JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 21, No. 2, pp. 373-380, Feb. 2020

Check for updates

효과적인 CUDA 디버깅을 위한 CUDA 커널 분리 소프트웨어 도구

양 정 아 · 박 태 정*
덕성여자대학교 IT미디어공학과 석사과정
덕성여자대학교 IT미디어공학과 교수

Software Tool to Separate CUDA Kernel for Effective CUDA Debugging

Jung Ah Yang · Taejung Park*
Master's Course, Department of IT and Media Engineering, Duksung Women's University, Seoul 01369, Korea
Professor, Department of IT and Media Engineering, Duksung Women's University, Seoul 01369, Korea

[요 약]

인공지능, 빅데이터 처리 등 다양한 분야에서 GPU를 이용한 병렬 처리 방법이 보편화 되면서 GPU 기반 병렬 처리에 대한 중요성이 점차 증가하고 있다. 그러나 GPU 기반 병렬 처리 소프트웨어의 개발 과정에서 보편적으로 널리 사용되는 디버거(Nvidia NSight)는 병렬 처리 기본 단위인 CUDA 커널(kernel)이 여러 개 존재할 때 내부적인 문제로 인하여 효율적인 디버깅을 수행할 수 없는 상황이 빈번하게 발생한다. 본 논문에서는 이러한 문제를 해결하기 위해 실행 중인 전체 CUDA 소스 코드에서 디버깅이 필요한 커널만을 추출하고 실제 실행 상황에서 해당 커널에 입력되는 입력값을 추출해서 단위 테스트 수행을 자동화하는 소프트웨어 도구의 개발을 제안한다. 제안하는 도구는 개발 중인 CUDA 병렬 코드의 실행 흐름을 그대로 유지한 채로 디버깅이 필요한 특정 커널에 초점을 맞춰 디버깅을 수행할 수 있는 코드를 자동으로 생성한다. 본 논문에서는 제안하는 소프트웨어 도구의 구조와 원리를 소개하고 실제 NSight만으로 디버깅이 불가능한 사례에 적용하여 성공적으로 디버깅을 수행하는 사례를 소개한다.

[Abstract]

With the advent of artificial intelligence and big data analysis, the parallel processing technique based on the GPU has been playing a critical role in these fields. Unfortunately, the common debugger, NSight from Nvidia often fails to work when there are lots of CUDA kernels, which are basic parallel execution units due to internal limitations of NSight. In this paper, we present a software tool which can extract the kernel in the problem and its input information in the CUDA source code without affecting the original parallel execution of CUDA codes. We also introduce a case study where our software tool has been applied to the actual software development process for GPU-based parallel processing to prove the validity of the proposed unit tester.

색인어 : CUDA, CUDA 커널 분리, 디버깅, NSight, 병렬 처리
Key word : CUDA, CUDA Kernel Separation, Debugging, NSight, Parallel Processing

http://dx.doi.org/10.9728/dcs.2020.21.2.373

Received 19 December 2019; **Revised** 31 January 2020
Accepted 25 February 2020

***Corresponding Author; Taejung Park**
Tel: +82-2-901-8339
E-mail: tjpark@duksung.ac.kr

Ⅰ. 서 론

최근 병렬 하드웨어의 성능 개선으로 그래픽스, 기계학습, 빅데이터 분석 등 실시간 처리나 대규모 데이터를 이용하는 기술들이 급속히 발전하고 있다. 특히, 그래픽 처리 장치(GPU; graphics processing unit)의 하드웨어 발전과 NVIDIA사의 GPGPU(general-purpose computing on graphics processing units) 기술인 CUDA(compute unified device architecture)의 개선으로 대규모 병렬처리 알고리즘 수행 능력이 향상되어 대규모 데이터의 고속 처리가 가능하게 되었다[1].

CUDA는 GPU 프로그래밍을 가능하게 하는 기술로 C/C++ 언어를 기초로 하는 확장 명령들로 구성된다. 병렬 처리를 위해 CUDA에서는 병렬 처리의 기본 단위라고 할 수 있는 함수 형태, 즉, 커널을 구현한 후 GPU 내에서 다중 스레드를 통해 실행한다. 특히 CUDA가 제공하는 병렬 처리 모델은 여러 스레드들이 단일 명령을 실행하는 형태(SIMT; Single Instruction, Multiple Threads)로 작동한다. 실제 구현을 위해 NVIDIA에서 제공하는 CUDA Toolkit은 Windows OS 환경에서의 개발을 위해 Microsoft사의 통합 개발 환경(IDE; intergrated development environment)인 Visual Studio를 지원한다. 특히, CUDA 프로그램 디버깅 및 프로파일링 도구인 Nsight[2]를 제공함으로써 IDE에서의 디버깅을 지원한다. Nsight에서는 일반적인 IDE 기반 디버거에서처럼 CUDA 코드에서 중단점을 지정하고 디버깅하면 사용자가 현재 CPU 코드 및 GPU 코드를 디버깅할 수 있도록 중단 지점마다 변수에 저장된 값, GPU 하드웨어 실행 단위인 warp[3]의 상태, 스레드 상태 등을 Visual Studio에서 쉽게 확인할 수 있다. 일반적인 병렬 알고리즘에서는 커널 자체는 병렬적으로 실행되지만 이러한 커널들이 여러 개 순차적으로 실행되며 따라서 실행 도중의 특정한 커널에 대한 디버깅이 실행되는 것이 일반적이다. 이러한 일반적인 CUDA 커널의 작동 양상은 [4]의 그림 2에서 확인할 수 있다. 그러나 이러한 일반적인 상황에서 Nsight의 작동이 느려지거나 작동이 멈춰 버리는 상황이 자주 발생한다.

본 연구에서는 Nsight를 사용한 CUDA 프로그램 디버깅 수행 시 실질적으로 발생하는 이러한 문제에 대해 논의하고 이 문제에 대한 해결책으로서 원하는 커널에 대한 CUDA 커널 단위 테스트 코드를 자동으로 생성하여 커널 별로 디버깅을 수행하는 실용적인 소프트웨어 도구를 제안한다. 또한 본 논문에서는 제안하는 방법을 실제 개발 중인 CUDA 코드에 적용한 사례를 소개하고 그 의의를 논의한다.

Ⅱ. CUDA 코드 디버깅 관련 문제

디버깅은 개발자가 프로그램의 정확성 또는 논리적인 오류를 검출하여 제거하는 과정이다. 실제로 문법적인 오류는 비교적 쉽게 검출해낼 수 있다. 그러나 논리적인 오류의 경우 디버거를 사용해서 프로그램 수행 상황을 보고 개발자가 판단하여

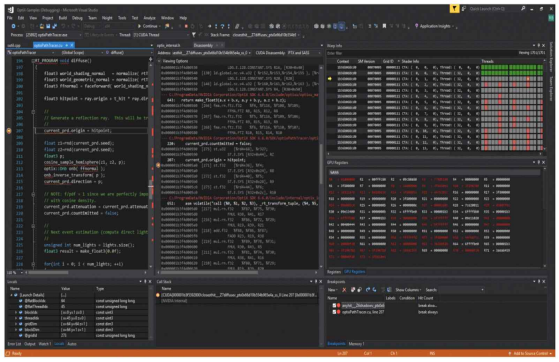

그림 1. Nsight Visual Studio 실행 화면[2]
Fig. 1. Nsight Visual Studio executing window [2]

논리적인 오류를 찾아내는 것이기 때문에 디버거를 사용하지 않는다면 에러 검출이 어렵다. 특히 다수의 스레드가 동시에 연산을 수행하는 병렬 프로그래밍의 경우 개발자가 수천 개의 스레드 상태를 전부 파악하기 어려워서 디버거의 역할이 중요하다.

일반적인 개발 과정에서와 마찬가지로 병렬 처리에서도 개발자가 디버거를 통해 스레드가 현재 어떤 명령을 수행 중인지, 연산을 처리한 결과는 무엇인지, 스레드의 상태는 어떠한지를 판단할 수 있어야 효율적으로 개발을 수행할 수 있다.

2-1 CUDA 디버깅 관련 기존 연구

이러한 디버깅의 한계는 본질적으로 단일 코어 기반의 순차적 프로그래밍과는 매우 다른 병렬 프로그래밍의 개발을 매우 어렵게 하며 그동안 GPU 기반 병렬 처리를 지원할 수 있는 몇 가지 방안들이 연구되었다. GPU에서 동시의 여러 데이터에 접근하기 위해 공유메모리를 각 warp마다 일정 memory bank로 나누는데, 각 스레드가 동시에 다른 bank에 접근할 때 발생하는 문제인 bank conflict와 공유 자원에 대해 동시 작업을 할 때 의도치 않은 결과를 가져오는 문제가 발생하는 race condition 문제 해결을 위해 CUDA 코드를 자동 분석하여 보고하는 방법이 제안되었다[5]. 수동으로 사용자가 그래픽 프로그래밍 전문가라는 가정하에 Microsoft Visual Studio를 사용하지 않는 디버깅 과정을 돕는 코딩 기술과 적용에 대해 제안된 바 있다[6]. 병렬 처리가 실시간 애플리케이션에 응용되는 경우가 많고 출력되는 데이터의 크기가 크고 복잡하여 발생하는 디버깅 문제를 해결하기 위하여 게임 물리 엔진을 이용하여 고속 실시간 병렬 처리 코드의 시각적 디버깅 방식이 제안된 바 있다[7]. 그러나 이 방식은 그 적용 분야가 실시간 애플리케이션이라는 한계가 있고 앞서 논의한 Nsight의 본질적인 한계를 극복하지 못하는 문제가 있다.

본 연구에서는 CUDA 개발 실무에서 Nsight의 본질적인 한계를 우회할 수 있는 실용적인 방안을 제안한다.

2-2 사용자 관점에서 디버깅의 한계

일반적으로 CUDA 코드 개발 과정에서는 Nsight를 사용해서 디버깅을 수행한다. 병렬 코드의 특성상, CUDA C 코드에서 커널 호출 시 할당한 블록 크기에 따라 사용하는 스레드 수가 달라지며 호출 시에는 동시에 스레드가 명령을 수행하더라도 분기나 연산 처리에 따라 스레드 별 실행하는 명령 지점이 달라질 수 있다. 따라서 CPU에서 수행하는 명령을 포함하여 GPU에서 사용하기 위해 호출된 CUDA 커널이 잦은 분기문이나 복잡한 연산을 포함하게 되면, 개발자가 Nsight를 사용하여 CUDA C 코드를 디버깅할 때 스레드별 상태나 변수에 저장된 값이 어떠한 과정을 통해 변경되었는지 구분하기 어려워진다.

CUDA 코드의 복잡성이 증가함으로써 나타나는 또 다른 문제는 Nsight가 CUDA 코드에서 전형적으로 발생하는 에러(메모리 포인터 관련 문제 등)가 방대한 코드 중에 어떠한 곳에서 발생했는지 파악할 수 있는 유용한 정보를 제공하지 못한다는 점이다.

2-3 디버깅 수행 중단 문제

또 다른 문제 중 하나는 복잡한 연산을 하는 커널들을 여러 개가 순차적으로 호출되는 상황의 경우 Nsight의 내부적 문제 또는 한계(예를 들어 전체 코드 단위의 실행을 파악하기 위한 메모리 사용 한계 등으로 추정)로 인해 Nsight가 느려지거나 이유 없이 중단되는 경우가 빈번하게 발생 된다는 사실이다. 그림 2에서는 실제로 복잡한 연산을 수행하는 다수의 커널을 호출하는 정상적으로 작동하는 CUDA 코드를 Nsight로 디버깅할 때 특정한 에러 정보 없이 디버깅 실행 중에 한 warp 내 스레드의 상태를 보여주는 Lane 창과 각 warp를 표시해주는 Warp 창에 아무런 정보도 보이지 않고 디버깅이 정지되는 문제가 발생하는 상황을 제시한다.

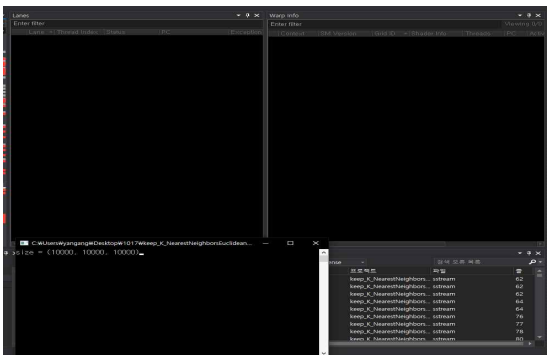

그림 2. 중단된 Nsight Lane, Warp 창
Fig. 2. stopped Nsight Lane, Warp window screen

Ⅲ. 커널 코드 분리를 통한 디버깅

3-1 작동원리

앞서 논의한 CUDA 디버깅에서 빈번하게 발생하는 문제들에 대한 한 가지 실질적인 해결 방법으로서, 본 논문에서는 에러가 존재하는 코드의 범위를 좁히고 Nsight에 가중되는 컴퓨팅 자원 소비로 인한 부담을 줄이기 위해서 전체 병렬 코드의 흐름 속에서 원래 코드에서의 실행 중 환경을 그대로 복사하고 디버깅을 원하는 병렬 커널 단위로 문제의 범위를 축소시킴으로써 Nsight의 연산 부하를 줄이는 방안을 제안한다.

그림 3은 제안하는 디버거의 전체적인 개요를 제시한다. i 번째 커널(그림에서 kernel i)을 디버깅해야 할 필요가 있다면 왼쪽의 전체 CUDA 코드 흐름 중 해당 커널에 공급되는 데이터(주로 큰 크기의 배열, 그림에서 array i-1)을 원래 코드를 실행하는 도중(런타임)에 확보해서 JSON 형식으로 저장한 후 디버깅하고자 하는 커널만을 별도의 CUDA 프로젝트로 이동시키고 JSON 형식으로 저장한 데이터를 다시 GPU로 전송하여 단일 커널을 실행, 디버깅할 수 있는 코드를 구축한다. 이후 Nsight를 이용해서 일반적인 CUDA 디버깅을 수행한다.

이 방법을 사용하면 Nsight 입장에서는 해당 커널 직전까지 여러 커널들의 실행 정보를 저장하지 않고 문제가 있는 커널에 대한 실행 정보만 관리하면 되기 때문에 연산 부하를 크게 줄여 원활한 디버깅을 수행할 수 있다.

3-2 세부 구현 설명

Python은 C/C++등 다른 프로그래밍 언어에 비해 배열 데이터 및 문자열의 용이한 처리가 가능한 특징이 있다. 따라서 본 논문에서 제안하는 방법은 문자열 분석의 용이성과 복잡한 CUDA 배열의 변형을 위해 Python 환경에서 구현하였다.

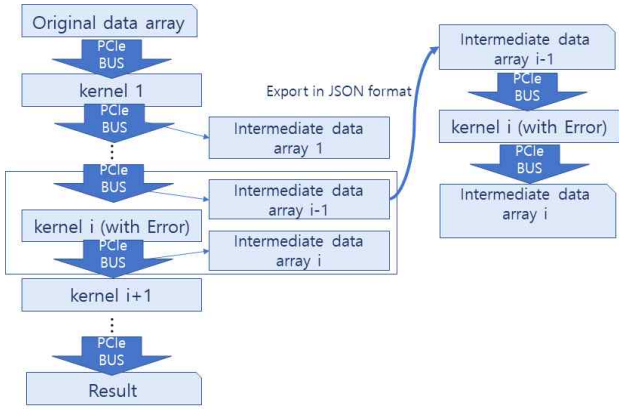

그림 3. 제안하는 커널 단위 디버거 전체 개요
Fig. 3. Overview of the proposed kernel unit debugger

1) CUDA 파일 파싱

파싱(Parsing)은 어떠한 파일에서 원하는 데이터를 특정 패턴이나 순서로 추출하여 가공하는 과정으로 정의된다. 본 연구에서 제안하는 방식을 구현하기 위해서는 Python 환경에서 CUDA 파일을 열어서 문자열 형태로 읽고 원하는 데이터를 추출하기 위해 C/C++과 CUDA 문법의 패턴과 규칙을 적용하여 파싱 과정을 수행할 필요가 있다.

보통 코딩 시, 파일 상단에 사용할 라이브러리 파일 포함 또는 매크로 상수 선언 등 전처리문을 사용하는 명령이 먼저 선언되는 것이 일반적이다. 다른 한편으로 함수의 경우 선언과 정의를 나눠서 구현하는 경우와 함수 정의를 main 함수 내에 모두 포함하는 경우도 있으나, 본 논문에서는 문제의 본질에 집중하기 위해 간단하게 함수 선언과 정의을 분리하지 않고 main 함수 내에서 모든 함수들을 정의하는 상황으로 한정한다. 향후 업그레이드를 통해 이러한 제약들을 하나씩 해결해 나갈 예정이다.

CUDA의 커널은 CPU에서 처리되는 호스트 코드 내에서 "<<< >>>" 기호를 사용하여 GPU 스레드 블록 크기를 지정하고 커널에서 필요로 하는 데이터를 인자로 전달하는 방식으로 호출한다. 구체적은 CUDA 커널 호출 형태는 아래와 같다.

```
kernelname<<<gridsize, blocksize>>>(arg1,arg2,...);
```

CUDA 프로그래밍을 할 때 커널 외에는 "<<< >>>"과 같은 기호를 사용하는 호출 방식을 사용하지 않기 때문에 CUDA 파일을 열어서 "<<< >>>" 기호가 사용된 부분을 찾은 결과를 커널로 간주한다. 위의 커널 호출 형태 패턴을 보면 "커널 명<<<스레드 블록 크기>>>(전달 인자);" 나타나는 것을 알 수 있다. 따라서 CUDA 파일에서 "<<< >>>" 이전 위치의 문자열을 커널 명, 내부 문자열을 스레드 블록 크기, 그리고 "<<< >>>" 기호 오른쪽의 여는 괄호부터 닫는 괄호까지의 문자열을 전달 인자로 간주하여 {커널 명 : [스레드 블록 크기, 전달 인자]} 형태로 Python에서 키(key)와 값(value)을 쌍으로 저장하는 Dictionary 형태에 저장한다.

2) 입력 데이터 추출 및 저장

CUDA 파일 내에 있는 커널을 분리해서 새로운 CUDA 파일에서 호출하고자 할 때 커널에 전달되는 데이터가 커널을 분리하지 않고 호출했을 때 전달되는 데이터와 값이 같아야 정상적인 커널 디버깅이 가능하다. 따라서 CUDA 프로그램을 실행하다가 분리하고자 하는 커널 호출 명령이 있는 부분 직전에 커널에서 필요로 하는 데이터들을 저장하도록 한다.

데이터를 저장할 때 프로그램 내 모든 변수의 값을 저장하는 것이 아니라 분리할 커널의 전달 인자로 사용될 데이터만 필요하다. 따라서 앞서 저장해둔 커널 딕셔너리 변수에서 분리할 커널 명을 딕셔너리 변수의 키로 접근하여 해당 커널이 필요로 하는 값을 얻는다. CUDA 파일에서 반복자를 사용하여 이 값들을 차례대로 순회하여 데이터를 저장하는 명령을 수행하도록 Python에서 문자열로 CUDA C 코드를 작성하여 CUDA 파일 내 분리할 커널이 호출되는 위치를 찾아 위치 직전에 문자열을 삽입하여 CUDA 파일을 실행했을 때 데이터 파일 작성을 수행할 수 있도록 한다.

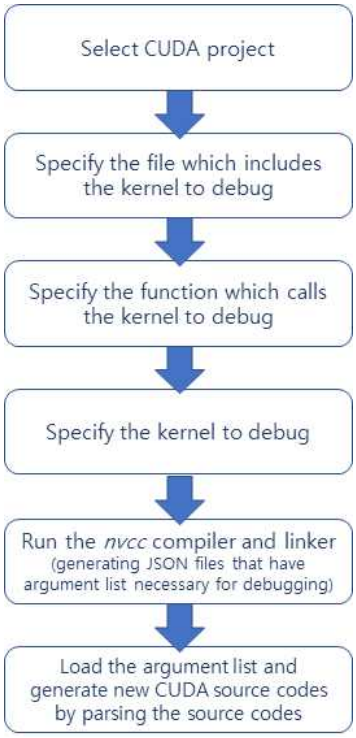

그림 4. CUDA 코드 분리 처리 알고리즘
Fig. 4. Algorithm for CUDA code separation

CUDA 프로그램에서는 크게 CPU(Central Processing Unit)에서 사용하는 변수(호스트 변수)와 GPU에서 사용하는 변수(디바이스 변수)로 구분된다. 호스트 변수는 CPU 메모리에 접근할 수 있지만 GPU 메모리에 접근할 수 없다. 파일을 쓰기 위한 함수인 fprintf()는 CPU에서수행하기 때문에 메모리 접근 제한으로 인해 디바이스 변수의 값을 바로 가지고 올 수 없다는 문제가 발생한다. 따라서 먼저 호스트 또는 디바이스 변수를 구분하기 위해 디바이스 변수에 "d_"를 붙여 변수 이름을 설정한다는 규칙을 적용하였다. 이후 디바이스 변수에 저장된 값을 복사하는 CUDA 함수인 cudaMemcpy()를 사용하여 디바이스 변수의 값을 호스트 변수에 복사하는 코드를 삽입하여 디바이스 변수에 저장된 데이터를 JSON 형식의 파일에 쓸 수 있도록 하였다.

결과적으로 CUDA 파일에 fprintf() 함수로 JSON 파일을 작

```
JSON += '\
FILE *fp = fopen("./Input.json", "w");\n \
fprintf(fp, "{\\n");\n'
for idx in range(len(self.gridandblock)):
    if idx >=2:
        JSON+= 'fprintf(fp, "    %d,\\n", '+self.gridandblock[idx]+');\n'
        self.gridandblock[idx] = 'shared'
        self.all_json_arg.append(self.gridandblock[idx])
    else:
        JSON += 'fprintf(fp, "    \\"kernelsize'+str(idx)+'\\" : [\\n");\n '
        JSON+= 'fprintf(fp, "    %d,\\n", '+self.gridandblock[idx]+'.x);\n'
        JSON+= 'fprintf(fp, "    %d,\\n", '+self.gridandblock[idx]+'.y);\n'
        JSON+= 'fprintf(fp, "    %d\\n", '+self.gridandblock[idx]+'.z);\n'
        JSON+= 'fprintf(fp, "], \\n"); \n'
        self.all_json_arg.append(self.gridandblock[idx])
```

그림 5. CUDA 파일에 삽입할 JSON 파일 작성하는 코드를 문자열로 처리하는 Python 코드
Fig. 5. Python code that treats code creating a JSON file to be inserted into a CUDA file as a string

성하는 코드를 삽입하기 위해 Python 환경에서 CUDA 파일을 열어서 C언어 문법으로 문자열을 만들어 Write() 함수로 CUDA 파일을 수정한다. 그림 5는 CUDA 파일에 C언어로 JSON 파일을 작성하는 코드를 삽입하는 Python 코드이다.

이 코드와 같이 CUDA 파일에 JSON 파일을 작성하는 코드를 통해 JSON 파일에 CUDA 프로그램의 변수 값을 저장하면 디버깅하려는 커널 직전까지 처리된 데이터 값을 JSON 파일을 통해 확인할 수 있다.

JSON 파일을 생성하는 과정을 자동화하기 위해 Python 환경에서 CUDA C 코드를 수정한 후 Python의 os 모듈을 사용하여 CUDA 컴파일 및 링크 명령어인 “nvcc –link [cuda 파일 경로]”를 수행하여 실행 파일을 빌드한다[8]. 실행 파일이 생성되면 python에서 os.system(“file.exe”) 명령으로 CUDA 프로그램을 실행하여 JSON 파일 생성을 완료한다.

표 1은 실제로 그림 4의 Python 코드를 실행하여 수정된 CUDA 파일을 실행했을 때 만들어진 JSON 파일이다. JSON 파일에서 키 값으로 사용된 문자열은 CUDA 프로그램의 변수명이며 CUDA 프로그램 내 변수가 가진 값과 JSON 파일에 저장된 값이 같음을 확인하였다.

3) 커널 단위 테스트 CUDA 파일 생성

기존 CUDA 파일에서 파싱한 전처리문 부분, 분리할 커널 구현부와 커널 정보(커널 명, 스레드 블록 크기, 전달 인자), 전달 인자로 사용될 데이터를 활용하여 분리된 커널만 실행할 수 있는 새로운 CUDA 파일을 작성한다. Python 환경에서 CUDA 파일을 수정하기 위해 문자열로 CUDA C 코드를 작성했던 것과 마찬가지로 Python에서 파싱해둔 데이터들을 토대로 CUDA C 코드를 문자열로 작성해 파일 쓰기를 수행한다.

기존 CUDA 파일에서 사용하는 라이브러리 파일이나 함수를 수정하지 않고 새로운 CUDA 파일에 작성함으로써 분리된 커널을 실행할 때 기존 CUDA 파일에서 실행하는 것과 같은 환경을 생성한다.

main 함수에서는 커널 호출과 함께 호스트와 디바이스 변수를 선언하여 전달 인자로 사용될 데이터를 변수에 저장한다. 데

표 1. CUDA 소스 코드 내 변수의 값이 저장된 JSON 파일
Table 1. JSON file to store values of variables in CUDA source code

```
{
    "kernelsize0":{
    20,
    1,
    1
  },
    "kernelsize0":{
    512,
    1,
    1
  },
  "numOfNeighbors" : 8,
  "numDataPoints" : 10000,
  "host_d_2kNNIndices":{
    7856,
    29,
    16,
    .
    .
    .
  }
}
```

이터는 JSON 파일로 저장되었으며 CUDA C에서 JSON 파일을 처리하기 위해 RapidJSON 라이브러리[9]를 사용하였다. 그림 6은 RapidJSON 코드를 추가하는 Python에서 구현한 CUDA C 문법 형태의 문자열이다.

전달 인자 데이터를 JSON 파일로 저장할 때 각 변수의 값을 모두 배열 형태로 처리했기 때문에 CUDA C에서 Rapid JSON을 사용하여 JSON 파일을 읽을 때 RapidJSON에서 제공하는 Size() 함수를 사용하여 데이터 배열 길이만큼 반복하는 형태를 구성한다.

반복문 내부에서는 JSON 파일의 키 값을 CUDA C의 호스트 변수명으로 사용하고 RapidJSON으로 데이터를 읽어서 선언된 호스트 변수에 저장한다. 디바이스 변수의 경우 바로 RapidJSON으로 읽은 데이터를 입력할 수 없기 때문에 임시로 호스트 변수를 만들어 데이터를 저장해두었다가 cudaMalloc() 함수로 디바이스 변수를 동적 할당하고 cudaMemcpy() 함수를 사용하여 앞에서 임시로 만든 호스트 변수의 데이터를 디바이스 변수로 복사하여 GPU에서 사용할 수 있도록 한다.

4) 생산성 향상 및 사용성 강화를 위한 GUI 환경 제공

본 논문에서 제안하는 방식은 생산성을 향상시키고 사용성을 강화하기 위한 옵션 중 하나로 GUI(graphical user interface) 환경에서 구현하였다.

```
RapidJSON = '#include "rapidjson/include/rapidjson/reader.h"\n\
#include "rapidjson/include/rapidjson/prettywriter.h"\n\
#include "rapidjson/include/rapidjson/filereadstream.h"\n\
#include "rapidjson/include/rapidjson/filewritestream.h"\n\
#include "rapidjson/include/rapidjson/writer.h"\n\
#include "rapidjson/include/rapidjson/document.h"\n\
#include "rapidjson/include/rapidjson/error/en.h"\n\
using namespace rapidjson;\n\n'
```

그림 6. RapidJSON 포함시키는 CUDA C 코드 형태의 문자열
Fig. 6. Python string in CUDA C code form to include RapidJSON

이러한 목표를 달성하기 위해 PyQt 라이브러리를 사용하여 CUDA 파일 수정 및 데이터 파싱, 새로운 CUDA 파일 작성을 사용자가 직접 커맨드 라인으로 실행시키지 않아도 간단히 GUI 버튼 클릭 이벤트로 전 과정을 처리할 수 있도록 하였다 (그림 7).

그림 7. CUDA 커널 단위 테스트 생성을 위한 GUI 윈도우
Fig. 7. The GUI window for creating CUDA kernel unit test

Ⅳ. 구현 및 테스트

본 논문에서 제안하는 방식의 검증을 위해 Python 3.7.4, CUDA 10.1 Toolkit, Visual Studio 2019 환경에서 구현하였고 실제 다른 연구를 위해 작성한 코드들 중 Nsight 디버깅 시에 중단 현상이 발생했던 코드를 대상으로 구현한 코드를 실행하여 테스트하였다.

4-1 스레드 상태 확인 용이성

CUDA 아키텍처는 하드웨어 측면에서는 한정적인 개수의 병렬 프로세서 코어를 가지고 있는 구조이다. 그러나 CUDA의 프로그래밍 모델이나 실행 모델[2]에서 볼 때는 물리적인 병렬 프로세서 코어 개수보다 훨씬 많은 동시 실행 스레드를 생성하게 된다. 실제로는 병렬 프로세서 코어 개수에 따라 물리적으로 동시에 실행 가능한 스레드의 개수가 한정되어 대략 (스레드 개수) / (코어 개수)만큼 순차적 실행이 이루어진다. 그러나 개발자나 사용자 입장에서 논리적으로는 모든 스레드가 병렬적으로 처리되는 것으로 인식되도록 보장하기 위한 여러 메커니즘들을 제공한다. CUDA의 이러한 물리적 구조와 논리적 구조의 차이 때문에 다수의 커널을 논리적으로 병렬적으로 실행하는 CUDA 프로그램 스레드마다 수행 중인 명령이 다를 수도 있는 한편, Nsight 및 하드웨어 용량의 제약으로 인해 디버깅해야 하는 커널이 많을수록 디버깅 작업 시에 커널별 처리 과정의 판단이 어려운 경향이 있다. 반면에 단일 커널을 실행하는 CUDA 코드는 디버거가 필요한 자원이 상대적으로 적기 때문에 분리된 커널을 용이하게 디버깅할 수 있다.

표 2. 단일 커널 예제
Table 2. Single kernel example

```
extern “C”
int main(){

FILE* fp = fopen("Input.json", "r");
char readBuf[65536];
FileReadStream is(fp, readBuf, sizeof(readBuf));
Document d;
d.ParseStream(is);
fclose(fp);
.
.
.
// kernel call
keep_K_NearestNeighborsEuclideanDistSelf<< <grid, block, shared >> >
(d_AllNNIndices, d_AllNNEuclideanDistances, d_2kNNIndices,
d_2kNNEuclideanDistances, hd, numOfNeighbors, numDataPoints);

return 0;
}
```

표 2에서는 main 함수에서 분리된 단일 커널을 호출하는 코드의 사례를 제시한다.

4-2 분리한 단일 커널에 대한 디버깅 수행

지금까지 논의한 내용을 구현하여 특정 커널 하나와 런타임을 통해 얻은 해당 커널에 투입될 입력 정보를 전체 코드에서 자동으로 분리한 후 디버깅을 수행하였다.

그 결과 다수의 커널을 호출하는 원본 CUDA 코드에서는 Nsight로 디버깅을 수행할 때 디버깅 작업이 중단되었으나 본 논문에서 제안하는 방식으로는 각 커널 단위로 하나씩 성공적으로 디버깅을 수행할 수 있었다.

이 결과는 비교적 간단하고 직접적인 방법으로 볼 수 있지만, 실제 전체 코드를 대상으로 하였던 Nsight 디버깅에서 파악하기 힘들거나 놓친 오류를 원활하게 확인할 수 있어 실질적인 측면에서 유용성이 매우 높았다.

그림 8에서는 여러 커널들이 실행되는 원본 코드에서는 디버깅이 불가능했던 코드를 단일 커널별로 구분하여 성공적으로 GPU 내부 정보를 파악하고 있는 화면을 제시한다.

Ⅴ. 결 론

5-1 결과 논의

본 연구는 Nsight의 단점을 극복하기 위한 시도로서, 복잡한 연산을 병렬로 처리하는 CUDA 코드 내에서 관심이 있는 특정 커널과 실행 정보를 분리하기 위하여 Python 환경에서 CUDA 파일을 파싱하여 관심이 있는 단일 커널로 구성되는 새로운 CUDA 파일과 원본 코드에서 실행 중 해당 커널에 투입되는 정보를 추출한다. 일반적으로 특정한 커널에 투입되는 정보는 이전 커널들에서 병렬 처리로 계산되어 관심 있는 커널로 투입되기 때문에 이 정보는 원본 CUDA 코드를 GPU에서 실행하는 도중에 추출해야 한다. 이후 추출한 데이터들을 기반으로 RapidJSON 라이브러리를 포함한 CUDA 코드를 문자열로 구성하여 새로운 CUDA 파일을 작성하였다. CUDA 파일 수정과 데이터 추출을 위한 CUDA 프로그램 컴파일 및 링크 작업을 Python에서 일괄적으로 처리함으로써 CUDA 커널 분리 과정을 구분하지 않고 한 번에 수행하도록 하였다. 또한, 사용성과 생산성 향상을 위하여 PyQt로 GUI 애플리케이션을 구현하고 커널 분리 작업을 수행할 수 있게 하였다. 그리고 CUDA 파일을 분리하기 위해 CUDA C 문법과 디바이스 변수나 커널을 호출하는 main 함수 앞에 접두어를 사용하는 규칙을 파싱 기준으로 적용함으로써 문법적인 예외가 발생하더라도 민감하게 오류가 발생하지 않도록 하였다. 실제 사용되는 CUDA 프로그램을 바탕으로 본 연구에 대한 실험을 진행하였을 때 이전에 발생했던 디버깅 동작 중지와 같은 문제를 해결할 수 있었으며 전보다 쉽게 스레드 상태나 변수 값을 확인함으로써 오류를 수정할 수 있어 매우 실용적인 방식으로 판단한다.

향후 현재 적용한 여러 가정과 제약을 극복하여 보다 유용한 도구로 발전시켜 나갈 계획이다.

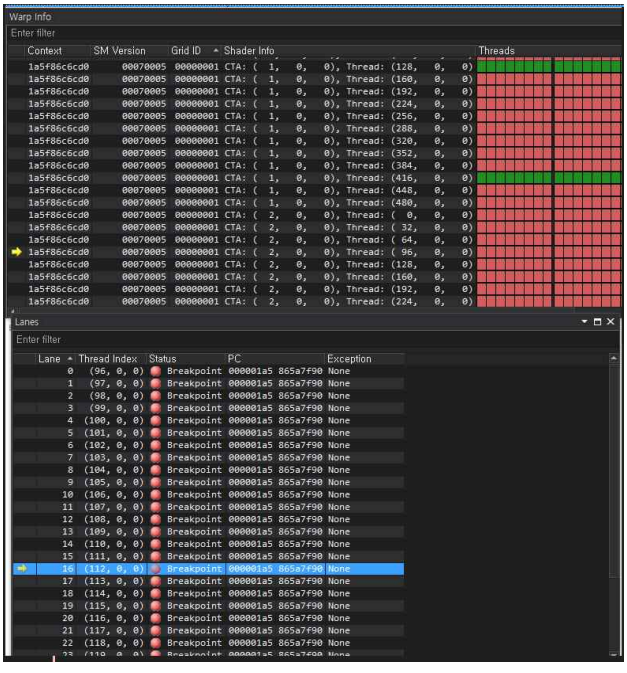

그림 8. 정상적인 디버깅 시 나타나는 Warp 정보와 Lanes 창
Fig. 8. Warp info and Lanes windows that appear during successful debugging

감사의 글

본 연구는 덕성여자대학교 2018년도 교내연구비 지원에 의해 수행되었음.

참고문헌

[1] H. Kim and T. Park, “Analysis of GPU-based Parallel Shifted Sort Algorithm as an Alternative to General GPU-based Tree Traversal,” *The Journal of Digital Contents Society*, Vol. 18, No. 6, pp. 1151-1156, October 2017.

[2] NVIDA Nsight Visual Studio Edition. Available: https://developer.nvidia.com/nsight-visual-studio-edition

[3] J. Cheng, M. Grossman, and T. McKercher, Professional CUDA C Programming, 1sted. Wrox, chapter 2 and 3, 2014.

[4] T. Park, “Optimization of Warp-wide CUDA Implementation for Parallel Shifted Sort Algorithm,” Journal of Digital Contents Society, Vol. 18, No. 4, pp. 739-745, July 2017.

[5] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic

analysis of CUDA programs. In STMCS, 2008.
[6] T. Park, “Development of Visualization and Debugging Environment for GPGPU Parallel Processing of Geometry Information Based on Game Engine,” Journal of The Korean Society for Computer Game, Vol. 30, No. 3, pp. 19-26, September 2017.
[7] WB. Langdon, “Debugging CUDA”, GECCO '11 Proceedings of the 13th annual conference companion on Genetic and evolutionary computation, pp. 415-422, July 2011
[8] NVIDIA. NVCC Commmand Options. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html
[9] RapidJSON Website. Available: https://rapidjson.org/

양정아(Jung Ah Yang)

2018년 : 덕성여자대학교 정보미디어대학 디지털미디어학과 학사 (공학사)
2020년 : 덕성여자대학교 공과대학 IT미디어공학과 대학원 석사

2020년~현재 : 스칼라웍스
※관심분야 : 컴퓨터그래픽스, 게임프로그래밍, 인공지능

박태정(Taejung Park)

1997년 : 서울대 전기공학부 (공학사)
1999년 : 서울대 전기공학부 대학원 (공학 석사, 반도체 물리 전공)
2006년 : 서울대 전기컴퓨터공학부 대학원 (공학박사, 컴퓨터 그래픽스 전공)

2006년～2013년 : 고려대학교 연구교수
2013년～2017년 : 덕성여자대학교 정보미디어대학 디지털미디어학과 조교수
2018년～현재 : 덕성여자대학교 공과대학 IT미디어공학과 부교수
※관심분야 : 컴퓨터그래픽스, 병렬처리, 인공지능, 수치해석, 3차원 모델링