

과제 #4

2024-27759 장효형

(20점) GPU 정보 확인하기 보고서에 다음 질문들에 대한 답을 서술하라.

(a) (4점) 다음 커맨드들의 의미와 실행 결과를 답하라. 결과가 너무 길면 적당히 앞부분만 잘라서 첨부하라.

1) `srn --partition=class1 --gres=gpu:4`: 공통 커맨드. slurm으로 실행하되, class1 partition에서 gpu를 4 개를 잡고 이후 커맨드를 실행한다.

2) `nvidia-smi`: GPU ID, 사용량, 현재온도 등의 정보를 알려주는 커맨드

```
shpc125@e1ogin3:~/hw4$ srun --partition=class1 --gres=gpu:4 nvidia-smi
srun: job 1002593 queued and waiting for resources
srun: job 1002593 has been allocated resources
Wed Nov 13 11:59:39 2024

+-----+
| NVIDIA-SMI 520.61.05      Driver Version: 520.61.05      CUDA Version: 11.8      |
+-----+
| GPU   Name           Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan   Temp   Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|=====+=====+
| 0     NVIDIA TITAN RTX      On      | 00000000:18:00.0 Off |          0M / 24576M |      0%      Default |
| 41%    33C    P8      14W / 280W | 0MiB / 24576MiB |             MIG M.   |
|-----+-----+
| 1     NVIDIA TITAN RTX      On      | 00000000:3B:00.0 Off |          0M / 24576M |      0%      Default |
| 41%    30C    P8      15W / 280W | 0MiB / 24576MiB |             N/A      |
|-----+-----+
| 2     NVIDIA TITAN RTX      On      | 00000000:86:00.0 Off |          0M / 24576M |      0%      Default |
| 41%    30C    P8      15W / 280W | 0MiB / 24576MiB |             N/A      |
|-----+-----+
| 3     NVIDIA TITAN RTX      On      | 00000000:AF:00.0 Off |          0M / 24576M |      0%      Default |
| 41%    28C    P8      11W / 280W | 0MiB / 24576MiB |             N/A      |
|-----+-----+

+-----+
| Processes: |
| GPU   GI   CI        PID   Type   Process name                      GPU Memory |
| ID   ID   ID              |              |                     Usage          |
|=====+=====+
| No running processes found |
+-----+
```

3) `nvidia-smi -q: --query`. More detailed information is available. 캡처된 사진에서는 제품명, 브랜드, accounting mode buffer size, GPU part number, serial number 등이 표기되어 있다.

```

shpc125@login3:~/hw4$ srun --partition=class1 --gres=gpu:4 nvidia-smi -q
srun: job 1002599 queued and waiting for resources
srun: job 1002599 has been allocated resources

=====NVSMI LOG=====

Timestamp                : Wed Nov 13 12:00:25 2024
Driver Version           : 520.61.05
CUDA Version             : 11.8

Attached GPUs            : 4
GPU 00000000:18:00.0
  Product Name           : NVIDIA TITAN RTX
  Product Brand          : Titan
  Product Architecture   : Turing
  Display Mode           : Disabled
  Display Active         : Disabled
  Persistence Mode       : Enabled
  MIG Mode
    Current              : N/A
    Pending              : N/A
  Accounting Mode        : Disabled
  Accounting Mode Buffer Size : 4000
  Driver Model
    Current              : N/A
    Pending              : N/A
  Serial Number          : 1324419054357
  GPU UUID               : GPU-441d4c71-3298-6ed8-ce42-699422c61d4d
  Minor Number           : 0
  VBIOS Version          : 90.02.2E.00.0C
  MultiGPU Board         : No
  Board ID               : 0x1800
  GPU Part Number        : 900-1G150-2500-000
  Module ID              : 0
  Inforom Version
    Image Version        : G001.0000.02.04
    OEM Object           : 1.1
    ECC Object           : N/A
    Power Management Object : N/A
  GPU Operation Mode
    Current              : N/A
    Pending              : N/A
  GSP Firmware Version   : N/A
  GPU Virtualization Mode
    Virtualization Mode  : None

```

4) clinfo. openCL 플랫폼 정보를 출력하는 커맨드. 플랫폼 버전, 디바이스 이름, 디바이스 버전, max compute units 등이 적혀 있다.

```

shpc125@elogin3:~/hw4$ srun --partition=class1 --gres=gpu:4 clinfo
srun: job 1002617 queued and waiting for resources
srun: job 1002617 has been allocated resources
Number of platforms                                1
Platform Name                                       NVIDIA CUDA
Platform Vendor                                     NVIDIA Corporation
Platform Version                                    OpenCL 3.0 CUDA 11.8.88
Platform Profile                                     FULL_PROFILE
Platform Extensions                                cl_khr_global_int32_base_atomics cl_khr_global
32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_f
cl_khr_3d_image_writes cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing cl_nv_compiler
ions cl_nv_device_attribute_query cl_nv_pragma_unroll cl_nv_copy_opts cl_nv_create_buffer cl_khr
64_base_atomics cl_khr_int64_extended_atomics cl_khr_device_uuid cl_khr_pci_bus_info cl_khr_exte
_semaphore cl_khr_external_memory cl_khr_external_semaphore_opaque_fd cl_khr_external_memory_opa
fd
Platform Host timer resolution                      0ns
Platform Extensions function suffix                 NV

Platform Name                                       NVIDIA CUDA
Number of devices                                   4
Device Name                                         NVIDIA TITAN RTX
Device Vendor                                       NVIDIA Corporation
Device Vendor ID                                   0x10de
Device Version                                      OpenCL 3.0 CUDA
Driver Version                                      520.61.05
Device OpenCL C Version                            OpenCL C 1.2
Device Type                                         GPU
Device Topology (NV)                               PCI-E, 18:00.0
Device Profile                                       FULL_PROFILE
Device Available                                    Yes
Compiler Available                                  Yes
Linker Available                                    Yes
Max compute units                                   72
Max clock frequency                                 1770MHz
Compute Capability (NV)                             7.5
Device Partition                                    (core)
  Max number of sub-devices                          1
  Supported partition types                           None
  Supported affinity domains                           (n/a)
Max work item dimensions                           3
Max work item sizes                                 1024x1024x64
Max work group size                                 1024
Preferred work group size multiple                  32
Warp size (NV)                                      32
Max sub-groups per work group                       0

```

(b) (4점) 실습 서버의 계산 노드에 장착된 GPU의 모델명과 노드당 장착된 GPU의 개수를 답하라.

-> NVIDIA TITAN RTX, 4개

(c) (4점) 실습 서버의 계산 노드에 장착된 GPU 하나의 메모리 크기를 MiB 단위로 답하라.
(MiB: 2^{20} bytes, MB: 10^6 bytes)

-> 24576MiB

(d) (4점) 실습 서버의 계산 노드에 장착된 GPU의 maximum power limit(W)과 maximum SM clock speed (Mhz)를 답하라.

-> 320W, 2100MHz

(e) (4점) 실습 서버의 계산 노드에 장착된 GPU를 OpenCL 을 이용해 사용할 때 Max work item dimension, Max work item size, Max work group size 를 답하라.

-> 3, 1024 * 1024 * 64, 1024

2 (80점) Matrix Multiplication with OpenCL

보고서에 다음 질문들에 대한 답을 서술하라

- 자신의 병렬화 방식에 대한 설명.

1) Padding

```
size_t local_work_size[2] = {BLOCK_SIZE / REG_SIZE, BLOCK_SIZE};
size_t global_work_size[2] = {
    ((M + BLOCK_SIZE - 1) / BLOCK_SIZE) * BLOCK_SIZE / REG_SIZE,
    ((N + BLOCK_SIZE - 1) / BLOCK_SIZE) * BLOCK_SIZE
};
```

```
for(int k=0; k < BLOCK_SIZE; ++k){
    if ((BLOCK_SIZE * t + k) < K && glbCol < N) {
        Bsub[k][locCol] = B[(BLOCK_SIZE * t + k) * N + glbCol];
    } else {
        Bsub[k][locCol] = 0.0;
    }
}
```

Opencl은 block 단위로 돌아가기 때문에 kernel 안에서 돌아가는 work group의 사이즈가 동일해야 한다. 자르고 남은 것을 별도로 연산하는 방법보다는 zero-padding을 붙여서 완전한 block이 들어갈 수 있게 한다. A도 마찬가지로 할당했다.

2) thread level Parallelism

```
// size_t global_work_size[2] = {M, N};
// size_t local_work_size[2] = {BLOCK_SIZE, BLOCK_SIZE};
```

위는 padding을 적용하지 않은 코드로, A의 행(M), B의 열(N)을 병렬로 처리했다. 이러면 A의 각 열이 블록 단위로 local_work_size[0]에 할당되고, B의 각 열이 local_work_size[1]에 할당되어 thread 병렬적으로 처리된다.

- 뼈대 코드 matmul.c의 각 부분에 대한 설명. matmul initialize, matmul, matmul finalize 함수 각각 에서 사용하는 OpenCL API 및 각 API에 대한 간략한 설명. (API 당 한문장이면 충분).

```

void matmul_initialize(int M, int N, int K) {
    // Get OpenCL platform
    err = clGetPlatformIDs(1, &platform, NULL);
    CHECK_ERROR(err);
    print_platform_info(platform);

    // Get OpenCL device (only 1)
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    CHECK_ERROR(err);
    print_device_info(device);

    // Create OpenCL context
    context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
    CHECK_ERROR(err);

    // Create OpenCL command queue
    queue = clCreateCommandQueue(context, device, 0, &err);
    CHECK_ERROR(err);

    // Compile program from "kernel.cl"
    program = create_and_build_program_with_source(context, device, "kernel.cl");

    // Extract kernel from compiled program
    kernel = clCreateKernel(program, "sgemm", &err);
    CHECK_ERROR(err);

    // Create GPU buffers
    a_d = clCreateBuffer(context, CL_MEM_READ_WRITE, M * K * sizeof(float), NULL, &err);
    CHECK_ERROR(err);
    b_d = clCreateBuffer(context, CL_MEM_READ_WRITE, K * N * sizeof(float), NULL, &err);
    CHECK_ERROR(err);
    c_d = clCreateBuffer(context, CL_MEM_READ_WRITE, M * N * sizeof(float), NULL, &err);
    CHECK_ERROR(err);
}

```

- 1) clGetPlatformIDs: platform(GPU 관리 소프트웨어)의 종류를 가져와 platform에 저장
- 2) clGetDeviceIDs: 1개의 GPU를 불러오기
- 3) clCreateContext: 공유 리소스를 관리하는 공간(컨텍스트)를 만들.
- 4) clCreateCommandQueue: command queue를 만들. 명령어를 받아서 device로 보냄.
- 5) create_and_build_program_with_source: kernel.cl 코드를 받아서 빌드
- 6) clCreateKernel: 빌드된 프로그램에서 커널(핵심 연산 코드) 생성
- 7) clCreateBuffer: 버퍼(데이터 저장소) 생성

```

void matmul(const float *A, const float *B, float *C, int M, int N, int K) {

    int BLOCK_SIZE = 16;
    int REG_SIZE = 8;
    int VEC_WIDTH = 4;
    // A, B 데이터를 GPU로 복사
    err = clEnqueueWriteBuffer(queue, a_d, CL_TRUE, 0, M * K * sizeof(float), A, 0, NULL, NULL);
    CHECK_ERROR(err);
    err = clEnqueueWriteBuffer(queue, b_d, CL_TRUE, 0, K * N * sizeof(float), B, 0, NULL, NULL);
    CHECK_ERROR(err);

    // 커널 인자 설정
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_d);
    CHECK_ERROR(err);
    err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_d);
    CHECK_ERROR(err);
    err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_d);
    CHECK_ERROR(err);
    err = clSetKernelArg(kernel, 3, sizeof(int), &M);
    CHECK_ERROR(err);
    err = clSetKernelArg(kernel, 4, sizeof(int), &N);
    CHECK_ERROR(err);
    err = clSetKernelArg(kernel, 5, sizeof(int), &K);
    CHECK_ERROR(err);

    size_t local_work_size[2] = {BLOCK_SIZE / REG_SIZE, BLOCK_SIZE};
    size_t global_work_size[2] = {
        ((M + BLOCK_SIZE - 1) / BLOCK_SIZE) * BLOCK_SIZE / REG_SIZE,
        ((N + BLOCK_SIZE - 1) / BLOCK_SIZE) * BLOCK_SIZE
    };
    // printf("%zu, %zu, %zu, %zu\n", global_work_size[0], global_work_size[1], local_work_size[0], local_work_size[1]);

    err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global_work_size, local_work_size, 0, NULL, NULL);
    CHECK_ERROR(err);

    // 결과 읽기
    err = clEnqueueReadBuffer(queue, c_d, CL_TRUE, 0, M * N * sizeof(float), C, 0, NULL, NULL);
    CHECK_ERROR(err);
}

```

8) clEnqueueWriteBuffer: CPU memory에서 GPU buffer로 데이터 전송.

9) clSetKernelArg: Kernel argument 세팅

10) clEnqueueNDRangeKernel: 디바이스에서 커널을 실행할 수 있게 예약

11) clEnqueueReadBuffer: GPU -> CPU로 데이터 전송

```

void matmul_finalize() {

    err = clReleaseMemObject(a_d);
    CHECK_ERROR(err);
    err = clReleaseMemObject(b_d);
    CHECK_ERROR(err);
    err = clReleaseMemObject(c_d);
    CHECK_ERROR(err);

    err = clReleaseKernel(kernel);
    CHECK_ERROR(err);
    err = clReleaseProgram(program);
    CHECK_ERROR(err);

    err = clReleaseCommandQueue(queue);
    CHECK_ERROR(err);
    err = clReleaseContext(context);
    CHECK_ERROR(err);
}

```

- 12) clReleaseMemObject: 메모리 object 해제
- 13) clReleaseKernel: kernel object 해제
- 14) clReleaseProgram: 빌드된 프로그램 해제
- 15) clReleaseCommandQueue: command queue 해제
- 16) clReleaseContext: 마지막으로 context 해제.

• 자신이 적용한 코드 최적화 방식을 분류하고, 각각에 대한 성능 실험 결과.

1) Sub matrix

```
__local float Asub[BLOCK_SIZE][BLOCK_SIZE];
__local float Bsub[BLOCK_SIZE][BLOCK_SIZE];
```

Pthread 등에서 이용된 tiling 기법을 도입했다. GPU off memory에 등록된 행렬들을 쪼개 GPU on memory에 값을 싣기 위해서 sub matrix를 도입하는 형태로 변형되어 이용되었다. BLOCK_SIZE는 clGetDeviceInfo with CL_DEVICE_LOCAL_MEM_SIZE로 찾아본 결과 32가 최대였다. 그래서 아래 항목과 함께 여러 조합을 실험해 보았다. (결과는 3)에 제시)

2) Increase num_operations per thread

```
float temp[C_SIZE];
for (int j=0; j<C_SIZE; ++j){
    temp[j] = 0.0;
}

const int NUM_TILES = (K + BLOCK_SIZE - 1) / BLOCK_SIZE;

for (int t = 0; t < NUM_TILES; ++t){

    for(int j=0; j<C_SIZE; ++j){

        // 웬지 모르겠는데 2D 행렬로 하면 더 빠름
        if ((glbRow + j * glbWSRow) < M && (BLOCK_SIZE * t + locCol) < K) {
            Asub[locRow + j * locBSRow][locCol] = A[(glbRow + j * glbWSRow) * K + (BLOCK_SIZE * t + locCol)];
        } else {
            Asub[locRow + j * locBSRow][locCol] = 0.0;
        }
    }
}
```

기존 Asub[locRow][locCol] = A[glbRow * K + BLOCK_SIZE * t * locCol]일 경우보다 스레드 당 연산량이 증가해 스레드 관리에 드는 latency를 절감할 수 있었다. 하나의 스레드 당 몇 개의 A 행 연산을 수행할지는 C_SIZE로 통제할 수 있으며, 위와 함께 여러 조합을 실험해 보았다(결과는 3)에 제시)

3) 결과

M=N=K=4096인 행렬을 ./run_performance.sh를 통해 실험해 본 후 GFLOPS 결과를 기록했다. BLOCK_SIZE = 32, C_SIZE = 32일 때 가장 좋은 성능을 냈다.

BLOCK_SIZE	C_SIZE	GFLOPS
64	64	1252.88
64	32	1117.33
64	16	725.86
32	32	1654.43
32	16	1142.24
32	8	719.25
16	16	963.82
16	8	1367.76
16	4	949.64
8	8	484.03
8	4	650.22
8	2	790.33