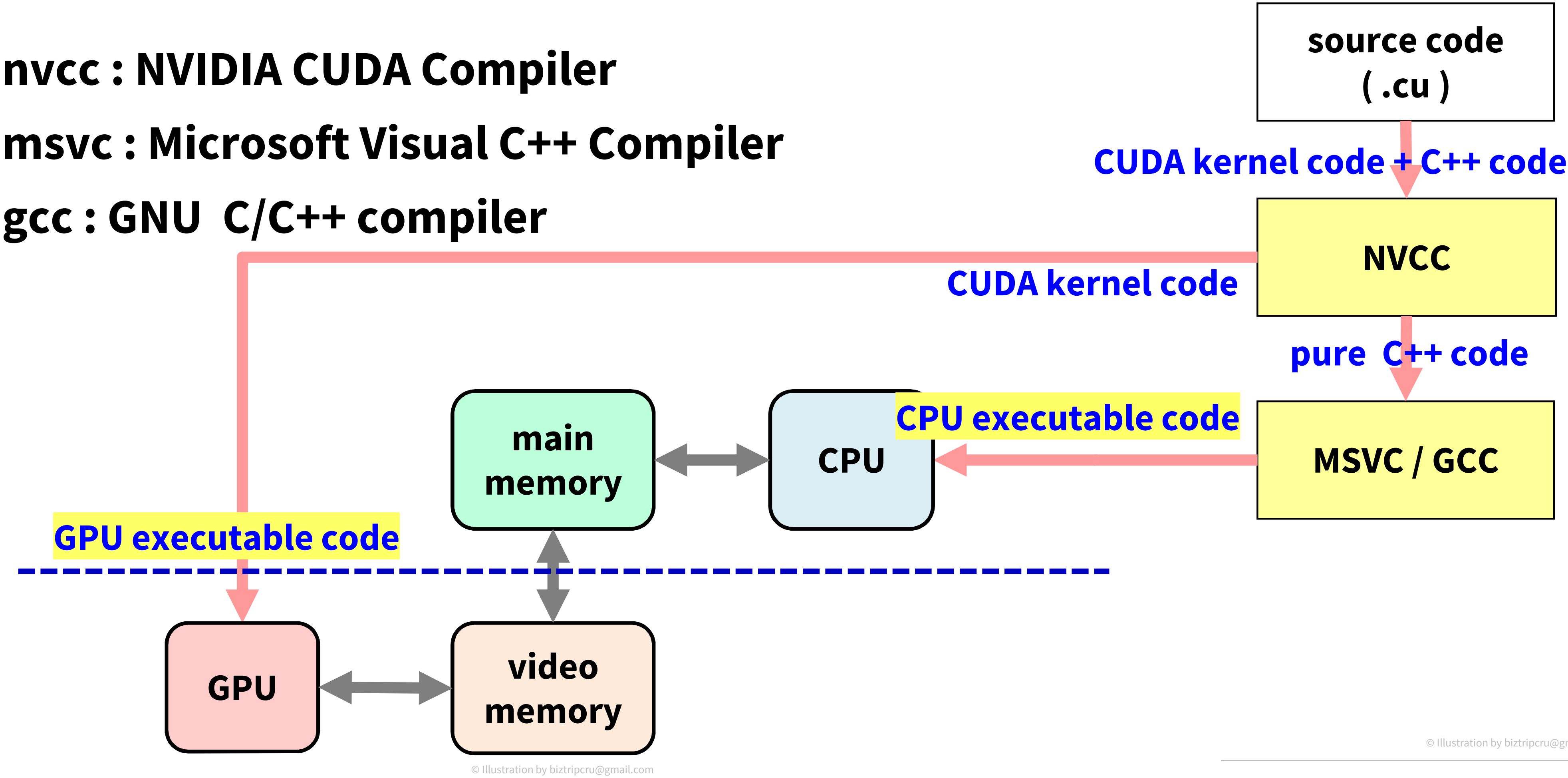
- **We have two sets of execution code**
  - for CPU (host part)
  - for GPU (device part)



© Illustration by biztripcru@gmail.com

CUDA 컴파일러 구조

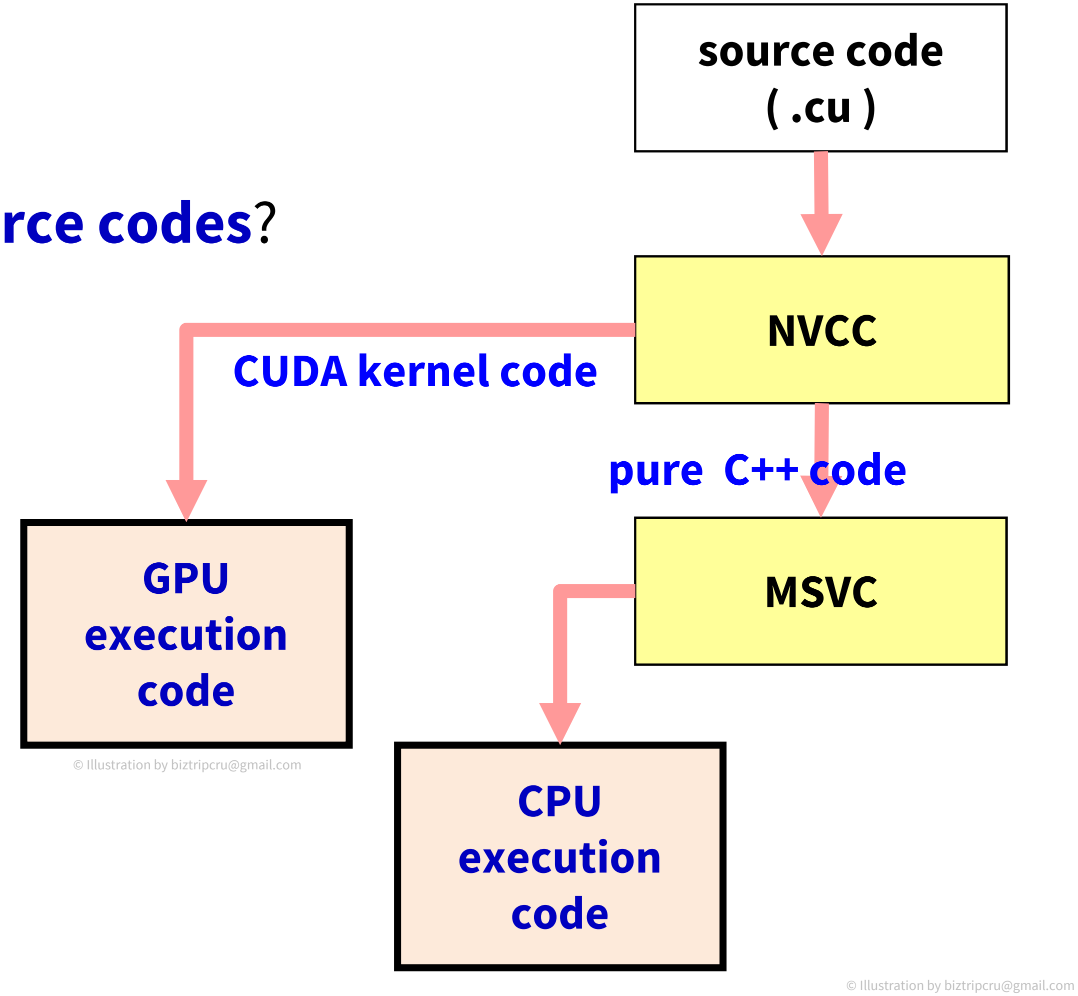
- nvcc : NVIDIA CUDA Compiler
- msvc : Microsoft Visual C++ Compiler
- gcc : GNU C/C++ compiler



© Illustration by biztripcru@gmail.com

CUDA 코드 생성

- **At the source code level,**
  - how can we distinguish those **source codes**?
  - Which one is for CPU ?
  - Which one is for GPU ?
  - And, what is
    the unit of compilations ?
    - 컴파일 단위?
  - file
    - 그 중간 어디쯤?
  - line → ASM in C/C++



CUDA programming model

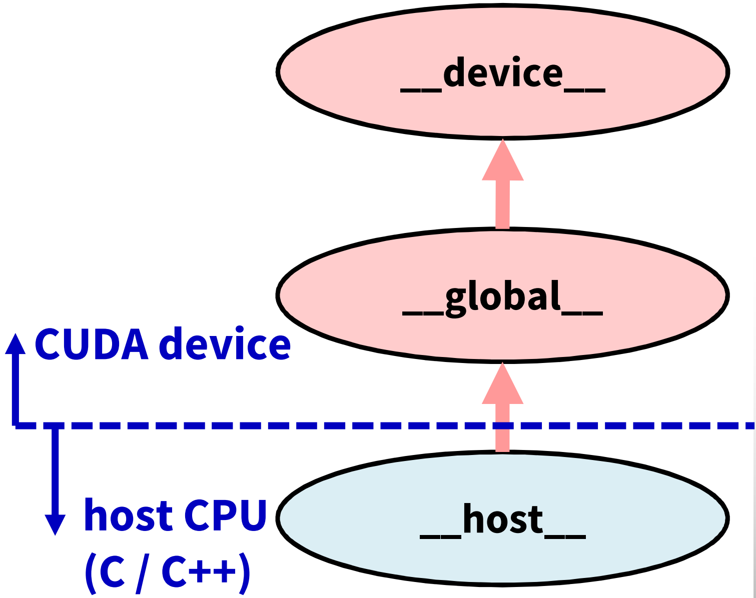
- **compilation unit ?**
  - **functions**
  - Each function will be assigned to CPU and/or GPU
- **how to distinguish them?**
  - **use PREFIX for each function**
  - **__host__** : can be called by CPU (**default**, can be omitted)
  - **__device__** : called from other GPU functions,
    cannot be called by the CPU
  - **__global__** : launched by CPU,
    cannot be called from GPU, must return void
  - __host__ and __device__ qualifiers can be combined

More on CUDA Function Declarations

- **__global__ defines a kernel 커널 function**
  - Each "__" consists of **two underscore 밑줄** characters
  - A kernel function must return void
- **__device__ and __host__ can be used together**
  - compiled twice (!) → 2번 컴파일 해서 각각 사용

	called by:	executed on:
__device__ float deviceFunc( )	device	device
__global__ void kernelFunc( )	host →	device
__host__ void main( )	host	host

© Illustration by biztripcru@gmail.com



© Illustration by biztripcru@gmail.com

Example Source Code:

```
__device__ inline void myAtomicAdd(unsigned long long int* address, unsigned long long int val) {
   unsigned long long int oldVal = *address;
   unsigned long long int newVal = oldVal + val;
   unsigned long long int readback;
   while ((readback = atomicCAS(address, oldVal, newVal)) != oldVal) {
      oldVal = readback;
      newVal = oldVal + val;
   }
}

__global__ void kernel(unsigned long long int* pCount) {
   myAtomicAdd(pCount, 1ULL);
}

__host__ int main(void) {
   unsigned long long int aCount[1]; // 64bit integer
   …
```

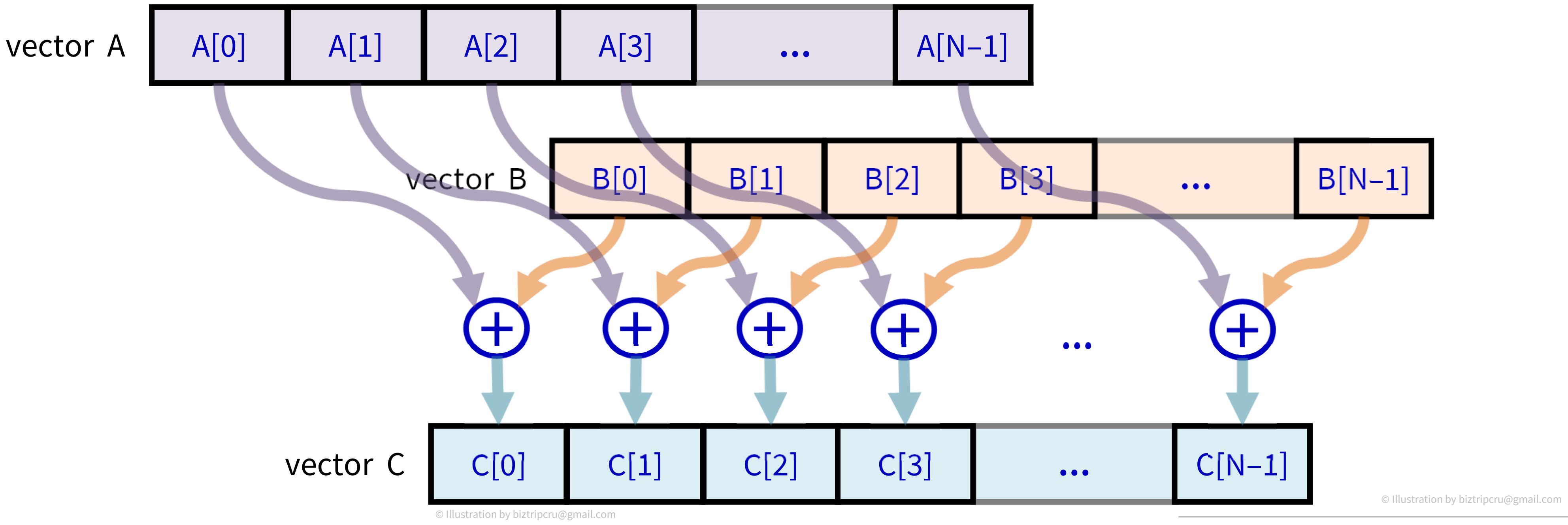
GPU executable functions

- **__device__ and __global__ prefixed functions**
  - __device__ and __global__ functions cannot have their address taken

- **CUDA language = C/C++ language with some restrictions:**
  - Can only access **GPU memory (CUDA memory, video memory)**
    - in new versions, can access host memory directly, with performance drawbacks.
  - No static variables
    - No static variable declarations inside the function
  - No recursion
    - in new versions, it is possible
  - No dynamic polymorphism

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**

Example: cpu-add.cu

```
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 };
   // calculate the addition
   for (register int i = 0; i < SIZE; ++i) {
      c[i] = a[i] + b[i];
   }
   // 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
   return 0;
}
```

Example: cpu-add.cu – result

- **execution result:**

~/secure/

```
linux/cuda-work > ./07a-cpu-add.exe
{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}
linux/cuda-work >
linux/cuda-work >
```

Considering the for loop

- **We use a for loop**

```
// calculate the addition
for (register int i = 0; i < SIZE; ++i) {
   c[i] = a[i] + b[i];
}
```

- **a single CPU does for i = 0, i = 1, ... i = SIZE–1**

Introducing the kernel function

- **kernel function:**
  - substitute for the **loop body**
  - with proper data values
- **real example:**

```
void add_kernel( int idx, const int* a, const int* b, int* c ) {
  int i = idx;
  c[i] = a[i] + b[i];
}
…
for (register int i = 0; i < SIZE; ++i) {
  add_kernel( i, a, b, c );
}
```

Example: cpu-kernel.cu

```
void add_kernel( int idx, const int* a, const int* b, int* c ) {
   int i = idx;
   c[i] = a[i] + b[i];
}

int main(void) {
   // host-side data
   ...
   // calculate the addition
   for (register int i = 0; i < SIZE; ++i) {
      add_kernel( i, a, b, c );
   }
   // print the result
   ...
   // done
   return 0;
}
```

Example: cpu-kernel.cu – result

- **execution result: the same result**

```
~/secure/
linux/cuda-work > ./07b-cpu-kernel.exe
{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}
linux/cuda-work >
```

The kernel function

- **another view**
  - CPU[0] executes add_kernel(0, …) with its own data
  - CPU[1] executes add_kernel(1, …) with its own data
  - CPU[2] executes add_kernel(2, …) with its own data
  - CPU[3] executes add_kernel(3, …) with its own data
  - ...
  - CPU[SIZE–1] executes add_kernel(SIZE–1, …) with its own data
  - done !

The kernel function 계속

- **we have a single CPU → sequential execution**
  - at time 0: CPU executes add_kernel(0, …) with its own data
  - at time 1: CPU executes add_kernel(1, …) with its own data
  - at time 2: CPU executes add_kernel(2, …) with its own data
  - at time 3: CPU executes add_kernel(3, …) with its own data
  - ...
  - at time (n – 1): CPU executes add_kernel(SIZE–1, …) with …
  - done !

The kernel function 계속

- **if we have multi-core CPU's → parallel execution !**
  - at time 0: CPU$^{[\text{core \#0}]}$ executes add_kernel(0, …) with its own data
  - at time 0: CPU$^{[\text{core \#1}]}$ executes add_kernel(1, …) with its own data
  - at time 1: CPU$^{[\text{core \#0}]}$ executes add_kernel(2, …) with its own data
  - at time 1: CPU$^{[\text{core \#1}]}$ executes add_kernel(3, …) with its own data
  - ...
  - at time $(n-1)/2$: CPU$^{[\text{core \#1}]}$ executes add_kernel(SIZE−1, …) with …
  - done !

The kernel function 계속

- **and for many-core GPU's ? → massively parallel execution !**
  - at time 0: GPU[core #0] executes add_kernel(0, …) with its own data
  - at time 0: GPU[core #1] executes add_kernel(1, …) with its own data
  - at time 0: GPU[core #2] executes add_kernel(2, …) with its own data
  - at time 0: GPU[core #3] executes add_kernel(3, …) with its own data
  - ...
  - at time 0: GPU[core #(SIZE–1)] executes add_kernel(SIZE–1, …) with …
  - done !
- **Now, explore the GPU kernels !**

내용 contents

- **CUDA programming model**
  - CUDA function declarations
  - vector addition example
- **CPU implementation**
  - for loop
- **kernel concept**
  - function, as loop-body

CPU Kernel

CPU 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

© 2021–2022. biztripcru@gmail.com. All rights reserved.
모든 저작권은 biztripcru@gmail.com 에게 있습니다.

Mathematical Notations $O(n \log n)$
Source Serif Pro