# HW 2: OpenCL GEMM Optimization

2021-01 인공지능 플랫폼 최적화

HW/SW Optimization for Machine Learning

박영준

# HW #2: OpenCL programming & optimization

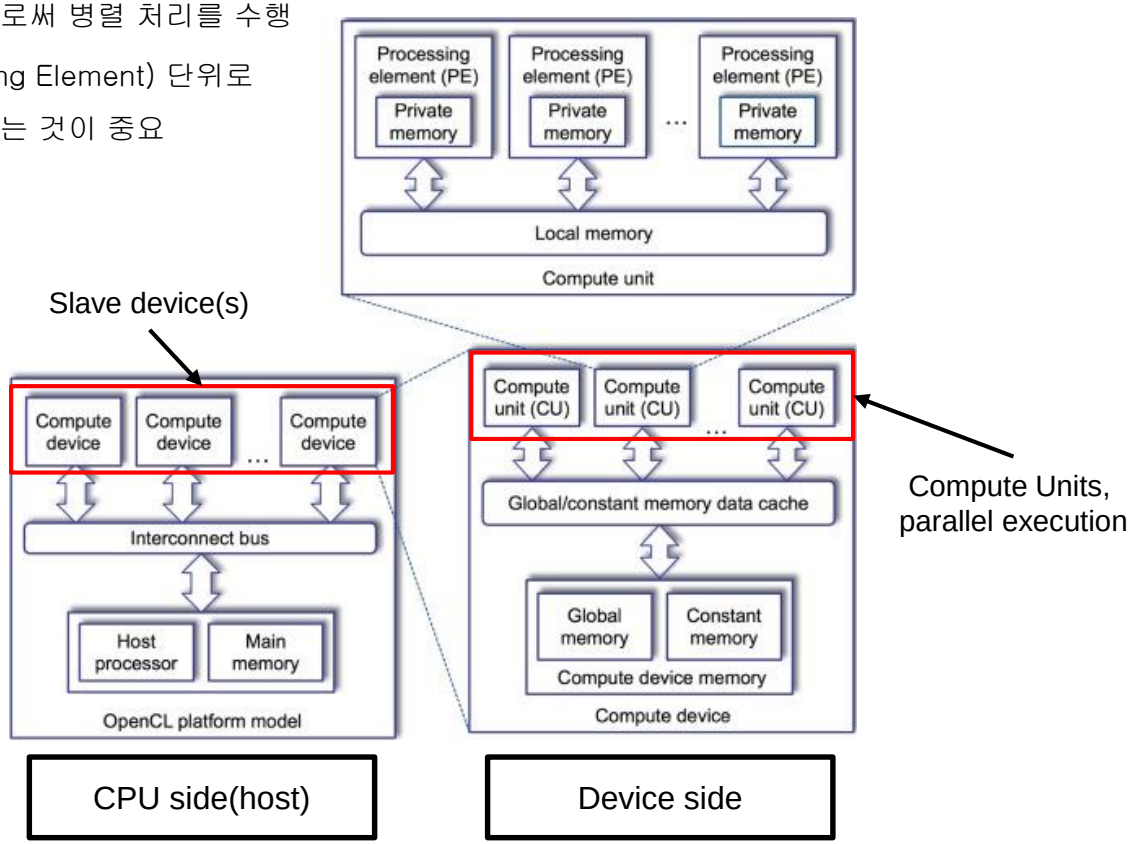
- 과제: 본 문서의 Step I ~ Step V 의 내용을 직접 수행하고 결과를 레포트로 제출
- 레포트에는 다음의 내용이 필수적으로 포함되어야 함
  - [Step I] OpenCL host/kernel을 구성하고 수행한 결과 스크린샷
  - [Step II] Naïve SGEMM 실행 결과 스크린샷
  - [Step III] Naïve SGEMM (cont'd) 실행 결과 스크린샷
  - [Step IV] SGEMM with loop unrolling 실행 결과 스크린샷
  - [Step V] SGEMM with vectorization 실행 결과 스크린샷
  - [Step II ~ V] 의 경우, 연산 검증 결과가 'PASSED' 이어야 함
- Deadline: 5/14 (Friday) 23:59:59
- Format: PDF or MS word file
- 문의: 유용승 (dydtmd1991@hanyang.ac.kr)

# OpenCL


- OpenCL
  - Open Computing Language, 이종 플랫폼을 위한 standard 병렬 프로그래밍 모델
  - CPU, GPU, FPGA, DSP 등 다양한 architecture에 적용 가능
  - Host(CPU side)에서는 할당될 작업들을 관리, slave(target device)에서는 할당된 작업들을 수행
- OpenCL의 특징
  - Data / task 병렬 프로그래밍 모델 지원
  - ANSI/ISO C99 기반 (OpenCL 1.x version)
  - 작성한 코드를 여러 디바이스에서 사용 가능 (코드 이식성이 높음)
- OpenCL의 한계
  - Target device에 따라 code 최적화가 어려울 수 있고, 프로그래밍의 복잡성이 높은 편
  - 단일 OS에서의 device만 프로그래밍 할 수 있음

# OpenCL

- OpenCL platform 모델
  - OpenCL에서는 각 slave device의 Compute Unit(CU)에 전체 작업을 나누어 할당함으로써 병렬 처리를 수행
  - 각 CU 단위, 각 PE(Processing Element) 단위로 전체 작업을 효율적으로 나누는 것이 중요

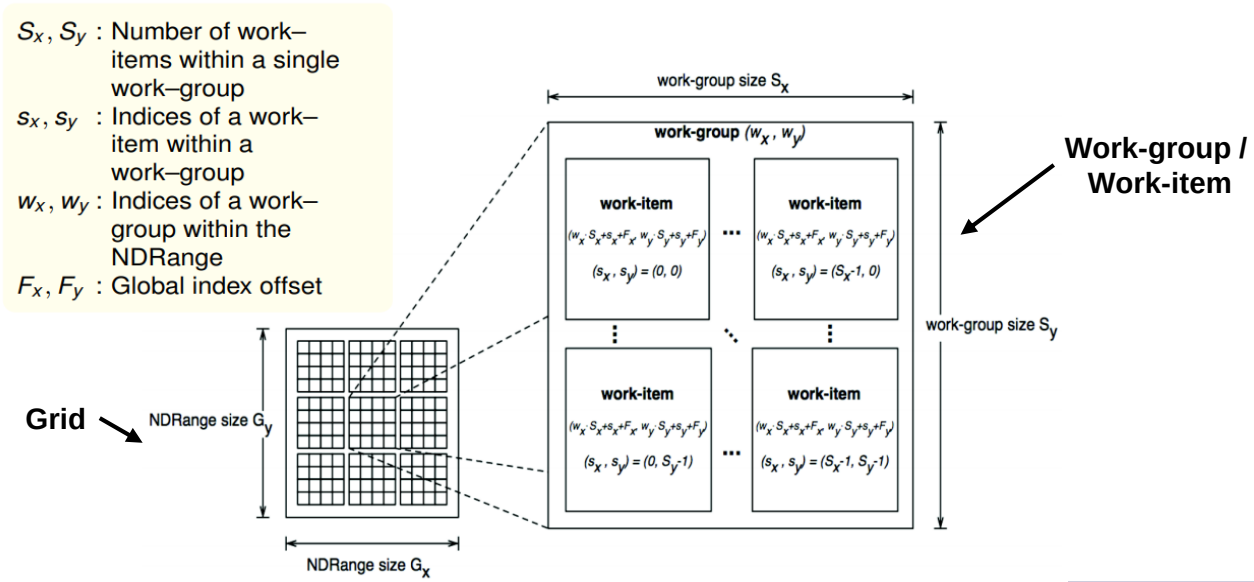


# OpenCL: Basics

- OpenCL 실행 과정
  - 1. Platform 및 device(사용할 수 있는 device) 정보 얻기
    - clGetPlatformIDs()
  - 2. 작업 할당에 필요한 객체, 메모리 등의 할당 및 초기화
    - Context: 전체적인 OpenCL 환경 변수 등을 정의, clCreateContext()
    - Command-Queue: device가 수행할 작업들을 순차적으로 담고 있는 queue, clCreateCommandQueue(),
    - Memory 할당: device의 작업에 필요한 메모리 영역 할당, clCreateBuffer()
    - Memory 초기화(write buffer): device 메모리 영역에 데이터 복사, clEnqueueWriteBuffer()
  - 3. OpenCL program (*.cl) build 및 kernel 객체 생성
    - Program 생성: kernel source 파일을 통해 OpenCL program 객체를 생성, clCreateProgramWithSource()
    - Program 빌드: 생성한 program 객체를 target device에 맞게 빌드, clBuildProgram()
    - Kernel 객체 생성: 빌드한 program에서 user가 사용할 이름의 kernel 객체를 생성, clCreateKernel()
  - 4. 커널 launch
    - Kernel launch를 위한 매개변수 인자(argument) 설정: clSetKerneArg()
    - kernel launch: clEnqueueNDRangeKernel()
  - 5. 커널 실행 결과 copy(device to host)
    - Device memory copy: Device 메모리로부터 데이터 복사(to host), clEnqueueReadBuffer()

# OpenCL: Basics

- OpenCL execution model
  - SIMD(Single Instruction, Multiple Data) 구조를 따름
    - 모든 thread가 같은 code를 수행하지만(single-instruction), access 하는 data가 각각 다름(multiple data)
  - Grid, work-group, work-item
    - Grid: 커널 launch 후 수행해야 할 모든 thread의 집합
    - Work-group: Grid 전체의 thread를 일정한 수로 나눈 group, local(shared) memory 등을 통해 서로 data를 교차하여 접근할 수 있음
    - Work-item: 하나의 thread와 1:1 대응됨
- Grid ~ work-group ~ work-item hierarchy



# OpenCL: Useful references

- 유용한 참고 자료

  OpenCL 1.2 standard

  url: https://www.khronos.org/registry/cl/

  OpenCL 1.2 reference page

  url: https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml

  OpenCL 1.2 reference card

  url: https://www.khronos.org/registry/cl/sdk/1.2/docs/OpenCL-1.2-refcard.pdf

  OpenCL Basics (from JULICH, FORSCHUNGSZENTRUM)

  url: https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/opencl/opencl-03-basics.pdf?__blob=publicationFile

# [Step I] Getting started

- Simple OpenCL host and OpenCL vectorAdd kernel
  - 간단한 OpenCL host 프로그램과 GPU를 타겟으로 하는 OpenCL kernel을 구성하여,
    GPU에서의 kernel 동작이 잘 이루어지는지 test
- Pre-requisite
  - (On ubuntu any releases) NVIDIA graphics driver & recent CUDA driver

    Reference:

    Installing a CUDA driver on Ubuntu OS:

    url: https://velog.io/@cychoi74/%EC%9A%B0%EB%B6%84%ED%88%AC-18.04-NVIDIA-%EB%93%9C%EB%9D%BC%EC%9D%B4%EB%B2%84-%EC%84%A4%EC%B9%98
  - 예제 host program 및 kernel source

    References:

    OpenCL tutorial: Getting started with OpenCL and GPU Computing,

    url: https://www.eriksmistad.no/getting-started-with-opencl-and-gpu-computing/

# [Step I] Getting started

- Simple OpenCL host and OpenCL vectorAdd kernel
  - opencl_host.c

```c
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MAX_SOURCE_SIZE (0x100000)

int main(void) {
    // Create the two input vectors
    int i;
    const int LIST_SIZE = 32;
    int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
    int *B = (int*)malloc(sizeof(int)*LIST_SIZE);
    for(i = 0; i < LIST_SIZE; i++) {
        A[i] = i;
        B[i] = LIST_SIZE - i;
    }

    // Load the kernel source code into the array source_str
    FILE *fp;
    char *source_str;
    size_t source_size;

    fp = fopen("opencl_kernel.cl", "r");
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
    source_str = (char*)malloc(MAX_SOURCE_SIZE);
    source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
    fclose( fp );

    // Get platform and device information
    cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
            &device_id, &ret_num_devices);

    // Create an OpenCL context
    cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

    // Create a command queue
    cl_command_queue command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

    // Create memory buffers on the device for each vector
    cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(int), NULL, &ret);
    cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(int), NULL, &ret);
    cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            LIST_SIZE * sizeof(int), NULL, &ret);

    // Copy the lists A and B to their respective memory buffers
    ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
    ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

    // Create a program from the kernel source
    cl_program program = clCreateProgramWithSource(context, 1,
            (const char **)&source_str, (const size_t *)&source_size, &ret);

    // Build the program
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel
    cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

    // Execute the OpenCL kernel on the list
    size_t global_item_size = LIST_SIZE; // Process the entire lists
    size_t local_item_size = 64; // Divide work items into groups of 64
    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
            &global_item_size, &local_item_size, 0, NULL, NULL);

    // Read the memory buffer C on the device to the local variable C
    int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
    ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), C, 0, NULL, NULL);

    // Display the result to the screen
    for(i = 0; i < LIST_SIZE; i++)
        printf("%d + %d = %d\n", A[i], B[i], C[i]);

    // Clean up
    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(a_mem_obj);    ret = clReleaseMemObject(b_mem_obj);    ret = clReleaseMemObject(c_mem_obj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);
    free(A);    free(B);    free(C);

    return 0;
}
```

# [Step I] Getting started

- Simple OpenCL host and OpenCL vectorAdd kernel
  - opencl_kernel.cl

```
1 __kernel void vector_add(__global const int *A, __global const int *B, __global int *C) {
2
3     // Get the index of the current element to be processed
4     int i = get_global_id(0);
5
6     // Do the operation
7     C[i] = A[i] + B[i];
8 }
```

  - get_global_id(0): Grid 전체의 work-item을 0번부터 size_of(Grid) – 1번까지 대응한 고유 id를 return
  - Khronos의 OpenCL API detail을 참고

# [Step I] Getting started

- Simple OpenCL host and OpenCL vectorAdd kernel
  - 실행 결과 출력부분

```c
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MAX_SOURCE_SIZE (0x100000)

int main(void) {
    // Create the two input vectors
    int i;
    const int LIST_SIZE = 32;
    int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
    int *B = (int*)malloc(sizeof(int)*LIST_SIZE);
    for(i = 0; i < LIST_SIZE; i++) {
        A[i] = i;
        B[i] = LIST_SIZE - i;
    }

    // Load the kernel source code into the array source_str
    FILE *fp;
    char *source_str;
    size_t source_size;

    fp = fopen("opencl_kernel.cl", "r");
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
    source_str = (char*)malloc(MAX_SOURCE_SIZE);
    source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
    fclose( fp );

    // Get platform and device information
    cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
            &device_id, &ret_num_devices);

    // Create an OpenCL context
    cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

    // Create a command queue
    cl_command_queue command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

    // Create memory buffers on the device for each vector
    cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(int), NULL, &ret);
    cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(int), NULL, &ret);
    cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            LIST_SIZE * sizeof(int), NULL, &ret);

    // Copy the lists A and B to their respective memory buffers
    ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
    ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

    // Create a program from the kernel source
    cl_program program = clCreateProgramWithSource(context, 1,
            (const char **)&source_str, (const size_t *)&source_size, &ret);

    // Build the program
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel
    cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

    // Execute the OpenCL kernel on the list
    size_t global_item_size = LIST_SIZE; // Process the entire lists
    size_t local_item_size = 64; // Divide work items into groups of 64
    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
            &global_item_size, &local_item_size, 0, NULL, NULL);

    // Read the memory buffer C on the device to the local variable C
    int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
    ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(int), C, 0, NULL, NULL);

    // Display the result to the screen
    for(i = 0; i < LIST_SIZE; i++)
        printf("%d + %d = %d\n", A[i], B[i], C[i]);

    // Clean up
    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(a_mem_obj);    ret = clReleaseMemObject(b_mem_obj);    ret = clReleaseMemObject(c_mem_obj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);
    free(A);    free(B);    free(C);

    return 0;
}
```

# [Step I] Getting started

- Simple OpenCL host and OpenCL vectorAdd kernel
  - compile the OpenCL host

    1. locate /CL/opencl command를 통해 (locate 패키지가 설치되어 있지 않은 경우, sudo apt-get install locate) opencl library의 경로 추적

```
yongseungyu@Hanyang-yongseungyu:~/Workspace_YongseungYu/Works_ETC$ locate /CL/opencl
/home/yongseungyu/NVIDIA_GPU_Computing_SDK/OpenCL/common/inc/CL/opencl.h
```

    2. gcc로 컴파일, OpenCL library를 링킹하고, 전술한 opencl library의 header 경로를 -I 로 include

```
gcc opencl_host.c -o opencl_example.exe -l OpenCL -I/home/yongseungyu/NVIDIA_GPU_Computing_SDK/OpenCL/commin/inc/
```

  - 프로그램 실행 예상 결과 (중간 실행 결과)

```
800 + 224 = 1024
801 + 223 = 1024
802 + 222 = 1024
803 + 221 = 1024
804 + 220 = 1024
805 + 219 = 1024
806 + 218 = 1024
807 + 217 = 1024
808 + 216 = 1024
809 + 215 = 1024
810 + 214 = 1024
811 + 213 = 1024
812 + 212 = 1024
813 + 211 = 1024
814 + 210 = 1024
815 + 209 = 1024
816 + 208 = 1024
817 + 207 = 1024
818 + 206 = 1024
819 + 205 = 1024
820 + 204 = 1024
```

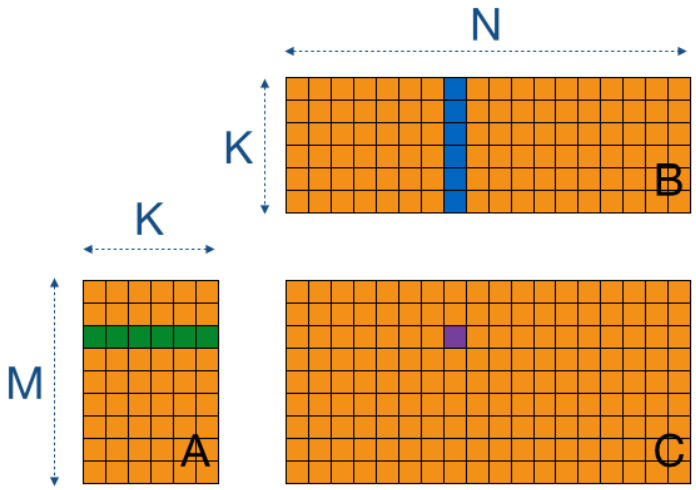
  - compile command 및 우측의 실행 결과를 screenshot으로 제출

# [Step II] Naïve SGEMM

- GEMM: GEneral Matrix Multiplication, 범용 행렬 곱셈
  - D = alpha * A x B + beta * C

    (alpha, beta: scalar, A, B, C, D: matrix)
  - 단순화하기 위해 이하에서는 alpha = 1, beta = 0

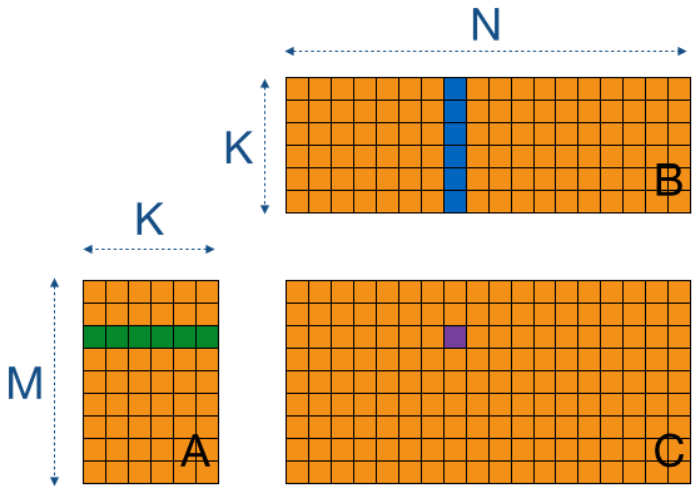
    (C = A x B)
  - 입력 matrix의 dimension:
    - Matrix A: M by K, matrix B: K by N -> result matrix C: M by N
    - C(MN) = A(MK) x B(KN) -> 일반적으로 array: [M, N, K]로 나타냄
- SGEMM: Single-precision GEMM
  - 4-byte float GEMM
  - Float type의 data로 이루어진 두 입력 행렬에 대한 GEMM



# [Step II] Naïve SGEMM

- [Step II]: Naive SGEMM, uniform dims
  - Grid의 work size = {64, 64}로 설정
  - GEMM dimension: [M, N, K] = [64, 64, 64]로 설정
  - 첨부된 커널의 /* fill */ 만을 채워서 완성
  - kernel file name: matmul_HW2.cl,
    kernel name: matmul_HW2
  - 첨부된 get_time() 함수를 이용, time check

```
예시:
double start_time, end_time;
start_time = get_time();
err = clEnqueueNDRangeKernel( ... );
end_time = get_time();
printf("Execution time:" %lf sec ₩n", end_time – start_time);
```



# [Step II] Naïve SGEMM

- [Step II]: Naive SGEMM, uniform dims
  - 실행 결과 예시 (결과가 PASSED 여야 함, FAILED이면 계산 결과 오류)

```
// HW_2
__kernel void matmul_HW2(
        const int N,
        const __global float *A,
        const __global float *B,
        __global float *C)
{
    int tidx = get_global_id(0); // i
    int tidy = get_global_id(1); // j

    if (tidx < N && tidy < N)
    {
        float Csub = 0.0f;
        for(int k = 0; k < N; k++) // k
            Csub += A[/* fill here */] * B[/* fill here */];

        C[/* fill here */] = Csub;
    }
}

```

```
cass@cass-gpu-server:~/aiplatform_course_HW/HW_2$ ./opencl_host_HW2.exe
Performance: 0.000020027 sec, result: PASSED
```

# [Step III] Naïve SGEMM (cont'd)

- [Step III]: Naive SGEMM, not uniform dims
  - Grid의 work size = {GEMM_M, GEMM_N} 로 설정
  - GEMM dimension: [M, N, K] = [2048, 1536, 1024]로 설정
  - Step II의 kernel을 기반으로 확장하여, square-matrix가 아닌 matrix에 대한 GEMM을 수행
  - 첨부된 커널의 /* fill */ 만을 채워서 완성
  - kernel file name: matmul_HW3.cl, kernel name: matmul_HW3
  - 첨부된 get_time() 함수를 이용, time check

# [Step III] Naïve SGEMM (cont'd)

- [Step III]: Naive SGEMM, not uniform dims
  - 실행 결과 예시 (결과가 PASSED 여야 함, FAILED이면 계산 결과 오류)

```
1 // HW_3
2 __kernel void matmul_HW3(
3         const int M,
4         const int N,
5         const int K,
6         const __global float *A,
7         const __global float *B,
8         __global float *C)
9 {
10     int tidx = get_global_id(0); // i
11     int tidy = get_global_id(1); // j
12
13     if (tidx < M && tidy < N)
14     {
15         /* fill here */
16     }
17 }
18
```

```
cass@cass-gpu-server:~/aiplatform_course_HW/HW_3$ gcc -O3 opencl_host_HW3.c -o opencl_host_HW3.exe -lOpenCL -I/usr/local/cuda-10.1/include/
cass@cass-gpu-server:~/aiplatform_course_HW/HW_3$ ./opencl_host_HW3.exe
HW3: Naive SGEMM, not uniform dims
Performance: 0.000158072 sec, result: PASSED
```

# [Step IV] SGEMM with loop unrolling

- [Step IV]: Loop unrolling
  - loop unrolling:

    임의의 loop에 대해 순차적인 몇 개의 index를 직접 inline 하는 기법

    + loop body의 work가 많아지므로 loop의 index 증가 및 조건 체크 부분의 overhead를 줄일 수 있다

    + loop 내에 dependency가 없는 경우, 병렬 처리를 할 수 있다

    - 프로그램 크기가 증가한다

    - unrolling 과정에서 register의 사용량이 많아지므로, 이로부터 performance 감소가 야기될 수 있다

**Normal**

```
for ( int i = 0; i < 100; ++i )
{
    delete( i );
}
```

**Loop Unrolling**

```
for ( int i = 0; i < 100; i += 5 )
{
    delete( i );
    delete( i + 1 );
    delete( i + 2 );
    delete( i + 3 );
    delete( i + 4 );
}
```

# [Step IV] SGEMM with loop unrolling

- [Step IV]: Loop unrolling
  - Grid의 work size, GEMM dimension은 [Step III]와 동일
  - Step III의 kernel을 기반으로 확장하여, kernel의 inner-most loop에 loop unrolling 기법을 적용
  - 첨부된 커널의 **/* fill */** 만을 채워서 완성
  - **kernel file name**: matmul_HW4.cl, **kernel name**: matmul_HW4
  - 첨부된 **get_time() 함수**를 이용, time check

# [Step IV] SGEMM with loop unrolling

- [Step IV]: Loop unrolling
  - 실행 결과 예시 (결과가 PASSED 여야 함, FAILED이면 계산 결과 오류)
  - Note: 실행 환경 및 target GPU에 따라 performance gain이 적거나 없을 수 있음

```
// HW_4
__kernel void matmul_HW4(
        const int M,
        const int N,
        const int K,
        const __global float *A,
        const __global float *B,
        __global float *C)
{
    int tidx = get_global_id(0); // i
    int tidy = get_global_id(1); // j

    if (tidx < M && tidy < N)
    {
        float Csub = 0.0f;

        for(int k = 0; k < K; k += 8) // k
        {
            if (k < K)
            {
                /* fill here */
            }
        }

        C[tidx * N + tidy] = Csub;
    }
}
```

```
cass@cass-gpu-server:~/aiplatform_course_HW/HW_4$ gcc -O3 opencl_host_HW4.c -o opencl_host_HW4.exe -lOpenCL -I/usr/local/cuda-10.1/include/
cass@cass-gpu-server:~/aiplatform_course_HW/HW_4$ ./opencl_host_HW4.exe
HW4: SGEMM with loop unrolling
Performance: 0.000100851 sec, result: PASSED
```

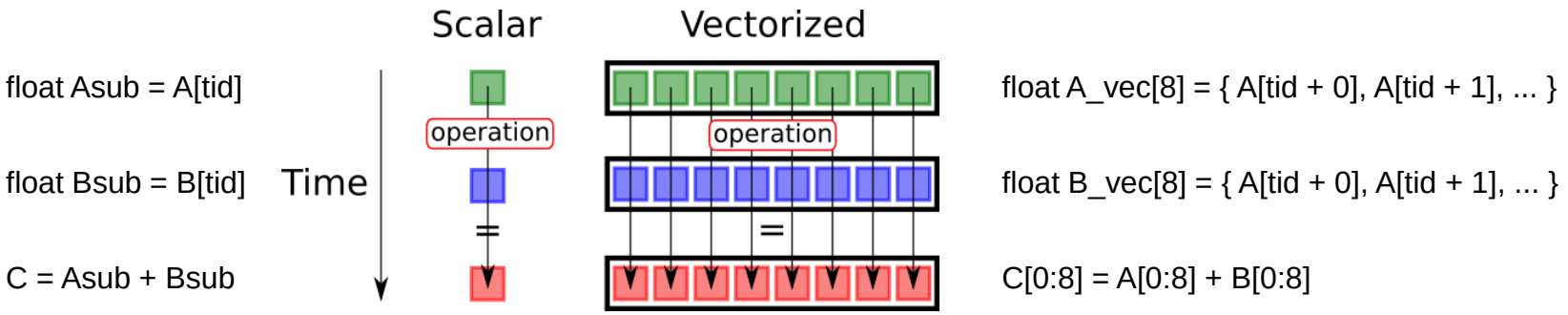
# [Step V] SGEMM with vectorization

- [Step V]: Vectorization
  - Vectorization in OpenCL

    하나의 work-item이 처리하는 scalar variable 몇 개를 묶어 실행하는 기법

    병렬 처리가 가능한 영역의 problem에 대해 scalable optimization 가능

    Hardware(ALU)의 vectorization 지원 여부 및 vectorization width에 따라 performance gain이 제한될 수 있음



# [Step V] SGEMM with vectorization

- [Step V]: Vectorization
  - Grid의 work size, GEMM dimension은 [Step III]와 동일
  - Step III의 kernel을 기반으로 확장하여, parameter A에 대해 vectorization 기법을 적용
  - 첨부된 커널의 **/* fill */** 만을 채워서 완성
  - **kernel file name**: matmul_HW5.cl, **kernel name**: matmul_HW5
  - 첨부된 **get_time() 함수**를 이용, time check

# [Step V] SGEMM with vectorization

- [Step V]: Vectorization
  - 실행 결과 예시 (결과가 PASSED 여야 함, FAILED이면 계산 결과 오류)
  - Note: 실행 환경 및 target GPU에 따라 performance gain이 적거나 없을 수 있음

```
// HW_5
__kernel void matmul_HW5(
        const int M,
        const int N,
        const int K,
        const __global float *A,
        const __global float *B,
        __global float *C)
{
    int tidx = get_global_id(1); // i
    int tidy = get_global_id(0); // j

    int vlen = 4;

    if (tidx < M && tidy < N)
    {
        float Csub = 0.0f;

        for(int k = 0; k < K; k++) // k
        {
            if (k < K)
            {
                float /* fill here */

                for (int l = 0; l < vlen; ++l)
                {
                    /* fill here */
                }
            }
        }
        C[tidx * N + tidy] = Csub;
    }
}

```

```
cass@cass-gpu-server:~/aiplatform_course_HW/HW_5$ gcc -O3 opencl_host_HW5.c -o opencl_host_HW5.exe -lOpenCL -I/usr/local/cuda-10.1/include/
cass@cass-gpu-server:~/aiplatform_course_HW/HW_5$ ./opencl_host_HW5.exe
HW5: SGEMM with vectorization
Performance: 0.000097990 sec, result: PASSED
```